XVT Portability Toolkit Guide

Published By

Providence Software Solutions, Inc.
201 Shannon Oaks Circle, Suite 200
Cary, NC 27511 USA

Revision History

First Printing ..... December, 1996 ..... DSC Release 4.5

Second Printing..... June, 1999..... DSC Release 5

Revised..... July, 2002 ..... DSC Release 5.5

Revised..... December, 2004 ..... DSC Release 5.6

Revised..... April, 2007 ..... DSC Release 5.8

Revised..... March, 2009 ..... DSC Release 2009.1

GUIDE

PREFACE

This *Guide* presents a basic yet thorough treatment of portable GUI programming with XVT's Portability Toolkit. XVT offers a Development Solution for C (DSC) and a Development Solution for C++ (DSC++). The XVT Portability Toolkit is the portable API for both DSC and DSC++.

This *Guide*, organized by subjects, complements the *XVT Portability Toolkit Reference*, which is an alphabetical listing of Application Programming Interface (API) elements. The *Guide* aims to get you programming as quickly as possible with XVT. On the other hand, you'll want to refer to the *Reference* throughout your development efforts.

How to Use the Guide

To get the most out of this *Guide*, XVT recommends the following approach:

- Read Chapters 1–5 to familiarize yourself with the XVT platform-independent approach to application programming.
- Read Chapters 6–9 to learn about the basic XVT building blocks.
- Read subsequent chapters to the depth required. The chapters are organized so that deeper levels of subject matter detail are treated in later chapters. Initially, your reading should concentrate on the top levels, to get a basic understanding. As you need more depth, you can read the chapters containing more detail.

If you are an XVT-Design user, work through the Tutorial chapter of the XVT-Design Manual. The XVT-Design Manual introduces you to using XVT-Design in conjunction with the XVT Portability Toolkit. The tutorial gives you an experiential appreciation for the capabilities of XVT-Design and the XVT Portability Toolkit. It introduces you to the GUI application structure and to laying out GUI objects, setting their attributes, constructing menubars, setting connections and seeing them in operation through Testmode, and entering source code in the Action Code Editor.

Other XVT Documentation

XVT provides many different documents, in PDF formats:

Release Notes and Installation Instructions
Platform-specific information about how to install the DSC is placed in the installation instructions. The release notes tell you what is new or changed with this release. This information, which includes environment configuration for your platform, appears on your distribution media, in the **install.txt** file.

XVT Portability Toolkit Guide
This manual presents a basic yet thorough treatment of portable GUI programming with the XVT Portability Toolkit (PTK).

XVT Portability Toolkit Reference
This documentation contains reference information for all API elements of the XVT PTK: portable attributes, events, data types, constants, and functions.

XVT Portability Toolkit Quick Reference
This small document encapsulates the *XVT Portability Toolkit Reference*, along with details about the XRC language.

XVT Platform-Specific Books
Each book contains information you need to use the PTK on a particular XVT-supported platform. The information about non-portable attributes and escape codes is especially useful.

XVT-Design Manual
This manual is both guide and reference for XVT-Design, the visual programming tool that is included with XVT Development Solution for C (DSC). It introduces you to portable GUI programming using XVT-Design.

XVT-PowerObjects GUI Components Pak 1

This document is both guide and reference for the GUI components, XVT-PowerObjects, included with the XVT Development Solution for C.

XVT Technical Notes

XVT provides Technical Notes on your distribution media, in the **doc** directory.

Guide to XVT Development Solution for C++

This manual presents an introduction to portable GUI programming using XVT-Architect and the DSC++ framework. It also introduces you to the application–document–view paradigm that is the cornerstone of XVT-based object-oriented programming.

XVT DSC++ Reference

This reference manual describes all the classes of the DSC++ framework.

XVT Development Solution for C ++ Quick Reference

This small document encapsulates the *XVT Power++ Reference*.

The documentation is designed to give you the information you need to use the XVT Portability Toolkit at all levels, from introductory (the *Guides*, particularly the first nine chapters) to advanced (the *References*).

About This Manual

XVT takes pride in its documentation, and continually seeks to improve it. If you find a documentation error, please contact XVT 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). Italso 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, for the names of other documents, and in cross-references to chapters inside the same document.

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).

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.

XVT CUSTOMER SUPPORT

When you buy an XVT product or an XVT maintenance agreement, you gain access to some of the most advanced application development assistance in the industry.

If you have problems or questions while using XVT products, you can talk to an XVT Customer Support Engineer. XVT Customer Support helps you make more effective use of XVT products, enabling you to get your application up and running as quickly as possible. Customer Support is available to customers who have purchased an XVT product andhave a current maintenance contract.

Please note that only one individual per purchased copy of an XVT product may request support. Questions will be taken only from the individual named on the software registration form.

Feel free to contact XVT Customer Support if you have a question, or would like to suggest a software enhancement or a change to any document. Your call is always welcome.

How Customer Support Works

XVT's Customer Support goal is to respond to all requests within twenty-four hours. As soon as we log your call into our system, you will receive a service request number.

If we have questions about your request, we ask that you respond to them within five working days. Please let us know if you need more time; otherwise, if we receive no response from you after five days, your service request will be closed.

What XVT Customer Support Provides

XVT Customer Support can serve you better if you understand what services are available.

This is what XVT's Customer Support can do:

- Provide "tips" to help you effectively use XVT functionality
- Explain XVT functionality and specifications
- Diagnose and analyze XVT-related application problems
- Suggest workarounds
- Suggest how to access native window system development tools
- Collect feedback for future product development

Keep in mind that XVT Customer Support *cannot* do the following:

- Design customer applications
- Debug user code
- Explain how to use operating systems, window systems, or compilers (except with regard to XVT application resources)
- Explain how to use native window system development tools
- Extensively teach XVT programming

If you need more help than Customer Service can provide, consider contacting XVT's Professional Services Group. More information about this group can be found on the last page of this section of this *Guide*.

Customer Support Services

XVT's Customer Support engineers can answer questions that arise from the use of a native GUI platform, or the operating system itself (see the following subsections, "Standard Customer Support Services" and "Extended Support Services"). When questions require investigation, you are given a follow-up reference number that identifies your inquiry in our Customer Support database. These pending requests for information are reviewed several times each day, to ensure a timely reply.

Standard Customer Support Services

XVT Customer Support personnel are experienced software developers that specialize in the use of XVT products on supported MS-Win32, Motif, and Macintosh computer systems. For registered named users, XVT offers the following standard services:

- Easy access by phone, fax, and electronic mail
- One day (maximum 24 hour) call backs after your initial call is received
- An assigned Service Request number when your questions are logged, so they are tracked and responded to efficiently
- Tips that speed the use of XVT functionality
- Explanations of XVT functionality and specifications to reinforce product manual descriptions
- Suggested workarounds for common development obstacles
- Suggestions for complementary products or native window system environment tools that might enhance your application or make you more productive
- Product enhancement requests are tracked and analyzed so that customers significantly influence future product development decisions

Extended Support Services

XVT recognizes that customers sometimes need comprehensive assistance—the type of assistance and advice that has a broader scope than just the XVT products that the customers have purchased. The XVT Customer Support Extended Service Contract is a convenient way to make use of XVT's development expertise. Extended Service allows you to:

- Get help debugging small user code examples
- Quick identification of common mistakes made during cross-platform development
- Get information about operating systems, window systems, or compilers you may not be familiar with (except with regard to XVT application resources)
- Get guidance about how to use native window system development tools that may be new to you

Note: You do not need to sign up for Extended Service in advance.

FTP Site

XVT's FTP site is available to all currently registered customers, and offers valuable information for XVT application developers. Technical papers and notes, programming examples, product updates, and programming utilities are available.

Support for XVT Software Purchased from Distributors

XVT products are sold around the world, often through an independent distributor licensed by XVT Software Inc. If you purchased your XVT product through an international distributor, your customer support requests must be routed through that distributor. If, on the other hand, you purchased your XVT product from an international office of XVT, you may contact XVT directly for support. Instructions for contacting XVT Customer Support are listed on the last page of this section of this *Guide*.

Information We Need to Help You

When you contact XVT Customer Support, please supply the following information:

- The name and version number of the product (for example, XVT/Win32 5.6)
- The product serial number (found on your distribution media)
- Your platform type (for example, IBM RS/6000)
- The operating system and version number (for example, HP-UX 11i, Solaris 9)
- The compiler and version number (for example, THINK C 3.0 or Sun One Studio 7)
- The window manager and version number (for example, Windows 2000)
- A detailed description of the problem, including information displayed with any internal error message—such as the called function, filename, and code line

Product Updates

Providence Software actively updates XVT. For most minor releases, and for all major releases, Providence supplies additions to or complete replacements for XVT documentation. As a service to our customers, all product updates are made available from Providence Software's FTP site which can be accessed from our website at:

www.xvt.com

Tip: Customers who are current on their XVT maintenance agreements can download product updates from the FTP site.

How to Contact Customer Support

You can contact Software Customer Support for XVT in several different ways:

- Telephone us at (919) 854-1800 x 201 (8:30 AM to 5:30 PM, Eastern Standard Time, Monday-Friday)
- Send us electronic mail via the Internet at xvt-support@xvt.com
- Write us at Providence Software Solutions, Inc., 201 Shannon Oaks Circle, Suite 200, Cary, NC 27511 USA

XVT's Consulting and Training Services

Providence offers extensive fee-based services to help customers use XVT products. Experienced professionals can help you learn GUI programming, or help you prototype, design, code, debug, and maintain your XVT applications.

In addition to consulting, Providence personnel also conduct on-site and public training classes in XVT and GUI programming techniques.

See Also: For more information about the Providence Software's professional services, contact us at:

Phone: (919) 854-1800

Email: sales@xvt.com

XVT LICENSE MANAGEMENT

You must have a valid license from Providence Software Solutions to use the XVT products. XVT products such as the `xrc` resource compiler, XVT Design and XVT Architect will not operate.

The XVT license can be either a node-locked license or a floating license. A node-locked license will permit you to use XVT on the individual computer you identify. A floating license will allow you to use a specified number of copies of your XVT product simultaneously; each individual license is supplied by a designated license server on your network.

Node-locked licenses

A node-locked license will give access to XVT on a particular computer, based on its unique computer identification. You first run a utility provided by Providence Software to determine your computer's id. Providence uses that id to generate a license file for your system.

The license file that is provided to you is named XVT.elm and must be placed in the bin directory under the root or top-most directory of your XVT installation. An environment variable named XVT_DSC_DIR or XVT_DSP_DIR must be set to point to the root of your XVT installation, so that the XVT.elm file can be found by XVT applications at $XVT_DSC_DIR/bin/XVT.elm or $XVT_DSP_DIR/bin/XVT.elm.

Floating licenses

Floating licenses are available on a local network. The number of users specified in your contract can connect to the license server when a license is needed.

A system on your network is designated as the server and runs an application named LMNetServer. This application checks its own license file, LMNetServer.elm. This file specifies the number of simultaneous licenses that can be made available. The LMNetServer.elm file must be located in the same directory as the LMNetServer executable is run from.

Users (or clients) of the floating license have an XVT.elm file on their system that identifies the location of the system with the LMNetServer license server running.

As with locked licenses, the XVT.elm file placed in the bin directory under the root or top-most directory of your XVT installation. An environment variable named XVT_DSC_DIR or XVT_DSP_DIR must be set to point to the root of your XVT installation, so that the XVT.elm file can be found by XVT applications at $XVT_DIR/bin/XVT.elm or $XVT_DSC_DIR/bin/XVT.elm.

1

INTRODUCTION TO THE XVT PORTABILITY TOOLKIT

XVT has implemented the XVT Portability Toolkit™ (PTK) as a thin layer on top of the native GUI Application Programming Interface (API). The PTK provides access to native functionality, without overloading your application's performance or size. A layered approach is efficient, native, and open, which allows access to native features. XVT's approach gives your customers the native look-and-feel results they expect.

1.1. The Elements of an XVT Application

1.1.1. Building Blocks

An XVT-based application usually comprises the following components:

- Code that defines the GUI components and layout of the application's interface
- Code that defines the application behavior of each GUI component, i.e., how (and when) they respond to events
- Source code modules that implement the functions needed by individual GUI components
- Additional source modules for the non-GUI parts of the application
- Text for the application's help system
- Bitmap images for the help system or other parts of the application to display

1.1.2. GUI Objects

A graphical user interface (GUI) has four main types of graphical objects: windows, dialogs, controls, and menus.

All the GUI objects provided by the XVT Portability Toolkit have a number of attributes that describe their appearance and behavior. For example, windows might be sizeable or iconizable, or might contain scrollbars or titlebars.

See Also: For a comparison of the XVT GUI components, and for information about attributes that are common to all of them, refer to Chapter 3, *GUI Elements*.
For more information on specific GUI object attributes, see Chapters 6, 7, 8, and 9: *Windows*, *Dialogs*, *Controls*, and *Menus*.

1.1.3. Events and Event Handlers

XVT bases its Portability Toolkit on a set of abstract, portable event representations. Abstract events deliver user and GUI system event data to GUI objects within your application.

An event handler is a function with the proper prototype for receiving events, meaning that it accepts a WINDOW and an EVENT* as arguments.

Most windows—and all dialogs—must be assigned an event handler to process the events generated during their lives. The exception is the screen window, which has no event handler because it receives no events. Windows can have unique event handlers, or multiple windows can share a common event handler.

See Also: For more information, see Chapter 4, *Events*.

1.2. XVT's Development Solutions

Using either one of XVT's Development Solutions, you can produce an extensive graphical application and only write a modest amount of new code.

Note: This *Guide* describes the functionality of the XVT Portability Toolkits, which are implemented in C. As shown in Figure 1.1, the XVT Portability Toolkits are utilized by both of XVT's Development Solutions.

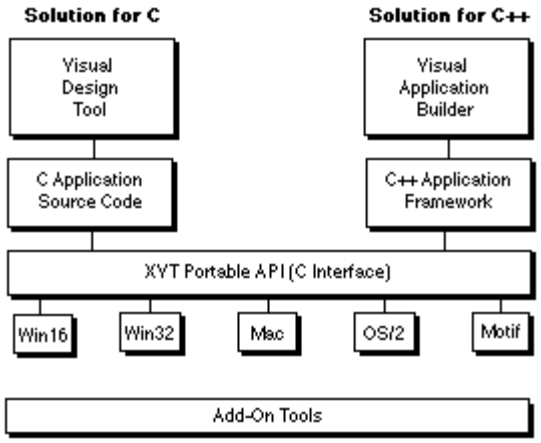


Figure 1.1. XVT Portability Toolkits — the foundation of a well-written, versatile, and maintainable application

The visual design tool and/or the visual application builder generates many of the files your application needs. The visual design tool generates the makefile, various source and header files, and a resource file. The visual application builder generates both Shell files and Factory files. The clean separation of code makes it easy for you to change and maintain your application. Thus, the visual design tool and the visual application builder are provided to assist you during the maintenance, as well as the development, of your applications.

Tip: If your application end users demand strict conformance to native look-and-feel, you can meet this demand by programming directly to the native GUI toolkits. For more information about how to program at this level using XVT products, refer to section 1.3.1.2 on page 1-5.

1.2.1. XVT Development Solution for C

XVT Development Solution for C (DSC) is an environment for writing and maintaining portable, extensible C application programs. Using XVT Development Solution for C, you can deploy your C programs to users working with a variety of windowing systems with a minimum amount of effort on your part. Incidentally, this *Guide*, the *XVT Portability Toolkit Guide*, also functions as the *Guide to XVT Development Solution for C*.

1.2.2. XVT Development Solution for C++

XVT Development Solution for C++ (DSC++) contains a robust, object-oriented application framework designed specifically for portable C++ GUI development.

This *Guide* mentions XVT Development Solution for C++ to complete your view of XVT's Development Solutions. For more information about XVT Development Solution for C++, see the *Guide to XVT Development Solution for C++*, a separate manual available from XVT.

1.3. Cross-platform GUI Development

1.3.1. Extensible Programming with XVT

As you develop your portable application, XVT lets you "extend" it beyond XVT's Portability Toolkits to native GUI functionality. This is a practical and powerful aspect of XVT's programming model, and something inherently available because of XVT's layered architecture.

XVT's approach to portability supports native look-and-feel and extensible programming, instead of GUI emulation. This approach requires developers to consider certain cross-platform issues: bugs within particular platforms, technical specification inconsistencies, and native look-and-feel differences.

Tip: To successfully develop XVT-based applications, XVT recommends that you port frequently in the early design stages. Also, you should have access to the proper hardware required for porting to different platforms.

1.3.1.1. System Attributes Feature

XVT's approach to GUI development includes a system attribute feature, supported by two general purpose functions. Given an attribute identifier, these functions either set a system attribute to a new value, or retrieve the current value of the attribute for the application. XVT supports two sets of attributes:

- Portable attributes (available on all XVT Portability Toolkits)
- Non-portable, platform-specific attributes

The platform-specific attributes found in each XVT Portability Toolkit let you access native GUI functionality that is not a part of the XVT portable programming interface.

1.3.1.2. Native Access Functions

In addition to platform-specific system attributes, each platform has a documented set of access functions that allow you to take advantage of unique features found in all native GUI systems. These functions let you interface directly with the native GUI toolkit. The XVT Portability Toolkits support the following platform-specific features:

- Native graphics device contexts
- Native window handles and identifiers
- Native event queue access (for manipulating native events)
- Keyboard translations
- Native fonts
- Custom task window configurability
- Native printing and graphical attribute options

See Also: For more information about system attributes, see section 2.4 in Chapter 2, *About the XVT API*.
For more information about specific non-portable attributes, see the *XVT Platform-Specific Books*.

1.3.2. Cross-platform Development Process for C

If you are programming with C, developing an XVT-based application involves following these general steps:

1. Analyze the requirements of your target audience with regard to such things as performance, locale, communication, equipment, external databases, functionality, and so forth.

2. Associate application behavior with GUI components, i.e., how (and when) objects behind the user interface communicate with one another.
3. Build source code modules that implement the functions needed by individual GUI components.
4. Build additional source modules for the non-GUI parts of the application.
5. Generate text for the application's help system.
6. Choose bitmap images for the help system or other parts of the application to display.
7. Write a makefile (whose template you select from the provided examples for the target compiler and platform).
8. Using the text editor of your choice, write a resource file, remembering that when programming with XVT, virtually every aspect of the user interface can be specified using resources.
9. Compile, link, and execute your application.

Application Files

XVT suggests you organize your C application in the following manner:

Makefile
An application makefile, using a template appropriate for the platform/compiler. Alternatively, you can use an IDE project file. The makefile or project file should link libraries from XVT's Portability Toolkit.

Module header and source files
Source (**.c**) and header (**.h**) files for the application module (task window), and for each window, menubar, and dialog in the project.

XVT Resource Compiler (XRC) file
This file defines the external resources of the project.

The general flow of the cross-platform development process for C is shown in Figure 1.2. For more details on the development process for C, refer to the *XVT Technical Overview* that describes the Development Solution for C.

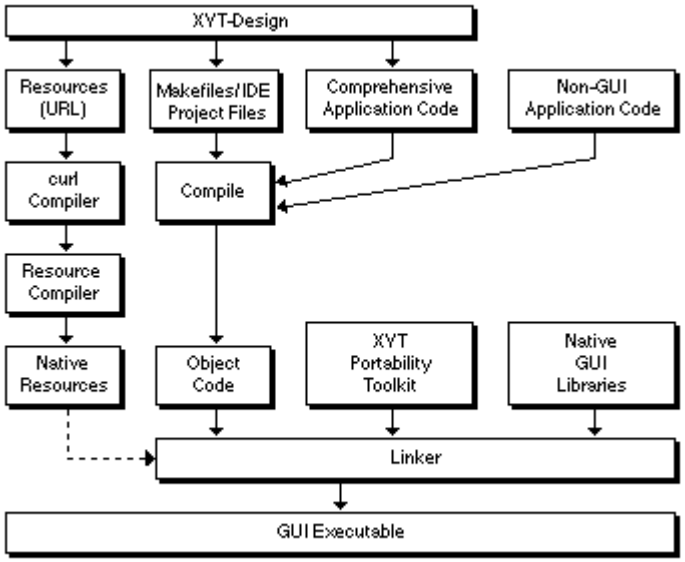


Figure 1.2. Important steps of "C" cross-platform development

1.3.3. Cross-platform Development Process for C++

If you are programming with C++, developing an XVT-based application using the visual application builder, it involves following these general steps:

1. Design and lay out your application using the visual application builder's Blueprint, Drafting Board, and Strata modules, as well as its editors.
2. Generate the Shell files, which include a C++ file and a header file for each application, document, and window class, as well as a startup file, a XRC file, and a makefile.
3. Generate the project's object Factory — a set of C++, header, and resource files that represent what you designed interactively with the visual application builder.
4. Generate a project file or makefile for your compiler, and add all necessary files.
5. Run **xrc** to compile XVT's XVT Resource Compiler (XRC) into a native resource file.
6. Modify the generated Shell files to implement the functionality of your application.
7. From the Shell files, interact with the Factory objects if you need to manipulate GUI objects at runtime.
8. Compile, link, and execute your application.

Application Files

An XVT C++ application is organized in the following manner:

Shell files
The Shell files include a C++ file and a header file for each application, document, and window class, as well as a startup file, a XRC file, and a makefile.

User files
The user files contain the application code that provides the behavior that is specific to your application.

Factory files
The Factory files contain the object information for your project. At runtime, your application code uses its Factory files to instantiate these objects.

The general flow of the cross-platform development process for C++ is shown in Figure 1.3. For more details on the development process for C++, refer to the *Technical Overview for XVT Development Solution for C++*.

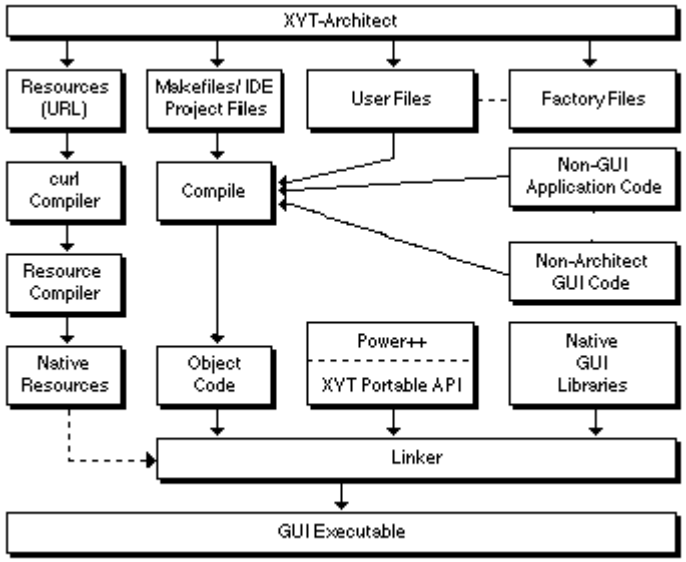


Figure 1.3. Important steps of "C++" cross-platform development

1.4. Getting the Most Out of the PTK

This section introduces you to the XVT Portability Toolkit (PTK) and to some of the utilities you will use with it: XVT's XVT Resource Compiler (XRC), the **xrc** resource compiler, and the **helpc** help text compiler.

This section also briefly mentions how the PTK allows you to internationalize and localize your applications.

1.4.1. XVT Portability Toolkits

The XVT Portability Toolkits are platform-specific C language libraries. As the foundation of XVT's visual programming model, they offer a consistent programming interface for all popular windowing systems.

Each Portability Toolkit implements the XVT interface over native GUI functionality. This ensures native look-and-feel, low overhead, interoperability with other applications, and access to native toolkits when required.

1.4.2. XVT's XVT Resource Compiler

Resources are specifications for menus, dialogs, windows, bitmap images, fonts, and strings—they are kept in a small, read-only database located outside your application's runtime address space. Resources do such things as:

- Set object attributes, such as those that determine the size, position, and alignment of windows, dialogs, and controls
- Establish an object's default appearance, such as initializing its label or title, and also controlling whether it is initially enabled or disabled
- Configure the menubars and menus for application windows

When your application needs a resource, the application requests the resource by an ID number. XVT or the native window system brings the resource into memory so it can be accessed. This saves space at runtime and makes it possible to construct resources without recompiling your C programs.

Furthermore, externalized strings and graphics allow your application to be run in more than one locale using localized resources, if that is a requirement for your organization. XVT provides pre-translated resources for five languages: Japanese, Italian, French, German, and English.

Most programmers find the XVT Resource Compiler (XRC) easy to learn. Since the XRC code is portable, you only need to define your resources once.

With every Portability Toolkit, XVT supplies a compiler for XRC, called **xrc**. You can port your XRC code to any supported XVT platform and compile it to the native resource format using the XVT

compiler, **xrc**. The **xrc** compiler reads specifications in the XVT Resource Compiler and generates specifications in the format appropriate to the native platform.

See Also: For more information about **xrc**, see Chapter 5, *Resources and XRC*.

1.4.3. XVT's helpc Help Text Compiler

XVT's online help feature provides a powerful, flexible, hypertext-based system for your applications:

- **Context-sensitive help**— Your XVT-based applications can provide context-sensitive help to users—in other words, help relevant to the current context or state.
- **Hypertext links** — XVT's help viewer includes hypertext links from the current context to additional topics. To activate a hypertext link, the user clicks on highlighted text.
- **Glossary links** — XVT's help viewer includes glossary links, which define a term or phrase. To activate a glossary link, the user clicks on underlined text.

To create a help system, you establish the context and presentation of the help system's links to the application—for example, an association between a window's creation and text specific to that window, or between a button's operation and another block of text.

You write help text with an editor, using XVT's Markup Language. Then, in the help text file, you specify topics, paragraphs, font changes, bitmap images, and formatting for the text.

XVT's **helpc** compiler reads the text file and creates a binary help file. Your application then calls either the native or XVT's Help Viewer to open a topic window and display the help according to the program's context.

See Also: For more information on this important functionality, see Chapter 22, *Hypertext Online Help*.

1.4.4. Multibyte Character Set and Localization Support

XVT includes support for application development for multiple locales and international languages. All XVT functions, including text edit object functions,handle multibyte strings. String processing API functions portably process multibyte strings.

Your XVT application can receive and process keyboard input that contains international (multibyte) characters. Input Method Editors (IMEs), provided by the native window systems or operating systems, can be used to enter composed characters.

Three multibyte codesets are explicitly supported: ASCII, Shift-JIS, and EUC. Character sets that can besupported must, at least, provide the invariant character set as a subset.

Note: XVT does not *directly* support the Unicode character set. An application can always use this character set by converting to the proper multibyte codeset when calling the XVT API.

1.4.4.1. Externalized Resource Files

XVT applications can allowthe user to select the language/locale of the user interface at application startup time. The user selects the resource file used by the application before invoking the application (DSC++) or before the application calls xvt_app_create (DSC).

All resources are separated from the executable code and can be selected at application startup time. This mainly affects the PC and Macintosh platforms, since the Motif platform has always provided separate resource files. Of course, running any localized application requires that the appropriate operating system, window system, and fonts are installed and set up correctly for the selected language and locale.

XVT has already localized its resources in English, Japanese, French, German, and Italian. For these languages, XVT provides localized files containing all of the standard resources used by the Portability Toolkits. Localized versions of the XVT standard help topics for each platform are also provided in these languages.

XVT's XRC compiler, **xrc**, handles quoted strings containing multibyte characters, including strings used for the following:

- menu and menu item titles
- window, dialog, and control titles
- edit control and text edit object initial text

- font family names
- font mapper native descriptors
- string resources
- user data

Note: XVT can support any left-to-right language. To see a complete list of supported languages, refer to Appendix A, *Languages and Codesets*.

1.4.4.2. More Support for Internationalized Applications

The help compiler, **helpc**, handles help text containing multibyte characters. The help viewer, **helpview**, displays help text containing multibyte characters.

Furthermore, file and pathnames may contain multibyte characters. All PTK functions and data types that accept file or pathname strings are multibyte capable.

The error processor, **errscan**, produces the error message file **ERRCODES.TXT**—you can localize this file for any language. Furthermore, attributes are provided that allow you to explicitly set the path to **ERRCODES.TXT**.

2

ABOUT THE XVT API

XVT's Portability Toolkit provides an application programming interface (API) layered on top of and abstracted from native GUIs. The Portability Toolkit's API forms the foundation of XVT's portable technology.

2.1. The XVT Normalized API Naming Convention

To make your coding easier, XVT uses the following normalized naming convention for its API functions:

xvt_object_operation_qualifier

where each portion of the name has this significance:

xvt_
: Unique XVT prefix to prevent API naming conflicts.

object
: Name of object or subsystem upon which the API function operates (e.g., font, image, win).

operation
: Operation that is performed (e.g., get, create, draw).

qualifier
: Optional qualification used to further specify the operation (e.g., win, font_size, image).

2.2. Objects, Inheritance, and Polymorphism

XVT organizes its normalized API (NAPI) around "objects." An object is an abstraction of user interface components (such as windows) or supporting facilities (such as the file system). The API is a collection of functions that operate on these objects.

2.2.1. Objects

XVT has identified the following "objects" and functional groupings:

Object:	Prefix:	Explanation:
Appl	xvt_app	Application object (global executable context)
Clipbrd	xvt_cb	Clipboard
Control	xvt_ctl	Functionality specific to controls (contrary to windows)
Container Extension	xvt_cxo	Container extension objects
Debug	xvt_debug	Debugging facility
Dialog	xvt_dlg	User-written dialog support
DlgMgr	xvt_dm	Dialog manager, controlling built-in dialogs
Drawable	xvt_dwin	Object supporting drawing operations (windows and pixmaps)
ErrorId	xvt_errid	Error message identifier
Error	xvt_errmsg	Error handling facility
Events	xvt_event	Event access
FontMap	xvt_fmap	Font mapper facility
Font	xvt_font	Font object
FileSys	xvt_fsys	File system under the application
GblMem	xvt_gmem	Global (Mac relocatable) memory management
Help	xvt_help	Help system
Image	xvt_image	Image object
I/Ostream	xvt_iostr	Input/Output byte stream
List	xvt_list	List box, list edit
Mem	xvt_mem	Memory allocation facility
Menu	xvt_menu	Application menu components
Palette	xvt_palet	Color palette object
Picture	xvt_pict	Picture object

Pixmap	xvt_pmap	Pixmap object
Print	xvt_print	Printing context
Rect	xvt_rect	Rectangle object
ResMgr	xvt_res	Resource manager
Screen	xvt_scr	Screen object
Scrollbar	xvt_sbar	Scrollbar object
Slist	xvt_slist	List of tagged strings
String	xvt_str	String operations
Timer	xvt_timer	Timer object
Text	xvt_tx	Portable, XVT look-and-feel text object
VisObj	xvt_vobj	Visible object (windows, dialogs, and controls)
Window	xvt_win	Visible window object-specific functionality

In most cases, the first argument to an API function consists of a handle identifying an object instance, like this:

```
xvt_win_get_cursor(window);
```

That argument is missing only for objects that always have just a single instance for each XVT application, like this:

```
xvt_fsys_set_dir_startup(void);
```

2.2.2. Inheritance and Polymorphism

Often the same operation can be performed on different objects. For example, you can set visibility on both windows and controls. Providing a separate call for the operation on each object would produce an unacceptably large API. Instead, XVT makes the API more efficient by using two concepts: inheritance and polymorphism.

Inheritance means that a function that applies to multiple objects is introduced by one object, then inherited by (i.e., applied to) other, more specialized ones. Since many XVT GUI objects are a specialization of a "visible object" (vobj), some key API functionality is defined by vobj, then inherited by windows (win), controls (ctl), drawable windows (dwin), and pixmaps (pmap).

A vobj function such as xvt_vobj_destroy applies to all visible objects. As a result, no separate function is required for destroying a control. Because it applies to multiple objects, a function such as xvt_vobj_destroy is called polymorphic.

To use polymorphic functions properly, you must understand how inheritance works. A function defined for one object applies to all objects inheriting from it, unless XVT specifies otherwise. The tree shown in Figure 2.1 defines the object inheritance for the XVT API.

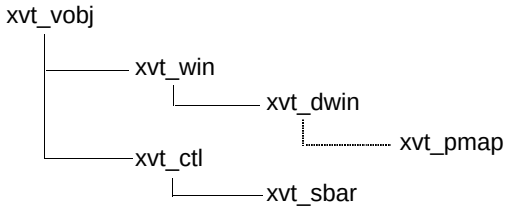


Figure 2.1. Object Inheritance within the XVT API

As you can see from Figure 2.1, xvt_sbar inherits the functionality of xvt_ctl and xvt_vobj, but does *not* inherit the functionality of xvt_dwin, which lies in a different branch of the tree.

Object specialization can also be restrictive, as indicated by the broken line between xvt_dwin and xvt_pmap. xvt_pmap inherits all of the drawing functionality of xvt_dwin. However, xvt_pmap does not inherit all parts of xvt_dwin's functionality. That is, xvt_pmap does not have an event handler and does not allow picture creation.

See Also: For complete information about all elements of the XVT API, see the *XVT Portability Toolkit Reference*.

2.3. Invoking XVT

Like other C programs, an XVT application starts with main. This allows you to specify command line arguments. You can also perform initialization unrelated to the GUI portion of your application, such as opening data files or establishing a network connection. Within main, you'll initialize the XVT library.

Tip: To initialize the XVT library:

Assign values to the XVT_CONFIG structure fields.

This is the XVT_CONFIG structure and its fields:

```
typedef struct {
        short menu_bar_ID;
        short about_box_ID;
        char *base_appl_name;
        char *appl_name;
        char *taskwin_title;
} XVT_CONFIG;
```

menu_bar_ID
: Stores the application's default menubar resource ID, which must be a valid XRC-based resource.

about_box_ID
: Stores the application's default About box resource ID, which must be a valid XRC-based resource. A value of zero indicates that XVT should use the default system About box.

base_appl_name
: Stores the application's base name. XVT uses this name when searching for help files, resource files, and so on.

appl_name
: Stores the application name used in the hyphenated window title required by some systems. This member can be overridden by setting the ATTR_APPL_NAME_RID attribute.

taskwin_title
: Specifies a name for the task window. This lets you provide a longer, more descriptive title, whereas the application name usually has character and length restrictions. This member can be overridden by setting the ATTR_TASKWIN_TITLE_RID attribute.

To ensure that all values are in a default state, your application should initialize the XVT_CONFIG structure to zero before using it. However, you must still specify all fields except the about_box_ID field.

After you initialize XVT_CONFIG, the main function invokes XVT by calling xvt_app_create. The xvt_app_create function does not return. After it is called, the application responds only to user actions by means of event-handling code.

See Also: For more information on event-handling code, see Chapter 4, *Events*.

2.4. System Attributes

XVT provides two functions for setting and retrieving system attributes. Given an attribute identifier, these functions either set a system attribute to a new value, or retrieve the current value of the attribute for the application:

- xvt_vobj_set_attr
- xvt_vobj_get_attr

Both functions use a long integer. The xvt_vobj_get_attr value always returns a long, which you should cast to the appropriate type. Likewise, xvt_vobj_set_attr requires a long argument, which is usually another data type cast to long in the argument list.

XVT supports two sets of attributes:

Portable attributes
Available on all XVT Portability Toolkits; includes system and window-specific attributes.

Platform-specific attributes
Provided in each XVT Toolkit to provide access to native GUI functionality not part of the XVT portable programming interface.

Example: The following code shows how an XVT application program would retrieve the screen dimensions and replace the default fatal error handler with its own function:

```
long XVT_CALLCONV1 task_event_handler(WINDOW win,
        EVENT *ep)
{
        int screen_height, screen_width;

        switch (event_p->type) {
        case E_CREATE:
                screen_height = (int)xvt_vobj_get_attr(NULL_WIN,
                        ATTR_SCREEN_HEIGHT);
                screen_width = (int)xvt_vobj_get_attr(NULL_WIN,
                        ATTR_SCREEN_WIDTH);
                xvt_vobj_set_attr(NULL_WIN,
                        ATTR_ERRMSG_HANDLER,
                        (long)my_handler);
                break;
        }
}       return (0L);
```

You can set or get several attributes before the call to xvt_app_create.

See Also: For more information about portable attributes, see the "Portable Attributes" portion of the *XVT Portability Toolkit Reference*.

For information about portable attributes that have been added or modified to support the ability to write internationalized XVT applications, see section 19.2.2 on page 19-22.
For information about individual non-portable attributes, see the *XVT Platform-Specific Books*.

2.5. Function Calling Convention Macro

The XVT Portability Toolkit contains one macro that defines function calling conventions: XVT_CALLCONV1. The macro's effect differs according to platform:

- On the XVT/Win32 platform, it defines the linkage convention of functions
- On XVT/Mac, it defines a C calling convention for linkage between C and C++ compiler-generated code
- On other platforms, it is defined as an empty macro

Use XVT_CALLCONV1 in all prototypes and headers for XVT callback functions (including event handlers and hook functions). Also use the XVT_CALLCONV1 macro in the declaration of the main function.

Tip: XVT recommends using this macro wherever possible. Although using it is not absolutely required, XVT includes it to ensure portability. You should definitely use XVT_CALLCONV1 in the following situations:

- If you are not using the recommended calling convention
- If the compiler (such as a C++ compiler) sets the calling convention differently than the XVT libraries were compiled and linked

Example: You should declare a callback function prototype like this:

```
BOOLEAN XVT_CALLCONV1 key_hook(...)
```

You should declare the main function like this:

```
int XVT_CALLCONV1 main(int argc, char **argv);
```

2.6. Symbols for Conditional Compilation

Portions of your source code can depend on the window system or file system used for compiling and running your application. For example, code specific to one window system could take advantage of a particular feature of that system.

When you #include the **xvt.h** file, the XVTheader files **xvt_env.h** and **xvt_plat.h** are included in your program. These two files define a number of compiler symbolic constants. At compile time, you can test these symbols with the usual preprocessor conditional operators, such as #if, ==, &&, and #endif. By using these operators to delimit the non-portable sections of your source code, you can make the preprocessor include or omit the sections appropriately, based on the value of the symbols at compile time.

The file **xvt_env.h** defines values for these symbols. Fingering or setting the values takes place in the file **xvt_plat.h**, specific to each platform.

2.6.1. Window System Symbols

Tip: To determine the window system on which you are compiling:

Test the symbol XVTWS for equality to the following symbols:

Symbol:	Window System:
MACWS	Macintosh
MTFWS	Motif
WIN32WS	Windows 32 platforms

Example: The following code tests for MS-Windows (Win32):

```
#if XVTWS == WIN32WS
      /* MS-Windows (32-bit) specific code */
#endif
```

2.6.2. File System Symbols

The following symbols define whether or not a particular file system is supported:

Symbol:	File System:
XVT_FILESYS_HPFS	High Performance
XVT_FILESYS_MAC	Apple Macintosh
XVT_FILESYS_NTFS	Windows XP. Vista (Win 32/64)
XVT_FILESYS_UNIX	UNIX

Set these values to TRUE or FALSE. At least one must be TRUE; however, you can set more than one to TRUE since some operating systems can support more than one file system.

Example: The following code tests for UNIX-like filenames:

```
#if (XV          T_FILESYS_UNIX)
          /* UNIX-specific file system code */
#endif
```

2.6.3. Operating System Symbols

Tip: To determine the operating system being used:

Test the symbol XVT_OS.

The **xvt_env.h** file defines the current values of these constants, which you can use to test the value of this symbol. XVT_OS has the same value as one of these constants.

Caution: Although you can use the XVT_OS macro to determine the filesystem or the window system, XVT strongly encourages you to use the XVTWS and XVT_FILESYS_* macros. They provide a more consistent and simpler way to determine file systems and window systems. Also, the supported values of the XVT_OS macro are subject to change between releases of XVT as support for various operating systems is added or removed.

2.6.3.1. Operating System Feature Symbols

Tip: To check for specific operating system features:

Test the following constants:

XVT_OS_BSD_SIGNALS
: Defined to TRUE if the operating system has BSD-style signal handling.

XVT_OS_BSD_TIMERS
: Defined to TRUE if the operating system has BSD-style timers.

XVT_OS_BSD_DIR
: Defined to TRUE if the operating system has BSD-style directory headers.

XVT_OS_BSD_GETWD
: Defined to TRUE if the operating system has a BSD-style getwd call.

XVT_OS_SYSV_SIGNALS
: Defined to TRUE if the operating system has SYSV-style signal handling.

XVT_OS_SYSV_TIMERS
: Defined to TRUE if the operating system has SYSV-style timers.

XVT_OS_SYSV_DIR
: Defined to TRUE if the operating system has SYSV-style directory headers.

XVT_OS_SYSV_GETCWD
: Defined to TRUE if the operating system has a SYSV-style getcwd call.

XVT_OS_ISUNIX
: Defined to TRUE if the operating system is a UNIX (V7 or later) operating system, FALSE otherwise. In particular, you can depend on select being available if this is defined.

XVT_OS_IS_SUNOS
: Defined to TRUE if the operating system is any version of SunOS.

XVT_OS_IS_MACOS
: Defined to TRUE if the operating system is a Macintosh operating system.

XVT_OS_IS_WINOS
: Defined to TRUE if the operating system is any version of MS-Windows.

2.6.4. Compiler Symbols

The symbol XVT_CC indicates the compiler being used.

Note: The supported values of the XVT_CC macro are subject to change between releases of XVT as support for various compilers is added or removed.

See Also: For information aboutcompilers supported for each XVT Portability Toolkit, see the *XVT Platform-Specific Books* and **readme** files for your particular platform.
To see a list of the currently defined compiler macros, see the **xvt_env.h** file.

2.6.4.1. Compiler Feature Symbols

The **xvt_env.h** file also defines a number of symbols that describe the features of the compiler in use.

XVT_CC_ISANSI
: Defined to TRUE if the compiler can handle ANSI C, FALSE if not. This macro is *not* equivalent to __STDC__. Many compilers only define __STDC__ to be 1 when they are in "strict ANSI" mode. This means that they support prototypes; **stdarg.h**; the keywords const, volatile, and signed; and (void*) pointers.

XVT_CC_PLUS
: Defined to TRUE if the compiler is a C++ compiler, FALSE if not. Note that there are several different types of C++, and this file does not try to classify them further. This macro is defined only if the C++ compiler is not a proper superset of C (i.e., it requires #extern or something to compile C).

XVT_CC_PROTO
: Defined to TRUE if prototypes can be used, FALSE otherwise.

This macro compiles function prototypes conditionally, like this:

```
#if XVT_CC_PROTO
extern void foo(int bar, float baz);
#else
extern void foo(bar, baz);
#endif
:
:
void
#if XVT_CC_PROTO
foo(int bar, float baz)
#else
foo(bar, baz)
int bar;
float baz;
#endif
{
        /* body of function foo */
}
```

If you use this style, you can maintain consistency between the declarations by using a prototyping compiler (for ANSI) followed by **lint** (for K&R). Alternatively, you can use the following prototype construction macros:

```
extern void foo XVT_CC_ARGS((int bar, float baz));
                :
                :
void foo XVT_CC_ARGL((bar, baz))
XVT_CC_ARG(int, bar)
XVT_CC_LARG(float, baz)
{
        /* body of function foo */
}
```

Note: Functions with no arguments are declared and defined like this:

```
extern void foo XVT_CC_NOARGS()
:
:
void foo XVT_CC_NOARGS()
{
        /* body of foo */
}
```

Caution: XVT explicitly does *not* recommend using FPROTO (a separate switch for function and declaration prototypes.) An ANSI compiler can legally throw out a prototype if it encounters a K&R-style definition for the same function.

2.6.4.2. Compile Time Optimization of XVT Applications

The XVT Portability Toolkit is implemented in two layers. The top API layer is called directly by the application. This layer performs error checking of all input parameters and sometimes other validation before calling the internal layer, which contains the implementation of the functionality.

XVT provides a compile time symbol, XVT_OPT, that generates additional optimization of the XVT Portability Toolkit. When this symbol is defined during compilation of your application files, XVT 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. This optimization does not eliminate all error checking from the XVT Portability Toolkit, 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 that this option only be used 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.

See Also: For detailed information about how to use the XVT_OPT symbol with your particular compiler, refer to the *XVT Platform-Specific Books*.

3

GUI ELEMENTS

This chapter introduces the basic graphical elements of GUI applications: windows, dialogs, controls, and menus. It discusses common data structures and events for these GUI objects and suggests techniques for dealing with them.

See Also: The chapters in the second section of this guide, *Windows*, *Dialogs*, *Controls*, and *Menus* discuss details specific to each particular GUI object.

3.1. GUI Object Definitions

Window
A user-interface object that presents information and lets the user interact with that information.

Dialog
A special type of window that contains controls. These controls display and gather additional information. Dialogs cannot contain graphics.

Container Object
Windows and dialogs are commonly referred to as "container" objects, because they serve as containers for the pictures, controls, and text of the application. Container objects enable such GUI objects to be moved and managed as collections.

Control
A standard user-interface object, such as a button, box, or edit control. Controls are placed in windows or dialogs to gather user input or display information.

Menu
A list of commands that the user can issue to the application. A menubar is associated with a window. A menubar and drop-down menus provide many easily available options for

controlling operations within the application, in a minimum amount of screen space.

3.2. Comparison of Dialogs and Windows

Dialogs are really specialized windows, designed to handle a specific task: presenting controls to the application user for selection and manipulation. However, XVT lets you place controls in windows as well. You might do this if your application has special needs, for example, putting graphics primitives and controls in the same container.

Table 3.1 compares dialogs and windows. The table demonstrates that most XVT functions that work with windows also work with dialogs in a consistent way.

Functionality	Dialogs	Windows
Event handlers	Yes	Yes
Events	Do not receive the following XVT events: E_UPDATE, E_MOUSE_*, E_*SCROLL, E_COMMAND, E_QUIT, E_FONT	Receive all XVT events (except for E_QUIT, which only the task window receives)
Coordinate systems	Relative to the client area of the dialog, starting at (0, 0)	Relative to the client area of the window, starting at (0, 0)
Drawing graphics primitives and text	Not available	Supported
Menus	Not available	Supported for all windows types except modal windows
Modality	Modal dialogs support the native GUI system's rules for modal dialogs; modeless dialogs may also show some platform-specific behavior	Modal windows support the native GUI system's rules for modal dialogs
Automatic traversal between controls	Supported by the native GUI system's dialog manager	Supported programmatically
User resize and/or move	Only move is supported, and only if the native GUI system supports user dialog moves	Fully supported if the window is created allowing moving and resizing
Decorations	Titlebars and close boxes are the only decorations that dialogs can possess, and on some native GUI platforms, some dialogs may not even have titlebars. These decorations are not optional.	On most window types, titlebars, border type, window border scrollbars, close boxes, and resize controls are supported and are optional.Modal windows allow only the decorations of the native system's modal dialogs.

Table 3.1. Comparison of dialogs and windows

3.3. Creating, Initializing, and Terminating GUI Objects

This section describes how to create, initialize, and terminate several types of GUI objects: resource-based, structure-based, and dynamic.

3.3.1. Resource-based GUI Objects

You can specify the following GUI objects as resources: windows, dialogs, controls, and menus. To create a resource-based object, you specify the object's definition in XVT's XVT Resource Compiler (XRC). The application then accesses it at runtime by means of the object's resource ID.

You can create resource-based objects in XVT-Design. To each object, XVT-Design assigns a symbolic identifier, which corresponds to a resource ID. Functions can access the object by its symbolic identifier. XVT-Design places symbolic identifier definitions for all GUI objects into the application module's header file. (By default this has the same name as the project.) XVT-Design also creates the XRC file with references to the identifiers (resource IDs) defined in the header file.

XRC definitions are external to your application and are used by window creation functions. Two functions, xvt_win_create_res and xvt_dlg_create_res, create window or dialog container objects, along with any controls.

3.3.1.1. Windows, Dialogs, and Controls

Resource-based windows and dialogs are useful when their definition is unknown or is changeable at compilation time.

In XRC, dialogs and windows are defined both in terms of their own attributes (i.e., resource ID, size, title, modality), and in terms of the individual controls that they contain. You can also define optional arbitrary data (USERDATA) for the window or dialog. The xvt_res_get_win_data or xvt_res_get_dlg_data functions can then retrieve this USERDATA.

Example: The following XRC and C code creates a window using resources (WINDOW_1 is this window's resource ID):

```
        /* XRC code to define WINDOW_1 */

WINDOW WINDOW_1 100 100 300 300 "A Sample Window" doc size
MENU_100 USERDATA "string1", "string2"
...

        /* Now the C code to create a window that is
        defined in XRC */

xvt_win_create_res(WINDOW_1, TASK_WIN, EM_ALL,
        a_window_eh, 0L);
```

Example: This XRC code defines a *modeless* dialog with one push button control:

```
#define OUR_DIALOG_ID 1000
...
DIALOG OUR_DIALOG_ID, 100, 100, 300, 200 "Sample Modeless Dialog"
MODELESS USERDATA "string1", "string2"

        BUTTON DLG_OK, 50, 50, 100, 30 "OK" DEFAULT
...
```

Example: This XRC code defines a *modal* dialog with one push button control:

```
#define OUR_DIALOG_ID 1000
...
DIALOG OUR_DIALOG_ID, 100, 100, 300, 200 "Sample Modeless Dialog"
MODAL USERDATA "string1", "string2"

        BUTTON DLG_OK, 50, 50, 100, 30 "OK" DEFAULT
...
```

Note: DLG_OK is a pre-defined control ID for the OK push button. (You must use this label to ensure portability.)

After you place the dialog defined above into the application's XRC file, you call xvt_dlg_create_res to invoke the dialog:

```
xvt_dlg_create_res(WD_MODELESS, OUR_DIALOG_ID,
        EM_ALL, a_dialog_eh, 0L);
```

Or, once the resource-based dialog definition exists, you could convert the definition into in-memory WIN_DEF data structures. You modify it if needed, and call xvt_dlg_create_def to create the dialog. The xvt_res_free_win_def function then frees the WIN_DEF array.

```
WIN_DEF *win_def_p;
...
win_def_p = xvt_res_get_dialog (OUR_DIALOG_ID);

/* Change the dialog from modeless to modal */

win_def_p->wtype = WD_MODAL;

xvt_dlg_create_def(win_def_p, EM_ALL, a_dialog_eh,
      0L);
xvt_res_free_win_def(win_def_p);
...
```

3.3.1.2. Menus

Usually, both the XRC file and the program refer to each item in a menu by means of constants.

Example: The code below shows a hierarchical menu structure as it would be defined in the XRC file:

```
MENUBAR 1000

MENU 1000            USERDATA "string1", "string2"
      SUBMENU 2000       "Options"
      SUBMENU 3000       "Attributes"

MENU 2000
      ITEM 2001          "Option #1" checkable checked
      ITEM 2002          "Option #2" disabled
      ITEM 2003          "Option #3"
      SEPARATOR
      ITEM 2004          "Last Option"

MENU 3000
      ITEM 3001          "Attribute #1"
      ITEM 3002          "Attribute #2" disabled
      SUBMENU 4000       "Nested Menu"
      SEPARATOR
      ITEM 3004          "Last Attribute"

MENU 4000
      ITEM 4001          "Item #1"
      ITEM 4002          "Item #2" disabled
      ...
```

This example defines a menubar with two submenus, each containing additional items. You can see these features in the XRC code:

- It defines a MENUBAR and gives it an ID of 1000.

- It defines a MENU, with the same ID as the MENUBAR. This tells **xrc** that this is the beginning of the actual MENUBAR definition. The SUBMENU references under MENU 1000 refer to subsidiary menus.
- MENU 2000 and MENU 3000 are subsidiary to MENU 1000. Each contains four candidate selection items, with the fourth item separated from the other three by a separator (platform-specific, but usually a separator appears as a dashed line).
- MENU 3000 has a a SUBMENU defined. This is an example of hierarchical menu definition: MENU 4000 is a child menu of MENU 3000, which is in turn a child of MENU 1000 (the menubar).

See Also: For more information about menus, including pop-up menus, see Chapter 9, *Menus*.
For details on xvt_res_get_menu_data, see the *XVT Portability Toolkit Reference*.

3.3.2. Structure-based GUI Objects

Structure-based GUI objects are created using an array of WIN_DEF data structures that is passed to xvt_win_create_def (for windows), xvt_dlg_create_def (for dialogs), or xvt_ctl_create_def (for controls).

WIN_DEF Data Structure

You initialize the contents of the WIN_DEF in two ways:

- Call xvt_res_get_win_def or xvt_res_get_dlg_def to generate an array of structures from resources
- Allocate an array of structures, initialize structure fields to zero, and assign values to the fields

In either case, the array of WIN_DEF structures contains information for creating windows or dialogs and their controls.

The following code shows the WIN_DEF structure:

```
typedef struct s_win_def {
        WIN_TYPE wtype;                         /* window type */
        RCT rct;                                /* creation rectangle */
        char *text;                             /* object title */
        UNIT_TYPE units;                        /* coordinate units */
        XVT_COLOR_COMPONENT *ctlcolors;         /* control colors */
        union {

                struct s_win_def_win {          /* Windows*/
                        short int menu_rid;     /* menu resource ID */
                        MENU_ITEM *menu_p;      /* menu tree */
                        long flags;             /* WSF_* flags */
                        XVT_FNTID ctl_font_id;  /* all control fonts */
                } win;

                struct s_win_def_dlg {          /* Dialogs */
                        long flags;             /* DLG_FLAG_* values */
                        XVT_FNTID ctl_font_id;  /* all control fonts */
                } dlg;

                struct s_win_def_ctl {          /* Controls */
                        short int ctrl_id;
                        short int icon_id;      /* for icons only */
                        long flags;             /* CTL* flags */
                        XVT_FNTID font_id;      /* control font */
                } ctl;

                struct s_win_def_tx {           /* Text Edit */
                        unsigned short attrib;  /* TX_* flags */
                        XVT_FNTID font_id;      /* text edit font */
                        short int margin;
                        short int limit;
                        short int tx_id;
                } tx;

        } v;
} WIN_DEF;
```

The first element ofa WIN_DEF array describes the container window or dialog. Subsequent array elements describe controls or text edit objects. The last element terminates the WIN_DEF array with an element whose wtype is W_NONE.

xvt_res_free_win_def should be called by your application to free array memory. (xvt_res_free_win_def automatically frees WIN_DEF structure font IDs, arrays of control component colors, text strings, and any MENU_ITEM arrays that are defined forthe menu_p menu pointer in the win substructure in the first array element.)

Example: This example creates a window using data from a WIN_DEF array initialized in application code:

```
WIN_DEF win_def_array[10];

static XVT_COLOR_COMPONENT win_colors[] = {
            {XVT_COLOR_FOREGROUND, COLOR_BLACK},
            {XVT_COLOR_BLEND, COLOR_WHITE},
            {XVT_COLOR_BACKGROUND, COLOR_BLUE},
            {XVT_COLOR_NULL, 0}};

static XVT_COLOR_COMPONENT ok_colors[] = {
            {XVT_COLOR_FOREGROUND, COLOR_GREEN},
            {XVT_COLOR_BACKGROUND, COLOR_BLACK},
            {XVT_COLOR_NULL, 0}};

XVT_FNTID button_font = xvt_font_create();
...
      /* initialze WIN_DEF array */
memset((char *)win_def_array, 0, 10*sizeof(WIN_DEF));

      /* Document window */
win_def_array[0].wtype                          = W_DOC
win_def_array[0].rct.top                        = 100;
win_def_array[0].rct.left                       = 100;
win_def_array[0].rct.bottom                     = 300;
win_def_array[0].rct.right                      = 300;
win_def_array[0].text                           = "Sample Window";
win_def_array[0].units                          = U_PIXELS;
win_def_array[0].ctlcolors                      = win_colors;
win_def_array[0].v.win.menu_rid                 = MENU_100;
win_def_array[0].v.win.flags                    = WSF_DECORATED;
win_def_array[0].v.win.ctl_font_id = NULL_FNTID;
      /* "OK" button */
win_def_array[1].wtype                          = WC_PUSHBUTTON;
win_def_array[1].rct.top                        = 10;
win_def_array[1].rct.left                       = 40;
win_def_array[1].rct.bottom                     = 20;
win_def_array[1].rct.right                      = 60;
win_def_array[1].text                           = "~OK";
win_def_array[1].ctlcolors                      = ok_colors;
win_def_array[1].v.ctl.ctrl_id                  = DLG_OK;
win_def_array[1].v.ctl.flags                    = CTL_FLAG_DEFAULT;
win_def_array[1].v.ctl.font_id                  = button_font;
...
      /* end array */

win_def_array[9].wtype = W_NONE;
...
xvt_win_create_def(win_def_array, TASK_WIN, EM_ALL,
                              a_window_eh, 0L);
...
xvt_font_destroy(button_font);
...
```

Alternatively, in the following code, xvt_win_create_def initializes the WIN_DEF array using the resource (XRC) definition for the window:

```
WIN_DEF *win_def_p;
```

```
...
win_def_p = xvt_res_get_win_def(WINDOW_1);

xvt_win_create_def(win_def_p, TASK_WIN, EM_ALL,
                                    a_window_eh, 0L);
...
xvt_res_free_win_def(win_def_p);
```

Window Attribute Flags

XVT defines window attributes as flags that can be logically OR'd together. The resulting combination is passed to one of the window creation functions. The following table lists the window-attribute flags (several of which are platform-specific):

WSF_NONE	No flags set
WSF_SIZE	Is user-sizeable
WSF_CLOSE	Is user-closeable
WSF_HSCROLL	Has horizontal scrollbar outside of the client area
WSF_VSCROLL	Has vertical scrollbar outside of the client area
WSF_DECORATED	A convenient combination of WSF_SIZE, WSF_CLOSE, WSF_HSCROLL, and WSF_VSCROLL
WSF_INVISIBLE	Is initially invisible
WSF_DISABLED	Is initially disabled
WSF_ICONIZABLE	Is iconizable (XVT/XM only)
WSF_ICONIZED	Is initially iconized
WSF_FLOATING	Is a floating window (XVT/Mac only)
WSF_SIZEONLY	Lacks border rectangles (XVT/Mac only)
WSF_NO_MENUBAR	Has no menubar of its own (see Note)
WSF_MAXIMIZED	Is initially maximized
WSF_DEFER_MODAL	Modal status deferred (not processed by xvt_win_create*)
WSF_PLACE_EXACT	Modal window is placed exactly where specified

Note: WSF_NO_MENUBAR implies that the window has no menubar. You can use this only with top-level windows; child windows never have menubars.

Dialog and Control Flags

To determine the initial state of a dialog, you specify two XVT flags in the flags field in the dlg substructure: DLG_FLAG_INVISIBLE, and DLG_FLAG_DISABLED. If you don't specify either, the dialog is both visible and enabled at creation time.

Tip: Avoid creating invisible or disabled modal windows and dialogs, because doing so can lock up your application.

Structure-based controls are created from a WIN_DEF data structure (of arrays) which is passed to xvt_ctl_create_def, xvt_win_create_def, or xvt_dlg_create_def.

Example: An array of WIN_DEF objects is used to create a dialog with a single push button (the third element of the array is the terminating WIN_DEF structure, with the wtype attribute set to W_NONE):

```
...
WIN_DEF win_def_array[3];
...
win_def_array [0].wtype                    = WD_MODELESS;
win_def_array [0].rct.left                 = 100;
win_def_array[0].rct.top                   = 100;
win_def_array[0].rct.right                 = 400;
win_def_arra y[0].rct.bottom               = 300;
win_def_array[0].text                      = "Sample Modeless
                                                Dialog";
win_def_array[0].units                     = U_PIXELS;
win_def_array [0].v.dlg.flags              = 0L;
win_def_array[1].wtype                     = WC_PUSHBUTTON;
win_def_array[1].rct.left                  = 50;
win_def_array[1].rct.top                   = 50;
win_def_array[1].rct.right                 = 150;
win_def_array[1].rct.bottom                = 80;
win_def_array[1].text                      = "OK";

win_def_array[1].units                      = U_PIXELS;
win_def_array [1].v.ctl.ctrl_id             = DLG_OK;
win_def_array[1].v.ctl.flags              = CTL_FLAG_DEFAULT;

win_def_array[2].wtype                     =      W_NONE;
                                                /* terminator */
...
```

Once the above WIN_DEF array has been constructed, create the dialog as follows:

```
xvt_dlg_create_def(win_def_array, EM_ALL,
      a_dialog_eh, 0L);
```

3.3.3. Dynamic Windows

Dynamic windows do not require external resource definitions. Your program can create them at any time.

Tip: To dynamically create windows:

Call xvt_win_create.

Specify all of the window's attributes (initial size, title text, menu resource ID, parent WINDOW, attribute flags, event handler, and application data) as arguments to xvt_win_create.

Example: The following code fragment dynamically creates a W_DOC window:

```
RCT rect;
...
rect.top = 100;
rect.left = 100;
rect.bottom = 300;
rect.right = 300;
...
xvt_win_create (W_DOC, &rect, "A Sample Window",
        MENU_100, TASK_WIN, WSF_DECORATED, EM_ALL,
        a_window_eh, 0L);
...
```

3.3.4. Initializing and Terminating Dialogs and Windows

3.3.4.1. Initializing After an E_CREATE

In response to E_CREATE events, XVT performs all initialization operations for windows and dialogs in their event handler functions. Initialization operations include the following:

- Allocating any window or dialog-specific data structures and attaching them to the window or dialog with xvt_vobj_set_data
- Initializing the various dialog controls (such as setting default text fields, control visibility and enabled properties, populating list boxes, etc.)
- Overriding the resource title and/or rectangle by the application's calling xvt_vobj_set_title and xvt_vobj_move in response to the E_CREATE event

Tip: On some systems, calling xvt_vobj_set_title or xvt_vobj_move can cause undesirable flashing. Instead, use xvt_res_get_dlg_def (or xvt_res_get_win_def) with xvt_dlg_create_def (or xvt_win_create_def) and change the title and/or rectangle in the WIN_DEF structure between these calls.

3.3.4.2. Terminating After an E_DESTROY

As mentioned earlier, a window or dialog's event handler receives an E_DESTROY event when xvt_vobj_destroy is called.
This is a good time to free the container's application data (with xvt_vobj_get_data), and to perform any other cleanup activities before it is destroyed.

Note: You cannot call xvt_win_get_ctl or xvt_vobj_get_data for any of the controls in a container during E_DESTROY. This means that if the controls have application data associated with them, you must either have pointers to this data in the window or dialog itself, or you must free the data earlier (for example, during the E_CLOSE or E_CONTROL event that caused the call to xvt_vobj_destroy).

3.4. Event Handler Functions

All creation functions for windows and dialogs require as a parameter the name of an event handler function.

XVT-Design automatically defines the event handler function and supplies its name to the container's creation function. It also provides the switch statement template in the event handler and supplies default code statements within some of the event cases.

The event handler function receives a pointer to an EVENT structure. A switch statement generally processes the type field of the EVENT structure. The type field matches one of the event identifiers shown in the table of the next section.

See Also: For an example of a dialog's event handler, see section 3.4.2 on page 3-15.
Also see Chapter 4, *Events*.

3.4.1. Handling Window and Dialog Events

The following table provides some information about the XVT events that are sent to your window's or dialog's event handler, and how to handle them:

XVT Event Sent to Event Handler	Comments and Suggested Actions to Perform upon Receiving Event
E_CREATE	Perform container initializations, modify appearance, allocate and attach application data to container and/or controls, initialize controls.
E_DESTROY	Perform cleanup, deallocate application data to container and/or controls.
E_FOCUS	Set or reset control states or contents (although usually handled by native windowing system).
E_SIZE	Modify layout of controls.
E_CLOSE	Close the container if it is appropriate.
E_CHAR	Received for characters that are not consumed by controls. Perform specific action based on the key pressed. On some platforms, controls consume character events. (Portable applications should avoid processing E_CHAR events in dialogs.)
E_CONTROL	Received for all controls manipulated by a user.
E_TIMER	Timer went off; perform some time-dependent action.
E_USER	Application-specific; no specific action recommended.
E_UPDATE	Window requires updating. Do any drawing operations. (Not sent to dialog event handlers.)
E_MOUSE_DOWN, E_MOUSE_UP, E_MOUSE_DBL, E_MOUSE_MOVE	Mouse click or motion. Select items; set or release mouse trapping; perform rubber-banding; etc. (Not sentto dialog eventhandlers.)
E_VSCROLL, E_HSCROLL	Scrollbar controls operated. Scroll window contents. (Not sent to dialog event handlers.)

E_COMMAND, E_FONT	Menu item selected or font selection dialog operated. Respond to menu selection, or call xvt_dwin_set_font. (Not sent to dialog event handlers.)
E_QUIT	System shutdown. (Not sent on all platforms. Sent only to task window event handler. Not sent to dialog or top level window event handlers.)

Table 3.2. Handling XVT events

Event Masking

XVT allows you to block (or "mask") specific event types from reaching window or dialog event handlers. To create an event mask, OR together the desired events (e.g., EM_MOUSE_DOWN | EM_UPDATE). By default, all XVT events are selected (the EM_ALL mask).

See Also: For information about event masking, see the "EM_* Constants" portion of the *XVT Portability Toolkit Reference*.
Also see Chapter 4, *Events*.

3.4.2. Event Handling for Controls

When a control-related event is reported, the WINDOW passed to the event handler identifies the control's parent window or dialog, and the win field in the CONTROL_INFO object is the WINDOW of the control itself. Similarly, the type field in the CONTROL_INFO object is the WIN_TYPE of the control itself. (To find the WIN_TYPE of the parent window or dialog, call xvt_vobj_get_parent, specifying the parent's WINDOW.)

XVT-Design automatically creates a switch statement within the E_CONTROL case of the window or dialog's event handler. In this switch statement, it inserts a case for each control contained in the window or dialog. The following example shows this structure.

Example: This example shows the event handler for a dialog.

```
long XVT_CALLCONV1 dlg_101_eh(WINDOW xdWindow,
        EVENT *xdEvent)
{
        short xdControlId = xdEvent->v.ctl.id;

        switch (xdEvent->type) {
        ...
                case E_CONTROL:
                /*
                        User operated control in window.
                */
                {

                switch(xdControlId) {
                case DLG_101_RADIOBUTTON_1: /* "Radio Button 1" */
                        {
                        xdCheckRadioButton(xdWindow,
                                WIN_101_RADIOBUTTON_1,
                                WIN_101_RADIOBUTTON_1,
                                WIN_101_RADIOBUTTON_1);
                        }
                        break;
                case DLG_101_PUSHBUTTON_2: /* "Push Button 2" */
                        {
                        }
                        break;
                ...
                default:
                        break;
                }
                }
                break;
        }
}       return (0L);
```

3.4.3. Event Handling For Menus

The following example shows the structure of a window's event handler for E_COMMAND events. Such events are generated when a user selects a menu item. For eachmenu item a function designed to handle that case is called.

XVT-Design inserts a call to a menu event handler function in the E_COMMAND event of the window (if the window is configured to have a menubar). The following example shows this structure.

```
/* define menu tags */

#define M_OPTION_1 ...
#define M_OPTION_2 ...
#define M_OPTION_3 ...
#define M_OPTION_4 ...

#define M_UTIL_1 ...
#define M_UTIL_2 ...
...
long XVT_CALLCONV1 win_101_eh(WINDOW xdWindow,
        EVENT *xdEvent)
{
        switch (xdEvent->type) {
        ...
        case E_COMMAND:
                {
                        do_MENU_BAR_2(xdWindow, xdEvent);
                }
                break;
        ...
        }
        return (0L);
}
void do_MENU_BAR_2(WINDOW xdWindow, EVENT *xdEvent)
{
        MENU_TAG tag = xdEvent->v.cmd.tag;
        ...
        switch( tag ) {
        ...
                case M_OPTION_1:
                        do_option_1(xdWindow);
                        break;
                case M_OPTION_2:
                        do_option_2(xdWindow);
                        break;
                case M_OPTION_3:
                        do_option_3(xdWindow);
                        break;
                case M_OPTION_4:
                        do_option_4(xdWindow);
                        break;
                case M_UTIL_1:
                        do_option_1(xdWindow);
                        break;
                case M_UTIL_2:
                        do_option_2(xdWindow);
                        break;
                ...
        }
...
}
```

3.5. Functions Common to Multiple GUI Objects

This section discusses some operations common to multiple GUI objects. Object-specific functions are discussed in the appropriate chapter (e.g., Chapter 6, *Windows*).

3.5.1. Determining Parent Windows

Tip: To determine the parent of any window:

Call xvt_vobj_get_parent.

Note: If called on a top-level window, xvt_vobj_get_parent returns TASK_WIN. It returns SCREEN_WIN as the parent of the task window, and returns NULL_WIN as the parent of SCREEN_WIN.

3.5.2. Window and Dialog Dimensions and Coordinates

Tip: To find the dimensions of a GUI object's client rectangle:

Call xvt_vobj_get_client_rect.

The coordinates returned are relative to the object, so the left and top coordinates are always zero.

Tip: To find the coordinates and dimensions of the entire GUI object:

Call xvt_vobj_get_outer_rect.

The rectangle includes object decorations, such as titlebars and scrollbars (for windows).

Tip: To translate coordinates from one container to another:

Call xvt_dwin_translate_points.

3.5.3. Controlling Keyboard Focus

Tip: To explicitly assign keyboard focus to a control, a window, or a dialog:

Call xvt_scr_set_focus_vobj.

Tip: To determine if a specific GUI object can be assigned keyboard focus:

Call xvt_vobj_is_focusable.

For a specified control or child window, this call activates the containing window hierarchy. The ability to assign focus is a static property of a visible object and is not affected by its current visibility or enabled state.

Tip: To find out which object receives keyboard events:

Call xvt_scr_get_focus_topwin.

Tip: To find out which top-level window or dialog is currently active:

Call xvt_scr_get_focus_topwin.

Implementation Note: The result of calling xvt_vobj_is_focusable on the same type of GUI object may vary between platforms. For example, a pushbutton can never gain focus under native look-and-feel for Macintosh, but can on Motif.

3.5.4. Controlling Window Stacking

Tip: To control the stacking of windows:

Call xvt_vobj_raise.

On some platforms, raising a top-level window can activate that window. Changing the stacking order of controls is not supported.

3.5.5. Setting and Getting Titles

Windows, dialogs, and some controls have a title, which is set when they are created.

Tip: To change titles at any time:

Call xvt_vobj_set_title.

For document windows, the function xvt_win_set_doc_title is similar, but it ensures that the title obeys appropriate user interface guidelines for the underlying toolkit.

Tip: To retrieve the title of a GUI object:

Call xvt_vobj_get_title.

3.5.6. Moving, Resizing, Disabling, and Hiding Objects

Normally, only the application user moves and resizes windows or dialogs, but you can also do this programmatically. Your application receives an E_SIZE event when a container is resized, but not when it is merely moved.

Tip: To move and/or resize a container:

Call xvt_vobj_move.

The rectangle passed to xvt_vobj_move is relative to the client rectangle of the container window. It is interpreted identically to the rectangle passed to the window or dialog creation functions. However, when you query the client rectangle of a window with a call to xvt_vobj_get_client_rect, the coordinates that are returned are relative to the top-left corner of the window; that is, the coordinate returned for the top-left is (0,0).

You can also use xvt_vobj_move to reposition controls within the container.

Tip: To hide an object without closing it:

Call xvt_vobj_set_visible with a FALSE argument.

Calling the same function with a TRUE argument reveals a hidden object.

Tip: To toggle the enabled/disabled state of a child window:

Call xvt_vobj_set_enabled.

This function also works on dialogs and controls.

3.5.7. Determining Creation Flags, Handles, and IDs

XVT allows the applicationto interact directly with any control. The functions described in this section allow you to ascertain the status of various XVT PTK GUI components.

3.5.7.1. Obtaining the Creation Flag of a Visible Object

Tip: To determine the current state of creation flags of a visible object (*vobj*):

Call xvt_vobj_get_flags.

This function returns the following types of creation flags:

- WSF_* values for windows
- DLG_FLAG_* values for dialogs
- CTL_FLAG_* values for controls

This function returns the values of the creation flags from their current state. For example, a window may have been created visible but may be hidden at a later point in time—the creation flag WSF_INVISIBLE would then be returned asone of the current creation flags.

Note: xvt_vobj_get_flags does not work with text edit objects. They do not have associated XVT WINDOWs and hence are not classified as visible objects (vobj), per se. The only way to retrieve a text edit's attributes (similar to creation flags) is to use xvt_tx_get_attr.

3.5.7.2. Obtaining a Control's Window Handle

Tip: To convert any XVT control ID to a WINDOW:

Call xvt_win_get_ctl.

3.5.7.3. Obtaining a Control's ID

Tip: To obtain the ID of a given control (by passing its WINDOW handle):

Call xvt_ctl_get_id.

xvt_ctl_get_id is the opposite of xvt_win_get_ctl because xvt_win_get_ctl returns a control's WINDOW handle given the control's ID.

3.5.8. Destroying GUI Objects

Tip: To destroy a window, dialog, or control:

Call xvt_vobj_destroy.

Tip: To destroy a pixmap:

Call xvt_pmap_destroy.

Destruction of a container results in all of the contained objects being destroyed as well.

4

EVENTS

XVT uses an event-driven programming paradigm. In other words, XVT applications respond to events whose order and timing is generally unpredictable. In this sense, no one part of an XVT application is "in control." Instead, various event-handling functions within the application process the events as they occur.

Typically, events are generated when users interact with the application. In some cases, though, the native GUI and/or window manager can generate events. Other events occur as by-products of user action.

Each event is delivered to an event handler, accompanied by two pieces of information: the WINDOW where the event was generated (or with which it is associated), and a pointer to an EVENT object (synthesized and allocated by XVT) that contains information about the event.

See Also: XVT-enforced rules govern the order in which some events occur; for details, see section 4.4.1 on page 4-12.

Figure 4.1 illustrates event flow control for XVT programs.

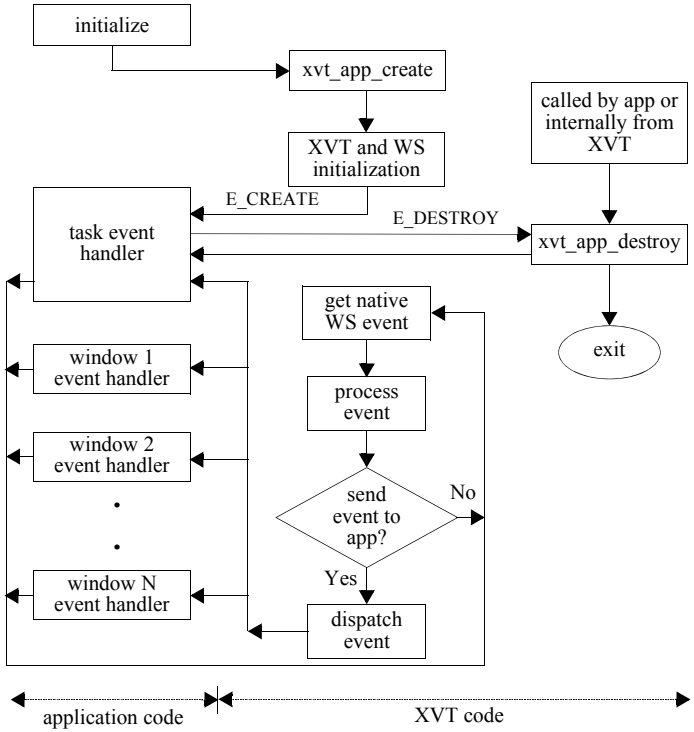


Figure 4.1. Control flow for XVT programs

Native Events

The underlying window system on which XVT runs can generate native GUI system events. XVT either ignores these orhandles them without involving the application. Your application can be notified when these events occur; however, both the events themselves and the notification method are non-portable.

See Also: For more information about accessing native events (including keystrokes) before they are processed by XVT, see section 4.4.3 on page 4-15.

4.1. Types of Events

This section discusses the types of events that XVT can send to event handlers, along with the information that accompanies them. All XVT events fall into one of three categories:

User Interaction Events

E_CLOSE
E_MOUSE_MOVE
E_MOUSE_DOWN
E_MOUSE_UP
E_MOUSE_DBL
E_MOUSE_SCROLL
E_CHAR
E_VSCROLL
E_HSCROLL
E_COMMAND
E_FONT
E_CONTROL

Window Management Events

E_CREATE
E_FOCUS
E_DESTROY
E_SIZE
E_UPDATE
E_QUIT

Other Events

E_USER
E_TIMER
E_HELP

Three types of event handlers dealwith events: task (window) event handlers, window event handlers, and dialog event handlers. The XVT task window has a separate handler, because although it is a WINDOW object, its behavior differs from ordinary windows and dialogs.

Note: Print windows don't have event handlers and therefore don't receive events.

The following table summarizes all XVT events, explaining their significance for each of the three types of event handlers.

XVT Event	Task Event Handler	Window Event Handler	Dialog Event Handler
E_CREATE	TASK_WIN has been created; first event sent to application.	Window has been created; first event sent to newly created window.	Dialog has been created; first event sent to newly created dialog.
E_DESTROY	Application is being terminated; last event sent to application.	Window has been closed; last event sent to window.	Dialog has been closed; last event sent to dialog.
E_FOCUS	Physical task window has lost or gained focus.	Window has lost or gained focus.	Dialog has lost or gained focus.
E_SIZE	Size of task window has been set or changed. Always sent on application startup; may be sent subsequently if task window is resized by user or via xvt_vobj_move.	Size of window has been set or changed; sent when window is created or subsequently resized by user or via xvt_vobj_move.	Size of dialog has been set or changed; sent when dialog is created or subsequently resized by xvt_vobj_move.
E_UPDATE	Physical task window requires updating.	Window requires updating.	Not sent.
E_CLOSE	Request to close (terminate) application.	Request to close window; user operated close menu item on window system menu, or operated close control on window frame. Not sent if Close on File menu is issued. Window not closed unless xvt_vobj_destroy is called.	Request to close dialog; user operated close menu item on dialog system menu, or operated close control on dialog frame. Dialog not closed unless xvt_vobj_destroy is called.

XVT Event	Task Event Handler	Window Event Handler	Dialog Event Handler
E_MOUSE_DOWN E_MOUSE_UP E_MOUSE_DBL E_MOUSE_MOVE E_MOUSE_SCROLL	Mouse click or motion.	Mouse click or motion.	Not sent.
E_CHAR	Character typed. (XVT/Win32 only)	Character typed.	On some platforms, controls consume character events; portable applications should avoid processing E_CHAR events in dialogs.
E_VSCROLL E_HSCROLL	Scrollbar control on frame operated.	Scrollbar control on frame operated.	Not sent.
E_COMMAND	User selected task window menu item command.	User selected window menu command.	Not sent.
E_FONT	User made aselection from task window menubar Font/Style menu, or Font Selection dialog.	User made aselection from window menubar Font/Style menu or Font Selection dialog.	Not sent.
E_CONTROL	User operated control in task window.	User operated control in window.	User operated control in dialog.
E_TIMER	Timer associated with TASK_WIN went off.	Timer associated with window went off.	Timer associated with dialog went off.
E_QUIT	System shutdown. (Sent only on those platforms which provide this information to XVT)	Not sent.	Not sent.
E_USER	Application-initiated.	Application-initiated.	Application-initiated.

XVT Event	Task Event Handler	Window Event Handler	Dialog Event Handler
E_HELP	Help has been requested for the window, or for an object or menu within the window (XVT/Win32 only).	Help has been requested for the window, or for an object or menu within the window.	Help has been requested for the dialog, or for an object within the dialog.

Table 4.1. XVT events summary

See Also: For more information about the task window, refer to section 6.1 on page 6-2.
For information about attributes that affect the task window, see the *XVT Platform-Specific Books*.

4.2. The EVENT Data Structure

XVT uses a common structure for EVENTs to tell an application what event occurred and, in most cases, to supply additional information about the event. This structure contains the member type, followed by a union v, which contains additional members that vary according to the particular event:

```
typedef struct {
        EVENT_TYPE type;
        union {
                ...             /* event-specific substructures */
        } v;
} EVENT, *EVENT_PTR;
```

Typically, you would structure an XVT event handler so that it branches to specific event-related code depending on the type of event, like this:

```
long XVT_CALLCONV1 a_window_eh(WINDOW win,
        EVENT *event_p)
{
        switch (event_p->type) {
        case E_CREATE:
                ...
        case E_DESTROY:
                ...
        }
}
```

EVENT_TYPE Definition

EVENT_TYPE has the following definition:

```
typedef enum _event_type {
        E_CREATE,           /* creation */
        E_DESTROY,          /* destruction */
        E_FOCUS,            /* window focus gain/loss */
        E_SIZE,             /* resize */
        E_UPDATE,           /* update */
        E_CLOSE,            /* close window request */
        E_MOUSE_DOWN,       /* mouse down */
        E_MOUSE_UP,         /* mouse up */
        E_MOUSE_MOVE,       /* mouse move */
        E_MOUSE_DBL,        /* mouse double-click */

        E_MOUSE_SCROLL, /* mouse scroll wheel event */
        E_CHAR,             /* character typed */
        E_VSCROLL,          /* horz. window scrollbar activity */
        E_HSCROLL,          /* vert. window scrollbar activity */
        E_COMMAND,          /* menu command */
        E_FONT,             /* font menu or dialog selection */
        E_CONTROL,          /* control activity */
        E_TIMER,            /* timer */
        E_QUIT,             /* application shutdown request */
        E_HELP,             /* help invoked */
        E_USER,             /* user-defined */
} EVENT_TYPE;
```

See Also: For details about each event type, see section 4.5 on page 4-16.

4.3. Event Handlers

When an event occurs, XVT notifies an application by invoking the application-defined event handler function for the appropriate window or dialog. XVT passes two arguments to the event handler:

- The WINDOW where the event occurred
- A pointer to an EVENT object, which contains specific information regarding the event

In XVT, only task windows, regular windows, and dialogs can have event handlers. Print windows, screen window, and controls cannot have them.

Each window or dialog in an XVT application *must* have an associated event handler function. Information about events that occur in windows or dialogs is sent to these event handlers, where the application can determine the type of event and the appropriate response.

Example: Here is a sample event handler structure:

```
long XVT_CALLCONV1 a_window_eh(WINDOW win,
        EVENT *event_p)
{
        switch (event_p->type) {
        case E_CREATE:
                /* code to handle window creation event */
                break;
        case E_DESTROY:
                /* code to handle window destruction event */
                break;
        case FOCUS:
                /* code to handle window focus change event */
                break;

                /* ... and so on for all events relevant to
                this window or dialog */

        default:
                /* ignore other events */
                break;
        }
        return (0L);
}
```

4.3.1. Sending Events

Tip: To send XVT events directly to window or dialog event handlers:

Call xvt_win_dispatch_event.

These events are not queued. Event handlers return a long value: 0 if successful, or –1 if not successful.

Note: You can send events to event handlers by calling the event handler directly, but XVT recommends that you use xvt_win_dispatch_event instead.

In some cases, an event might be automatically passed up a window hierarchy to the event handler of a parent or task window. For example, mouse events for a disabled child window will be translated to its parent window's coordinates, and sent automatically to the parent window's event handler. Except in such cases, though, automatic event transference doesn't occur.

However, you can send events to the event handler of any window (including the parent window of another window) at any time.

4.3.2. Recursive Calls to Event Handlers

An XVT event can occur whenever your application passes control to XVT, either by calling an XVT function or returning from an event handler. When window management events (for example, E_UPDATE or E_FOCUS) occur, an event handler might be called even if it hasn't returned from handling a previous event.

As a result, you must design your program to execute recursively. This isn't difficult, because most event processing is associated with user events, which do not cause recursive calls unless xvt_app_process_pending_events is called, in which case any pending native events are flushed to event handlers. On the other hand, it is possible for a window management event, such as E_UPDATE, to occur during the processing of a user event or other window event.

Tip: XVT recommends that you follow these guidelines for avoiding recursion problems:

Minimize processing during window management events.
For example, when an update event occurs, update only the window. When a focus event occurs, adjust the menus if necessary, but try not to do much else. In particular, don't do anything to cause another window management event during the processing of a window management event. For instance, don't set the focus to a window or dialog (with xvt_scr_set_focus_vobj, xvt_dlg_create_res, or xvt_dm_post_note) during an update event. To prevent such problems, XVT restricts the use of many functions during the processing of an E_UPDATE event (see section 4.3.3).

Minimize use of global variables.
In particular, avoid modifying the value of a global variable during the processing of a window management event. For example, if the global variable window holds the WINDOW argument to the event handler, you will likely find that your event processing isn't re-entrant; as a result, you'll be in trouble if the event handler is called recursively. Instead, pass the WINDOW as an argument to every function called directly or indirectly from the event handler, and eliminate the global window.

Avoid functions that implicitly cause recursive update events. The functions that commonly cause these problems include the following:

xvt_app_process_pending_events
xvt_dwin_update
xvt_dm_post_note
xvt_vobj_move
xvt_vobj_destroy
xvt_scr_set_focus_vobj
Any window and dialog creation functions

Note: Since xvt_app_process_pending_events causes recursion as a normal part of its behavior, all of the above suggestions apply to it. xvt_app_process_pending_events is most commonly used during computation-intensive operations that can effectively lock the user interface. By calling this function frequently during such operations, an application can continue to handle events normally. While xvt_app_process_pending_events can slightly delay computations, this is usually a modest price for ensuring that your application remains responsive to the user.

See Also: For a list of functions that can cause recursive update events (and whose use XVT restricts during the processing of E_UPDATE events), see section 4.3.3, next.

4.3.3. E_UPDATE Restrictions

When called during the processing of an E_UPDATE event, several XVT functions can cause unwanted recursive behavior within the event handler. This is usually due to side effects that these functions cause within the context of an E_UPDATE event. For example, calling a function that causes an E_UPDATE to be generated from within the processing of a previous update event can cause endless recursion.

To avoid these problems, XVT restricts use of these functions in the context of E_UPDATE event processing.

Note: If you absolutely *must* make one of these calls during an E_UPDATE, you can use the ATTR_SUPPRESS_UPDATE_CHECK attribute in a call to xvt_vobj_set_attr. XVT recommends that you *not* do this unless it is essential to your application. For details, see the *XVT Portability Toolkit Reference*.

Function Calls Illegal During E_UPDATE Event Processing

Under normal circumstances, you cannot make the following function calls during E_UPDATE event processing. If you call these functions during an E_UPDATE, XVT issues an error.

xvt_app_process_pending_events
xvt_cb_* (except xvt_cb_has_format)
xvt_ctl_check_radio_button
xvt_ctl_create_def
xvt_ctl_def_dialog
xvt_ctl_def_window
xvt_ctl_res_dialog
xvt_ctl_res_window
xvt_ctl_set_checked
xvt_ctl_set_text_sel
xvt_dm_post_ask
xvt_dm_post_error
xvt_dm_post_file_open
xvt_dm_post_file_save
xvt_dm_post_font_sel
xvt_dm_post_message
xvt_dm_post_note
xvt_dm_post_warning
xvt_dwin_invalidate_rect
xvt_dwin_scroll_rect
xvt_dwin_update
xvt_list_* (not including “querying” functions)
xvt_menu_*
xvt_pmap_destroy
xvt_print_*
xvt_sbar_set_*
xvt_scr_set_focus_vobj
xvt_tx_add_par
xvt_tx_append
xvt_tx_clear
xvt_tx_create
xvt_tx_create_def
xvt_tx_destroy
xvt_tx_move
xvt_tx_rem_par
xvt_tx_reset
xvt_tx_resume
xvt_tx_scroll_hor
xvt_tx_scroll_vert
xvt_tx_set_*
xvt_tx_suspend
xvt_vobj_destroy
xvt_vobj_move
xvt_vobj_set_enabled
xvt_vobj_set_palet
xvt_vobj_set_title
xvt_vobj_set_visible
xvt_win_create

xvt_win_set_caret_pos
xvt_win_set_caret_size
xvt_win_set_caret_visible
xvt_win_set_doc_title

Note: Although xvt_dm_post_error and xvt_dm_post_warning appear in the list above, the error signaled by calling them is posted by the XVT "last chance" error handler after the completion of the E_UPDATE event.

4.4. Managing Events

This section discusses some techniques for managing events. To manage events in your application, you can:

- Anticipate the order in which events are received by understanding XVT's event ordering rules
- Prevent some events from reaching event handlers by masking them
- Access native GUI system events by using XVT-provided "hook" functions

4.4.1. Event Ordering Rules

XVT enforces certain rules regarding the ordering of events. If you make no assumptions about event ordering other than those in the rules below, your applications will be less error-prone, and wil port more quickly.

Note: The rules listed below might not be maintained if an error condition occurs.

Event Ordering Rules

1. The first event that a window receives is an E_CREATE, and the last event is an E_DESTROY. XVT guarantees this pair of events for each window.
2. During the processing of an E_CREATE, any XVT window operation for that window is valid. If the window was created initially visible, then the window will be visible at the time of the E_CREATE. (This also holds true for dialogs.)
3. During the processing of an E_DESTROY, you can't call any XVT functions that refer to the WINDOW being destroyed, except xvt_vobj_get_data.

4. The minimum sequence of events that an application receives for a new window or dialog is E_CREATE, E_SIZE, E_DESTROY. On some platforms, performing certain operations during a window's E_CREATE (such as creating a dialog) can cause an E_SIZE event to be delivered to the window before the completion of the E_CREATE callback. This violates the pairing of E_CREATE / E_SIZE events. If the application needs the size of a window during the processing of an E_CREATE, call xvt_vobj_get_client_rect or xvt_vobj_get_outer_rect. These functions will return correct values by the time an E_CREATE event is generated.
5. Applications are guaranteed to receive an E_FOCUS (FALSE) (or an E_DESTROY), for every E_FOCUS (TRUE).
6. E_CHAR events are only received between E_FOCUS (TRUE) and E_FOCUS (FALSE) events.
7. A mouse double-click is represented like this: E_MOUSE_DOWN, E_MOUSE_UP, E_MOUSE_DBL, E_MOUSE_UP. Under some conditions, the final E_MOUSE_UP may not be delivered if the mouse has moved outside the window before being released.
8. When window resizing operations cause an E_UPDATE, the E_SIZE is always sent to the application before the E_UPDATE.
9. The TASK_WIN has identical event semantics to other windows, with the E_QUIT event added. E_QUIT can be received any time after the E_SIZE event is received.
10. Dialog event handlers do not receive the following events: E_COMMAND, E_FONT, E_MOUSE_*, E_QUIT, E_*SCROLL, and E_UPDATE.
11. Dialog event handlers can receive the following events: E_CHAR, E_CONTROL, E_CLOSE, E_CREATE, E_DESTROY, E_FOCUS, E_HELP, E_SIZE, E_TIMER, and E_USER.

Note: Event ordering rules for E_CREATE, E_DESTROY, and E_FOCUS events are identical for dialogs and for windows. Aside from the above rules, no ordering of events is defined or guaranteed.

4.4.2. Event Masking

XVT allows you to block (or "mask") certain event types from reaching window or dialog event handlers. At a minimum, XVT ensures that these event types do not reach the event handler. For some types, on some platforms, XVT masks at the level of the native GUI system, so that these event types are never generated.

The following two XVT functions support event masking:

- xvt_win_set_event_mask
  Sets an event mask for a specified window or dialog.
- xvt_win_get_event_mask
  Gets the current event mask for a specified window or dialog.

Additionally, XVT provides the following set of constants to represent masks for each XVT event type:

```
#define EM_NONE ...
#define EM_ALL ...
#define EM_CREATE ...
#define EM_DESTROY ...
#define EM_FOCUS ...
#define EM_SIZE ...
#define EM_UPDATE ...
#define EM_CLOSE ...
#define EM_MOUSE_DOWN ...
#define EM_MOUSE_UP ...
#define EM_MOUSE_MOVE ...
#define EM_MOUSE_DBL ...
#define EM_MOUSE_SCROLL ...
#define EM_CHAR ...
#define EM_VSCROLL ...
#define EM_HSCROLL ...
#define EM_COMMAND ...
#define EM_FONT ...
#define EM_CONTROL ...
#define EM_TIMER ...
#define EM_QUIT ...
#define EM_HELP ...
#define EM_USER ...
```

EM_NONE means that no events are sent to the event handler; EM_ALL means that all events are sent to the event handler. EM_ALL is the default constant.

Event mask constants are defined so that they can be easily OR'd together when specified as the mask argument to xvt_win_set_event_mask. Perhaps the easiest way to understand event masking is to remember that you specify the events that you want the event handler to receive.

Example: If you wanted to mask all E_MOUSE_MOVE and E_CHAR events from a window's event handler, you would call xvt_win_set_event_mask like this:

```
xvt_win_set_event_mask(my_win, ~(EM_MOUSE_MOVE | EM_CHAR));
```

Some XVT platforms can mask certain events at the native GUI system level. For example,the result of masking mouse move events at the server level under X-based XVT platforms is that these events never reach the client application, and client-server network traffic is greatly reduced.

4.4.3. Defining Event and Keyboard Hooks

XVT provides functions for accessing native GUI system events (including keystrokes) before they are processed by XVT. These "hook" functions let you examine, modify, reroute, or even discard such events. Each function is platform-dependent, because the events it processes are platform-specific.

XVT provides source code for the keyboard hook functions on each XVT platform. There is no default event hook function.

Tip: To specify replacement hook functions to XVT:

Call xvt_vobj_set_attr, with ATTR_EVENT_HOOK and ATTR_KEY_HOOK.

The address of your replacement hook function is passed to xvt_vobj_set_attr, and replaces the default hook function.

Example: The calls to xvt_vobj_set_attr look like this:

```
xvt_vobj_set_attr(NULL_WIN, ATTR_EVENT_HOOK, (long)
      event_hook);
xvt_vobj_set_attr(NULL_WIN, ATTR_KEY_HOOK, (long)
      key_hook);
```

The name of the hook function used by the application replaces event_hook and key_hook. (You choose the name and code the hook function.)

See Also: For more details on the keyboard hook functions, see the *XVT Platform-Specific Books*.

4.4.4. Application Errors

Normally, applications clean up in response to the E_DESTROY event sent to the task window. However, abnormal exits might not generate this event.

XVT lets you create your own error handler function so you can deal with abnormal exits and perform appropriate cleanup activities. You can register your function by using xvt_vobj_set_attr.

In any case, the final XVT behavior is to display a fatal error dialog box and then terminate the application.

See Also: The example in section 2.4 shows how to specify an application-specific error handler.
For more information about error handling, see Chapter 21, *Diagnostics and Debugging*.

4.5. Descriptions of XVT Events

The sections that follow (organized alphabetically) describe each event type. Only the part of the EVENT_TYPE union that applies to the event under discussion is shown.

See Also: To see the complete EVENT_TYPE structure, see section 4.2 on page 4-6.
For additional details about XVT events, see the "Events" portion of the *XVT Portability Toolkit Reference*.

4.5.1. E_CHAR Events and Virtual Key Codes

Partial Event Structure

```
typedef struct {
      EVENT_TYPE type;/* E_CHAR */
      union {
            ...
            struct s_char {
                  XVT_WCHAR ch;              /* wide character */
                  BOOLEAN shift;             /* Shift key? */
                  BOOLEAN control;           /* Control or
                                                Option key? */
                  BOOLEAN virtual_key;       /* virtual key? */
                  unsigned long modifiers;   /* key bit field
                                                modifiers */
            } chr;
            ...
      } v;
} EVENT;
```

XVT sends an E_CHAR event to the event handler for a WINDOW when the user types a character or virtual key code into a window. The E_CHAR event is delivered only to the event handler of the window which has the keyboard focus, and then only if a control has not absorbed the character event for its own use. When the WINDOW event handler receives an event, the WINDOW argument specifies the window in which the event occurred, and the EVENT pointer defines an EVENT structure with fields in the chr union expressing event-specific information. The ATTR_PROPAGATE_NAV_CHARS attribute allows your application to control the propagation of character events from controls to windows.

If the key is held down and auto-repeat occurs, a separate event is generated for each repetition. Consequently, repeated characters do not require special handling.

Implementation Note: In XVT/Win32 the task window's event handler receives E_CHAR events only if the task window is drawable (platform-specific attribute ATTR_WIN_PM_DRAWABLE_TWIN is set to TRUE). On XVT/Mac, E_CHAR events also are delivered to XVT dialogs. However, to maintain portability, you should not process character events sent to dialogs.

Processing Characters

The EVENT substructure chr contains the character code member (ch) which is an XVT_WCHAR. XVT_WCHAR is an encapsulation of the ANSI wchar_t type, although this implementation may vary depending on the support supplied by native ANSI C libraries. Applications should not assume that the size of this field is equal to a short.

Multibyte-aware applications must use the XVT function xvt_str_convert_wc_to_mb before assigning a wide character to a multibyte string array or before processing the character with other XVT functions. In a switch statement test of a wide character, a multibyte application must also compare the ch character to a wide character constant.

It is recommended, though not required, that single-byte applications also call xvt_str_convert_wc_mb. However, single-byte applications can always cast XVT_WCHAR characters to char as long as the character is not a virtual key (and does not rely on the virtual key—high byte—portion of the XVT_WCHAR). In multibyte applications, this method does not work because the high byte portion is necessary for representing normal character keys.

See Also: For more information about processing strings in a multibyte-aware application, see section 19.2.5 on page 19-25.

Text Edit Object Events

If the window with focus contains a text edit object, and the text edit object is active, your application must take special action to process the character event. (Your application can determine if the text edit object is active by calling the following two functions in succession: xvt_scr_get_focus_vobj and xvt_vobj_get_type.) In such a case, when a user types a character in the text edit object, the E_CHAR event is sent to the window's event handler. Your event handler should propagate the character event to the text edit object by calling xvt_tx_process_event.

Shift and Control Characters

The BOOLEAN members of the chr substructure, shift and control, indicate whether the Shift or Control keys were held down while a character was typed. However, if the user types an uppercase character or a control character (such as '\t' or '\b'), the true value of the character code is in ch, so your application doesn't have to look at shift or control to see what was actually typed. In fact, your application should use the shift and control members sparingly, because doing so may make it less portable.

Implementation Note: On XVT/Mac, the shift field does not always report the depression of a Shift key. In some cases, the character is converted prior to event creation (just like happens with an uppercase character).

Modifier Keys

In addition to the shift and control fields, the modifiers field is a general way for detecting a pressed modifier key (Control key, Option key, Alt key, etc.). This field holds bit-wise flags to indicate one or more modifier keys selected. All available modifier keys are passed in the E_CHAR event for use by the application. The following constants are defined for bit positions in the modifiers field and indicate which corresponding key or keys are held down:

XVT_MOD_KEY_NONE
: No modifier keys are pressed (and only this bit is set in the modifiers field).

XVT_MOD_KEY_SHIFT
: Shift key is pressed (either XVT_MOD_KEY_LSHIFT bit or XVT_MOD_KEY_RSHIFT bit also set on platforms that can detect individual Left or Right Shift keys).

XVT_MOD_KEY_CTL
: Control key is pressed.

XVT_MOD_KEY_ALT
: Alt key is pressed.

XVT_MOD_KEY_LSHIFT
: Left Shift key is pressed on platforms that can detect Left Shift key (XVT_MOD_KEY_SHIFT bit also set).

XVT_MOD_KEY_RSHIFT
: Right Shift key is pressed on platforms that can detect Right Shift key (XVT_MOD_KEY_SHIFT bit also set).

XVT_MOD_KEY_CMD
: Command key is pressed.

XVT_MOD_KEY_OPTION
: Option key is pressed.

XVT_MOD_KEY_COMPOSE
: Compose key is pressed (available on XVT/XM only).

Implementation Note: On XVT/Win32, XVT_MOD_KEY_RSHIFT and XVT_MOD_KEY_LSHIFT are not reported. XVT_MOD_KEY_CMD and XVT_MOD_KEY_OPTION are reported on XVT/Mac only. XVT_MOD_KEY_COMPOSE is reported on XVT/XM only.

Virtual Keys

XVT virtual key values are the K_* values (F1, Home key, etc.) defined in the **xvt_defs.h** header file. Virtual keys in character events may be detected in several ways.

For the ASCII character code set only, values of the ch field greater than UCHAR_MAX indicate a virtual key (except for K_DEL, which is less than UCHAR_MAX).

The virtual_key member of the chr substructure is also set to TRUE to identify virtual key characters. In multibyte applications, virtual key codes may conflict with some multibyte character encodings. Therefore, the virtual_key field must be validated for multibyte applications.

Alternatively, the most general means for testing for a virtual key (regardless of charactercode set) is to pass the EVENT structure to the xvt_event_is_virtual_key utility function, which then determines if the character in the E_CHAR event is a virtual key.

Implementation Note: Do not use the Control key for keyboard shortcuts (mnemonics), because the native platforms for XVT/Win32 use the Control key with menu accelerators. Also, on XVT/Mac, the Option key

generates non-ASCII characters.
XVT/Mac stores these characters into the ch member, and they can be handled normally but their use may make your application non-portable.

See Also: For details about virtual key codes, see "Key Codes" in the *XVT Portability Toolkit Reference*.

Key Hook Attribute

You can change the mapping of raw key codes (as generated by the keyboard) to XVT virtual key codes, or add new codes, by changing the default key hook function. This is done with the function xvt_vobj_set_attr and the attribute ATTR_KEY_HOOK.

The parameters passed to a key hook function vary between platforms. Parameters also depend on whether your XVT application is capable of processing multibyte characters (by setting ATTR_MULTIBYTE_AWARE to TRUE):

Single-byte mode
Hook functions receive only platform-specific data.

Multibyte-aware mode
Key hook functions on all platforms receive a pointer to the EVENT structure (E_CHAR event), in addition to platform-specific information. This is necessary because only the hook function knows if it is mapping a passed character to a virtual key in a multibyte-aware environment and can set the virtual_key member properly.

Note that the interface for multibyte hook functions is called only if ATTR_MULTIBYTE_AWARE is set to TRUE, otherwise the single-byte (default) interface is used.

See Also: For details on the key hook functions, see ATTR_KEY_HOOK, described in the appendices of the *XVT Platform-Specific Books*.

Example: The following code processes characters delivered in E_CHAR events:

```
long XVT_CALLCONV1 win_eh(WINDOW win, EVENT * ep)
{
    static int x = LEFT_MGN, y = 0;
    char mbc[XVT_MAX_MB_SIZE + 1];
    int len, width;
    ...
    switch (ep->type) {
        ...
        case E_CHAR:
            if (y == 0) {
                y = doc.height;
                xvt_dwin_set_caret_pos(win, x, y);
                xvt_dwin_set_caret_visible(win, TRUE);
            }
            if (xvt_event_is_virtual_key(ep)) {
                /* don't process virtual characters */
                return;


            len = xvt_str_convert_wc_to_mb(mbc,
                                ep->v.chr.ch);
            if ((len == 0) ||
                 ((len == 1) && !xvt_str_is_alnum(mbc)))
                /* only process alphanumeric characters */
                return;

            width = xvt_dwin_get_text_width(win, mbc, 1);
            if (x + width > doc.rct.right) {
                if (++doc.curline >= doc.maxlines) {
                    xvt_dm_post_note(
                        "Characters don't fit!");
                    return;
                }
                x = LEFT_MGN;
                y += doc.height;
            }
            xvt_dwin_draw_text(win, x, y, mbc, 1);
            x += width;
            xvt_dwin_set_caret_pos(win, x, y);
            save_char(ep->v.chr.ch);
            ...
            break;
        ...
    }
}
```

4.5.2. E_CLOSE Events

Partial Event Structure

```
typedef struct {
      EVENT_TYPE type;          /* E_CLOSE */
      union {
                                /* no union member */
      } v;
} EVENT;
```

XVT sends an E_CLOSE event to the event handler for a window or dialog in response to the user clicking its "close box." Windows that aren't created with either the WSF_CLOSE or WSF_DECORATED flags won't have a "close box." (Choosing Close on the File menu doesn't generate an E_COMMAND event, since it's really a menu selection.)

The WINDOW argument tells the event handler which dialog or window the user tried to close. No additional information is needed to process this event, so the EVENT structure contains none.

When this event is received, the window or dialog hasn't actually been closed; your application must call xvt_vobj_destroy to accomplish that. Additional events (such as E_FOCUS) may then be generated for the window, and your application must be prepared to handle them. The last event for a window will be an E_DESTROY (discussed later in this section).

If E_CLOSE is ignored, then no window is closed, and nothing in the application changes. This distinction is important. Typically, applications check the state of the window upon receiving an E_CLOSE event. If the state indicates that the contents of the window or dialog have been saved (for example), then the application can simply call xvt_vobj_destroy. If, however, the contents have not been saved, the application can display a dialog asking if the user wishes to save or discard changes, so that the changes can be preserved before calling xvt_vobj_destroy.

E_CLOSE events are generated for the task window, regular windows, and dialogs. They are not generated for print windows, pixmaps, controls, or screen windows.

Note: If the task window's event handler receives an E_CLOSE event, and the application calls xvt_vobj_destroy(TASK_WIN), then the application is terminated.

E_CLOSE Example

In the following code, the application provides the function OK_to_close elsewhere:

```
long XVT_CALLCONV1 a_window_eh(WINDOW win, EVENT * ep)
{
	switch (ep->type) {
	case E_COMMAND:
		switch (ep->v.cmd.tag) {
		case M_FILE_CLOSE:
			if (OK_to_close(win))
				xvt_vobj_destroy(win);
			break;
		}
		break;
	case E_CLOSE:
		if (OK_to_close(win))
			xvt_vobj_destroy(win);
		break;
	}
}
```

4.5.3. E_COMMAND Events

Partial Event Structure

```
typedef struct {
	EVENT_TYPE type;		/* E_COMMAND */
	union {
		...
		struct s_cmd {
			MENU_TAG tag;		/* menu item tag */
			BOOLEAN shift;		/* Shift key down? */
			BOOLEAN control;	/* Control (Option) key
						   down? */
		} cmd;
		...
	} v;
} EVENT;
```

XVT generates an E_COMMAND event when the user makes a menu selection (or causes a menu selection by typing a menu-accelerator key). However, selections from Font/Style menus generate E_FONT events, not E_COMMAND events.

The WINDOW referenced in the event handler is associated with the menubar from which the selection was made. Within the cmd substructure (in the EVENT structure referenced in the event handler), the tag member refers to the menu item chosen. Tags can be any integer value between 1 and MAX_MENU_TAG (32000).

If you want to vary the behavior of a command when the Shift, Control, or both keys are down, examine the shift and/or control members to see if the user pressed one of these modifier keys when making the menu selection.

Typically, applications are structured so that the menu tags found in the EVENT structure for E_COMMAND events are parsed via switch statements. These statements are coded in either the event handler or in a function called by the event handler which you can design to handle all menu-initiated operations.

E_COMMAND events are generated only for task windows and top-level windows. Dialogs, child windows, and windows created without menubars cannot have menubars, so their event handlers will never receive E_COMMAND events.

E_COMMAND Example

The following code handles command events in a window event handler. The macros M_FILE_CLOSE and M_FILE_QUIT are defined in **xvtmenu.h**, which is included by **xvt.h**.

```
#include <xvt.h>

static void
do_menu(WINDOW win, MENU_TAG cmd, BOOLEAN shift,
        BOOLEAN control)
{
        switch (cmd) {
        case M_FILE_CLOSE:
                do_close(win);
                break;
        case M_FILE_QUIT:
                xvt_app_destroy();
                break;
        }
}
long XVT_CALLCONV1 a_window_eh(WINDOW win, EVENT * ep)
{
        switch (ep->type) {
        case E_COMMAND:
                do_menu(win, ep->v.cmd.tag, ep->v.cmd.shift,
                 ep->v.cmd.control);
                break;
        }
}
```

Additionally, applications commonly need to pass E_COMMAND events to the task window's event handler, even though the menu is attached to another window. For example, in cases such as the opening and closing of files, you can call

xvt_win_dispatch_event(TASK_WIN, ep) in a window's event handler to pass the events. This allows all processing for some common events to occur in a single place.

4.5.4. E_CONTROL Events

Partial Event Structure

```
typedef struct {
        EVENT_TYPE type;                    /* E_CONTROL */
        union {
                ...
                struct s_ctl {              /* E_CONTROL */
                        short id;           /* control's resource ID */
                        CONTROL_INFO ci;    /* control info */
                } ctl;
                ...
        } v;
} EVENT;
```

XVT sends an E_CONTROL event to the event handler for a window or dialog in response to the user operating a control in that window or dialog. XVT also generates an E_CONTROL event when the underlying window system indirectly modifies the state of a control.

The WINDOW argument passed to the event handler refers to the window or dialog containing the control, not the WINDOW that represents the control itself.

The WINDOW that references the control itself is contained in the win field of the CONTROL_INFO structure contained in the event structure for E_CONTROL events. This WINDOW is used to identify the control for functions that operate on controls.

The ctl substructure of the EVENT structure describes what happened to the control. Its id member is the numeric ID assigned when the control was created. (The id must be unique among the controls within any one window.) The ci member supplies additional control-specific information.

The type member of the CONTROL_INFO structure gives the type of control. Control types are defined in the WIN_TYPE enumeration by the prefix WC_*. All WC_* control types are allowed in both windows and dialogs.

XVT doesn't generate E_CONTROL events for the optional scrollbars placed alongside edges of windows as decorations, but generates E_HSCROLL and E_VSCROLL events instead. However, scrollbar controls within windows or dialogs do generate E_CONTROL events.

Event handlers for task windows, regular windows, and dialogs can all receive E_CONTROL events.

What to Do with an E_CONTROL Event

What should your application do upon receiving an E_CONTROL event? The following design approach works well.

Tip: To handle an E_CONTROL event:

1. Branch on the control ID.
2. Infer the control type based on your ID numbering scheme. This lets you determine which substructure in the union within the CONTROL_INFO structure is appropriate for the event, and for the control.
3. Check the contents of the control-specific substructure.
4. Based on that control's properties and behavior, perform operations that reflect your application's needs for that event.

   For a button control event, do whatever action is initiated by pressing the button. For a check box or radio button, set the control's state appropriately with xvt_ctl_set_checked or xvt_ctl_check_radio_button.

Some controls have no additional information associated with them, while others have several pieces of information.

Note: XVT does not automatically check radio buttons and check boxes. Also, in many cases you will want to defer making permanent changes to the application model until a subsequent button is clicked or a menu command is issued, confirming that it's okay to set these controls. This is a design issue subject to your discretion.

See Also: For information about properties and behaviors of controls, as well as the events they can generate, see Chapter 8, *Controls*.

4.5.5. E_CREATE Events

Partial Event Structure

```
typedef struct {
      EVENT_TYPE type;        /* E_CREATE */
      union {
                              /* no union member */
      } v;
} EVENT;
```

XVT sends an E_CREATE event to the event handler for a WINDOW immediately after the window or dialog has been created. This event is guaranteed to be the first event received by the event handler. E_CREATE is also guaranteed to be the first event sent to the task event handler. The task event handler receives the E_CREATE event after the application calls xvt_app_create.

The WINDOW argument tells the event handler which dialog or window has been created. No additional information is needed to process this event, so the EVENT structure contains none.

In response to E_CREATE events, applications usually perform some or all of the following operations (although you aren't limited to these):

- Initialize the window or dialog (set event masks, initialize structures, etc.)
- Attach window or dialog-specific data via xvt_vobj_set_data
- Set or modify the appearance, title, decorations, attributes, size, or position of the window or dialog
- Create controls to be placed in a window, or initialize controls in windows or dialogs

Do not perform drawing operations during window initializations; your application automatically receives an update event, and in response to this, you can draw in the window.

Note: When the task window's event handler receives an E_CREATE event, the implication is that the application itself has been created (along with the task window). Application-level initializations (and operations such as creating your initial top-level document windows) are best performed in response to an E_CREATE event in the task event handler.

4.5.6. E_DESTROY Events

Partial Event Structure

```
typedef struct {
      EVENT_TYPE type;        /* E_DESTROY */
      union {
                              /* no union member */
      } v;
} EVENT;
```

XVT sends an E_DESTROY event to the event handler of a window or dialog to notify your application that the WINDOW is about to be

destroyed. Typically, an event handler receives an E_DESTROY event soon after your application has called xvt_vobj_destroy. The purpose of the event is to give your application a chance to free memory it has allocated for application data associated with the WINDOW being destroyed.

An E_DESTROY event is guaranteed to be the last event an event handler receives. An E_DESTROY event sent to the task event handler is guaranteed to be the last event received by the application.

The WINDOW argument tells the event handler which window or dialog has been destroyed. No additional information is needed to process this event, so the EVENT structure contains none.

Responding to E_DESTROY Events

In response to E_DESTROY events, applications will usually perform some or all of the following operations (although you aren't limited to these):

- Perform window or dialog-specific terminations (free structures, save window states, close files, etc.)
- Free window or dialog-specific data (set earlier by xvt_vobj_set_data) by calling xvt_vobj_get_data
- Remove the window from application-specific window registration or management data structures

Note: When the task window's event handler receives an E_DESTROY event, the implication is that the application itself is about to be terminated (along with the task window). So, E_DESTROY is a good cue to perform application-level cleanup and termination.

The only XVT WINDOW function that can be called during the processing of an E_DESTROY event is xvt_vobj_get_data for the window or dialog being destroyed. Also, the application data value for controls in windows or dialogs cannot be accessed in response to an E_DESTROY event; by this time, these controls are essentially "dead." If you need to access the application data of a control before the parent window or dialog is destroyed, you can do this in response to the dialog or window's E_CLOSE event, before issuing any calls to xvt_vobj_destroy for the window or dialog.

4.5.7. E_FOCUS Events

Partial Event Structure

```
typedef struct {
      EVENT_TYPE type;                  /* E_FOCUS */
      union {
            ...
            BOOLEAN active;             /* is the window or dialog
                                           gaining the focus? */
            ...
      } v;
} EVENT;
```

An E_FOCUS event is generated when a window or dialog gains or loses the focus (conditions known as activation and deactivation). A WINDOW gains or loses the focus as a result of the following actions:

- The user clicks on a focusable area of a window
- Any active keyboard navigation occurs
- Your application calls xvt_scr_set_focus_vobj

In each case, XVT notifies the application that the focus has changed.

For a given window, focus activation events are always guaranteed to be paired with either a subsequent focus deactivation event or, if the window has been closed while it has focus, an E_DESTROY event. Deactivation events are always followed by activation events, and vice versa, until the window is closed.

Responding to Focus Changes

Windows and dialogs often respond to focus changes in the following ways:

- Set the cursor in a window or dialog to some application-specific shape, which implies that the focus is now here (or has gone).
- Check the clipboard to see what formats are available and, if any can be used, enable the Paste command on the Edit menu. Otherwise, disable the Paste command. This is necessary because activation may have resulted from another application.
- If there is a text or graphic selection and the focus was lost, show it by reversing the selected text (draw a black rectangle

with a drawing mode of M_XOR). If the focus was gained, then redraw the text in XOR mode.

Do not attempt to deal with the following issues, which XVT handles automatically:

- Activating or deactivating the title (caption) bar, the scrollbars, or the size box
- Removing or restoring a blinking caret (do not call xvt_dwin_set_caret_visible when you get an activate or deactivate event)

Focus Deactivate Events

On a focus deactivate event, don't worry about the menubar or the clipboard but, if text or graphics were selected, consider showing them as unselected. Many applications will not have to do anything in response to E_FOCUS events.

Note: On a focus deactivate event, you never know whether the user switched from one window to another within the same application or between different applications.

Focus Example

The following code fragments illustrate how to enable and disable items on the **Edit** menu when a window gains the focus:

```
static CB_FORMAT paste_fmt;

static void update_menus(WINDOW win)
{
        BOOLEAN paste_enable = TRUE;

        if (xvt_cb_has_format(CB_APPL, APPL_FORMAT))
                paste_fmt = CB_APPL;
        else if (xvt_cb_has_format(CB_PICT, NULL))
                paste_fmt = CB_PICT;
        else if (xvt_cb_has_format(CB_TEXT, NULL))
                paste_fmt = CB_TEXT;
        else
                paste_enable = FALSE;

        xvt_menu_set_item_enabled(win, M_EDIT_PASTE,
                paste_enable);
        xvt_menu_set_item_enabled(win, M_EDIT_CLIPBOARD,
                paste_enable);
}
```

```
long XVT_CALLCONV1 a_window_eh (WINDOW win, EVENT * ep)
{
        switch (ep->type) {
        case E_FOCUS:
                if (ep->v.active)
                        update_menus(win);
                break;
        }
}
```

4.5.8. E_FONT Events

Partial Event Structure

```
typedef struct {
        EVENT_TYPE type;                    /* E_FONT */
        union {
                ...
                struct s_efont {
                        XVT_FNTID font_id;  /* logical font ID */
                } font;
                ...
        } v;
} EVENT;
```

An E_FONT event is generated when the user chooses an item from the Font Selection dialog or the Font/Style menu (on platforms that have it).

Implementation Note: On XVT/XM and XVT/Mac, you can add a Font/Style menu to your application by using a DEFAULT_FONT_MENU statement in the XRC file. The XVT/Win32 platform only has a Font Selection dialog, not a Font/Style menu.

E_FONT events are very similar to E_COMMAND events, in that they can represent menu selections. However, E_FONT events represent only selections from the Font/Style menu or Font Selection dialog.

E_FONT events are sent only to window-event handlers. Dialog-event handlers never receive E_FONT events, because dialogs lack menubars. However, if a dialog WINDOW ID is passed as a parameter in an application call to xvt_dm_post_font_sel, the dialog's event handler can receive an E_FONT event.

An E_FONT event represents a user's selection and specification of a logical font. A logical font is a description of a desired physical font—a particular implementation of a font installed on a window system. The XVT_FNTID member of the E_FONT event represents the user's selection.

The entire logical font represents the last font established with xvt_menu_set_font_sel, as modified by the user's menu or dialog choice. In other words, the logical font in the E_FONT event reflects the user's modifications to either the Font/Style menu or Font Selection dialog.

Note: Because the E_FONT event only notifies you of a Font/Style menu or Font Selection dialog choice, XVT does not automatically set check marks on the Font/Style menu, change the default logical font in the Font Selection dialog, or change the logical font in the window.

See Also: For more information about logical fonts, see Chapter 15, *Fonts and Text*.

Responding to E_FONT Events

When your application receives an E_FONT event, it must determine whether to apply the user's Font/Style menu or Font Selection dialog selection. If so, it must do two things:

- Call xvt_menu_set_font_sel, passing it a copy of the XVT_FNTID member contained in the EVENT, so that the displayed Font/Style menu check marks and default Font Selection dialog logical font reflect the selection made
- Store the logical font somewhere (using xvt_font_copy), so that the update code can call xvt_win_set_font to make sure any text that it draws uses that logical font

E_FONT Example: Displaying Text Objects

The following code displays four text objects, represented by this array of structures:

```
#define NUM_OBJS 4

static struct {                         /* information about each
                                                object */
        char *text;                     /* text */
        PNT pos;                        /* starting position */
        RCT bounds;                     /* bounding rectangle */
        XVT_FNTID font_id;              /* font */
} obj[NUM_OBJS] = {
        {
                "This is the first sentence.",
                {50, 10}
        },
        {
                "This is the second sentence.",
                {125, 150}
        },
        {
                "This is the third sentence.",
                {200, 100}
        },
        {
                "This is the fourth sentence.",
                {275, 200}
        }
};
```

During application initialization, the font_id member of each object is set and the bounding rectangle is calculated:

```
static void startup(void)
{
        int i;
        WINDOW win;

        win = xvt_win_create(W_DOC, XVT_MAX_WINDOW_RECT,
                        "FONT", WIN_MENUBAR, TASK_WIN,
                        WSF_SIZE|WSF_CLOSE, EM_ALL, win_eh, 0L);
        for (i = 0; i < NUM_OBJS; i++) {
                obj[i].font_id = xvt_font_create();
                xvt_font_set_family(obj[i].font_id,
                                XVT_FFN_HELVETICA);
                xvt_font_set_style(obj[i].font_id, XVT_FS_NONE);
                switch (i % 3) {
                case 0:
                        xvt_font_set_size(obj[i].font_id, 20);
                        break;
                case 1:
                        xvt_font_set_size(obj[i].font_id, 10);
                        break;
                case 2:
                        xvt_font_set_size(obj[i].font_id, 12);
                }
                set_bounds(win, i);
        }
        xvt_menu_set_item_enabled(win, M_FILE_NEW, TRUE);
}

static void
set_bounds(WINDOW win, int n)
{
        int ascent, descent, width;

        xvt_font_map(obj[n].font_id, win);
        xvt_font_get_metrics(obj[n].font_id, NULL,
                &ascent, &descent);
        xvt_dwin_set_font(win, obj[n].font_id);
        width = xvt_dwin_get_text_width(win,
                obj[n].text, -1);
        xvt_rect_set(&obj[n].bounds, obj[n].pos.h,
                obj[n].pos.v - ascent, obj[n].pos.h + width,
                obj[n].pos.v + descent);
}
```

When the user clicks within the bounding rectangle of one of the four objects, the global variable sel_obj is set to that object's subscript in the obj array, and the function invert_selection is called to select it. Or, if the clicked-on object is already selected, sel_obj is set to NO_OBJ and no object is selected.

E_FONT Example: Menus and Dialogs

The Font/Style menu and Font Selection dialog are set to correspond to the currently selected object, or to have no check marks (in the menu) if no object is selected. This serves two purposes:

- When an object is selected, the user can pull down the Font/Style menu or bring up the Font Selection dialog and see what its attributes are
- If the user actually specifies a new font, the font_id passed along with the E_FONT event reflects the new font for the object

If the user pulls down the Font/Style menu when no object is selected, the absence of check marks shows that no font can be changed. If the user brings up the Font Selection dialog when no object is selected, the default font is the same as that returned by xvt_font_create.

```
#define NO_OBJ -1
static int sel_obj = NO_OBJ;
static void
invert_selection(WINDOW win)
{
	if (sel_obj != NO_OBJ) {
		DRAW_CTOOLS tools;
		CBRUSH hollow_cbrush;

		hollow_cbrush.pat = PAT_HOLLOW;
		hollow.cbrush.color = COLOR_BLACK;
		xvt_dwin_get_draw_ctools(win, &tools);
		xvt_dwin_set_draw_mode(win, M_XOR);
		xvt_dwin_set_std_cpen(win, TL_PEN_BLACK);
		xvt_dwin_set_cbrush(win, &hollow_cbrush);
		xvt_dwin_draw_rect(win, &obj[sel_obj].bounds);
		xvt_dwin_set_draw_ctools(win, &tools);
	}
}
static void
fix_font_menu(WINDOW win)
{
	if (sel_obj != NO_OBJ)
		xvt_menu_set_font_sel(win, obj[sel_obj].font_id);
	else {
		xvt_menu_set_font_sel(win, NULL_FNTID);
	}
}
```

E_FONT Example: Handling the E_FONT

When an E_FONT event occurs, the font_id passed in the EVENT structure reflects the new logical font for the currently selected object, since the menus were previously set up to reflect that object's logical font. Here is the code that handles E_FONT events:

```
static void
do_font(WINDOW win, XVT_FNTID font_id)
{
        RCT rct;

        if (sel_obj != NO_OBJ) {
                xvt_menu_set_font_sel(win, font_id);
                rct = obj[sel_obj].bounds;/* old bounds */
                inflate_rect(&rct, 1);
                xvt_dwin_invalidate_rect(win, &rct);
                                                /* old bounds */
                xvt_font_copy(obj[sel_obj].font_id, font_id,
                        XVT_FA_ALL);
                set_bounds(win, sel_obj);
                rct = obj[sel_obj].bounds;/* old bounds */
                inflate_rect(&rect, 1);
                xvt_dwin_invalidate_rect(win, &rct);
                                                /* old bounds */
        }
}
long XVT_CALL_CONV1 win_eh(WINDOW win, EVENT * ep)
{
        switch (ep->type) {
        ...
        case E_FONT:
                do_font(win, ep->v.font.font_id);
                break;
        ...
        }
}
```

4.5.9. E_HELP Events

Partial Event Structure

```
typedef struct {
      EVENT_TYPE type;                  /* E_HELP */
      union {
            ...
            struct s_help {
                  WINDOW obj;           /* target for help--window,
                                           dialog, or control */
                  MENU_TAG tag;         /* target for help--menu
                                           item */
                  XVT_HELP_TID topic;
                                        /* help topic, usually
                                           NULL_TID */
            } help;
            ...
      } v;
} EVENT;
```

An E_HELP event is generated when the application user requests online help. Usually your application does not have to handle E_HELP explicitly, since the help system handles this event automatically and does not pass it on to your application's event handlers.

You can process the help event yourself if you're writing your own help system or creating special-case help services for certain containers.

Only one of the three members of the s_help structure is relevant for any single E_HELP event, depending on the type of object for which the user requested help.

If the user requests help for a window, dialog, or control:

- The WINDOW member of s_help contains the identifier of that object
- The MENU_ITEM member of s_help is NULL
- The XVT_HELP_TID member of s_help contains NULL_TID

If the user requests help for a menu item:

- The MENU_ITEM member of s_help is the identifier of the menu item for which help is requested
- The WINDOW member of s_help is NULL_WIN
- The XVT_HELP_TID member of s_help contains NULL_TID

If the user requests help for a specific topic (rather than for a specific GUI object):

- The XVT_HELP_TID member of s_help contains the topic identifier
- The WINDOW member of s_help is NULL_WIN
- The MENU_ITEM member of s_help is NULL

See Also: For more information on handling help events, see Chapter 22, *Hypertext Online Help*.

4.5.10. E_HSCROLL and E_VSCROLL Events

Partial Event Structure

```
typedef struct {
      EVENT_TYPE type;              /* E_VSCROLL or E_HSCROLL */
      union {
            ...
            struct s_scroll_info {
                  SCROLL_CONTROL what; /* location */
                  short pos;                  /* thumb position */
            } scroll;
            ...
      } v;
} EVENT;
```

XVT sends an E_HSCROLL event to a window's event handler to notify your application that the user has operated one of the scrollbars that are part of a window's frame. Only windows with the WSF_HSCROLL, WSF_VSCROLL, or WSF_DECORATED flag set at creation time receive these events. These flags cannot be added or removed later.

These events are not sent to dialog event handlers, because dialog windows cannot have scrollbars.

The what member indicates which part of the scrollbar was operated, as shown below:

```
typedef enum {                /* site of scrollbar activity */
      SC_NONE,                /* nowhere (event ignored) */
      SC_LINE_UP,             /* one line up */
      SC_LINE_DOWN,           /* one line down */
      SC_PAGE_UP,             /* previous page */
      SC_PAGE_DOWN,           /* next page */
      SC_THUMB,               /* thumb repositioning */
      SC_THUMBTRACK           /* dynamic thumb tracking */
} SCROLL_CONTROL;
```

The various parts of an XVT scrollbar are shown in Figure 4.2. The interpretation of line and page is entirely up to your application. Each click on the scrollbar generates a separate E_HSCROLL or E_VSCROLL event. If the user holds the mouse button down, a sequence of events occurs.

Member what is equal to SC_THUMBTRACK while the user drags the thumb, and SC_THUMB when the user stops dragging. In these cases, member pos indicates the current position of the thumb relative to the range and proportion of the scrollbar. The range must have been previously set with a call to xvt_sbar_set_range. If no such call was made, the range is undefined, so pos is meaningless.

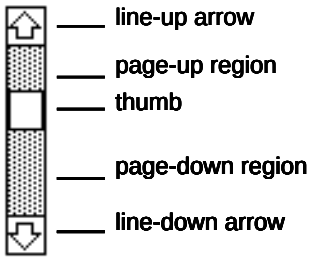


Figure 4.2. Parts of a scrollbar

Usually, your application will do the same thing for both SC_THUMBTRACK and SC_THUMB scrollbar events; if you prefer not to track the thumb dynamically, you can safely ignore SC_THUMBTRACK occurrences because you will always get a corresponding SC_THUMB when the user stops dragging.

XVT doesn't do anything to the window or to the scrollbar when one of these events occurs.

Tip: To scroll the contents of the window:

Call xvt_dwin_scroll_rect.

Tip: To show the thumb in a new position:

Call xvt_sbar_set_pos.

See Also: Functions that affect scrollbars are also discussed in section 8.3.7 in Chapter 8, *Controls*.
For more details about scrolling, see Chapter 13, *Scrolling*.

Scrolling Example

The following code reads a fileinto memory, and lets the user scroll both horizontally and vertically through the text.

The application data for a window is a pointer to a structure containing the following information:

```
typedef struct {
        int nlines;                 /* number of text lines */
        char **lines;               /* array of pointers to text */
        int maxlines;               /* max capacity of array */
        int line_height;            /* height of line in win */
        int maxwidth;               /* max line width */
        PNT org;                    /* origin (for scrolling) */
        ...
} DOC;
```

This discussion addresses only these fields: nlines, maxwidth, line_height, and org. The nlines and maxwidth fields indicate how much data the window must scroll vertically and horizontally, respectively. The line_height field indicates how many pixels constitute one line of text in the current physical font. The org field maintains the position of the window's upper-left corner relative to the data being displayed.

When drawing in the window, it will be necessary to convert from data-relative coordinates to window-relative coordinates. This is done by subtracting the origin org from the data-relative coordinates It's okay if the result is outside the window's client area, since XVT clips the output appropriately. These issues are discussed in the E_UPDATE section (section 4.5.19).

When a file is first read, when the window is resized, or when the font changes, the scrollbars are set with this function:

```
static void scroll_sync(WINDOW win, int height,
        int width)
{
        DOC * d;
        d = get_doc_data(win);
        xvt_sbar_set_range(win, VSCROLL, 0,
                d->nlines + height / d->line_height);
        xvt_sbar_set_range(win, HSCROLL, 0,
                d->maxwidth + width);
        xvt_sbar_set_proportion(win, VSCROLL,
                height / d->line_height);
        xvt_sbar_set_proportion(win, HSCROLL, width);
        xvt_sbar_set_pos(win, VSCROLL,
                d->org.v / d->line_height);
        xvt_sbar_set_pos(win, HSCROLL, d->org.h);
}
```

The height and width arguments are the height and width of the window. The range is set to contain all the data (nlines or maxwidth), plus an extra window-sized amount in each direction. This allows the user to scroll up to one page past the end of data, which is typical of many word processing applications. The vertical scrollbar is stated in units of lines (based on the line_height); the horizontal (maxwidth) is stated in units of pixels.

The scrollbar proportions are set to the amount of data visible within the window. This allows platforms that support proportional scrollbars to work correctly.

Finally, the scrollbar positions are set based on the current position of the window within the data.

When an E_HSCROLL or E_VSCROLL event occurs, the function do_scroll is called:

```
static void
do_scroll(WINDOW win, SCROLL_TYPE type,
        SCROLL_CONTROL what, short pos)
{
        DOC * d;
        RCT rct;
        PNT old_org;
        int dh, dv, maxorg, page;

        xvt_dwin_update(win);

        d = get_doc_data(win);
        old_org = d->org;
        xvt_vobj_get_client_rect(win, &rct);

        switch (type) {
        case HSCROLL:
                switch (what) {
                case SC_LINE_UP:
                        d->org.h = max(0, old_org.h - HINTERVAL);
                        break;
                case SC_LINE_DOWN:
                        d->org.h = min(d->maxwidth,
                                old_org.h + HINTERVAL);
                        break;
                case SC_PAGE_UP:
                        d->org.h = max(0, old_org.h - rct.right);
                        break;
                case SC_PAGE_DOWN:
                        d->org.h = min(d->maxwidth,
                                old_org.h + rct.right);
                        break;
                case SC_THUMB:
                        d->org.h = pos;
                        break;
```

```
                default:
                        break;
                }
                break;
        case VSCROLL:
                maxorg = d->nlines * d->line_height;
                page = rct.bottom / d->line_height *
                        d->line_height;
                switch (what) {
                case SC_LINE_UP:
                        d->org.v = max(0,
                                old_org.v - d->line_height);
                        break;
                case SC_LINE_DOWN:
                        d->org.v = min(maxorg,
                                old_org.v + d->line_height);
                        break;
                case SC_PAGE_UP:
                        d->org.v = max(0, old_org.v - page);
                        break;
                case SC_PAGE_DOWN:
                        d->org.v = min(maxorg, old_org.v + page);
                        break;
                case SC_THUMB:
                        d->org.v = pos * d->line_height;
                        break;
                default:
                        break;
                }
        }
        dh = old_org.h - d->org.h;
        dv = old_org.v - d->org.v;
        if (dh != 0 || dv != 0) {
                xvt_dwin_scroll_rect(win, &rct, dh, dv);
                xvt_sbar_set_pos(win, VSCROLL,
                        d->org.v / d->line_height);
                xvt_sbar_set_pos(win, HSCROLL, d->org.h);
        }
}

long XVT_CALLCONV1 win_eh(WINDOW win, EVENT * ep)
{
        switch (ep->type) {
        ...
        case E_HSCROLL:
                do_scroll(win, HSCROLL, ep->v.scroll.what,
                        ep->v.scroll.pos);
                break;
        case E_VSCROLL:
                do_scroll(win, VSCROLL, ep->v.scroll.what,
                        ep->v.scroll.pos);
                break;
        ...
        }
}
```

Features of the do_scroll Function

The following are several important things to note in do_scroll:

- Before doing anything else, you must call the XVT function xvt_dwin_update. This causes any pending updates to be processed before you do the scrolling. Without this call, these updates would be processed using the new origin, even though they applied to the value of the origin before the window was scrolled.
- You must change the origin stored in the window's application data before calling xvt_dwin_scroll_rect. This is necessary because xvt_dwin_scroll_rect will generate one or more E_UPDATE events before it returns. For the drawing to be done correctly by the update code, it will need the new value of the origin.
- Make sure that the origin stays within the range of allowed values; calls to max and min guarantee this. Recall that the vertical scrollbar is stated in units of lines, and the horizontal is stated in units of pixels. The vertical size of a line is the line_height, based on the physical font; horizontally, an arbitrary number of pixels is used. Similarly, vertical pages must be an integral multiple of line_height; horizontally, the width of the client area is sufficient.
- Finally, after computing the new origin, call xvt_dwin_scroll_rect to scroll the window's contents, and call xvt_sbar_set_pos to set the new position of the thumb.

See Also: For more information about scrolling, see Chapter 13, *Scrolling*.

4.5.11. E_MOUSE_DBL Events

Partial Event Structure

```
typedef struct {
      EVENT_TYPE type;                  /* E_MOUSE_DBL */
      union {
            ...
            struct s_mouse {
                  PNT where;            /* location of event */
                  BOOLEAN shift;        /* Shift key down? */
                  BOOLEAN control;      /* Control key down? */
                  short button;         /* button number */
                  XVT_INT32 scroll_x;   /* scroll delta in the X axis -
                                           E_MOUSE_SCROLL only */
                  XVT_INT32 scroll_y;   /* scroll delta in the Y axis -
                                           E_MOUSE_SCROLL only */
            } mouse;
            ...
      } v;
} EVENT;
```

XVT sends an E_MOUSE_DBL event to a window's event handler for an XVT window in response to the user double-clicking a mouse button while the mouse pointer is in the client area of the window.

Dialogs do not send mouse events to their event handlers. The location of the mouse cursor in window-relative coordinates is stored in the where member. The states of the Shift and Control keys are stored in the shift and control members. The button member specifies the button; it can have the values 0, 1, or 2, but only 0 and 1 are fully portable.

Double-click Definition

A double-click is defined as a button-down action that rapidly follows a button-up action. Each platform defines "rapidly" according to its own tolerances. XVT reports the button-up action separately as an E_MOUSE_UP event. A second E_MOUSE_UP follows the E_MOUSE_DBL as soon as the user lets up on the button.

Hence, four events result (in this order) from a double-click:

```
E_MOUSE_DOWN
E_MOUSE_UP
E_MOUSE_DBL
E_MOUSE_UP
```

The application does not necessarily receive all four events. When one of the events occurs, the application might take action (such as

bringing up a dialog box) that precludes receiving the other events. See the example below.

If your application must handle single-click events but ignore double-click events, you should either set an application flag when an E_MOUSE_DBL event occurs and then ignore E_MOUSE_UP events when that flag is set, or ignore E_MOUSE_UP events entirely.

See Also: For more information about mouse events, see the discussion of dragging in section 4.5.13 on page 4-48.

E_MOUSE_DBL Example

Tip: Before reading this example, review section 4.5.8 on page 4-31.

In this example, when the user double-clicks within an object's bounding rectangle, the application makes sure that object is selected (whether it already is or not) and then opens a dialog box that shows its point size:

```
static void do_double(WINDOW win, PNT where)
{
        int dbl_obj;
        long size;

        if ((dbl_obj = find_obj(where)) != NO_OBJ) {
                if (sel_obj != dbl_obj) {
                        invert_selection(win);
                        sel_obj = dbl_obj;
                        invert_selection(win);
                }
        size = xvt_font_get_size(obj[sel_obj].font_id);
        xvt_dm_post_note("%d points", size);
        }
}
long XVT_CALLCONV1 win_eh(WINDOW win, EVENT * ep)
{
        switch (ep->type) {
        ...
        case E_MOUSE_DBL:
                do_double(win, ep->v.mouse.where);
                break;
        ...
        }
}
```

The application does not receive an E_MOUSE_UP event following the E_MOUSE_DBL because a modal dialog box appears immediately upon receipt of the E_MOUSE_DBL. The mouse up event goes to the dialog box, not to an XVT window and, because mouse events in dialogs are not sent to your application, it is ignored.

4.5.12. E_MOUSE_DOWN Events

Partial Event Structure

```
typedef struct {
      EVENT_TYPE type;                  /* E_MOUSE_DOWN */
      union {
            ...
            struct s_mouse {
                  PNT where;            /* location of event */
                  BOOLEAN shift;        /* Shift key down? */
                  BOOLEAN control;      /* Control key down? */
                  short button;         /* button number */
                  XVT_INT32 scroll_x;   /* scroll delta in the X axis -
                                           E_MOUSE_SCROLL only */
                  XVT_INT32 scroll_y;   /* scroll delta in the Y axis -
                                           E_MOUSE_SCROLL only */
            } mouse;
            ...
      } v;
} L

EVENT;
```

XVT sends an E_MOUSE_DOWN event to a window's event handler in response to the user clicking a mouse button while the mouse pointer is in the client area of the window.

In XVT, dialogs do not send mouse events to their event handlers. The native platform handles mouse events in dialogs. The location of the mouse cursor in window-relative coordinates is stored in the where member. The states of the Shift and Control keys are stored in the shift and control members. The button member specifies the button; it can have the values 0, 1, or 2, but only 0 and 1 can be generated on all platforms.

When the user releases the button, a separate E_MOUSE_UP event can be generated.

Normally, a click is defined as an E_MOUSE_DOWN /E_MOUSE_UP pair; however, on some platforms, if the cursor is moved outside the window in which the E_MOUSE_DOWN event occurred, the corresponding E_MOUSE_UP event might be dropped or sent to a different window. If E_MOUSE_MOVE events occur between E_MOUSE_DOWN and E_MOUSE_UP, the user is dragging the mouse.

If the mouse has moved out of all XVT windows, you will not get the event at all.

Tip: To guarantee that you get an E_MOUSE_UP, even if it occurs outside the window in which the E_MOUSE_DOWN occurred:

Trap the mouse with xvt_win_trap_pointer and subsequently release it with xvt_win_release_pointer.

See Also: For more information about E_MOUSE_UP events, see section 4.5.14 on page 4-52.

E_MOUSE_DOWN Example

In the following code fragment, the object in which the E_MOUSE_DOWN occurred is noted (in the variable down_obj), but nothing is done unless an E_MOUSE_UP occurs in the same object.

```
static void do_mouse(WINDOW win, EVENT_TYPE type,
        PNT where)
{
        static int down_obj = NO_OBJ;

        switch (type) {
        case E_MOUSE_DOWN:
                down_obj = find_obj(where);
                break;
        case E_MOUSE_UP:
                if (down_obj == find_obj(where)) {
                        /* unselect selected object */
                        invert_selection(win);
                                        if (down_obj == sel_obj)

                                /* user wanted to unselect */
                                sel_obj = NO_OBJ;
                        else
                                /* select new object */
                                sel_obj = down_obj;
                        /* show new selection, if any */
                        invert_selection(win);
                }
                down_obj = NO_OBJ;
        }
}

long XVT_CALLCONV1 win_eh(WINDOW win, EVENT * ep)
{
        switch (ep->type) {
        case E_MOUSE_DOWN:
        case E_MOUSE_UP:
                do_mouse(win, ep->type, ep->v.mouse.where);
                break;
        ...
}
```

See Also: For more details about E_MOUSE_UP events, see section 4.5.14 on page 4-52.

4.5.13. E_MOUSE_MOVE Events

Partial Event Structure

```
typedef struct {
      EVENT_TYPE type;                  /* E_MOUSE_MOVE */
      union {
            ...
            struct s_mouse {
                  PNT where;            /* location of event */
                  BOOLEAN shift;        /* Shift key down? */
                  BOOLEAN control;      /* Control key down? */
                  short button;         /* button number */
                  XVT_INT32 scroll_x;   /* scroll delta in the X axis -
                                           E_MOUSE_SCROLL only */
                  XVT_INT32 scroll_y;   /* scroll delta in the Y axis -
                                           E_MOUSE_SCROLL only */
            } mouse;
            ...
      } v;
} EVENT;
```

XVT sends an E_MOUSE_MOVE event to a window'sevent handler in response to the user moving the mouse pointer in the client area of an XVT window.

In XVT, dialogs do not send mouse events to their event handlers. The native platform handles mouse events in dialogs. The location of the mouse cursor in window-relative coordinates is stored in the where member. Thev.mouse.shift, v.mouse.control, and v.mouse.button fields are not valid for this event.

Dragging

XVT generates E_MOUSE_MOVE events whether the mouse button is down or not. Most applications take action on mouse movements only when the button is down (that is, only when E_MOUSE_MOVE events occur between E_MOUSE_DOWN and E_MOUSE_UP events). This is called dragging.

Tip: To ensure that your application gets the E_MOUSE_UP event:

> Trap the mouse with a call to xvt_win_trap_pointer and release it with a call to xvt_win_release_pointer.

XVT generates E_MOUSE_MOVE events as often as the mouse is moved, but you can't rely on their frequency or regularity because other tasks may be going on concurrently.

To help you implement auto-scrolling, E_MOUSE_MOVE events are generated continuously when the mouse is trapped, even if it isn't physically moved.

Because E_MOUSE_MOVE events are passed to a window's event handler whether your application does anything with them or not, make sure that you don't waste time before returning when your event handler gets an event that it ignores. In addition, because of the extra processing overhead associated with XVT's sending the mouse move events to the event handler each time, it is often desirable to have XVT not send the event to an event handler at all, especially if you are ignoring them.

Tip: You might want to use XVT's event masking feature to specify that, for certain windows, events should not be sent to their event handlers.

See Also: For more information about trapping the mouse, see Chapter 14, *Cursors and Carets*.
For information about event masking, see section 4.4.2 on page 4-14.

E_MOUSE_MOVE Example

The following code fragments demonstrate what happens during mouse move events:

- Allowing the user to specify the size of an object to be created by dragging out a rubberband rectangle
- Transforming the rectangle
- Drawing the rubberband rectangle

Specifying the Size of the Rectangle

```
static void do_mouse(WINDOW win, EVENT * ep)
{
	static PNT last_pnt;
	static enum {DR_OUTLINE, DR_NONE}
				drag_type = DR_NONE;
	static BOOLEAN first_move;
	static RCT outline;

	switch (ep->type) {
	case E_MOUSE_DOWN:
		drag_type = DR_OUTLINE;
		outline.left = ep->v.mouse.where.h;
		outline.top = ep->v.mouse.where.v;
		last_pnt = ep->v.mouse.where;
		xvt_win_trap_pointer(win);
		first_move = TRUE;
		break;
	case E_MOUSE_MOVE:
		if (drag_type == DR_NONE)
			return;
		if (last_pnt.h == ep->v.mouse.where.h &&
		 last_pnt.v == ep->v.mouse.where.v)
			return(TRUE);/* didn't really move */
		if (!first_move)
			/* erase old rect */
			rubber_rect(win, &outline);
		first_move = FALSE;
		outline.right = ep->v.mouse.where.h;
		outline.bottom = ep->v.mouse.where.v;
		last_pnt = ep->v.mouse.where;
			/* draw new rect */
		rubber_rect(win, &outline);
		break;
	case E_MOUSE_UP:
		if (drag_type == DR_NONE)
			return;
		xvt_win_release_pointer();
		if (!first_move)
			/* erase rubber rect */
			rubber_rect(win, &outline);
		normalize_rect(&outline, &outline);
		create_object(win, &outline);
		drag_type = DR_NONE;
		break;
	}
}
```

```
long XVT_CALLCONV1 win_eh(WINDOW win, EVENT * ep)
{
        switch (ep->type) {
        ...
        case E_MOUSE_DOWN:
        case E_MOUSE_UP:
        case E_MOUSE_MOVE:
                do_mouse(win, ep);
                break;
        ...
        }
}
```

The event handler sends mouse-related events to do_mouse. The variable drag_type indicates whether dragging is in effect. It is set on an E_MOUSE_DOWN and cleared on an E_MOUSE_UP. Each E_MOUSE_MOVE event erases the previous rubberband rectangle, if there was one; the variable first_move determines this. The code then draws a rectangle having the new dimensions.

During the E_MOUSE_DOWN, the program traps the mouse with a call to xvt_win_trap_pointer, which guarantees it will get the E_MOUSE_UP. When the E_MOUSE_UP is retrieved, the code erases the rubberband rectangle, releases the mouse, and creates an object having the bounding rectangle that the user specified by dragging the rubberband.

Note: The state variables drag_type, last_pnt, first_move, and outline are declared to be static, because their values must persist between calls to do_mouse. Alternatively, you could store them in a data structure associated with the window, using the function xvt_vobj_set_data.

Transforming the Rectangle

The function normalize_rect (shown below) transforms a rectangle to ensure that its top is not greater than its bottom, and its left is not greater than its right. This is necessary because the user can move the mouse in any direction after pressing the button; that is, the starting and ending points of the dragging operation can be anywhere relative to each other.

```
static void normalize_rect(RCT * norm_rctp, RCT * rctp)
{
        xvt_rect_set(norm_rctp, min(rctp->left,
                                rctp->right),
         min(rctp->top, rctp->bottom),
         max(rctp->left, rctp->right),
         max(rctp->top, rctp->bottom));
}
```

Drawing the Rubberband Rectangle

The function rubber_rect draws a rubberband rectangle, using a PAT_HOLLOW rectangle, a PAT_RUBBER pen, and the function xvt_dwin_draw_rect:

```
static void rubber_rect(WINDOW win, RCT * rctp)
{
	RCT rct;
	DRAW_CTOOLS t;

	xvt_app_get_default_ctools(&t);
	t.pen.pat = PAT_RUBBER;
	t.brush.pat = PAT_HOLLOW;
	t.mode = M_XOR;
	xvt_dwin_set_draw_ctools(win, &t);
	normalize_rect(&rct, rctp);
	xvt_dwin_draw_rect(win, &rct);
}
```

4.5.14. E_MOUSE_SCROLL Events

Partial Event Structure

```
typedef struct {
	EVENT_TYPE type;			/* E_MOUSE_SCROLL */
	union {
		...
		struct s_mouse {
			PNT where;		/* location of event */
			BOOLEAN shift;		/* Shift key down? */
			BOOLEAN control;	/* Control key down? */
			short button;		/* button number */
			XVT_INT32 scroll_x;	/* scroll delta in the X axis -
						E_MOUSE_SCROLL only */
			XVT_INT32 scroll_y;	/* scroll delta in the Y axis -
						E_MOUSE_SCROLL only */
		} mouse;
		...
	} v;
} EVENT;
```

XVT sends an E_MOUSE_SCROLL event to a window's event handler in response to the user operating a mouse scroll wheel while the mousepointer is in the client area of the window.

In XVT, dialogs do not send mouse events to their handlers. The location of the mouse cursor in window-relative coordinates is stored in the where member. The states of the Shift and Control keys are stored in the shift and control members. The button member is not valid for this event. The vertical scroll delta is stored in the scroll_y member. A positive value indicates that the scroll wheel was rotated forward, away from the user; a negative value indicates that the scroll wheel was rotated backward, toward the user. The horizontal scroll delta is stored in the scroll_x member. A positive value indicates that the scroll wheel was rotated left, counter-clockwise to the user; a negative value indicates that the scroll wheel was rotated right, clock-wise to the user.

Special note for X-Motif users; X handles mouse scroll events as mouse button presses. Depending upon the platforms the mouse driver needs to be properly configured to map scroll wheel motion to a button number. XVT, using the defaults, maps button 4 to a positive scroll_y value; button 5 to a negative scroll_y value; button 6 to a positive_x value; and button 7 to a negative scroll_x value. Mouse driver configuration information, not all protocols are supported on all platforms or by all mouse hardware.

Option “XAxisMapping” “N1 N2”

Specifies which buttons are mapped to motion in the X direction in wheel emulation mode. Button number N1 is mapped to the negative X axis motion and button number N2 is mapped to the positive X axis motion. Default: no mapping.

Option “YAxisMapping” “N1 N2”

Specifies which buttons are mapped to motion in the Y direction in wheel emulation mode. Button number N1 is mapped to the negative Y axis motion and button number N2 is mapped to the positive Y axis motion. Default: “4 5”.

Option “ZAxisMapping” “X”

Option “ZAxisMapping” “Y”

Option “ZAxisMapping” “N1 N2”

Option “ZAxisMapping” “N1 N2 N3 N4”

Set the mapping for the Z axis (wheel) motion to buttons or another axis (**X** or **Y**). Button number N1 is mapped to the negative Z axis motion and button number N2 is mapped to the positive Z axis motion. For mice with two wheels, four button numbers can be specified, with the negative and the positive motion of the second wheel mapped respectively to buttons number N3 and N4. Default: “4 5 6 7”.

4.5.15. E_MOUSE_UP Events

Partial Event Structure

```
typedef struct {
      EVENT_TYPE type;                  /* E_MOUSE_UP */
      union {
            ...
            struct s_mouse {
                  PNT where;            /* location of event */
                  BOOLEAN shift;        /* Shift key down? */
                  BOOLEAN control;      /* Control key down? */
                  short button;         /* button number */
                  XVT_INT32 scroll_x;   /* scroll delta in the X axis -
                                           E_MOUSE_SCROLL only */
                  XVT_INT32 scroll_y;   /* scroll delta in the Y axis -
                                           E_MOUSE_SCROLL only */
            } mouse;
            ...
      } v;
} EVENT;
```

XVT sends an E_MOUSE_UP event to a window's event handler in response to the user releasing a mouse button while the mouse pointer is in the client area of the window.

In XVT, dialogs do not send mouse events to their event handlers. The location of the mouse cursor in window-relative coordinates is stored in the where member. The states of the Shift and Control keys are stored in the shift and control members. The button member specifies the button; it can have the values 0, 1, or 2, but only 0 and 1 are portable across all platforms.

An E_MOUSE_UP event is not necessarily preceded by an E_MOUSE_DOWN or E_MOUSE_DBL event because those button actions might have taken place outside of the XVT window. Therefore, you usually want to code your application so that spurious E_MOUSE_UP events are ignored.

Similarly, an E_MOUSE_UP event isn't necessarily generated every time an E_MOUSE_DOWN or E_MOUSE_DBL event is because the button might have been released outside of the window. To ensure that this doesn't happen, you can trap the mouse with a call to xvt_win_trap_pointer.

Example: See the examples in sections 4.5.11, 4.5.12, and 4.5.13.

4.5.16. E_QUIT Events

Partial Event Structure

```
typedef struct {
    EVENT_TYPE type;                /* E_QUIT */
    union {
        ...
        BOOLEAN query;              /* query only? */
        ...
    } v;
} EVENT;
```

XVT sends an E_QUIT event to the task event handler to notify your application that the user has initiated a system shutdown. Only task windows can send these events to their event handlers.

Implementation Note: Only XVT/Win32 generates this event.

If your application has a **Quit** or **Exit** item on a menu (File, usually), this event isn't generated when the user chooses that item—a normal E_COMMAND event is generated instead. E_QUIT is reserved for those cases where the native operating system can tell applications that the system is performing a system-wide shutdown; it is not an event that the user can directly generate.

See the Implementation Note later in this section for more detailed information on how to use E_QUIT in a portable manner.

Types of E_QUIT Events

The following are the two types of E_QUIT events:

Member query is TRUE

The application should not actually quit (by calling xvt_app_destroy), but should prepare to quit. Usually, if there are any unsaved documents, the application will have to query the user about each via a dialog box containing three buttons: Save, Discard, and Cancel. The application should do the following in each case:

Save

Save the document and, if that is successful, close the window and go on to the next document's save dialog.

Discard

Don't save the document, but just close the window and go on to the next document's save dialog.

Cancel
: Return from the event handler without showing any more save dialogs. XVT understands that quitting is not okay.

After the user has been queried about every unsaved document, and has not clicked the Cancel button for any of them, the application should call xvt_app_allow_quit and then return from the task event handler to tell XVT that the application is willing to quit. However, it shouldn't quit (by calling xvt_app_destroy) because other applications also may have to be queried, and one of them might decline to quit.

Member query is FALSE
: The application was previously queried with an E_QUIT event with query set to TRUE, and it has already determined that quitting is okay, and told XVT about it by calling xvt_app_allow_quit. It should immediately call xvt_app_destroy. No documents have to be saved because they were taken care of earlier.

Instead of generating this event (query being FALSE), XVT might instead simply call xvt_app_destroy itself (perhaps from the E_CLOSE case of a task event handler). Therefore, don't put any cleanup code in your case for E_QUIT. Do all cleaning up in your task event handler in response to the E_DESTROY event.

If your task event handler receives an E_COMMAND event for tag M_FILE_QUIT, your application should take action similar to case query TRUE, except that, instead of calling xvt_app_allow_quit, it should just call xvt_app_destroy. If the user clicked Cancel for any save dialog, don't call xvt_app_destroy.

Remember that E_QUIT is different from E_CLOSE. An E_QUIT event is not sent to the application when the user attempts to close the task window, or any other window or dialog. In those cases, E_CLOSE events are sent, and the event handlers can choose either to call xvt_vobj_destroy or ignore the event. E_QUIT events are only generated on those systems which can notify applications of a system-wide shutdown.

Implementation Note: E_QUIT events never occur on XVT/Mac or XVT/XM. As a result, code that handles them can be debugged only on XVT/Win32, a query-only E_QUIT is generated when the user closes the windowing system. If an application does not call xvt_app_allow_quit, then the attempt to shut down is cancelled. XVT applications don't have to do anything special about quitting under the Mac MultiFinder, as long as they handle the Quit item on the File menu in the proper way.

E_QUIT Example

In the following code fragments, when a query-only E_QUIT event occurs, or when the user chooses Quit (or Exit) from the File menu, the function quit_approved is called to determine if quitting is okay. If it is, then the XVT function xvt_app_allow_quit is called in the first case, and xvt_app_destroy is called in the second case.

```
static void do_menu(WINDOW win, MENU_TAG cmd,
        BOOLEAN shift, BOOLEAN control)
{
        ...
        switch (cmd) {
        ...
        case M_FILE_QUIT:
                if (quit_approved())
                        xvt_app_destroy();
                break;
        ...
        }
}
long XVT_CALLCONV1 task_window_eh(WINDOW win, EVENT * ep)
{
        ...
        switch (ep->type) {
        ...
        case E_COMMAND:
                do_menu(win, ep->v.cmd.tag, ep->v.cmd.shift,
                        ep->v.cmd.control);
                break;
        ...
        case E_QUIT:
                if (ep->v.query) {
                        if (quit_approved())
                                xvt_app_allow_quit();
                }
                else
                        xvt_app_destroy();
                break;
        ...
        }
}
```

In a real application that has documents associated with windows, quit_approved would ask the user about each unsaved document. But in this simplified example, it just asks the user whether changes should be saved.

If the user agrees, the actual saving is done by the function save_document (not shown here), which returns TRUE if the saving goes okay and FALSE if it fails. In the case of failure, quit_approved then returns FALSE, just as it does when the user clicks the Cancel button, and the quitting procedure is aborted. If saving succeeds, or if the

user elects to discard changes, the window is closed with a call to discard_window (also not shown) and quit_approved returns TRUE.

```
static BOOLEAN quit_approved(void)
{
      switch (xvt_dm_post_ask("Save", "Discard", "Cancel",
       "Save changes?")) {
      case RESP_DEFAULT:
            if (!save_document())            /* try to save document */
                  return(FALSE);
                                             /* fall through */
      case RESP_2:
            discard_window();                /* discard window */
            return(TRUE);
      case RESP_3:
            return(FALSE);                   /* abort quit process*/
      }
}
```

4.5.17. E_SIZE Events

Partial Event Structure

```
typedef struct {
      EVENT_TYPE type;                       /* E_SIZE */
      union {
            ...
            struct s_size {
                  short height;              /* new height */
                  short width;               /* new width */
            } size;
            ...
      } v;
} EVENT;
```

The event handler for a window, dialog, or task window receives an E_SIZE event for any of the following reasons:

- XVT sends an E_SIZE event—indicating the initial size—to the event handler for a window, dialog, or task window immediately following an E_CREATE event. Recall that the E_CREATE event is sent to notify your application that the window, dialog, or task window has been successfully created. Note that, on some platforms, performing certain operations during a window's E_CREATE (such as creating a dialog) might cause an E_SIZE event to be delivered to the window before the completion of the E_CREATE callback.
- XVT sends an E_SIZE event to the event handler for a window when the user resizes the window using the border decorations. Only windows with the WSF_SIZE or

WSF_DECORATED attribute set at creation time can have window resizing border decorations. On XVT/Win32, an E_SIZE might also be sent to the task window in response to a border decoration resize.

- XVT sends an E_SIZE event to a window or dialog event handler as a result of your application calling xvt_vobj_move. The height and width fields give the new dimensions of the client area.

Responding to E_SIZE Events

When your application gets an E_SIZE event, you don't have to redraw anything; XVT generates a separate E_UPDATE if necessary. However, you should adjust anything dependent on the size of the client area upon receipt of an E_SIZE event, such as the range or proportion of the scrollbars or the scale of a picture. If the effect of increasing or decreasing the size of the window is merely to show less (or more) of what is in the client area, then you don't have to do anything else when an E_SIZE event occurs; XVT automatically clips anything drawn in the window to fit the new size.

If your application wants to change the scale of a picture when the size of the window changes, you should make any necessary adjustments to your data structures and call xvt_dwin_invalidate_rect. This will generate one or more E_UPDATE events, which will take care of redrawing the window contents to the new scale.

E_SIZE Example

In the following code fragments, the scrollbar proportions are adjusted when the window is resized.

```
static void scroll_sync(WINDOW win, int height,
        int width)
{
        DOC * d;

        d = get_doc_data(win);

        xvt_sbar_set_range(win, VSCROLL, 0,
         d->nlines + height / d->line_height);
        xvt_sbar_set_range(win, HSCROLL, 0,
                        d->maxwidth + width);

        xvt_sbar_set_proportion(win, VSCROLL,
         height / d->line_height);
        xvt_sbar_set_proportion(win, HSCROLL, width);

        xvt_sbar_set_pos(win, VSCROLL,
                        d->org.v / d->line_height);
        xvt_sbar_set_pos(win, HSCROLL, d->org.h);
}


long XVT_CALLCONV1 win_eh(WINDOW win, EVENT * ep)
{
        switch (ep->type) {
        ...
        case E_SIZE:
                scroll_sync(win, ep->v.size.height,
                 ep->v.size.width);
                break;
        ...
        }
}
```

See Also: For more details about E_HSCROLL and E_VSCROLL events, see section 4.5.10 on page 4-38.

4.5.18. E_TIMER Events

Partial Event Structure

```
typedef struct {
        EVENT_TYPE type;                    /* E_TIMER */
        union {
                ...
                struct s_timer {
                        long id;            /* timer ID */
                } timer;
                ...
                } v;
} EVENT;
```

XVT sends an E_TIMER event to the event handler of a window, dialog, or task window to notify your application that a specified time interval has elapsed.

The application program establishes timers on a per-window or per-dialog basis. Once a timer is set, E_TIMER events are sentto the event handler of the specified window or dialog at a specified regular time interval. The task window can also have timers.

Using Timers

Tip: To set a timer with a millisecond interval as an argument:

Call xvt_timer_create.

Tip: To turn off a timer:

Call xvt_timer_destroy.

When a window is closed via xvt_vobj_destroy, then all timers for the window are killed automatically. It is unnecessary and invalid to call xvt_timer_destroy to kill the timer for a window when it gets an E_DESTROY.

An application can have more than one timer; the maximum number is implementation-dependent. The function xvt_vobj_get_attr and attribute ATTR_NUM_TIMERS can be used to find out how many timers are available. However, the value returned may be incorrect on systems where timers are a resource shared between multiple applications. To deal with this problem, check the return code from xvt_timer_create to find out if a timer was available at the time when the function call was made.

XVT provides at least one timer per window or dialog, even for those platforms that only support a single timer. However, this may

not be true if a platform has a finite number of timers shared between multiple applications.

When writing applications that depend on timers with a certain interval, make sure that the interval is achievable on all platforms where the application must run.

An application must not depend on any minimum latency between the theoretical timeout and the actual E_TIMER delivery. For non-preemptive systems, this delay can be arbitrarily long if another application is active. XVT guarantees that timers send events *no sooner than specified*.

Tip: Common uses for XVT timers are timed demos, connection timeouts, polling of input devices, and crude animation. Timers are not useful for situations demanding a high degree of precision.

E_TIMER Example

The following code sets a timer when a window is created. E_TIMER events are then received sometime after the passage of 1000 milliseconds, and cause a status bar to be updated. Additional E_TIMER events continue to be generated at 1000-millisecond intervals until the window or the timer is destroyed.

```
long XVT_CALLCONV1 a_window_eh(WINDOW win, EVENT * ep)
{
	switch (ep->type) {
	case E_CREATE:
		xvt_timer_create(win, 1000);
		break;
	case E_TIMER:
		update_status_bar(win);
		break;
	}
}
```

4.5.19. E_UPDATE Events

Partial Event Structure

```
typedef struct {
      EVENT_TYPE type;                  /* E_UPDATE */
      union {
            ...
            struct s_update {
                  RCT rct;              /* update rectangle */
            } update;
            ...
      } v;
} EVENT;
```

XVT sends an E_UPDATE event to the event handler for an XVT window when part or all of the client area of the window needs to be redrawn. This event might be generated as the result of window creation, a change in the stacking order of the windows, user manipulation of the window, or by functions such as xvt_dwin_scroll_rect and xvt_dwin_invalidate_rect.
Pending E_UPDATE events can also be expedited by xvt_dwin_update.

Note: E_UPDATE events are only generated to regular window event handlers; dialog event handlers do not receive them.

The rct field gives the bounding rectangle of the area that needs tobe drawn, in window-relative coordinates.

Don't assume that only one E_UPDATE event is generated when a window needs to be redrawn. XVT may send several E_UPDATE events for different areas of the window, or may combine the areas into a single bounding rectangle. Also, you can't make any assumptions about when E_UPDATE events will occur; they may occur anytime after the receipt of the window's E_CREATE event.

Drawing and E_UPDATE Events

In response to an E_UPDATE event, you should at least draw the part that needs updating. If you draw more than that, XVT may, for efficiency, temporarily reduce the clipping area so that only the part that needs updating can actually be drawn.

Tip: It is usually best to organize your application so that most, if not all, drawing occurs in response to E_UPDATE events, rather thandrawing things as you progress. That way the occurrence of an update event will be the usual case rather than the exception, and the program is likely to be simpler and more reliable. When the data structure representing the contents of a window changes, don't draw the

changes immediately. Instead, after making changes to the data structure, induce an update event with xvt_dwin_invalidate_rect.

A newly created visible, regular window (non-dialog, non-pixmap, and non-print) always gets an E_UPDATE event for its entire client area shortly after being created, so it is not necessary to draw into a new window.

Inducing E_UPDATEs

Don't induce an E_UPDATE event when it's important to draw right away, to keep up with the user or to show animation. For example, when the user selects an object with the mouse, immediately draw whatever is required to show the selection; waiting for the E_UPDATE event may cause a noticeable delay.

In addition, don't induce an E_UPDATE event when the user operates a scrollbar. The window will scroll much faster if you move some pixels already there with a call to xvt_dwin_scroll_rect, rather than repainting the entire window.

During E_UPDATE, do not call xvt_app_process_pending_events.

Updating Rectangles

When you are calling xvt_dwin_invalidate_rect several times to invalidate disjoint areas of the window, it may be advantageous to call xvt_app_process_pending_events between calls to xvt_dwin_invalidate_rect. This allows each update rectangle to be handled individually. Otherwise, the several disjoint update rectangles may be merged into a single rectangle, causing your application to update more of the screen than is needed. If you do this, take into account that there will be a recursive call to your window's event handler.

Caution: Many XVT functions cannot be called during the processing of an E_UPDATE event—calling them causes XVT to issue an error. For example, do not call xvt_app_process_pending_events. For more information and a complete list of functions that you cannot call during an E_UPDATE event, see section 4.3.3 on page 4-10.

E_UPDATE Example

Example: In the following code fragments, the function do_update is called when an E_UPDATE event is received:

```
static void do_update(WINDOW win, RCT rct)
{
        DRAW_CTOOLS t;
        DOC * d;
        int i, ascent, descent, y, bottom;

        xvt_dwin_get_draw_ctools(win, &t);
        xvt_dwin_clear(win, t.backcolor);

        d = get_doc_data(win);
        xvt_dwin_set_font(win, d->font_id);
        xvt_dwin_get_font_metrics(win, NULL, &ascent,
                                &descent);
        bottom = rct.bottom + d->line_height;

        /* start with first visible line */
        for (i = d->org.v / d->line_height,
                                y = rct.top + ascent;
                i < d->nlines && y < bottom;
                i++, y += d->line_height) {
                rct.top = y - ascent;
                rct.bottom = y + descent;
                if (xvt_dwin_is_update_needed(win, &rct)) {
                        xvt_dwin_draw_text(win, MGN - d->org.h, y,
                         d->lines[i], -1);
                }
        }
}

long XVT_CALLCONV1 win_eh(WINDOW win, EVENT * ep)
{
        RCT rct;

        switch (ep->type) {
        ...
        case E_UPDATE:
                xvt_vobj_get_client_rect(win, &rct);
                do_update(win, rct);
                break;
        ...
        }
}
```

This code is typical for updating a window containing text. First, it clears the window by filling it with its current background color. The loop then draws each line that is part of the update rectangle, using xvt_dwin_is_update_needed to check. The variable i refers to the data for each line, while y keeps track of each line's location. The

process stops either when it reaches the end of the data, or when it moves off the bottom of the client area.

4.5.20. E_USER Events

Partial Event Structure

```
typedef struct {
      EVENT_TYPE type;                  /* E_USER */
      union {
            ...
            struct s_user {
                  long id;              /* application ID */
                  void *ptr;            /* application pointer */
            } user;
            ...
      } v;
} EVENT;
```

XVT does not generate an E_USER event. The E_USER event lets you pass custom events to window or dialog event handlers. It is used in conjunction with xvt_win_dispatch_event, which sends E_USER events (or any other XVT event that you might wish to synthesize), in a non-queued fashion, to an event handler. (The application can also call an event handler directly, but XVT recommends that you use xvt_win_dispatch_event.)

You can distinguish various kinds of E_USER events by giving them unique IDs, the administration of which is up to your application. The structure for an E_USER event also contains a member for a pointer, which you can use for whatever purpose you want.

The range of legal values for the id field is 0 to 32767. All values greater than 32767 are reserved for future XVT use.

xvt_win_dispatch_event directly (synchronously) sends an event to an event handler, and then returns when the event handler has finished processing the event. XVT does not support an event queue.

5

RESOURCES AND XRC

This chapter explains how to use XVT's XVT Resource Compiler (XRC) to specify resources for menus, dialogs, windows, and strings. You can then use the **xrc** compiler to translate your XRC specification into a resource script or binary file that you can use on a particular XVT platform.

You can create XRC resources in XVT-Design without having to write XRC code. XVT-Design can generate controls, menus, windows, dialogs and other resource objects, along with their XRC file. If you use XVT-Design, you won't need most of the information in this chapter, except as a general background.

See Also: For information about using the **xrc** tool and for detailed descriptions of the XRC resource language, refer to the *XVT Portability Toolkit Reference*.
For information about dynamically binding resources to a multibyte-aware (internationalized) application, see sections 19.2.7 and 19.4.7 in Chapter 19, *Multibyte Character Sets and Localization*.

5.1. Resources

Resources are specifications for menus, dialogs, windows, controls, strings, bitmap images, and fonts that are kept in a small, read-only database located outside your application's runtime address space. When your application needs a resource, the application requests the resource by an ID number (referred to as the resource ID or RID). XVT or the native window system then reads the resource into memory so it can be accessed. This saves space at runtime and makes it possible to construct resources without recompiling your C programs.

Example: For example, consider this XVT call. It accesses a dialog resource by ID:

```
xvt_dlg_create_res(WD_MODAL, 101, EM_ALL,
    eventhandler_fcn, 0L);
```

In this example, the xvt_dlg_create_res function searches the resources for a dialog specification numbered 101, which specifies the size of the dialog, the type and position of the controls, the labels on the controls, and so on. It then displays the dialog on the screen.

5.1.1. Predefined Resources

XVT supplies some predefined resources whose resource IDs are above 30000. XVT reserved the RIDs above 30000 for its own use. Any resources you create in your application must be positive numbers less than 30000.

Note: XVT supplies some predefined dialog IDs in the range 9050 to 9099.

See Also: For more information on XVT's predefined dialogs, see section 7.3 on page 7-6.

5.1.2. Other System-Specific Resources

There are usually other system-specific resources. These are needed by the window system, rather than by XVT itself, and should be in the resource database as well. For example, Macintosh applications need a "bundle" resource, and MS-Windows applications need icon resources.

5.1.3. Binary Resources

The resources that are accessed at runtime are called *binary* resources. They may be located in a separate file from the executable application, they may be bound into the executable file, or they may be in a special part of the file (on the Mac, the resource fork).

See Also: For details on system-specific resources, see your window system documents.

5.2. Portable Resource Concepts

5.2.1. Creating Portable Resources with XRC

Each platform has its own native resource language for describing resources in a text file. XVT provides a XVT Resource Compiler (XRC) that lets you write resources for menus, dialogs, windows, strings, images, and fonts. XVT's XRC compiler (**xrc**) translates your XRC specification into the native resource language. By using XRC, you don't have to learn a new native resource language whenever you port to a new platform.

See Also: For information about describing native resources, see the *XVT Platform-Specific Books*.

Figure 5.1 shows two ways to create portable resources. You can manually code the resources into a text file in XRC. Or you can develop the resources with XVT-Design™, an XVT utility product that interactively produces XRC.

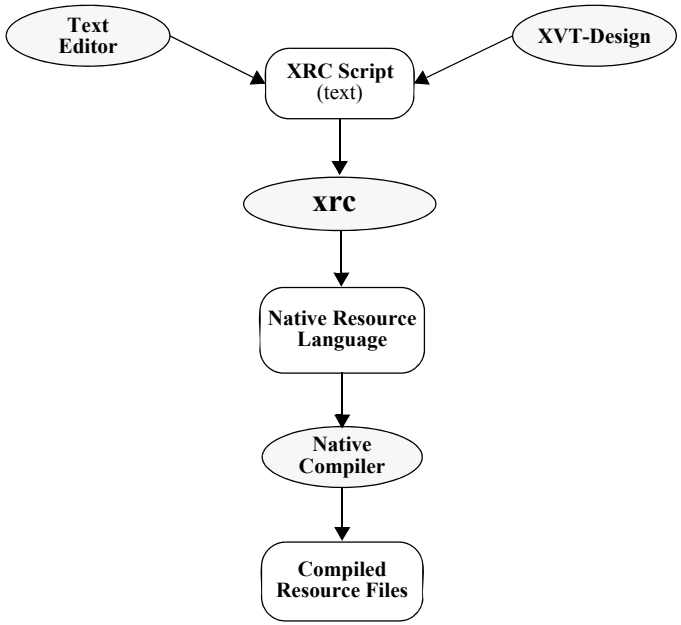


*Figure 5.1. Building resources with **xrc***

You compile the XRC script with **xrc** (for any of the XVT platforms) using a platform-specific resource compiler.

XVT-Design was developed to simplify the creation of resources. It allows you to directly generate controls, menus, windows, dialogs and the other resource objects as they will appear on the user's screen. When you are satisfied with their placement and appearance, XVT-Design generates a XRC file that works with XVT, as well as a C Language framework for your application's user interface.

See Also: For more information about platform-specific resource compilers, see the *XVT Platform-Specific Books*.

5.2.2. General Rules for Coding Resources

This section lists the rules that you must follow when coding resources for XVT applications, whether in XRC or in a native resource language:

- The standard XVT resources for menus, dialogs, windows, strings, etc., must be present in all applications. The files **xrc.h** and **xrc_plat.h** define these resources. If you're using xrc, you get them automatically when you #include the **xrc.h** file in your application XRC file. If not, you'll have to extract them from **xrc.h** and **xrc_plat.h**. Do this only if you're experienced in using XRC and the native resource compiler.
- Don't assume that you can use native resources already coded for an existing application when you recode it for XVT. This will usually work, but may require special handling of non-XVT features.
- If a window or dialog has a navigation object installed, observe the numbering requirements for the Default and Cancel buttons. The default button ID must be DLG_OK, and the cancel button must have a resource ID of DLG_CANCEL.
- Make the base name of the XRC file (the part before the **.xrc** suffix) the same as the base_appl_name field of the XVT_CONFIG structure passed to the XVT function xvt_app_create.

Note: While all the rules are important, the first rule listed above is vital.

XVT-Design automatically follows all resource-coding rules.

See Also: For more information on DLG_* control IDs, see the *XVT Portability Toolkit Reference*.

5.2.3. Resources for Internationalized Applications

When writing an international XVT application, resources become an integral aspect of the application design and your software development process. For example, when running **xrc**, you will need to make sure that the correct header files for your locale are available and that you have defined a LANG_* constant in the source file, or on the command line, to include the localized header file version for your target locale.

See Also: For more detailed information about how to provide resources for international XVT applications, see sections 19.2.7 and 19.4.7 in Chapter 19, *Multibyte Character Sets and Localization*.

5.2.4. XVT Coordinate Units for Resources

In XRC, and also in WIN_DEF arrays, XVT coordinate units describe the position and dimensions of windows, dialogs, and controls. XVT defines three types of coordinate units:

- pixels
- chars
- semichars

The semantics differ for pixels versus chars and semichars.

See Also: For more information about these coordinate units, see the units XRC statement in the *XVT Portability Toolkit Reference*.

5.2.4.1. Pixels

When you generate XRC with units of pixels (or an array of WIN_DEF structures with a UNIT_TYPE of U_PIXELS), then XVT interprets all specified sizes and locations as exact pixel values.

Example: If you specify a push button to be 24 pixels high, then the push button will be 24 pixels high, regardless of the platform or system font. If the system font happens to be 16 pixels high, then 24 is a good value for a push button. However, if the system font is 26 pixels high, then 24 produces a badly scaled push button.

See Also: For more information on pixel coordinates and drawing, see Chapter 10, *Coordinate Systems*, and Chapter 11, *Drawing and Pictures*.

5.2.4.2. Chars and Semichars

By using chars or semichars (UNIT_TYPE of U_CHARS or U_SEMICHARS), you can specify "device-independent" sizes for objects. XVT defines semichars and chars relative to the size of the system font on any given platform:

- A char is similar to a semichar, except that a char is the same height as the system font and is the width of an average character in the system font
- A semichar is 1/8 the height of the system font and 1/4 the width of an average character in the system font

Example: If you specify a push button to be 12 semichars high (i.e., 1 1/2 times the height of the system font), then on the platform with the 16-pixel system font, the push button will be 24 pixels high, and on the platform with the 26-pixel system font, the push button will be 39 pixels high.

5.2.4.3. Scaling Controls and Dialogs

Specifying controls in semichars or chars scales the controls to keep the same amount of space around the label. You should specify geometry for sizes (width, height) as well as for coordinates (x, y) in the same coordinate units. In a dialog, this scales the entire dialog correctly so that controls won't grow or shrink, or run into or away from each other.

5.2.5. Formatting GUI Objects for Different Platforms

XVT-Design generates macros for "tweaking" the size and spacing of dialogs, windows, and controls. The XRC files that XVT-Design creates define all dialogs, windows, and controls in terms of the XRC_RECT macro. This section discusses this macro and tells how to override it.

The XRC_RECT macro changes the size and spacing of rectangles created in XRC files. XVT defines the XRC_RECT macro in terms of the macros XRC_DEST_WIDTH, XRC_DEST_HEIGHT, XRC_SRC_WIDTH, and XRC_SRC_HEIGHT, as follows:

```
scaled_x = x * XRC_DEST_WIDTH / XRC_SRC_WIDTH
scaled_y = y * XRC_DEST_HEIGHT / XRC_SRC_HEIGHT
scaled_w = w * XRC_DEST_WIDTH / XRC_SRC_WIDTH
scaled_h = h * XRC_DEST_HEIGHT / XRC_SRC_HEIGHT
```

XRC_RECT automatically uses these macros to change the size and spacing of XRC rectangles. They define the height and width of the "system font" on the "source" platform (where the XRC files were generated) and the "destination" platform (where the generated application is built). These macros allow the GUI objects defined in the XRC file to be scaled proportionally to the system font to ensure that they have an acceptable appearance.

Tip: Even if you are not using XVT-Design-generated XRC files, you might want to copy these macros from an XVT-Design-generated XRC file into your XRC file and use XRC_RECT for windows, dialogs, and controls.

5.2.5.1. Overriding the Macros

XVT-Design always gives usable default values to the XRC_SRC_* and XRC_DEST_* macros. However, if you want to override these macros to resize your XVT-Design generated application, you can reset them before the XRC_RECT macro definition, at the top of the XRC file created by XVT-Design, or by specifying them explicitly on the xrc execution line of the application's makefile.

If the controls, windows, and dialogs have the wrong size or wrong spacing on your destination platform, you can change either or both of the XRC_SRC_* and XRC_DEST_* macros. Increasing the XRC_SRC_* macro values results in smaller, more closely spaced controls, and decreasing them results in larger, more widely spaced controls.

Changing the XRC_DEST_* macro values has the reverse effect. Decreasing them produces smaller, more closely spaced controls, and increasing them produces larger, more widely spaced controls.

Implementation Note: If the destination platform is Win32, you should change only the XRC_SRC_* macro values, since the XRC_DEST_* macros are in terms of
chars or semichars. On other platforms, you should define values for both the XRC_SRC_* and XRC_DEST_* macros, since by default the XRC_DEST_* values are set to those of the corresponding XRC_SRC_* values, and no change in spacing takes effect.

Example: On XVT/XM, placing the following lines at the start of your XRC file before XRC_RECT is defined causes controls to appear larger and more widely spaced:

```
#define     XRC_SRC_WIDTH      8
#define     XRC_SRC_HEIGHT     16
#define     XRC_DEST_WIDTH     12
#define     XRC_DEST_HEIGHT    24
```

5.3. XRC Language Specification

The XRC language adheres to these guidelines:

- XRC scripts generally follow the same lexical rules as C.
- Identifiers must start with a letter. They can consist of up to 31 letters, numbers, and underscores. Keywords are case-insensitive, but #defined identifiers are case-sensitive.
- Comments begin with /* and end with */ (C style comments), or begin with // and terminate at the end of the line (C++ style comments).
- A backslash at the end of a line indicates that the line is continued onto another line. The backslash and the line-ending characters are ignored. A backslash can occur in the middle of a string, or even in the middle of a token.
- Strings must be surrounded by double quotes (""). A double quote or a backslash appearing in a string must be preceded by a backslash.
- Two sequential strings are concatenated together to form a single string. For example, "ABC" "DEF" is the same as "ABCDEF". The two strings can be separated by any amount of whitespace (including none).
- When certain escape sequences starting with a backslash are inside a string, **xrc** processes them like this:

\" becomes		"
\\	becomes	\
\n becomes		line feed character
\t	becomes	tab character

In addition, a backslash followed by 1, 2, or 3 octal digits is converted to the equivalent 8-bit character. Legal values are \0 to \377. Although **xrc** accepts any of these values and outputs them to the native resource file, they might not map to legal characters on particular platforms, or they might map to different characters on different platforms.

Backslash escape sequences not processed by **xrc** pass through untouched (e.g., \b, \94, \x7AOD). However, this could change in future versions of **xrc**.

- Backslashes appearing in pathnames in #include statements are not treated specially, and do not have to be escaped by doubling them. That is, a path such as **\pgm\rsrc.xrc** can be typed literally in an #include statement.
- Whitespace is any sequence of comments, spaces, tabs, carriage returns, or line feeds. Whitespace is optional except when needed to separate two adjacent identifiers.
- A text line can be terminated by a line feed (decimal 10), a carriage return (decimal 13), a line feed followed immediately by a carriage return, or a carriage return followed immediately by a line feed. This allows the XRC compiler to process any text file without first converting it to the native text-file format.
- In an XRC script, specifications for strings, windows, dialogs, and images can appear in any order. Within a dialog or window specification, however, the statement order matters. Submenus and items appear in a menu in the order that the statements appear. Lines from #transparent statements are output in the order in which those statements appear.
- Whenever an integer constant is needed, a constant expression can appear also. Such an expression must consist only of integer constants and these tokens:

(	&&	<=	^
)	||	>=	<>
+	!	<<	?
–	==	>>	:
*	%	&	
/	!=	|	

The || and && operators can result in short-circuit evaluation in the following sense: If the right-hand side does not have to be evaluated in order to determine the value of the expression and it contains undefined preprocessor symbols, those symbols are simply ignored. Of course, if the right-hand side contained a symbol that was defined, it would have been replaced by its definition before the operator was evaluated.

- Integer constants are taken as octal constants if they begin with a zero; otherwise they are taken as decimal.

5.4. Writing XRC Scripts

The best way to begin writing an XRC script is to study this chapter along with any XVT-provided examples, each of which has an associated XRC file.

The standard XVT menus and dialogs for each platform are contained in a file called **xrc_plat.h**. This file is included by **xrc.h**, which itself should be included by all XRC scripts.

Tip: The **xrc_plat.h** file also serves as an XRC example. You might find it useful to print out and study this file.

See Also: For detailed descriptions of the XRC resource language, refer to the *XVT Portability Toolkit Reference*.

5.5. Compiling XRC

For information about compiling XRC with **xrc**, see the *XVT Portability Toolkit Reference*. Keep in mind that, while XRC input is the same for all platforms, the **xrc** compiler's output script varies from one target platform to another.

5.6. Sample XRC Script

The following file was generated by XVT-Design. This is **fontmap.xrc**, from the Font Mapper (**samples/design/fontmap**) example in the sample area:

```
#ifndef APPNAME
#define APPNAME fontmap
#endif
#ifndef QAPPNAME
#define QAPPNAME "fontmap"
#endif
#ifndef PROJECT
#define PROJECT fontmap.9
#endif

#ifndef NO_STD_FONT_MENU
#define NO_STD_FONT_MENU
#endif

#include "xrc.h"
#scan "fontmap.h"

#ifndef XRC_SRC_WIDTH
#define XRC_SRC_WIDTH          8
#endif
#ifndef XRC_SRC_HEIGHT
#define XRC_SRC_HEIGHT        13
#endif
#if XVTWS == WINWS
UNITS SEMICHARS
#undef  XRC_DEST_WIDTH
#define XRC_DEST_WIDTH        4
#undef  XRC_DEST_HEIGHT
#define XRC_DEST_HEIGHT       8
#elif XVTWS == WMWS
UNITS CHARS
#undef  XRC_DEST_WIDTH
#define XRC_DEST_WIDTH        1
#undef  XRC_DEST_HEIGHT
#define XRC_DEST_HEIGHT       1
#else
UNITS PIXELS
#ifndef XRC_DEST_WIDTH
#define XRC_DEST_WIDTH XRC_SRC_WIDTH
#endif
#ifndef XRC_DEST_HEIGHT
#define XRC_DEST_HEIGHT XRC_SRC_HEIGHT
#endif
#endif

#define XRC_RECT(x,y,w,h) ((x)*XRC_DEST_WIDTH)\
        XRC_SRC_WIDTH,((y)*XRC_DEST_HEIGHT)\
        XRC_SRC_HEIGHT,((w)*XRC_DEST_WIDTH)\
        XRC_SRC_WIDTH,((h)*XRC_DEST_HEIGHT)\
        XRC_SRC_HEIGHT


MENUBAR FONTMAP_MENUBAR
```

```
MENU FONTMAP_MENUBAR
SUBMENU FILE_MENU "~File"
        SUBMENU MAPPING_MENU "~Mappings"
        SUBMENU OPTIONS_MENU "~Options"
#if (XVTWS!=WINWS) && defined(XVT_WIN_MENU)
        XVT_WIN_MENU
#endif
        DEFAULT_HELP_MENU

MENU FILE_MENU
        ITEM FILE_MENU_OPEN "~Open Mappings..."
        ITEM FILE_MENU_MERGE "~Merge Mappings..."
        ITEM FILE_MENU_SAVE "~Save Mappings..."
        ITEM FILE_MENU_GENERATE_XRC "Generate ~XRC file..."
        ITEM FILE_MENU_QUIT "~Quit"

MENU MAPPING_MENU
        ITEM MAPPING_MENU_CREATE "~Change Font Mapping"
        ITEM MAPPING_MENU_DELETE "~Delete Custom Mapping"
        ITEM MAPPING_MENU_CLEAR "Delete ~All Custom Mappings"

MENU OPTIONS_MENU
        ITEM OPTIONS_MENU_APPLICATION_DIALOG\
                "Use Application Font ~Dialog" CHECKABLE CHECKED
        ITEM OPTIONS_MENU_SHOW_MAPPINGS "Show Custom ~Mappings"\
                CHECKABLE

MENUBAR TASK_MENUBAR

MENU TASK_MENUBAR
        DEFAULT_FILE_MENU
        DEFAULT_EDIT_MENU
#if (XVTWS!=WINWS) && defined(XVT_WIN_MENU)
        XVT_WIN_MENU
#endif
        DEFAULT_HELP_MENU


WINDOW WIN_FONT_MAPPER XRC_RECT(61,169,339,380)\\
        "Font Mapper" DOC CLOSE INVISIBL E ICONIZABLE\
        FONTMAP_MENUBAR
        TEXT WIN_FONT_MAPPER_FAMILY_TEXT\
                XRC_RECT(24,36,68,14) "Family"
        TEXT WIN_FONT_MAPPER_SIZE_TEXT\
                XRC_RECT(180,36,58,14) "Size"
        TEXT WIN_FONT_MAPPER_STYLE_TEXT XRC_RECT(264,36,67,14)\
                "Style"
        TEXT WIN_FONT_MAPPER_NATIVE_TEXT XRC_RECT(-9,180,348,14) ""
        LISTBOX WIN_FONT_MAPPER_STYLE XRC_RECT(227,60,112,88)\
                MULTIPLE
        LISTBOX WIN_FONT_MAPPER_MAPPINGS XRC_RECT(-11,296,350,84)
        GROUPBOX WIN_FONT_MAPPER_PORTABLE_ATTRS XRC_RECT\
                (-33,12,372,144) "Portable Font Attributes" CENTER_JUST
        GROUPBOX WIN_FONT_MAPPER_NATIVE_BOX XRC_RECT\
                (-33,156,372,48) "Native Mapped Font" CENTER_JUST
        GROUPBOX WIN_FONT_MAPPER_MAPPINGS_BOX XRC_RECT\
                (-33,272,372,108) "Custom Font Mappings" CENTER_JUST
        BUTTON WIN_FONT_MAPPER_CREATE_BUTTON\\
#if XVTWS == MTFWS
                XRC_RECT(36,117,204,25)\\
#else
                XRC_RECT(36,120,204,19)\\
```

```
#endif
                "Change Font Mapping"
        LISTEDIT WIN_FONT_MAPPER_FAMILY XRC_RECT(24,60,144,95)
        LISTEDIT WIN_FONT_MAPPER_SIZE XRC_RECT(180,60,72,95)

DIALOG APP_FONT_DLG XRC_RECT(69,115,360,160)\
"Application Font Dialog" MODAL
        BUTTON APP_FONT_DLG_OK\
#if XVTWS == MTFWS
                XRC_RECT(190,127,110,25)\
#else
                XRC_RECT(190,130,110,19)\
#endif
                "OK" DEFAULT
  BUTTON APP_FONT_DLG_CANCEL\
#if XVTWS == MTFWS
                XRC_RECT(60,127,110,25)\
#else
                XRC_RECT(60,130,110,19)\
#endif
                "Cancel"
        TEXT APP_FONT_DLG_TEXT1 XRC_RECT(10,10,68,14) "Family"
        TEXT APP_FONT_DLG_TEXT2 XRC_RECT(160,10,58,14) "Size"
        TEXT APP_FONT_DLG_TEXT3 XRC_RECT(240,8,67,14) "Style"
        LISTBUTTON APP_FONT_DLG_FAMILY XRC_RECT(10,30,144,95)
        LISTBUTTON APP_FONT_DLG_SIZE XRC_RECT(160,30,72,95)
        LISTBOX APP_FONT_DLG_STYLE XRC_RECT(240,30,112,88) MULTIPLE
```

6

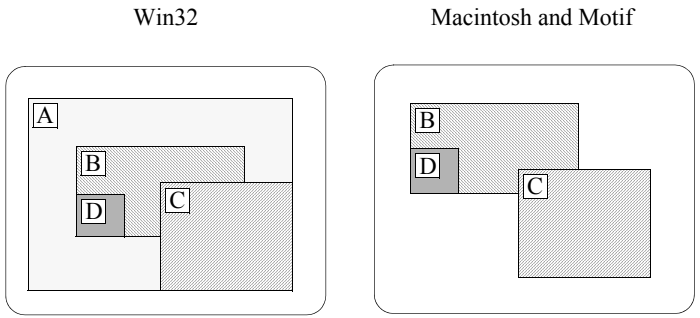
WINDOWS

XVT-Design produces source code and resources that create, size and locate windows. It also produces the event handling mechanism for the window and all its controls. This chapter contains background information about windows. If you create windows with XVT-Design, you won't need much of the information in this chapter.

Windows are the basic building blocks in XVT programs. They provide an application "work area" for the user, presenting information and allowing the user to interact with that information. Windows are containers for graphics, font-based text, and controls, and they can have an associated menubar.

XVT provides five types of windows, as shown in Figure 6.1.

- Screen window
- Task window
- Top-level windows
- Child windows
- Modal windows



A = Task window
B = Top level window and parent to D
C = Top level window
D = Child window to B (could be modal or non-modal)

Figure 6.1. Window relationships on different platforms. XVT/XM has a floating task window the size of the menubar (not shown here).

When the user (or the native windowing system) interacts with a window, events are sent to the window's event handler. The behavior and flow of the entire application center on event handlers. (This is also true of dialogs—see Chapter 7, *Dialogs*.)

Note: In XVT, several functions manipulate dialogs and controls as well as windows. Consequently, some of the functions mentioned in this chapter also appear in Chapter 7, *Dialogs*, and Chapter 8, *Controls*.

6.1. Screen and Task Windows

On start-up, an XVT application creates two windows: a screen window and a task window.

6.1.1. Screen Window

The screen window represents the physical display screen. Its boundaries and dimensions reflect the pixel extent of the physical screen. It receives no events.

Every XVT application has its own screen window. Programmatically, it is represented by the macro SCREEN_WIN, which is a valid XVT WINDOW that can be passed to many of the XVT functions that require a WINDOW handle.

6.1.2. Task Window

The task window is a virtual window with its own event handler. It represents the application, process, or logical task. On some platforms, it is represented by a distinct visible object with its own window coordinates. On other platforms, its representation corresponds to the entire screen, and its coordinates are that of the screen.

XVT-Design creates the taskwindow for you. XVT-Design refers to it as the "Application" module. Design places source code for the task window and its event handler in a file which, by default, has the same name as the project.

The task window has several purposes:

- It serves as a logical container for all top-level (document) XVT windows. On some platforms, it also serves as a visible (physical) container for top-level XVT windows (with an exception made for "detached windows").
- It provides a visible (physical) location for the application menubar.
- It represents the application task and provides a place to receive application events. An application is terminated by closing the task window. The task window event handler receives an E_DESTROY event indicating the termination of the application.

Note: For implementing your application's task window event handler, the concept of a task window is portable across all platforms. However, its appearance and behavior with respect to the user is platform-specific.

Table 6.1 describes the characteristics of the task window on every XVT platform:

	Task window is represented by a distinct visible object	Task window corresponds to the entire screen	Task window has a "ghost" window	Task window is drawable
XVT/Mac	—	3	—	—
XVT/Win32	3 [1]	3 [1]	—	3 [3]
XVT/XM	—	3	3 [2]	—

[1] The task window visibly contains top-level windows, or, if you set the ATTR_WIN_PM_NO_TWIN attribute, the screen window contains top-level windows.

[2] A "ghost" window provides a visible location for the application menu when there are no other windows with a visible menu. (ATTR_X_DISPLAY_TASK_WIN controls whether the task window menubar appears when there are no other menubar-carrying windows visible on the screen.)

This "ghost" window should not be confused with the task window itself. The primary (and only) thing you can do with a ghost window is to set its menubar, and thus, the "ghost" window provides one way for the user to close the application.

[3] You can make the task window drawable (so that it accepts drawing functions) by setting the ATTR_WIN_PM_DRAWABLE_TWIN attribute.

Table 6.1. Characteristics of the task window on each supported XVT platform

The TASK_WIN constant represents the task window. As with SCREEN_WIN, you can pass TASK_WIN as a parameter to many, but not all, XVT functions that require a WINDOW parameter.

The task window event handler always receives the first and last events for your XVT application: E_CREATE and E_DESTROY. You can terminate an application by calling xvt_app_destroy, which translates to xvt_vobj_destroy(TASK_WIN).

See Also: For more information about the task window, see TASK_WIN in the *XVT Portability Toolkit Reference*.
For information about attributes that affect the task window, see the *XVT Platform-Specific Books*.

6.2. Top-level, Child, and Modal Windows

6.2.1. Top-level Windows

Top-level windows—also called regular or document windows—contain the application's controls and graphics (see Figure 6.2 and Figure 6.3). The application creates them as needed. Top-level windows are independent of one another.

Top-level windows can possess the following window decorations: border scrollbars, titlebars, and window resizing and closing controls. You can specify all of these attributes when you create the window. Top-level windows can also have a menubar associated with them.

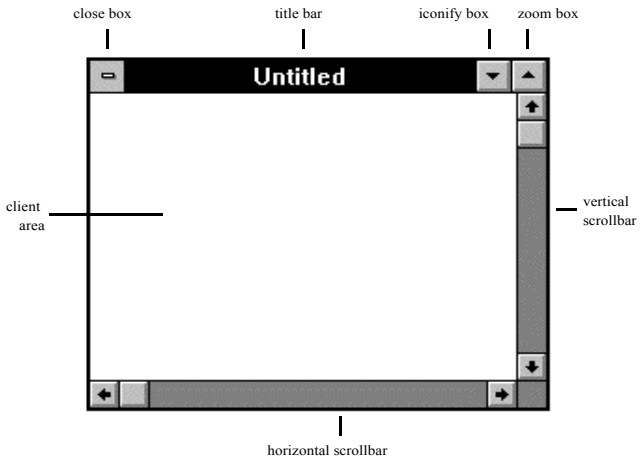


Figure 6.2. Top-level window on MS-Windows

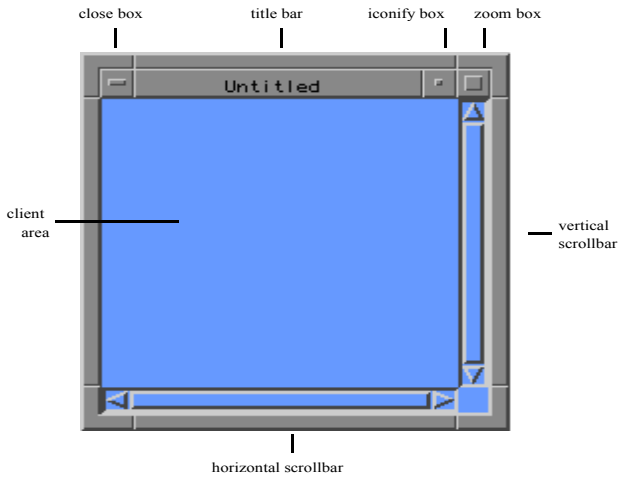


Figure 6.3. Top-level window on Motif

6.2.2. Child Windows

Child windows can be nested within top-level windows, or within other child windows; they move with their parent window and are clipped to the parent's boundaries. Only child windows can be nested within another window. Like top-level windows, child windows can have scrollbars. They can have a plain border, single border, or invisible border.

See Also: For more information about child windows, see section 6.7.

6.2.3. Modal Windows

The purpose of a modal window is to block the users' interaction with any other application window except the modal window itself. Modal windows have a different look-and-feel on each platform, because they conform with the required style of that platform's window manager. Like child windows, modal windows are associated with a parent window. The important difference between modal windows and other types of windows is that modal windows *must* be answered before the application can change its state. It is the responsibility of the application to destroy a modal window after the user responds to the choices offered in that window.

See Also: For more information about modal windows, see section 6.3.3.4.

6.3. XVT WINDOWs and Window Types

To refer to a specific XVT window, you'll use a descriptor of type WINDOW:

```
typedef long WINDOW;
```

Caution: Direct information about the contents of a window descriptor is not available to your application. Don't assume that a WINDOW is a window pointer or window handle. Operations that are allowed on windows are performed by XVT functions that take a WINDOW as one of their arguments.

6.3.1. NULL_WIN Symbol

When you have to assign or compare an object of type WINDOW to NULL, use the symbol NULL_WIN like this:

```
if (win == NULL_WIN)
    ...
```

6.3.2. WIN_TYPE Data Type

The data type WIN_TYPE is used whenever the type of window must be specified, such as when a window is created:

```
typedef enum {          /* type of window */
    W_DOC,              /* document window */
    W_PLAIN,            /* single (plain) border window */
    W_DBL,              /* double border window */
    W_NO_BORDER,        /* window with no border */
    W_MODAL,            /* modal document window */
    W_SCREEN,           /* not to be used to create windows */
    W_TASK,             /* not to be used to create windows */
    ...                 /* other WINDOW types (dialogs,
                           controls) */
} WIN_TYPE;
```

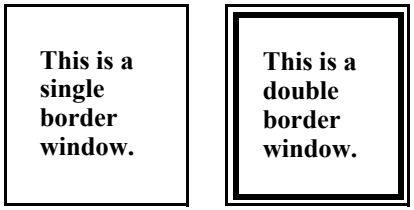


Figure 6.4. Single- and double-border windows

6.3.3. Window Types

This section explains five types of windows: W_DOC, W_PLAIN, W_DBL, W_NO_BORDER, and W_MODAL, and defines the client area of a window.

6.3.3.1. W_DOC (Top-level Windows)

In this *Guide*, W_DOC windows are usually referred to as regular, document, or top-level windows. They can have border scrollbars, titlebars, and window resizing and closing controls. XVT top-level windows can also have a menubar associated with them. When you create a window, you can specify all of these window attributes (sometimes called window decorations). However, once a window has been created, these attributes cannot be changed.

Note: The parent of a W_DOC window is always either TASK_WIN or SCREEN_WIN. W_DOC windows are always top-level windows; they cannot be child windows.

6.3.3.2. W_PLAIN and W_DBL

W_PLAIN and W_DBL are simple rectangular windows that can't possess border controls or decorations. If a native GUI platform has no concept of a double-bordered window, then these two window types look the same.

6.3.3.3. W_NO_BORDER

The type W_NO_BORDER is for a window without any border at all. Such a window can be created only if it is a child window. (Child windows can only be of type W_NO_BORDER or W_PLAIN; they are explained more fully later in this chapter.)

Tip: To find out a window's type:

Call xvt_vobj_get_type.

You can receive as a return value the window type W_PRINT, which XVT uses to indicate a print window.

6.3.3.4. W_MODAL (Modal Windows)

A modal window prevents user interaction with any other window of an application (including the parent window which may be modal itself) until some user-initiated action causes the modal window to be dismissed. When a user initiates a request for dismissal, the application *must* destroy the modal window by calling

xvt_vobj_destroy. After a modal window is destroyed, focus returns to the window which previously had focus. The Portability Toolkit also preserves the order of a modal chain (a stack of modal windows or dialogs).

Note: Modal windows do not support menubars.

See Also: For more information on menubars and their relationship to windows, see Chapter 9, *Menus*.

Modal Window Look-and-Feel

Modal windows are implemented using the native object best suited to providing modality on each platform. A W_MODAL window may have characteristics of a top-level window, a child window, or a dialog. Moreover, the look-and-feel of this object is platform-specific—it will have the physical appearance most appropriate for modality on a particular platform. Modal windows follow native look-and-feel guidelines for decorations (borders, system menus, etc.) and stacking order.

Parent Window

The following types of windows are valid parents for a modal window:

- Screen window (W_SCREEN)
- Task window (W_TASK)
- Top-level window (W_DOC, W_DBL, W_PLAIN)
- Dialog (WD_MODAL, WD_MODELESS)
- Another different modal window (W_MODAL)

The parent window of a modal window is returned by calling xvt_vobj_get_parent. If the parent is destroyed, the modal window is also destroyed automatically. Modal windows with the *screen* window as the parent are destroyed automatically when the *task* window is destroyed (as with top-level windows and dialogs).

Creation Flags

The W_MODAL window supports the following creation flags:

- WSF_DISABLED
- WSF_INVISIBLE
- WSF_PLACE_EXACT

All other WSF_* flags produce runtime warnings.

Creation Rectangle

The creation rectangle specifies the width and height for the modal window. The modal window follows native look-and-feel guidelines for position with respect to its parent. The creation flag, WSF_PLACE_EXACT, (used only for modal windows) insures that the modal window respects the top and left fields of the creation rectangle (RCT) structure. With this flag, the creation position is specified relative to the coordinate system of the parent window.

Implementation Note: On XVT/Win32, the position specified by the creation rectangle is respected even when the creation flag WSF_PLACE_EXACT is not used.

Modal Window Behavior

A modal window's *enabled* state does not depend on the state of its parent, unlike other parent/child relationships. A modal window cannot be created from an *invisible* parent, and the parent of a modal window cannot be made invisible (by calling xvt_vobj_set_visible) while the modal window exists. An error results from either case.

E_CHAR character events are delivered to the window handlers of modal windows. Your application should process these character events as it would for any other top-level window. Your options for keyboard navigation (inside modal windows) are: 1) use the XVT navigation object (described in section 6.6 on page 6-14), and let it automatically provide keyboard navigation for your modal windows, or 2) implement your own navigation mechanism.

The portable attribute ATTR_PROPAGATE_NAV_CHARS controls the delivery of those character events necessary for navigation to windows (including modal windows). This attribute is automatically set if you have chosen to use the XVT navigation object.

XVT Portability Toolkit functions that accept document windows as arguments will also accept modal windows. You can create all XVT controls, text edit objects, and custom controls in modal windows. Support is provided for drawing in modal windows as well (for example, E_UPDATE events and use of drawing tools DRAW_CTOOLS). Functions that return top-level windows as arguments, including any window enumeration functions, can also return modal windows.

Note: If you call xvt_vobj_move to move the parent of a modal window, the modal window does not move relative to the screen window.

6.3.4. Client Area

All windows possess a client area, which is the inner rectangular area of the window that is used by the application.

All drawing operations are performed in the client area. In addition, controls can be placed in the client area. The dimensions of the client area are those specified for the window when it is created, or when it is resized via xvt_vobj_move.

The function xvt_vobj_get_client_rect always returns the dimensions of a window's client area, with the origin at (0,0). xvt_vobj_get_outer_rect returns the dimensions of the entire window, including window decorations such as the titlebar and scrollbars and the top-left coordinates relative to the parent window.

6.4. Creating Windows

You can create XVT windows in three different ways, based on the needs of your application. In all cases, a WINDOW is returned from each creation. This value is your reference to the new window. You can create as many windows as your application needs, within the limits of each native GUI windowing system.

XVT-Design can automatically create windows. See XVT-Design Manual for more details.

When specifying the size of a window initially, you can use the XVT_MAX_WINDOW_RECT constant to create a top-level window that occupies its entire container.

6.4.1. Dynamic Windows

Dynamic windows are created by your program, without any need for external resource definitions.

Tip: To dynamically create windows at any time:

Call xvt_win_create.

All of the window's attributes (initial size, title text, menu resource ID, parent WINDOW, attribute flags, event handler, and application data) are specified as arguments to xvt_win_create.

See Also: For more information about creating windows, see section 3.3 in Chapter 3, *GUI Elements*.

6.4.2. Resource-based Windows

Resource-based windows are created from external resource definitions. To create a resource-based window, you specify the window's definition in XVT's XVT Resource Compiler (XRC). The application accesses it at runtime by means of the object's resource ID. The xvt_win_create_res function then creates the window.

You can create a resource-based window in XVT-Design. XVT-Design assigns a symbolic identifier, which corresponds to a resource ID, to the window. Functions can access the window by its symbolic identifier.

Resource-based windows are useful when the definition of a window is unknown or changeable at compilation time. You can optionally define arbitrary data (USERDATA) for the window. This USERDATA can then be called by xvt_res_get_win_data.

See Also: For details about xvt_res_get_win_data, see the *XVT Portability Toolkit Reference*.
For information about the XRC syntax for creating windows, see section 3.3.1 in Chapter 3, *GUI Elements*.

6.4.3. Structure-based Windows

Structure-based windows are created from an array of WIN_DEF data structures that are passed to xvt_win_create_def. The xvt_win_create_def function allows you to create a container and all of its contained components in a single call.

See Also: The WIN_DEF structure is common to windows, dialogs, and controls. For details, see section 3.3.2 in Chapter 3, *GUI Elements*.

6.5. Replacing and Retrieving Window Event Handlers

Whenever you create an XVT task, top-level, or child window, you must specify an event handler for it. Later, you can substitute a new event handler for the window, using xvt_win_set_handler. In addition, you can retrieve a window's current event handler via xvt_win_get_handler.

For every window or dialog that you create, XVT-Design automatically defines the event handler function and supplies its name to the container's creation function. XVT-Design also creates an event handler for the task window.

See Also: For more information about event handling for windows, see section 3.4 in Chapter 3, *GUI Elements*.

6.6. Keyboard Navigation in Windows

Keyboard navigation is the use of keyboard input, in lieu of mouse pointing and clicking, to interact with GUI objects. Generally, native look-and-feel for keyboard navigation includes using the Tab key and Shift-Tab key (back-tab) to traverse through a list of controls. Alternatively, the user may type character keys (associated with mnemonic characters) to select an object directly. A mnemonic character is preceded by a tilde (~) in the title text and displayed with an underline to users. Groups of controls (such as radio buttons) may be traversed with Arrow keys.

Unlike XVT dialogs which automatically provide keyboard navigation to users, XVT windows require special handling to implement keyboard navigation. The XVT_NAV navigation object encapsulates the navigation list of controls, child windows, and custom controls for a particular window. The navigation object allows you to specify the navigation order for your application's windows. Any control mnemonic character set in the control's title will be processed automatically on platforms on which control mnemonics are supported in native look-and-feel (XVT/Win32).

Tip: To create a navigation object for a specified WINDOW:

Call xvt_nav_create with a valid SLIST of XVT GUI objects.

Tip: To destroy a navigation object:

Call xvt_nav_destroy.

Tip: To retrieve the navigation object associated with a WINDOW:

Call xvt_win_get_nav.

GUI objects may be added or removed from and existing navigation object by calling xvt_nav_add_win or xvt_nav_rem_win, respectively. xvt_nav_list_wins provides a list of GUI components in a navigation object.

XVT-Design automatically creates the XVT_NAV object for you when you enable the Navigation check box using XVT-Design's window attribute editor.

See Also: For detailed information about the xvt_nav_* functions, refer to their descriptions in the *XVT Portability Toolkit Reference*.

6.7. Working with Child Windows

A child window is one that is hierarchically related to a parent window. Child windows have several characteristics in common:

- Is a full-fledged window, with its own client area, coordinate system, drawing tools, clipping rectangle, etc.
- Is always clipped to the boundaries of its client area
- Can have children of its own
- Can receive any window-oriented events, such as E_MOUSE_MOVE or E_UPDATE
- Is fixed relative to the client area of its parent unless changed by a call to xvt_vobj_move (if its parent moves, the child window moves also)
- Is unaffected by resizing of its parent (however, although its position and size aren't changed, how much of it can be seen does change)
- Is not subject to being moved, resized, or closed by the user (child windows do not have resizing and closing controls or titlebars)

The only valid window types for child windows are W_PLAIN, W_MODAL, and W_NO_BORDER (W_NO_BORDER is exclusively reserved for child windows).

6.7.1. Benefits of Child Windows

XVT child windows allow you to nest windows within windows, and to establish a hierarchical model for defining the relationships between windows. Neither would be possible with top-level windows only, because they are independent of one another. Other than the fact that top-level windows generally are bounded by the client area of the task window, no top-level window is nested within any other top-level window.

When child windows are created, their positioning rectangle is relative to the client area of their parent window. In addition, child windows move synchronously with their parent, and are clipped to their parent's boundaries.

A child window inherits whether it is visible and enabled from its parent. If the parent is disabled, then all child windows within that parent are disabled. However, if a child is disabled, then all mouse events from the child window are transformed into the parent's coordinate system and sent to the parent window's event handler.

6.7.2. Determining Parent Windows

Tip: To determine the parent of any window:

Call xvt_vobj_get_parent.

Note: If called on a top-level window, xvt_vobj_get_parent returns TASK_WIN. It returns SCREEN_WIN as the parent of the task window, and returns NULL_WIN as the parent of SCREEN_WIN.

6.7.3. Listing Window Descendants

Tip: To list the titles and window handles of controls, windows, and dialogs (only if W_SCREEN is parent) which have a specified parent WINDOW:

Call xvt_win_list_wins.

xvt_win_list_wins returns an SLIST in which each SLIST_ELT element contains a WINDOW handle and a window title string (if appropriate for the window type). xvt_win_list_wins is non-recursive–in other words, it lists only the immediate descendants of the container in the order of their creation. xvt_win_list_wins does not return text edit objects because they do not possess WINDOW handles.

See Also: xvt_scr_list_wins in the *XVT Portability Toolkit Reference* for more information about this function which lists all top-level windows and dialogs in an XVT application.

6.7.4. Enumerating Windows

Tip: To apply a function to the list of controls, child windows, and dialogs (only if W_SCREEN is parent) that have a specified parent WINDOW:

Call xvt_win_enum_wins.

Your application-supplied callback function is called once for each window in the descendant list with the window handle (WINDOW) and a pointer to application data passed as arguments. You should avoid creating new descendant windows or destroying existing (enumerated) descendant windows during the enumeration process.

The type XVT_ENUM_CHILDREN prototypes the callback function that your application passes to xvt_win_enum_wins. Be careful not to cause inadvertent recursion when writing this callback function.

Example: This code demonstrates the use of xvt_win_enum_wins to set application data for controls and later free the same data and destroy the controls:

```
BOOLEAN XVT_CALLCONV1 create_app_data (WINDOW win,
                                       long data)
{
        int id = xvt_ctl_get_id(win);
        char *ctl_data;

        ctl_data = (char *) xvt_mem_alloc(30);

        xvt_str_sprintf(ctl_data, (char *)data, id);

        xvt_vobj_set_data(ctl_win,(long)ctl_data);

        return TRUE;
}


BOOLEAN XVT_CALLCONV1 free_app_data (WINDOW win,
                                     long data)
{
        DATA_PTR ctl_data =
                    (DATA_PTR)xvt_vobj_get_data(win);

        NOREF(data);

        if (ctl_data != NULL)
                xvt_mem_free(ctl_data);

        xvt_vobj_destroy(win);

        return TRUE;
}
```

```
long XVT_CALLCONV1 win_eh(WINDOW win, EVENT *ep)
{
        switch (ep->type) {
                ...
                case E_CREATE:
                        ...
                        xvt_win_enum_wins(win, create_app_data,
                                        (long) "Control ID = %d",
                                        NULL);
                        ...
                        break;

                case E_CONTROL:
                        switch (ep->v.ctl.id) {
                                ...
                                case CLOSE_BUTTON:
                                        xvt_win_enum_wins(win, free_app_data,
                                                NULL, NULL);
                                        ...
                                        break;
                                ...
                        }
                        break;
                ...
        }
        ...
}
```

6.8. Associating Application Data with Windows

Frequently, you'll want to associate your own data with a window. Doing so allows you to keep window-related data with the window, rather than maintaining it somewhere else. In a word processor, for example, the application data might be the text of the document and all of its related attributes.

Tip: To associate data with a window:

Use the function xvt_vobj_set_data to set a long word that XVT keeps with each window.

Tip: To retrieve the value:

Call xvt_vobj_get_data.

Tip: You can also set a window's application data value as part of the window creation function call. However, it is better programming style to create and attach window-specific data in the window's event handler, in response to the E_CREATE event. XVT recommends that you perform all window-specific initializations using this approach.

Because each window has its own event handler, attaching application data to a window is an efficient way to program in XVT.

Each time an event is sent to an event handler, the window's WINDOW is also sent. You can then call xvt_vobj_get_data at the top of the event handler, making the data easily accessible to the application.

Note: XVT recommends that you perform window-specific termination operations in response to the E_DESTROY event, which might include freeing data structures previously attached to the window's application data field, as well as those of its controls. Furthermore, XVT recommends that you reset the application data fields to zero in case they are referenced again later in the application. Once a window is destroyed, allocated memory that was referenced by the window's application data field can't be retrieved.

Usually the data is a pointer to a structure, in which case you'll have to cast the argument to xvt_vobj_set_data and the return value from xvt_vobj_get_data. To prevent a compiler warning, use the macro PTR_LONG to cast a pointer to a long.

6.9. Updating Windows

This section explains drawing in windows and how clipping is implemented. Other functions, such as determining parent windows, window dimensions, and front-most windows, are common to multiple GUI objects.

See Also: For more information about common functions, see section 3.5 in Chapter 3, *GUI Elements*.

6.9.1. Drawing

It's a good idea to postpone drawing in a window until an E_UPDATE event occurs, because both initial drawing and repair of damage can then be handled together.

Tip: To draw in a window:

1. Revise whatever internal data structures you're maintaining.
2. Call xvt_dwin_invalidate_rect.

   This tells XVT that a part of the window has to be updated. Later, XVT generates an appropriate E_UPDATE event.

Of course, sometimes you have to draw in real time, for instance when the user is dragging a shape in a drawing program. In this case, the drawing functions should be executed immediately. When an E_UPDATE does occur, you can speed up output by only drawing

shapes and text that are in the update rectangle (which is supplied in the E_UPDATE event).

Tip: To force all pending E_UPDATE events to be processed immediately (resulting in one or more calls to your event handler function):

Call xvt_dwin_update.

You should make this call just prior to calling xvt_dwin_scroll_rect.

6.9.2. Clipping

Anything your application draws in a window is clipped to the client area, so that nothing spills out into border areas or decorations. The default clipping rectangle of a window is its client area.

Tip: To restrict the drawing area to just part of the client area:

Call xvt_dwin_set_clip.

Tip: To determine the existing clipping rectangle:

Call xvt_dwin_get_clip.

The clipping rectangle you set for a window remains until you change it with xvt_dwin_set_clip. During an update, XVT clips all drawing operations to the intersection of the update area and the clipping rectangle. If you accidentally leave the clipping area too small, you may not update as much as you should during an update.

Tip: To be safe, when you restrict the clipping rectangle with xvt_dwin_set_clip, be sure to restore it as soon as possible. Do this with a NULL second argument to xvt_dwin_set_clip.

6.10. Window Titles

Document windows and modal windows have a title, which is originally set when the window is created. You can change the title at any time by calling xvt_vobj_set_title. The function xvt_win_set_doc_title is similar, but it ensures that the title obeys appropriate user interface guidelines for the underlying toolkit.

You can retrieve the title of a window with xvt_vobj_get_title. xvt_vobj_get_title is also used to get the text of dialogs and controls.

6.11. Window Scrollbars and Scrolling

The WSF_HSCROLL and WSF_VSCROLL flags, specified during window creation, indicate whether the window has horizontal and/or vertical scrollbars. Scrollbars have no default range, so you need to set the ranges yourself.

Tip: To set scrollbar ranges for a document window:

Call xvt_sbar_set_range.

Once the range of a scrollbar is set, you should set its proportion indicator.

Tip: To set a scrollbar's proportion:

Call xvt_sbar_set_proportion.

The indicator shows the amount of data that is visible in a window, as compared to the total amount of data.

Example: For example, if your document contains 500 lines, and your window can display 50 lines, then the scrollbar range would be 0 to 500, and the proportion would be 50.

See Also: For more information about scrolling, see Chapter 13, *Scrolling*.

6.11.1. Proportional Scrollbars

Proportional scrollbars are not supported on all platforms, but you don't have to think about this when writing an XVT program. If you set the range and proportions correctly, the scrollbars behave appropriately on all platforms. On platforms without proportional scrollbars, the call to xvt_sbar_set_proportion simply reduces the scrollbar range, which has the desired effect.

6.11.2. Scrolling

When the user operates a window's scrollbar, XVT generates the appropriate event. In responding to this event, your application usually updates the window, possibly by scrolling part of it with xvt_dwin_scroll_rect. It should then change the position of the thumb by calling xvt_sbar_set_pos. The thumb indicates the position of the window's contents relative to the document as a whole.

Note: Scrollbars do not automatically perform scrolling operations within the client area. Scrollbars only notify your window's event handler that they were manipulated. It is up to you to respond to the scrollbar events.

6.12. Other Window Operations

Tip: To get a list of all windows (top-level as well as dialogs):

Call xvt_scr_list_wins.

This function returns the window titles and WINDOW descriptors in the form of an SLIST**.** (For more information about SLIST, see section B.1 in Appendix B, *Utilities*.)

Implementation Note: The xvt_scr_list_wins function does not return SCREEN_WIN or TASK_WIN in its list ofwindows. (On XVT/Win32, TASK_WIN will be listed if it was created as a drawable window.)

Tip: To determine a WINDOW's type:

Call xvt_vobj_get_type.

Tip: To get the top-level or modal window that has keyboard focus or contains a child window with focus:

Call xvt_scr_get_focus_topwin.

This function is useful for getting the top-level container window, for operations such as menu changes.

Tip: To draw a borderless client-area-sized rectangle in a specified window, using the specified COLOR argument:

Use xvt_dwin_clear.

The xvt_dwin_clear function can also quickly set the background color of windows. (Dialogs cannot be referenced by this function, since the client area of a dialog is not drawable in XVT.)

6.13. Window Manipulation Functions

Table 6.2 summarizes the XVT functions that you can use to manipulate windows.

These functions also work with dialogs or controls. However, their behavior may differ when used with them.

XVT Function	Effect on Task Window	Effect on Top-level and Modal Windows	Effect on Child Window
xvt_vobj_set_title	Sets the title text of TASK_WIN	Sets the title of the window	Ignored
xvt_vobj_get_title	Gets the title of TASK_WIN	Gets the window title	Returns NULL
xvt_vobj_set_enabled	Not allowed	Disables or enables specified window and all child WINDOWS	Disables or enables specified window and child WINDOWS
xvt_vobj_set_visible	Not allowed	Shows or hides specified window and child WINDOWS	Shows or hides specified window and child WINDOWS
xvt_vobj_get_client_rect	Gets client rectangle of TASK_WIN	Gets client rectangle of specified window	Gets client rectangle of specified window
xvt_vobj_get_outer_rect	Gets outer rectangle of TASK_WIN	Gets outer rectangle of specified window	Gets outer rectangle of specified window
xvt_vobj_get_parent	Returns SCREEN_WIN (xvt_vobj_get_parent on SCREEN_WIN returns NULL_WIN)	Returns TASK_WIN (or parent of modal window)	Returns parent WINDOW
xvt_vobj_move	Moves/resizes TASK_WIN (if applicable)	Moves/resizes specified window	Moves/resizes specified window

XVT Function	Effect on Task Window	Effect on Top-level and Modal Windows	Effect on Child Window
xvt_vobj_destroy	Terminates application (all windows) via E_DESTROY to task event handler	Terminates window (and all child windows and contained modal windows) via E_DESTROY to window's event handler	Terminates window (and all child windows) via E_DESTROY to window's event handler
xvt_vobj_raise	Raises TASK_WIN to the top (if applicable)	Raises specified window to the top	Raises specified window to the top
xvt_vobj_get_type	Returns W_TASK	Returns WIN_TYPE of window (W_DOC, W_PLAIN, W_DBL, W_MODAL)	Returns WIN_TYPE of child window (W_PLAIN or W_NO_BORDER)
xvt_vobj_set_data	Sets app data	Sets app data	Sets app data
xvt_vobj_get_data	Returns app data	Returns app data	Returns app data
xvt_scr_set_focus_vobj	Sets keyboard focus to TASK_WIN (only on XVT/Win32 when ATTR_WIN_PM_DRAWABLE_TWIN is set)	Sets keyboard focus to specified window	Sets keyboard focus to specified window
xvt_scr_get_focus_vobj	Returns TASK_WIN if it has focus (only on XVT/Win32 when ATTR_WIN_PM_DRAWABLE_TWIN is set)	Returns top-level window if it has the focus	Returns child window or control if it has the focus
xvt_scr_get_focus_topwin	N/A	Gets the top-level window with focus (the active window)	N/A
xvt_win_set_handler	Sets the event handler for TASK_WIN	Sets the event handler for top-level window	Sets the event handler for child window
xvt_win_get_handler	Gets the event handler for TASK_WIN	Gets the event handler for top-level window	Gets the event handler for child window

XVT Function	Effect on Task Window	Effect on Top-level and Modal Windows	Effect on Child Window
xvt_sbar_set_range	Sets scrollbar range for TASK_WIN border scrollbar (XVT/Win32 only)	Sets scrollbar range for top-level window border scrollbar [1]	Sets scrollbar range for top-level window border scrollbar
xvt_sbar_get_range	Gets scrollbar range for TASK_WIN border scrollbar (XVT/Win32 only)	Gets scrollbar range for top-level window border scrollbar [1]	Gets scrollbar range for child window border scrollbar
xvt_sbar_set_pos	Sets scrollbar position for TASK_WIN border scrollbar (XVT/Win32 only)	Sets scrollbar position for top-level window border scrollbar [1]	Sets scrollbar position for child window border scrollbar
xvt_sbar_get_pos	Gets scrollbar position for TASK_WIN border scrollbar (XVT/Win32 only)	Gets scrollbar position for top-level window border scrollbar [1]	Gets scrollbar position for child window border scrollbar
xvt_sbar_set_proportion	Sets scrollbar proportion indicator for TASK_WIN border scrollbar (XVT/Win32 only)	Sets scrollbar proportion indicator for top-level window border scrollbar [1]	Sets scrollbar proportion indicator for child window border scrollbar
xvt_sbar_get_proportion	Gets scrollbar proportion indicator for TASK_WIN border scrollbar (XVT/Win32 only)	Gets scrollbar proportion indicator for top-level window border scrollbar [1]	Gets scrollbar proportion indicator for child window border scrollbar

[1] N/A for modal or top-level plain windows.

Table 6.2. Window manipulation functions

Caution: If any of the above windows are created without scrollbars, and you call xvt_sbar_set_* or xvt_sbar_get_*, XVT issues an error.

See Also: For more information about dialogs and controls, see Chapter 7, *Dialogs*, and Chapter 8, *Controls*.

7

DIALOGS

XVT dialogs (sometimes referred to as dialog boxes), are similar to XVT windows. Typically, dialogs serve as containers for controls, and provide a means for presenting selection options to the user. Most native GUI platforms have some kind of dialog manager, which facilitates keyboard traversal of controls.

7.1. Modal and Modeless Dialogs

XVT provides two types of dialogs, modal and modeless. Dialog modality determines whether the application is frozen until the user responds to the dialog (see Figure 7.1). Dialogs are specified by using the WIN_TYPE enumeration:

```
typedef enum e_win_type {       /* type of window */
    ...
    WD_MODAL,                   /* modal dialog */
    WD_MODELESS,                /* modeless dialog */
    ...
} WIN_TYPE;
```

Type is specified when the dialog is created, and can't be changed. A dialog's type determines its behavior.

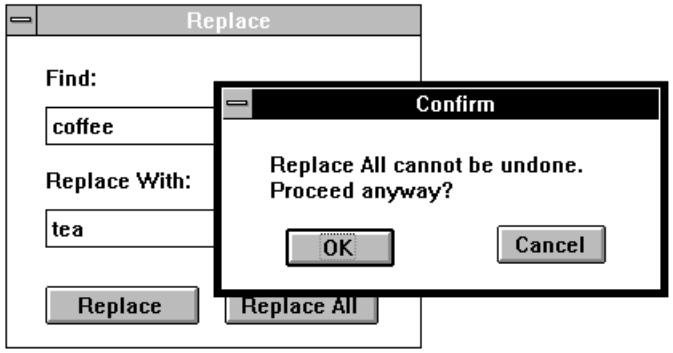


Figure 7.1. Modal and modeless dialogs

Modal Dialogs

Modal dialogs freeze an application until the user responds. In other words, a modal dialog forces the user to respond to it; once the user response has been received and acted upon, you must destroy the dialog by calling xvt_vobj_destroy. As you might expect, you should use modal dialogs only when the user must respond before the application can continue.

Tip: XVT provides several predefined modal dialogs for frequently performed operations (see section 7.3 on page 7-6).

Example: You would use a modal dialog to ask if the user wants to save the changes made to a document before closing the window. Because this question must be answered before the application can continue, it makes sense to have the dialog containing this question "lock" the application until the dialog is responded to and, finally, dismissed.

The response to a modal dialog is almost always an E_CONTROL event, which tells the handler that the user has finished working with the dialog. Often, this is a push button control event, perhaps signaling a user response to this dialog. Once such an event has been processed, the dialog event handler must call xvt_vobj_destroy, producing an E_DESTROY event and thus destroying the dialog. Only then does the call that created the dialog (either xvt_dlg_create_def or xvt_dlg_create_res) return to the application.

Tip: When you invoke a window or a modeless dialog from a modal dialog, you should immediately dismiss the modal dialog so that the

newly created window or modeless dialog can receive events and be useful.

Example: The following code shows how you could structure an application to define, create, and manage a modal dialog. This dialog contains a push button which, when pressed, signals the application that the user has responded, and to destroy the dialog. (You could also use this code for modeless dialogs, but the application would continue to operate after the dialog creation function returns.)

```
...
/* Create modal dialog using resource-based definition */

xvt_dlg_create_res (WD_MODAL, OUR_DIALOG_ID, EM_ALL,
        a_dialog_eh, 0L);

/* The user responded; the application can continue. */
...

long XVT_CALLCONV1 a_dialog_eh (WINDOW dlg,
        EVENT *event_p)
{
        switch (event_p->type) {
        /* If user pressed OK pushbutton, close the dialog. */
        case E_CONTROL:
                if (event_p->v.ctl.id == DLG_OK)
                        xvt_vobj_destroy (dlg);
                break;

        /* If user operates close control on the dialog frame,
                close the dialog. */

        case E_CLOSE:
                xvt_vobj_destroy (dlg);
                break;
        ...
        }
        return (0L);
}
```

Often your dialogs will be more complex, but their basic program structure should follow the approach shown above.

See Also: For more information about E_CONTROL events, see section 4.5.4 on page 4-25.

Modeless Dialogs

Modeless dialogs behave much more like windows. The function called to create a modeless dialog (either xvt_dlg_create_def or xvt_dlg_create_res) immediately returns to the application without waiting for a user response to the dialog. The application can continue to operate, processing and generating events in a normal manner. The modeless dialog remains on the screen until xvt_vobj_destroy is called to destroy it.

As they do with modal dialogs, control-related events come into the modeless dialog's event handler in the form of E_CONTROL events. Usually, a particular control event (for example, the pressing of a push button), signals the event handler that the user has responded to this dialog. The states of any of the relevant controls in the dialog can be queried, processed, and/or saved, and then xvt_vobj_destroy can be called to destroy the modeless dialog.

7.2. Defining and Creating Dialogs

You can define and create XVT dialogs in two ways: as resource-based dialogs or in-memory structures.

Caution: As a general guideline, do not attempt to create a dialog while processing an E_UPDATE event in a window event handler. Some native GUI windowing systems cause XVT to generate E_UPDATE events when a dialog is destroyed, so that the contents of a previously obscured window can be repaired. On these systems, creating a dialog while processing an E_UPDATE event can cause an E_UPDATE event to be recursively generated to the window event handler, causing the dialog to be created again, and so on. By default, XVT prevents you from making this mistake, although you can explicitly override this restriction by using the attribute ATTR_SUPPRESS_UPDATE_CHK.

A similar problem can occur when creating dialogs during the processing of E_FOCUS events. XVT has no restriction for this case, so you should take care when attempting to do this.

7.2.1. Resource-based Dialogs

Tip: To create a resource-based dialog:

Call xvt_dlg_create_res.

In XRC, dialogs are defined in terms of both their own attributes (resource ID, size, title, modality, etc.), and the individual controls that they contain. By referring to the dialog's resource ID, you can create a dialog simply by calling xvt_res_get_dialog. You can optionally define arbitrary data (USERDATA) for the dialog. This USERDATA can then be called by xvt_res_get_dlg_data.

See Also: For more information, see section 3.3.1 in Chapter 3, *GUI Elements*. Also see xvt_res_get_dlg_data in the *XVT Portability Toolkit Reference*.

7.2.2. In-memory Structures

You can use in-memory structures to define and create dialogs, as well as windows.

Tip: To define an in-memory dialog and its controls:

> Call xvt_dlg_create_def in conjunction with an array of WIN_DEF structures.

No matter how the dialog is defined and created, you should structure your application to respond to events by means of the dialog's event handler. In XVT, you cannot add controls to dialogs after creation, so you should include all of the needed controls either in the XRC file or as objects in the WIN_DEF array passed to xvt_dlg_create_def.

Tip: A way of getting around this restriction is to create some controls as initially invisible, then make them visible as needed. This simulates adding new controls to dialogs.

See Also: For more details on the WIN_DEF structure and its use, see section 3.3.2 in Chapter 3, *GUI Elements*.

Modal versus Modeless Dialogs

For modal dialogs (of WIN_TYPE WD_MODAL), the creation function returns only when the dialog is destroyed (via xvt_vobj_destroy). For modeless dialogs (of WIN_TYPE WD_MODELESS), creation functions return immediately after the dialog has been created, and the application program continues to function in an event-driven manner. As with modal dialogs, modeless dialogs are removed when xvt_vobj_destroy is called for the dialog.

For modeless dialogs, dialog creation functions return the dialog's WINDOW. For modal dialogs, the WINDOW is also returned, but because the function waits until the modal dialog is destroyed before returning, the WINDOW is not valid and should not be used.

Note: Some native GUI windowing systems can provide a close dialog control on the dialog's frame; if the user selects this frame control, an E_CLOSE event is generated. However, xvt_vobj_destroy must still be called for the dialog to be destroyed.

7.3. Predefined Dialogs

XVT supports several common dialog designs. You can use them as follows:

If you want to:	Use this function:
Ask the user a yes or no question	xvt_dm_post_ask
Allow the user to choose a color	xvt_dm_post_color_sel
Allow the user to change drawing tools	xvt_dm_post_ctools_sel
Put up a note or error alert	xvt_dm_post_error, xvt_dm_post_note, or xvt_dm_post_warning
Display a set of font alternatives	xvt_dm_post_font_sel
Put up a note and terminate the application	xvt_dm_post_fatal_exit
Get a string typed by the user	xvt_dm_post_string_prompt
Put up an About box	xvt_dm_post_about_box
Prompt the user for a filename for input or output	xvt_dm_post_file_open or xvt_dm_post_file_save

Note: The last two dialog boxes (open and save) let the user scan filenames, change directories, and switch drives.

7.4. Dialog Manipulation Functions

XVT functions that manipulate dialogs often perform similar operations on windows and, in some cases, on controls. To use these functions, you need the WINDOW for the dialog of interest. This is always available within the dialog's event handler. Also, for modeless dialogs, the dialog creation functions return a valid WINDOW.

Table 7.1 shows the dialog manipulation functions.

XVT Function	Effect on Dialog
xvt_vobj_set_title	Sets dialog title (not displayed for some dialogs on some platforms)
xvt_vobj_get_title	Returns title of dialog (even if it is not displayed)
xvt_vobj_set_enabled	Disables (or hides) specified dialog
xvt_vobj_set_visible	Shows specified dialog (For a modal dialog, xvt_vobj_set_visible(dlg, FALSE) is *not* supported on some platforms and is not recommended on *any* platform)
xvt_vobj_get_client_rect	Gets client rectangle of specified dialog
xvt_vobj_get_outer_rect	Gets outer rectangle of specified dialog
xvt_vobj_get_parent	Returns SCREEN_WIN
xvt_vobj_get_type	Returns dialog WIN_TYPE (WD_MODAL or WD_MODELESS)
xvt_vobj_move	Moves/resizes specified dialog
xvt_vobj_destroy	Terminates dialog and all controls
xvt_vobj_get_data	Returns app data
xvt_vobj_set_data	Sets app data
xvt_scr_set_focus_vobj	Sets keyboard focus to specified dialog
xvt_scr_get_focus_vobj	Returns dialog NULL_WIN if a dialog has the keyboard focus
xvt_win_get_handler	Gets the event handler for dialog
xvt_win_set_handler	Sets the event handler for dialog

Table 7.1. Dialog manipulation functions

8

CONTROLS

Controls are the most common object of user interaction. XVT supports a wide variety of controls: push buttons, check boxes, radio buttons, edit fields, combo controls, static text, list boxes, scrollbars, group boxes, notebooks, icons, and text edit objects. An assortment of XVT controls is shown in Figure 8.1.

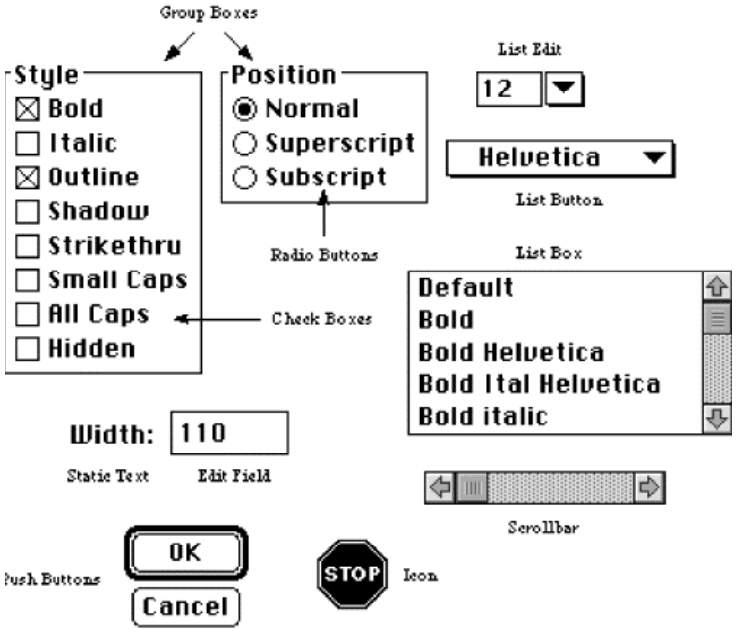


Figure 8.1. Controls that can appear in dialog boxes

E_CONTROL Events and Event Handlers

Events related to controls are called E_CONTROL events. These events are sent to the event handler of the control's parent window. This is because, unlike windows and dialogs, controls normally lack event handlers. (However, you can structure your code so that your controls do have event handlers; see section 3.4.2 on page 3-15.)

The E_CONTROL event contains information specific to each control. From the type and ci fields in the CONTROL_INFO structure, you can determine how the user manipulated the control. And, because controls have short IDs associated with them (which you can arbitrarily assign), you can build switch statements into the code that handles your E_CONTROL events, to respond to specific controls.

Creating Controls

You can define controls as resources bundled with a dialog or window, or add them at runtime to existing windows. You can't add controls at runtime to existing dialogs, because most native GUI dialog handlers don't allow this. However, you can circumvent this by creating a control as invisible. And, as previously described, you can create windows and dialogs, along with a group of controls, at runtime by using in-memory data structures (see section 7.2.2 on page 7-5).

Working with Controls

XVT allows the application to interact directly with any control. Some function calls might get or set the control's state or change its attributes, while other calls are more control-specific, for example, inquiring about a list's selection.

Many of the functions that manipulate windows and dialogs also work for some or all control attributes, along with many control-specific manipulation functions.

See Also: For details on the XRC syntax for creating controls, see the *XVT Portability Toolkit Reference*.
For more information about creating windows and dialogs as in-memory data structures, see Chapter 6, *Windows*, and Chapter 7, *Dialogs*.

8.1. Creating and Defining Controls

You can create and lay out controls in XVT-Design. See XVT-Design Manual for more details.

In XVT, you can define and create controls in three ways. These flexible methods are very similar to those used to create windows and dialogs:

Dynamic Controls

Dynamic controls are created by your program, without relying on external resource definitions. This type of control creation is restricted to windows, because several native GUI platforms do not allow for dynamic control additions to a dialog after it has been created. The xvt_ctl_create function can create dynamic controls in windows at any time from within an XVT application. You specify all attributes of the control—initial size, title text, ID, the parent WINDOW, attribute flags, application data, etc.—as arguments to xvt_ctl_create.

Resource-based Controls

Resource-based controls are created by your application from an external resource definition. You specify this definition in XVT's XVT Resource Compiler (XRC), and the application accesses it at runtime by means of either a dialog or window resource ID. The individual controls within the window or dialog belong to the overall window or dialog resource definition. To create resource-based controls, you use xvt_dlg_create_res and xvt_win_create_res.

Structure-based Controls

As with windows and dialogs, you can create controls from a WIN_DEF data structure. For controls the function called is xvt_ctl_create_def.

See Also: For more information about creating resource-based controls, see section 3.3.1 on page 3-4.
For more information about creating controls from a WIN_DEF data structure, see section 3.3.2 on page 3-7.

8.2. Control Event Structures

As it does for windows and dialogs, the WIN_TYPE enumeration defines control types.

```
typedef enum e_win_type {  /* type of window */
        ...
        WC_PUSHBUTTON       /* push button */
        WC_EDIT             /* edit field */
        WC_LBOX             /* list box */
        WC_ICON             /* icon */
        WC_VSCROLL          /* vertical scrollbar */
        WC_HSCROLL          /* horizontal scrollbar */
        WC_TEXT             /* static text */
        WC_LISTEDIT         /* list edit combo control */
        WC_LISTBUTTON       /* list button combo control */
        WC_CHECKBOX         /* check box */
        WC_RADIOBUTTON      /* radio button */
        WC_GROUPBOX         /* group box */
        WC_TEXTEDIT         /* XVT text edit object */
        WC_TREEVIEW         /* treeview control*/
        ...
} WIN_TYPE;
```

EVENT and CONTROL_INFO Data Structures

The EVENT and CONTROL_INFO data structures in the EVENT structure notify the application of control-related events:

```
typedef struct s_event {
        EVENT_TYPE type;
        union {
                ...
                struct s_ctl {                  /* E_CONTROL */
                        short id;               /* control's ID */
                        CONTROL_INFO ci;        /* control info */
                } ctl;
                ...
        } v;
} EVENT, *EVENT_PTR;
```

The ctl substructure of the EVENT structure contains the control's ID (assigned during the creation of the control), and a CONTROL_INFO structure (the field ci), which contains more control-related event information.

Based on the type of the control whose event is being reported, the appropriate substructure in the CONTROL_INFO structure is filled in:

```
typedef struct s_ctlinfo {
  WIN_TYPE type;                    /* WC_* */
  WINDOW win;                       /* WINDOW of control */
  union {
        struct s_pushbutton {
              int reserved;         /* reserved (unused) */
        } pushbutton;

        struct s_radiobutton {
              int reserved;         /* reserved (unused) */
        } radiobutton;

        struct s_checkbox {
              int reserved;         /* reserved (unused) */
        } checkbox;

        struct s_scroll {           /* scrollbar action */
              SCROLL_CONTROL what;
                                    /* activity site */
              short pos;            /* thumb position */
        } scroll;

        struct s_edit {
              BOOLEAN focus_change;
                                    /* event a focus change? */
              BOOLEAN active;       /* if so: gaining focus? */
        } edit;
        struct s_statictext {
              int reserved;         /* reserved (unused) */
        } statictext;

        struct s_lbox {             /* list box action */
              BOOLEAN dbl_click;    /* double click? */
        } lbox;

        struct s_listbutton {
              int reserved;         /* reserved (unused) */
        } listbutton;

        struct s_listedit {
              BOOLEAN focus_change;
                                    /* event a focus change? */
              BOOLEAN active;       /* if so: gaining focus? */
        } listedit;

        struct s_groupbox {
              int reserved;         /* reserved (unused) */

        } groupbox;
```

```
        struct s_textedit {
            BOOLEAN focus_change;
                                        /* event a focus change? */
            BOOLEAN active;             /* if so: gaining focus? */
        } edit;

        struct s_treeview {
            XVT_TREEVIEW_NODE node; /* Node */
            BOOLEAN sgl_click; /* Single click*/
            BOOLEAN dbl_click; /* Double click */
            BOOLEAN expanded; /* Node was expanded*/
            BOOLEAN collapsed; /* Node was collapsed*/
        } treeview;

        struct s_icon {
            int reserved;               /* reserved (unused) */
        } icon;
    } v;
} CONTROL_INFO, * CONTROL_INFO_PTR;
```

See Also: For more information on handling events related to controls, see section 3.4.2 on page 3-15.

8.3. Descriptions of XVT Controls

This section describes the controls that you can use in XVT. The descriptions cover all information about controls, including components, attributes, event-related information returned to event handlers about the control, manipulation functions, and usage restrictions.

Note: Unless the descriptions specify otherwise, you can set control attributes at definition and/or creation time (i.e., in XRC or in WIN_DEF objects), and your program can change them at runtime.

8.3.1. Push Buttons

Push button controls let the user invoke an action: when a user pushes the button, the application is notified of the action.

Push Button Information

WIN_TYPE	WC_PUSHBUTTON
XRC object name	BUTTON
Restrictions	None

Push Button Attributes

"Default pushbutton"	Set at creation time only in XRC or WIN_DEF structure
Size / location	xvt_vobj_move, xvt_vobj_get_client_rect, or xvt_vobj_get_outer_rect
Label text	xvt_vobj_set_title or xvt_vobj_get_title
Label justification	Set at creation time only, in XRC with *_JUST flags, and in WIN_DEF with CTL_FLAG_*_JUST flags (ignored on some platforms)
Font	xvt_ctl_set_font or xvt_ctl_get_font
Color	xvt_ctl_set_colors or xvt_ctl_get_colors
Visibility	xvt_vobj_set_visible
Enable	xvt_vobj_set_enabled
Focus	xvt_scr_set_focus_vobj or

	xvt_scr_get_focus_vobj
WINDOW info	xvt_win_get_ctl or xvt_vobj_get_type
Parent	xvt_vobj_get_parent
Control ID	xvt_ctl_get_id

Event Handling

```
typedef struct s_ctlinfo {
        WIN_TYPE type;                  /* WC_PUSHBUTTON */
        WINDOW win;                     /* WINDOW of control */
        union {
                ...
                struct s_pushbutton {
                        int reserved;
                                        /* reserved (unused) */
                } pushbutton;
                ...
        } v;
} CONTROL_INFO, * CONTROL_INFO_PTR;
```

No additional control-specific information is needed; an E_CONTROL for a WC_PUSHBUTTON type implies that the push button was pressed.

8.3.2. Check Boxes

Check boxes let the user select from a group of checkable items. The major difference between check boxes and push buttons is that check boxes display a state (checked or unchecked), while push buttons simply provide a selection control with no state attribute.

Check boxes are often visually clustered to represent a set of switches that the user can either turn on or off. However, there is no relationship or automatic behavior among the check box controls in a group; they are completely autonomous.

Check boxes differ from radio buttons (discussed in the next section) in that the user can switch any number of check boxes on or off, while radio buttons allow only one selection per group.

XVT does not automatically set the state of check boxes when the check box control is clicked by the user; you must do this explicitly from the event handler.

Tip: To set the state of check boxes:

Call xvt_ctl_set_checked.

Check Box Information

WIN_TYPE	WC_CHECKBOX
XRC object name	CHECKBOX
Restrictions	None

Check Box Attributes

Checked state	xvt_ctl_set_checked (dynamically and at creation time), xvt_ctl_is_checked
Size / location	xvt_vobj_move, xvt_vobj_get_client_rect, or xvt_vobj_get_outer_rect
Label text	xvt_vobj_set_title or xvt_vobj_get_title
Label justification	Set at creation time only, in XRC via *_JUST flags, and in WIN_DEF via CTL_FLAG_*_JUST flags (ignored on some platforms)
Font	xvt_ctl_set_font or xvt_ctl_get_font
Color	xvt_ctl_set_colors or xvt_ctl_get_colors
Visibility	xvt_vobj_set_visible (also can be set at creation time)
Enable	xvt_vobj_set_enabled (also can be set at creation time)
Focus	xvt_scr_set_focus_vobj or xvt_scr_get_focus_vobj
WINDOW info	xvt_win_get_ctl or xvt_vobj_get_type
Parent	xvt_vobj_get_parent
Control ID	xvt_ctl_get_id

```
typedef struct s_ctlinfo {
        WIN_TYPE type;                  /* WC_CHECKBOX * /
        WINDOW win;                     /* WINDOW of control */
        union {
                ...
                struct s_checkbox {
                        int reserved;
                                        /* reserved (unused) */
                } checkbox;
                ...
        } v;
} CONTROL_INFO, * CONTROL_INFO_PTR;
```

No additional control-specific information is needed; an E_CONTROL for a WC_CHECKBOX type implies that the check box was pressed.

8.3.3. Radio Buttons

Radio buttons are similar to check boxes. Like check boxes, the user can turn radio buttons on or off. However, radio buttons differ from check boxes in two respects: only one radio button in the group can be on (the rest must be off), and radio buttons have a different shape.

The XVT Portability Toolkit does not automatically implement groups of radio buttons. When you create a radio button group, you must maintain information (an array of radio button WINDOWs) indicating which radio buttons belong to that group. Also, make sure the IDs for those radio button controls are continuous. For example:

```
#define RA1 1
#define RA2 2
#define RA3 3
```

XVT-Design automatically generates code to implement radio button groups. See XVT-Design Manual for more details.

When you are notified that the user pushed a radio button control (an E_CONTROL event), you need to pass your radio button group WINDOW array to xvt_ctl_check_radio_button, along with the WINDOW for the radio button that you want to set. This function ensures that the selected radio button is on, and that the other radio buttons in the group are off.

Tip: To convert any XVT control ID to a WINDOW:

Call xvt_win_get_ctl.

Radio Button Information

WIN_TYPE	WC_RADIOBUTTON
XRC object name	RADIOBUTTON
Restrictions	None

Radio Button Attributes

Checked state	xvt_ctl_check_radio_button (dynamically and at creation time), xvt_ctl_is_checked
Size / location	xvt_vobj_move, xvt_vobj_get_client_rect, or xvt_vobj_get_outer_rect
Label text	xvt_vobj_set_title or xvt_vobj_get_title
Label justification	Set at creation time only, in XRC via *_JUST flags, and in WIN_DEF via CTL_FLAG_*_JUST flags (ignored on some platforms)
Font	xvt_ctl_set_font or xvt_ctl_get_font
Color	xvt_ctl_set_colors or xvt_ctl_get_colors
Visibility	xvt_vobj_set_visible (also can be set at creation time)
Enable	xvt_vobj_set_enabled (also can be set at creation time)
Grouping	Set at creation time
Focus	xvt_scr_set_focus_vobj or xvt_scr_get_focus_vobj
WINDOW info	xvt_win_get_ctl or xvt_vobj_get_type
Parent	xvt_vobj_get_parent
Control ID	xvt_ctl_get_id

Event Handling

```
typedef struct s_ctlinfo {
        WIN_TYPE type;                  /* WC_RADIOBUTTON */
        WINDOW win;                     /* WINDOW of control */
        union {
                ...
                struct s_radiobutton {
                        int reserved;
                                        /* reserved (unused) */
                } radiobutton;
                ...
        } v;
} CONTROL_INFO, * CONTROL_INFO_PTR;
```

No additional control-specific information is needed; an E_CONTROL for a WC_RADIOBUTTON implies that the radio button was pressed.

8.3.4. Static Text

Static text controls let you place read-only text strings into a dialog or window. These are not text strings in the XVT graphical text sense, but rather true controls without user interaction attributes.

Static Text Information

WIN_TYPE	WC_TEXT
XRC object name	TEXT
Restrictions	None

Static Text Attributes

Size / location	xvt_vobj_move, xvt_vobj_get_client_rect, or xvt_vobj_get_outer_rect
Text	xvt_vobj_set_title or xvt_vobj_get_title
Text justification	Set at creation time only, in XRC via *_JUST flags, and in WIN_DEF via CTL_FLAG_*_JUST flags (ignored on some platforms)
Font	xvt_ctl_set_font or xvt_ctl_get_font
Color	xvt_ctl_set_colors or xvt_ctl_get_colors

Visibility	xvt_vobj_set_visible
Enable	xvt_vobj_set_enabled (a FALSE value might be ignored on some platforms that do not natively support disabled static text)
WINDOW info	xvt_win_get_ctl or xvt_vobj_get_type
Parent	xvt_vobj_get_parent
Control ID	xvt_ctl_get_id

Event Handling

```
typedef struct s_ctlinfo {
        WIN_TYPE type;                  /* WC_TEXT */
        WINDOW win;                     /* WINDOW of control */
        union {
                ...
                struct s_statictext {
                        int reserved;
                                        /* reserved (unused)*/
                } statictext;
                ...
        } v;
} CONTROL_INFO, * CONTROL_INFO_PTR;
```

No events are generated; users cannot manipulate this control.

8.3.5. Edit Fields

Edit field controls let the user input a text string to the application. These controls vary in their appearance and behavior depending on the native GUI platform. For example, some systems provide small scrollbars for these controls on one or both ends of the control. Also, platforms handle the text scrolling differently.

However, edit field controls always report events whenever the text string is modified or the keyboard focus is gained (or lost).

XVT edit field controls are always one line high, which overrides the rectangle height specified in the creation call.

Edit Field Information

WIN_TYPE	WC_EDIT
XRC object name	EDIT
Restrictions	None

Edit Field Attributes

Size / location	xvt_vobj_move, xvt_vobj_get_client_rect, or xvt_vobj_get_outer_rect
Edit field text	xvt_vobj_set_title or xvt_vobj_get_title
Text justification	Set at creation time only, in XRC via *_JUST flags, and in WIN_DEF via CTL_FLAG_*_JUST flags (ignored on some platforms)
Font	xvt_ctl_set_font or xvt_ctl_get_font
Color	xvt_ctl_set_colors or xvt_ctl_get_colors
Visibility	xvt_vobj_set_visible
Enable	xvt_vobj_set_enabled
Current text selection	xvt_ctl_get_text_sel or xvt_ctl_set_text_sel
Focus	xvt_scr_set_focus_vobj or xvt_scr_get_focus_vobj
WINDOW info	xvt_win_get_ctl or xvt_vobj_get_type
Parent	xvt_vobj_get_parent
Control ID	xvt_ctl_get_id

Event Handling

```
typedef struct s_ctlinfo {
        WIN_TYPE type;              /* WC_EDIT */
        WINDOW win;                 /* WINDOW of control */
        union {
                ...
                struct s_edit {
                    BOOLEAN focus_change;
                                    /* is it a focus change? */
                    BOOLEAN active;
                                    /* if so: gaining focus? */
                } edit;

                struct s_statictext {...}
                ...
        } v;
} CONTROL_INFO, * CONTROL_INFO_PTR;
```

Based on the event information above, you can determine the following:

- If focus_change is FALSE, then the contents of the edit field changed. You can call xvt_vobj_get_title to determine the new contents of the edit field, and compare this with the previous contents of the edit field.
- If focus_change is TRUE, and active is TRUE, then the edit field gained the keyboard focus.
- If focus_change is TRUE, and active is FALSE, then the edit field lost the keyboard focus.

Changing Behavior of Keys with Event Hooks

You can intercept a key sequence in an edit field, to implement a specific action. For example, you might want to make the Enter key terminate input to the edit field and move the user to the "next" field in the dialog or window. To accomplish this, your application must use event hooks. The application must capture the keystrokes from the native windowing system before they are processed by XVT.

Tip: To use event hooks:

Use the ATTR_EVENT_HOOK attribute in the event handler of the dialog or window container of the edit field, like this:

```
long XVT_CALLCONV1 dlg_eh(WINDOW win, EVENT *ep)
{
    switch(ep->type) {
        ...
        case E_CONTROL:
            switch(ep->v.ctl.id) {
                ...
                case EDIT_FIELD_1:
                    if (ep->v.ctl.ci.v.edit.active &&
                    ep->v.ctl.ci.v.edit.focus_change)
                    xvt_vobj_set_attr(NULL_WIN,
                                ATTR_EVENT_HOOK,
                               (long)my_event_hook);
                    if (!ep->v.ctl.ci.v.edit.active &&
                    ep->v.ctl.ci.v.edit.focus_change)
                    xvt_vobj_set_attr(NULL_WIN,
                          AT      TR_EVENT_HOOK, NULL);
                    break;
                ...
            }
        ...
    }
    return (0L);
}
```

Within the function my_event_hook, you perform the platform-specific operations to process the keystrokes in the edit field. For example, your code might watch for an ENTER keystroke, then call:

```
xvt_scr_set_focus_vobj(next_ec);
```

where next_ec is of type WINDOW and refers to the next control to which you want to navigate.

See Also: For details about how to structure your my_event_hook program, see ATTR_EVENT_HOOK in the "Non-Portable Attributes and Escape Functions" appendix of the appropriate *XVT Platform-Specific Book*.

8.3.6. List Boxes

List boxes let the user make single or multiple selections from a scrollable list of candidate selections. List boxes generate E_CONTROL events to yourdialog or window event handler when the user clicks or double-clicks on an item in the list box. You won't receive any events in your application when the user scrolls the list box; this behavior is handled automatically by XVT via the native list box control.

Many XVT functions, all starting with the prefix xvt_list_*, work with list boxes (as well as other list-oriented controls, such as combo controls).

SLISTs

To specify text strings for a list box and retrieve single or multiple selections, you use an XVT data type called an SLIST (short for String LIST). SLISTs are abstracted lists of strings, where each element contains a string pointer, anindex (starting at 0), and a long data value.

Several XVT functions, all prefixed with xvt_slist_*, manipulate SLISTs.

When you add an SLIST to a list box, only the strings are maintained. List boxes cannot maintain the data pointers associated with each SLIST element. When xvt_list_get_sel is called, the long data words of the returned SLIST are filled withthe element's index in the list box.

Note: SLISTs cannot contain carriage returns, line feeds, or new lines.

See Also: For a list of the functions you can use to manipulate SLISTs, see section B.1 in Appendix B, *Utilities*.

List Box Information

WIN_TYPE	WC_LBOX
XRC object name	LISTBOX
Restrictions	None

List Box Attributes

Size / location	xvt_vobj_move, xvt_vobj_get_client_rect, or xvt_vobj_get_outer_rect
Selection mode	Single (default), multiple or read only (all defined at creation time only)
Font	xvt_ctl_set_font or xvt_ctl_get_font
Color	xvt_ctl_set_colors or xvt_ctl_get_colors
Visibility	xvt_vobj_set_visible
Enable	xvt_vobj_set_enabled
Current selection(s)	xvt_list_get_sel, xvt_list_get_first_sel, xvt_list_is_sel, xvt_list_get_sel_index, or xvt_list_set_sel
List count	xvt_list_count_all or xvt_list_count_sel
List contents	xvt_list_add, xvt_list_clear, xvt_list_get_all, xvt_list_rem, or xvt_list_get_elt
List updating	xvt_list_suspend or xvt_list_resume
Focus	xvt_scr_set_focus_vobj or xvt_scr_get_focus_vobj
WINDOW info	xvt_win_get_ctl or xvt_vobj_get_type
Parent	xvt_vobj_get_parent
Control ID	xvt_ctl_get_id

Event Handling

```
typedef struct s_ctlinfo {
        WIN_TYPE type;          /* WC_LBOX */
        WINDOW win;             /* WINDOW of control */
        union {
                ...
                struct s_lbox {
                                /* list box action */
                BOOLEAN dbl_click;
                                /* double click (vs. single) */
                } lbox;
                ...
        } v;
} CONTROL_INFO, * CONTROL_INFO_PTR;
```

The dbl_click field tells whether the user double-clicked in the list box, or simply single-clicked. No other events are reported for this control.

Implementation Note: According to Macintosh look-and-feel guidelines, clicking below the last item in a list box causes an E_CONTROL event to be delivered, and the application must clear any selection. On all other platforms, no event is generated.

Example: Because list boxes can be complex, here is an example of how you might write part of an XVT application using list boxes. In this example, an XRC-defined dialog holds a single-selection list box containing eight items. When a user clicks one of the list box items, the code calls xvt_scr_beep, which signals that a single-click selection was made. A double-click signals the application to get the selection that was double-clicked, then close the dialog.

First, here is the XRC code for defining the dialog and list box control:

```
/* XRC dialog and control definitions */
...
#define DIALOG_11000
#define LISTBOX_11001
...
DIALOG DIALOG_1,100, 100, 400, 500
 "Sample Dialog" MODELESS
        LISTBOX LISTBOX_1, 30, 30, 200, 300
...
```

Here is the dialog and list box creation code, and the dialog's event handler:

```
...
xvt_dlg_create_res (WD_MODELESS, DIALOG_1, EM_ALL,
        dialog_1_eh, 0L);

long XVT_CALLCONV1 dialog_1_eh(WINDOW win, EVENT *ep);
{
        SLIST slist;
        WINDOW lbox_win;

        switch (ep->type) {
        ...
        case E_CREATE:

                /* This is the best place to fill the list box
                */
                lbox_win = xvt_win_get_ctl(win, LISTBOX_1);

                slist = xvt_slist_create;
                xvt_slist_add_at_elt (slist, (SLIST_ELT)NULL,
                        "Hawaii", 0L);
                xvt_slist_add_at_elt (slist, (SLIST_ELT)NULL,
                        "Alaska", 0L);
                xvt_slist_add_at_elt (slist, (SLIST_ELT)NULL,
                        "Texas", 0L);
                xvt_slist_add_at_elt (slist, (SLIST_ELT)NULL,
                        "Wisconsin", 0L);
                xvt_slist_add_at_elt (slist, (SLIST_ELT)NULL,
                        "New York", 0L);
                xvt_slist_add_at_elt (slist, (SLIST_ELT)NULL,
                        "Colorado", 0L);
                xvt_slist_add_at_elt (slist, (SLIST_ELT)NULL,
                        "Washington", 0L);
                xvt_slist_add_at_elt (slist, (SLIST_ELT)NULL,
                        "Florida", 0L);

                xvt_list_add (lbox_win, -1, slist);
                xvt_slist_destroy (slist);
                break;

        case E_CONTROL:

                if (ep->v.ctl.id == LISTBOX_1) {
                        if (ep->v.ctl.ci.v.lbox.dbl_click ==
                                        TRUE) {
                                sel_slist = xvt_list_get_sel
                                        (xvt_win_get_ctl(win, LISTBOX_1));
                                xvt_vobj_destroy (win);
                        }
                        else
                                xvt_scr_beep();
                }
                break;
                ...
                }
        return (0L);
}
```

8.3.7. Scrollbars

Horizontal and vertical scrollbar controls are similar in many ways to the scrollbars that you can define as part of a window's border decorations. The only difference is that they are in fact controls, not border decorations. However, you can handle them the same as window border scrollbars.

The what field, of type SCROLL_CONTROL, indicates which part of the scrollbar was operated, as shown below:

```
typedef enum {              /* site of scrollbar act */
      SC_NONE,              /* nowhere (event ignored) */
      SC_LINE_UP,           /* one line up */
      SC_LINE_DOWN,         /* one line down */
      SC_PAGE_UP,           /* previous page */
      SC_PAGE_DOWN,         /* next page */
      SC_THUMB,             /* thumb repositioning */
      SC_THUMBTRACK         /* dynamic thumb tracking */
} SCROLL_CONTROL;
```

The interpretation of line and page is entirely up to your application. Each individual click on the scrollbar generates a separate E_CONTROL event. If the user holds the mouse button down, a sequence of events occurs.

The what field is equal to SC_THUMBTRACK while the user drags the thumb, and SC_THUMB when the user stops dragging. In these cases, the pos field indicates the current position of the thumb relative to the range of the scrollbar. The range must have been previously set with a call to xvt_sbar_set_range. If no such call was made, the range is undefined, so pos is meaningless. If the call was made, pos is relative to the range for the scrollbar.

Scrollbar Information

WIN_TYPE	WC_VSCROLL WC_HSCROLL
XRC object name	SCROLLBAR
Restrictions	None

Scrollbar Attributes

Size / location	xvt_vobj_move, xvt_vobj_get_client_rect, or xvt_vobj_get_outer_rect
Color	xvt_ctl_set_colors or xvt_ctl_get_colors
Visibility	xvt_vobj_set_visible
Enable	xvt_vobj_set_enabled
Scrollbar thumb position	xvt_sbar_set_pos or xvt_sbar_get_pos
Scrollbar range	xvt_sbar_set_range or xvt_sbar_get_range
Scrollbar proportion indicator	xvt_sbar_set_proportion or xvt_sbar_get_proportion
Focus	xvt_scr_set_focus_vobj or xvt_scr_get_focus_vobj
WINDOW info	xvt_win_get_ctl or xvt_vobj_get_type
Parent	xvt_vobj_get_parent
Control ID	xvt_ctl_get_id

Event Handling

```
typedef struct s_ctlinfo {
        WIN_TYPE type;          /* WC_VSCROLL or WC_HSCROLL */
        WINDOW win;             /* WINDOW of control */
 un       ion {
                ...
        struct s_scroll {
                                /* scrollbar action */
    SCROLL          _CONTROL what;
                                /* site of activity */
            short pos;          /* thumb position */
 }          scroll;
                ...
 }          v;
} CONTROL_INFO, * CONTROL_INFO_PTR;
```

The what field indicates which part of the scrollbar control was manipulated. The pos field indicates the new position of the scrollbar thumb (when what is SC_THUMB*).

8.3.8. List Button

An XVT list button control is a combination of two other control types—a push button and a selection list. (Such controls are sometimes referred to as "combo controls" for this reason.) A list button can be described as a list box that can be displayed in two ways:

- As a push button whose text label represents the current selection in the list (when the control is not being used)
- As a list box (when the control is being used)

The list box part of the list button is transitory—it appears only when the list button is pressed. When the user selects from the list, the list box part of the control disappears, leaving the selected text in the list button. (If the list button list is empty, then the list button label is also empty.)

The events generated from list buttons are similar to those generated from list boxes except that, because double-clicks aren't supported in list buttons, the event merely signals that the user made a selection from the list.

For list boxes, XVT supports all of the xvt_list_* functions supported for list buttons, so list buttons utilize SLIST objects in the same way as do list boxes.

An XVT list button control's static portion is always one line high. The height specified in the creation call determines the height of the control with the list portion visible.

List Button Information

WIN_TYPE	WC_LISTBUTTON
XRC object name	LISTBUTTON
Restrictions	xvt_ctl_set_text_sel cannot be used; no concept of a "selection" exists in terms of the text label in the button

List Button Attributes

Size / location	xvt_vobj_move, xvt_vobj_get_client_rect, or xvt_vobj_get_outer_rect
Selection mode	always single select mode (not settable)
Label text	xvt_list_get_sel or

	xvt_list_set_sel
Label justification	Set at creation time only, in XRC via *_JUST flags, and in WIN_DEF via CTL_FLAG_*_JUST flags (ignored on some platforms)
Font	xvt_ctl_set_font or xvt_ctl_get_font
Color	xvt_ctl_set_colors or xvt_ctl_get_colors
Visibility	xvt_vobj_set_visible
Enable	xvt_vobj_set_enabled
Current selection(s)	xvt_list_get_sel, xvt_list_get_first_sel, xvt_list_is_sel, xvt_list_get_sel_index, or xvt_list_set_sel
List count	xvt_list_count_all or xvt_list_count_sel
List contents	xvt_list_add, xvt_list_clear, xvt_list_get_all, xvt_list_rem, or xvt_list_get_elt
List updating	xvt_list_suspend or xvt_list_resume
Focus	xvt_scr_set_focus_vobj or xvt_scr_get_focus_vobj
WINDOW info	xvt_win_get_ctl or xvt_vobj_get_type
Parent	xvt_vobj_get_parent
Control ID	xvt_ctl_get_id

Event Handling

```
typedef struct s_ctlinfo {
        WIN_TYPE type;              /* WC_LISTBUTTON */
        WINDOW win;                 /* WINDOW of control */
        union {
                ...
                struct s_listbutton {
                        int reserved;
                                    /* reserved (unused) */
                } listbutton;
                ...
        } v;
} CONTROL_INFO, * CONTROL_INFO_PTR;
```

No additional control-specific information needed; an E_CONTROL for a WC_LISTBUTTON type implies that a selection was made from the list component. Use one of the xvt_list_get_* functions to get selection information.

Miscellaneous Information About List Buttons

Here are some additional considerations regarding XVT list buttons:

- A list selection made by either the user or the application updates the button text portion automatically.
- The text label in the button always reflects the current list selection. If there is no selection, or if the list is empty, no text is displayed in the text portion.
- Deleting a list item that is the current selection, by definition, results in an empty text label. Deleting other non-selected items does not change the text label of the control.
- Adding items to the list does not cause those items to be the current selection. The exception to this is when the first list item is added. In this case, the first item is automatically selected and thus displayed.
- The current selection of a list is maintained after the list itself disappears. In other words, the program can inquire and/or change the selection at any time, even when the list is not visible.
- The list button bounding rectangle describes the control area as if the list component were being displayed.

8.3.9. List Edit

An XVT list edit control is a combination of two other control types: an edit field, and a selection list. (Such controls are sometimes referred to as "combo controls" for this reason.) A list edit is an edit field control that possesses an alternate, or shorthand, text input source: a pop-up list of text items. The user can operate a list edit control in two ways:

- By making a selection from the list (which is then inserted into the edit field component)
- By directly editing the edit field's contents component

The list box part of the list edit is transitory; it appears only when the part of the list edit control that displays the list component is pressed.

If the list edit list component is empty, the list button edit field can still contain text, and doesn't have to be empty.

The events generated from list edits are quite similar to those generated for edit field controls. This is because you are notified when the control gains or loses the keyboard focus and when the edit field contents are modified. In addition, list edit events are generated whenever the user makes a selection from the pop-up list.

Tip: To determine the text string displayed in the edit field component:

Call xvt_vobj_get_title.

Most of the xvt_list_* functions supported for list boxes and list buttons are also supported for list edits.

List edits utilize the SLIST objects in the same way as do list boxes and list buttons. However, there is one important distinction: list edits have no concept of a current list selection. The list component is secondary to the edit field in the case of list edit controls. Because of this, you can't use any of the xvt_list_*_sel list selection-related functions with list edit controls.

An XVT list edit control's static portion is always one line high. The height specified in the creationcall determines the height ofthe control with the selection list portion visible.

List Edit Information

WIN_TYPE	WC_LISTEDIT
XRC object name	LISTEDIT
Restrictions	You can't use any of the xvt_list_*_sel* selection-oriented functions with list edits; XVT errors result if you attempt this

List Edit Attributes

Size / location	xvt_vobj_move, xvt_vobj_get_client_rect, xvt_vobj_get_outer_rect
Selection mode	always single select mode (not settable)
Label text	xvt_vobj_set_title / xvt_vobj_get_title
Font	xvt_ctl_set_font or xvt_ctl_get_font

Color	xvt_ctl_set_colors or xvt_ctl_get_colors
Visibility	xvt_vobj_set_visible
Current text selection	xvt_ctl_get_text_sel or xvt_ctl_set_text_sel
List count	xvt_list_count_all
List contents	xvt_list_add, xvt_list_clear, xvt_list_get_all, xvt_list_rem, or xvt_list_get_elt
List updating	xvt_list_suspend or xvt_list_resume
Enable	xvt_vobj_set_enabled
Text justification	Set at creation time only, in XRC via *_JUST flags, and in WIN_DEF via CTL_FLAG_*_JUST flags (ignored on some platforms)
Focus	xvt_scr_set_focus_vobj or xvt_scr_get_focus_vobj
WINDOW info	xvt_win_get_ctl or xvt_vobj_get_type
Parent	xvt_vobj_get_parent
Control ID	xvt_ctl_get_id

Event Handling

```
typedef struct s_ctlinfo {
        WIN_TYPE type;          /* WC_LISTEDIT */
        WINDOW win;             /* WINDOW of control */
        union {
                ...
                struct s_listedit {
                    BOOLEAN focus_change;
                            /* did the edit field
                                    part change focus? */
                    BOOLEAN active;
                            /* if so, focus gained? */
                } listedit;
                ...
        } v;
} CONTROL_INFO, * CONTROL_INFO_PTR;
```

Based on the list edit event information in the table above, you can also determine the following:

- If focus_change is FALSE, then the contents of the edit field changed. You can call xvt_vobj_get_title to determine the new contents of the edit field, and compare this with the previous contents of the edit field.
- If focus_change is TRUE, and active is TRUE, then the edit field gained the keyboard focus.
- If focus_change is TRUE, and active is FALSE, then the edit field lost the keyboard focus.

Miscellaneous Information for List Edits

Here are some additional considerations to keep in mind when using XVT list edits:

- A list selection made by either the user or the application updates the edit field component automatically. The entire edit field contents become selected.
- Calling xvt_vobj_set_title on a list edit sets the contents of the edit field, but does*not* automatically select the contents of the edit field.
- Typing into the edit field component of a list edit control may cause a matching item in the list to be automatically selected. (Whether the matching item is selected depends on the look-and-feel of the particular native GUI toolkit.)
- E_CONTROL events with focus change information might not be sent when the focus is changed between the edit field and list components of list edit controls.
- xvt_ctl_set_text_sel sets the selection or the insertion point for the edit field part of a list edit. Note that calling xvt_ctl_set_text_sel also sets the input focus to the control.

8.3.10. Group Boxes

XVT group box controls provide a way to draw an annotated rectangle around (and behind) a group of controls in a window or dialog. The group box rectangle has an embedded label or title, which appears on the upper line of the rectangle, and can be either left, centered, or right depending on the text justification flags for the control.

A group box defines those controls that are within its boundaries as a set. This does not imply that a group box is the parent of controls contained within it; no such relationship exists in XVT. Group boxes are like static text in that they provide no interaction capability or subsequent events; they are for annotation purposes only.

XVT automatically places group boxes at the back of a dialog or window, behind all other controls. Overlapping group boxes are not supported.

Note: You can use group boxes with any controls except XVT's text edit objects, which are not considered native controls on any platforms.

Group Box Information

WIN_TYPE	WC_GROUPBOX
XRC object name	GROUPBOX
Restrictions	You cannot call xvt_scr_set_focus_vobj for group boxes. In addition, nested group boxes are not allowed.

Group Box Attributes

Size / location	xvt_vobj_move, xvt_vobj_get_client_rect, or xvt_vobj_get_outer_rect
Annotation text	xvt_vobj_set_title or xvt_vobj_get_title
Annotation justification	Set at creation time only, in XRC via *_JUST flags, and in WIN_DEF via CTL_FLAG_*_JUST flags (ignored on some platforms)
Font	xvt_ctl_set_font or xvt_ctl_get_font

Color	xvt_ctl_set_colors or xvt_ctl_get_colors
Visibility	xvt_vobj_set_visible (platform-specific, can be ignored)
Enable	xvt_vobj_set_enabled (platform-specific, can be ignored)
WINDOW info	xvt_win_get_ctl or xvt_vobj_get_type
Parent	xvt_vobj_get_parent
Control ID	xvt_ctl_get_id

Event Handling

```
typedef struct s_ctlinfo {
        WIN_TYPE type;              /* WC_GROUPBOX */
        WINDOW win;                 /* WINDOW of control */
        union {
                ...
                struct s_groupbox {
                        int reserved;
                                    /* reserved (unused) */
                } groupbox;
                ...
        } v;
} CONTROL_INFO, * CONTROL_INFO_PTR;
```

No events are generated; users cannot manipulate this control.

8.3.11. Notebooks

Notebook controls allow the user to define multiple child windows for the same area of a window. Figure 8.2 shows a Win32 notebook.

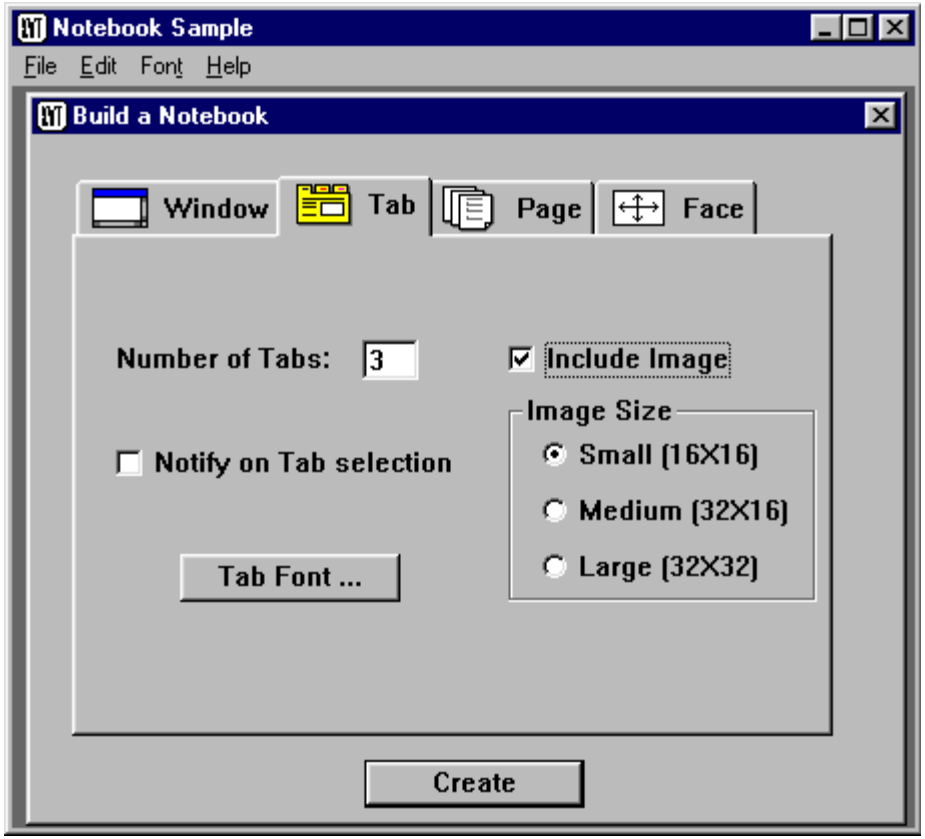


Figure 8.2. Win32 notebook control

A notebook is made up of a hierarchy of three different objects: tabs, pages, and faces (XVT WINDOW), as shown in Figure 8.3. It is simply a container for it's tabs.

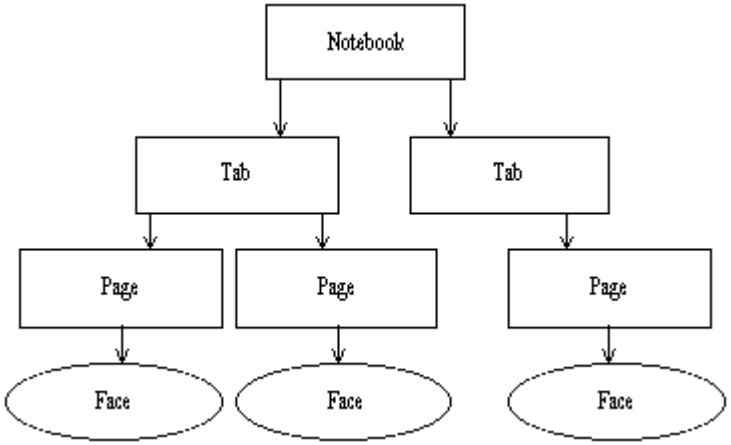


Figure 8.3. Notebook hierarchy

A tab has a visual button the user clicks to select the tab. Each tab can have multiple pages.

A page is conceptual. A page can have one face. If a tab has more than one page, the programmer must come up with a way for the user to change pages. For example, you may provide "Next" and "Prev" buttons or a list button with page titles on each face for tabs with multiple pages.

A face is a standard XVT child window.

8.3.11.1. Notebook Creation

The notebook control is more complicated to create than most other XVT controls. Notebook controls can be created dynamically, or they may be structure based. They cannot be created from resources. (See section 8.1.) Once the notebook has been created, the other objects (tabs, pages, and faces) must be created dynamically. Without one or more tabs, a notebook is not useful.

Creating Tabs

The function xvt_notebk_add_tab will create a tab for the specified notebook control. Tabs are identified by an index that represents their order in the notebook control (0 based). Without one or more pages, a tab is not useful.

Creating Pages

The function xvt_notebk_add_page will create a page for the specified tab within the specified notebook control. Pages are identified by an index that represents their order inthe tab (0 based). Without a face, a page is not useful.

Creating Faces

There are three functions for creating faces: xvt_notebk_create_face for dynamic creation, xvt_notebk_create_face_def for structure-based creation, and xvt_notebk_create_face_res for resource-based creation. A face is created for a specific page in a specific tab in a specific notebook. Since each face is an XVT child window, each has it's own event handler.

Example: The following code will create the notebook in Figures 8.2 and 8.3 above:

```
RCT aRct;
WINDOW aNotebk;
WIN_DEF *aFace;
XVT_IMAGE anImage = NULL;

/* Build notebook control */
xvt_rect_set(&aRct, 20, 20, 400, 300);
aNotebk = xvt_ctl_create(WC_NOTEBK, &aRct,
        "Notebook", xdWindow, 0L, 0L, 2);

/* Build Window Tab */
anImage =  xvt_image_read_bmp("window.bmp");
xvt_notebk_add_tab (aNotebk, 0, "Window", anImage);
xvt_notebk_add_page(aNotebk, 0, 0, "Page0",0L);
aFace = xvt_res_get_win_def(WINDOW_FACE);
xvt_notebk_create_face_def(aNotebk, 0, 0,
        aFace, EM_ALL, WINDOW_FACE_eh, 0L);
xvt_res_free_win_def(aFace);
xvt_image_destroy(anImage);

/* Build Tab Tab */
anImage =  xvt_image_read_bmp("tab.bmp");
xvt_notebk_add_tab (aNotebk, 1, "Tab", anImage);
xvt_notebk_add_page(aNotebk, 1, 0, "Page0",0L);
aFace = xvt_res_get_win_def(TAB_FACE);
xvt_notebk_create_face_def(aNotebk, 1, 0,
        aFace, EM_ALL, TAB_FACE_eh, 0L);
xvt_res_free_win_def(aFace);
xvt_image_destroy(anImage);

/* Build Page Tab */
anImage =  xvt_image_read_bmp("page.bmp");
xvt_notebk_add_tab (aNotebk, 2, "Page", anImage);
xvt_notebk_add_page(aNotebk, 2, 0, "Page0",0L);
aFace = xvt_res_get_win_def(PAGE_FACE);
xvt_notebk_create_face_def(aNotebk, 2, 0,
        aFace, EM_ALL, PAGE_FACE_eh, 0L);
xvt_res_free_win_def(aFace);
xvt_image_destroy(anImage);

/* Build Face Tab */
anImage =  xvt_image_read_bmp("face.bmp");
xvt_notebk_add_tab (aNotebk, 3, "Face", anImage);
xvt_notebk_add_page(aNotebk, 3, 0, "Page0",0L);
aFace = xvt_res_get_win_def(FACE_FACE);
xvt_notebk_create_face_def(aNotebk, 3, 0,
        aFace, EM_ALL, FACE_FACE_eh, 0L);
xvt_res_free_win_def(aFace);
xvt_image_destroy(anImage);

/* Activate first tab, first page. */
xvt_notebk_set_front_page(aNotebk, 0, 0);
```

Notebook Information

WIN_TYPE	WC_NOTEBK
XRC object name	Not applicable
Restrictions	Cannot be created in dialogs

Notebook Attributes

Size / location	xvt_vobj_move, xvt_vobj_get_client_rect, or xvt_vobj_get_outer_rect
Font	xvt_ctl_set_font (changes all tab fonts)
Color	xvt_ctl_set_colors or xvt_ctl_get_colors (not supported on XVT/Win32)
Visibility	xvt_vobj_set_visible
Enable	xvt_vobj_set_enabled
Focus	xvt_scr_set_focus_vobj or xvt_scr_get_focus_vobj
WINDOW info	xvt_win_get_ctl or xvt_vobj_get_type
Parent	xvt_vobj_get_parent
Control ID	xvt_ctl_get_id

Event Handling

```
typedef struct s_ctlinfo {
        WIN_TYPE type;          /* WC_NOTEBK */
        WINDOW win;             /* WINDOW id of the control */
        union {
                …
                struct s_notebk {
                        WINDOW face;
                        short  tab_no;
                        short  page_no;
                } notebk;
        } v;
} CONTROL_INFO, *CONTROL_INFO_PTR;
```

An event for the notebook control is sent when the user selects a tab. The face field is the face that currently has focus. The tab_no field is the tab that was selected. The page_no field is the front page for the notebook (face is the face for the current page).

Events for a face will go to the face's event handler.

Miscellaneous Information About Notebooks

The following work with tabs, pages, and faces within a notebook control:

Add a page to a specific tab	xvt_notebk_add_page
Add a tab to a notebook control	xvt_notebk_add_tab
Create a face for a page dynamically	xvt_notebk_create_face
Create a face from an array of data structures	xvt_notebk_create_face_def
Create a face from a resource file	xvt_notebk_create_face_res
Enumerate through all the pages and apply a function to each page	xvt_notebk_enum_pages
Get the face in the notebk at tab and page	xvt_notebk_get_face
Get the front page for a notebook	xvt_notebk_get_front_page
Get the number of pages in the tab specified	xvt_notebk_get_num_pages
Get the number of tabs in a notebk	xvt_notebk_get_num_tabs
Get the data associated with a page	xvt_notebk_get_page_data
Get the page, tab, andnotebk associated with a specific face	xvt_notebk_get_page_from_face
Get the page title	xvt_notebk_get_page_title
Get the image for a tab	xvt_notebk_get_tab_image
Get the title for a tab	xvt_notebk_get_tab_title
Remove a page attached to a tab	xvt_notebk_rem_page
Remove a tab	xvt_notebk_rem_tab
Set the data for a page	xvt_notebk_set_page_data
Set the title for a page	xvt_notebk_set_page_title
Set the front page	xvt_notebk_set_front_page
Set the tab image	xvt_notebk_set_tab_image
Set the tab title	xvt_notebk_set_tab_title

8.3.12. HTML Controls

HTML controls give XVT applications the ability to draw text and images in a window using HTML data. The file location for the data is specified with a well-formed Universal Resource Locator (URL) string containing either a local file path or an Internet address. The HTML control is a WINDOW of type WC_HTML.

HTML is a common format and the ability to leverage HTML within production applications is a powerful feature that extends the utility of an XVT application. The types of tasks you can perform with the HTML control and the HTML Rendering library include:

- Drawing HTML data in application windows
- Working with URL events

- Formatting and updating the rendering area with such elements as borders and scroll bars
- Obtaining information about pages
- Launching a native browser from an XVT application

Note: It is important to remember that the XVT HTML control, while useful, is not a full Internet browser; it does not contain advanced features such as history and Plug-in support.

Platform Differences

As with most XVT controls, the HTML control relies on the underlying native platform implementation of the control. The XVT application will reflect those native differences. It is important to remember that the intrinsic operation of the application will be maintained across the different platforms, even though the manifestation of the behavior may vary.

In the case of the HTML control, XVT/Win32 renders both local and external/Internet HTML files in the same lightweight COM object window that is embedded in the application. On the Macintosh platform, local URLs are displayed in the application, while external URLs, 'mailto' and default FTP clients for 'ftp://' activities will be displayed in the default system browser. The *Platform-Specific Book* for each native platform contains additional, specific information on the implementation.

Activities

Rendering HTML is an automatic, encapsulated activity, completely handled by the native control. There is little need to intervene in the drawing process. The duties of the XVT application are to set the URL to be rendered and query the current URL setting.

The application can also define a URL intercept handler. This is useful when the application neeeds to maintain a history list of visited URLs or when needing to impose a degree of control over the URL value. If a URL intercept handler is defined, the application is notified, through its intercept handler whenever a URL is to be set on the HTML control; either by the application setting the URL, or when the user follows a link in the displayed HTML.

Note: The following list is unique behavior for the HTML control:

- If the HTML rendered in the control doesn't fit within the client area, scrollbars will appear if necessary.

- The HTML control cannot gain focus. xvt_vobj_is_focusable will always return FALSE.
- The client and outer rectangles are the same.
- xvt_vobj_set_title has no effect, xvt_vobj_get_title returns the title of the HTML page as it would appear at the top of a browser.

HTML Rendering Library Functions

The following functions are available in the HTML Rendering Library:

xvt_html_get_url

Get the URL value of an HTML Control

xvt_html_set_url

Set the URL value of the HTML Control, causing the HTML to be displayed

xvt_html_get_url_intercept

Retrieve the URL Intercept Handler for HTML Control

xvt_html_set_url_intercept

Set the URL Intercept Handler for HTML Control

xvt_scr_launch_browser

Launch the OS Default Web Browser

xvt_html_refresh

Request a refresh of the currently active page in the browser

xvt_html_home

Request the browser navigate to the system's home page

xvt_html_back

Go to the previous page visited, if there is one

xvt_html_forward

Go to the following page, if there is one

xvt_html_stop

Tell the browser to stop trying to load the page

xvt_html_search

Request the browser navigate to the system's search page

These commands can be used to:

- Build a history list
- Redirect URLs
- Temporarily override the intercept handler by saving the current handler and restoring it later with a call to xvt_html_set_url_intercept.
- Retrieve the current intercept handler from one HTML control and assign it to another HTML control
- Save the intercept handler, assign a new intercept handler, and "preprocess" URLs with the new intercept handler before invoking the original intercept handler. This technique can be used to effectively chain together a series of intercept handlers.

Example: This code uses xvt_html_get_url_intercept to get the URL intercept handler so that it can be assigned to another HTML control.

```
/* Get the URL intercept handler */
XVT_HTML_URL_INTERCEPT_HANDLER urlIH =
xvt_html_get_url_intercept(myHTMLCtl);
xvt_html_set_url_intercept(myNewHTMLCtl, urlIH);
```

Example: This code uses xvt_html_set_url_intercept to set an URL intercept handler to redirect URLs.

```
BOOLEAN myInterceptHdlr(WINDOW win, char **url)
{
    char localURL[] = "http://www.xvt.com";
    char errURL[] = "file://c:/my_app/errpage.htm";


    /* If URL is not local, redirect to error page. */
    if (strncmp(*url, localURL, sizeof(localURL)) == 0)
    {
        /* url was allocated using xvt_mem_alloc.
           According to documentation, if we want to
           change url, it must be freed using
           xvt_mem_free to avoid memory leaks. */
       xvt_mem_free(*url);
```

```
        /*  Allocate memory for url based on the length of
            our new URL */
         *url = xvt_mem_alloc(sizeof(errURL));

         /* Copy the new URL into url */
         strcpy(*url, errURL);
    }
   /* Returning TRUE notifies calling function to process
       the URL in url Returning FALSE notifies the
       calling function not to process the URL in url */
    return TRUE;
}


long XVT_CALLCONV1 myWindow_eh(WINDOW win, EVENT *ep)
{
    switch(ep->type)
    {
        case E_CREATE:
        ...
        xvt_html_set_url_intercept(xvt_win_get_ctl(win,
           HTML_CTL), myInterceptHdlr);
        ...
        break;
...
}
return (*save_eh)(win, ep);
}
```

Detailed information on the xvt_html_* functions can be found in the *XVT Portability Toolkit Reference*.

8.3.13. Icons

XVT icon controls let you display platform-specific icons in dialogs and windows. The actual description (or resource definition) of an icon is handled differently for each XVT platform. However, once icons are described, XVT can portably include them in windows and dialogs, as well as supporting some types of operations on them.

When creating icon controls with xvt_win_create_def and xvt_ctl_create_def, you use WIN_DEF data structures. The WIN_DEF structure contains a substructure used for icon control definitions:

```
typedef struct s_win_def {
        WIN_TYPE wtype;
        RCT rct;
        char *text;
        UNIT_TYPE units;
        union {
                ...
                struct {
                        int ctrl_id;
                        int icon_id;
                        long flags;
                } ctl;
                ...
        } v;
} WIN_DEF;
```

When creating icon controls, set the icon_id field to the ID of an icon resource. You must define the icon resource non-portably in your XRC or native resource file.

Icon controls do not generate events in XVT, and cannot be assigned the keyboard focus.

See Also: For more information about resource definitions for icons, see the *XVT Platform-Specific Books*.

Icon Information

WIN_TYPE	WC_ICON
XRC object name	ICON
Restrictions	See caveats below

Icon Attributes

Size / location	xvt_vobj_move, xvt_vobj_get_client_rect, or xvt_vobj_get_outer_rect
Visibility	xvt_vobj_set_visible
WINDOW info	xvt_win_get_ctl or xvt_vobj_get_type
Parent	xvt_vobj_get_parent
Control ID	xvt_ctl_get_id

Here are some caveats regarding icon controls:

- xvt_vobj_destroy is supported only for icons in windows.
- Icon controls in windows can only be created through xvt_ctl_create_def, xvt_win_create_def, and xvt_win_create_res. The xvt_ctl_create function cannot be used, because it lacks the arguments needed to specify the resource ID of the icon.
- Icon controls can be created in dialogs via xvt_dlg_create_res and xvt_dlg_create_def.

Event Handling

```
typedef struct s_ctlinfo {
        WIN_TYPE type;              /* WC_ICON */
        WINDOW win;                 /* WINDOW of control */
        union {
                ...
                struct s_icon {
                    int reserved;
                                    /* reserved (unused) */
                } icon;
                ...
        } v;
} CONTROL_INFO, * CONTROL_INFO_PTR;
```

Icons cannot generate events, because the user cannot manipulate them.

8.3.14. Text Edit Objects

An XVT text edit object is a fully functional, multi-line edit control supporting many of the functions of other controls. These functions include:

xvt_ctl_get_colors	xvt_vobj_get_client_rect
xvt_ctl_get_id	xvt_vobj_get_data
xvt_ctl_get_font	xvt_vobj_get_outer_rect
xvt_ctl_set_colors	xvt_vobj_get_parent
xvt_ctl_get_type	xvt_vobj_is_focusable
xvt_ctl_set_font	xvt_vobj_raise
xvt_scr_set_focus_vobj	xvt_vobj_set_data
xvt_scr_get_focus_vobj	xvt_vobj_set_enabled
xvt_vobj_move	xvt_vobj_translate_points

Strictly speaking, text edit objectsare not native controls at all. They are implemented on top of the XVT API, using graphics primitives, graphics text, and XVT windows to create the object components. This is why you can place text edit objects only in windows, not in dialogs.

Text Edit Information

WIN_TYPE	WC_TEXTEDIT
XRC object name	TEXTEDIT

8.3.14.1. Text Edit Capabilities

The text edit system is quite flexible. It allows you to:

- Insert and manipulate the text from your program, or you can let the user edit the text with the usual keyboard and mouse conventions
- Control whether the text is wrapped to a margin
- Have a border around the text edit area
- Limit the number of charactersper paragraph (useful in form entry applications), and limit the text to one paragraph
- Force the text to be read-only
- Control whether cut, copy, or paste are allowed
- Control whether horizontal and vertical auto-scrolling are enabled
- Change any attributes (wrapping, margin, physical font, etc.) at any time

Because of its flexibility, you canuse the text edit system for a wide variety of applications. Two typical applications are form entry, when you have several, completely unrelated text edit areas per window (Figure 8.4), and file editing (with or without wrapping), in which the document occupies the entire window (Figure 8.5).

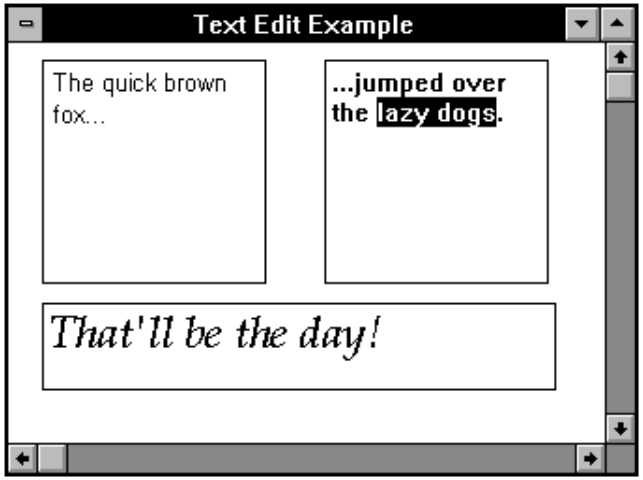


Figure 8.4. Three text edit objects in a window

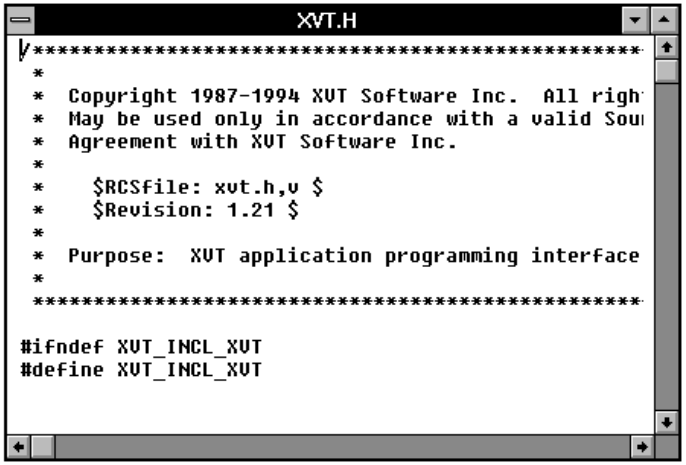


Figure 8.5. A window showing part of a single, borderless text edit object

8.3.14.2. Text Edit Terminology and Geometry

Text Edit Objects

A text edit object consists of text divided into paragraphs, each terminated with a carriage return. If word wrapping is enabled, each paragraph can appear as one or more lines of text.

View Rectangle

The length of a line and the total number of lines can be very large, so normally not all of the text appears on the screen. Only the part within the view rectangle can be seen. The application program or the user can scroll the text so that the unseen part enters the view rectangle.

Border Rectangles

Optionally, you can request the text edit system to draw a border rectangle around the view rectangle. If you do so, the view rectangle is inset by 4 pixels inside the border rectangle. On the bottom it may be inset by a few more pixels, because the bottom coordinate of the view rectangle that you request may be reduced so that an integral number of text lines will appear.

Coordinates

Coordinates for the border and view rectangles are always window-relative. Although each text edit object looks like a window, it is not.

Figure 8.6 shows the relationships between the screen, the window, and the border and view rectangles of a text edit object. The text around the screen indicates that only a fraction of the text actually shows in the view rectangle.

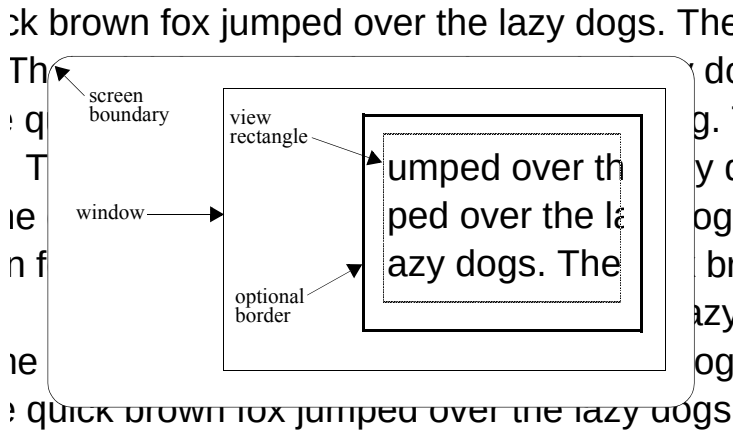


Figure 8.6. Text edit object's view and border rectangles

8.3.14.3. Using the Text Edit System

This section describes techniques for manipulating text edit objects:

- Creating text edit objects
- Text edit handles
- Text edit events
- Setting and getting properties
- Loading text
- Retrieving text
- Selecting text
- Tab support
- Scrolling
- Clearing and deleting text

Creating Text Edit Objects

Tip: To create one or more text edit objects:

Call xvt_tx_create, xvt_tx_create_def, xvt_win_create_res, or xvt_win_create_def.

You specify the window in which the object appears, the border rectangle, the logical font, the margin, the character limit, and the attributes.

Attributes (specified by TX_* constants) determine the following characteristics:

- If text scrolls automatically when the user drags the mouse outside of the view rectangle
- If there is a border
- If cut, copy, and paste are allowed
- If more than one paragraph can be entered
- If the text can be edited
- If word wrap is enabled

See Also: For a complete list of text attributes that you can set, refer to the *XVT Portability Toolkit Reference*.

Tip: To form the attribute argument:

OR together desired constants (e.g., TX_ONEPAR | TX_READONLY).

The margin is the maximum number of pixels allowed per line when word wrap is enabled. A word that would extend past the margin is placed on the next line. If the word is longer than a line, it is split at the margin. Use the character limit for fixed-length fields in form-entry applications; this limit restricts the number of characters that can be typed into a paragraph.

After the text edit object is created, you can change all of the above characteristics, except for the window. In addition, you can use the xvt_tx_set_* functions to change the colors used for text, border, and background, and the average number of characters between tab stops.

WIN_DEF Data Structures

When specifying text edit objects with xvt_win_create_def and xvt_tx_create_def, you use WIN_DEF data structures. The WIN_DEF structure contains a substructure designed for text edit object definitions:

```
typedef struct s_win_def {
        WIN_TYPE wtype;
        RCT rct;
        char *text;
        UNIT_TYPE units;
        XVT_COLOR_COMPONENTS *ctlcolors;
        union {
                ...
                struct {
                        unsigned short attrib;
                        XVT_FNTID font_id;
                        short int margin;
                        short int limit;
                        short int tx_id;
                } tx;
                ...
        } v;
} WIN_DEF;
```

The fields in the tx substructure reflect the attributes of text edit objects, and are the same attributes specified when you create a text edit object with the function xvt_tx_create.

TXEDIT Handles

Text edit objects are identified by a TXEDIT handle which is equivalent to a WINDOW handle.

Tip: To get the TXEDIT value for a text edit object, given its resource ID:

Call xvt_win_get_tx.

Tip: To get the resource ID for a text edit object, given the TXEDIT value:

Call xvt_ctl_get_id.

Tip: To retrieve text edit objects for a particular window or for the entire application from a list internally maintained by the Portability Toolkit:

Call xvt_tx_get_next_tx.

xvt_tx_get_next_tx is passed a text edit object and a window handle. If the text edit object is NULL_TXEDIT, then the first text edit object in the list is returned. Otherwise, the text edit object passed must be valid, and the text edit object that immediately follows it in the list is returned.

If a valid window is passed to xvt_tx_get_next_tx, then xvt_tx_get_next_tx returns the next text edit object belonging to the window that was passed. If the passed window is NULL_WIN, then the next text edit object in any window is returned.

NULL_TXEDIT is returned when the end of the list is reached.

Implementation Note: You should not assume consistency between platforms or between sequences of calls to xvt_tx_get_next_tx concerning the order in which text edit objects are returned.

Example: This code fragment shows how to use xvt_tx_get_next_tx to clear all text edit objects in an application:

```
TXEDIT tx = NULL_TXEDIT;
...
while (NULL_TXEDIT !=
            (tx = xvt_tx_get_next_tx(tx, NULL_WIN)))
      xvt_tx_clear(tx);
```

Event Handling

```
typedef struct s_ctlinfo {
        WIN_TYPE type;              /* WC_TEXTEDIT */
        WINDOW win;                 /* WINDOW of control */
        union {
                ...
                struct s_textedit {
                    BOOLEAN focus_change;
                                    /* is it a focus change? */
                    BOOLEAN active;
                                    /* if so: gaining focus? */
                } textedit;
                ...
        } v;
} CONTROL_INFO, * CONTROL_INFO_PTR;
```

Based on the event information above, you can determine the following:

- If focus_change is FALSE, then the contents of the text edit object have changed. You can call xvt_tx_get_line to determine the new contents of the text edit object, and compare this with the previous contents of the text edit object.
- If focus_change is TRUE, and active is TRUE, then the text edit object gained the keyboard focus.
- If focus_change is TRUE, and active is FALSE, then the text edit object lost the keyboard focus.

Setting Properties

Tip: To change various properties of a text edit object:

```
xvt_tx_set_attr
xvt_tx_set_limit
xvt_tx_set_margin
xvt_tx_set_tabstop
```

Most of these functions cause the text to reset, which, among other things, re-wraps every paragraph. You can also reset a text edit object explicitly with xvt_tx_reset.

Note: If the text edit colors (foreground, background, and border) have not been set explicitly, these colors can be changed indirectly using a call to xvt_win_set_colors.

Getting Properties

Tip: To retrieve current values, call one of the following functions:

xvt_tx_get_attr
xvt_tx_get_limit
xvt_tx_get_margin
xvt_tx_get_tabstop
xvt_tx_get_view

Tip: To retrieve various text edit sizes, call one of the following functions:

xvt_tx_get_num_chars
: Number of characters in a given line.

xvt_tx_get_num_lines
: Total number of lines in a text edit object.

xvt_tx_get_num_par_lines
: Number of lines in a given paragraph.

xvt_tx_get_num_pars
: Total number of paragraphs in a text edit object.

Loading Text

Tip: To load text into a previously created text edit object:

Call xvt_tx_add_par.

Tip: To change existing text:

Call xvt_tx_append, xvt_tx_rem_par, and xvt_tx_set_par.

Tip: When loading a large amount of text (with multiple calls to xvt_tx_add_par, for instance), you can save a lot of time by suspending screen updating and word wrapping with xvt_tx_suspend. When you want updating to resume, call xvt_tx_resume.

Retrieving Text

Tip: To retrieve the text from a text edit object (e.g., for searching or saving to a file):

Call xvt_tx_get_line, which returns a pointer to the text.

You must bracket a text-retrieving call to xvt_tx_get_line with two special calls to lock and unlock the text buffer, and you must do this for each line you want to access. This eliminates the overhead of copying the text to a buffer if it could be accessed in place.

Selecting Text

Tip: To explicitly set the text selection:

Call xvt_tx_set_sel.

Normally, of course, the user selects text interactively by dragging the mouse or using the arrow keys with the Shift key held down.

Tip: To get the current selection:

Call xvt_tx_get_sel.

When the user clicks in the view rectangle of a text edit object, XVT automatically makes that object active and displays a caret.

Text Edit Tab Stops

Text edit controls automatically provide support for tabs. XVT sets tab stops at multiples of a predetermined tab stop distance. The tab stop distance depends on the font and size of the text—it is calculated as eight times an average character width. Tab characters in the text edit text indicate that the next character after the tab is to be positioned at the next tab stop to the right of the current position. Tab characters are not converted to spaces in the text.

Tip: To put tabs into the text programmatically:

Call xvt_tx_add_par, xvt_tx_append, or xvt_tx_set_par.

Reading text containing tabs from a text edit object with the xvt_tx_get_line function returns a character string containing the ASCII tab character.

Scrolling

There are two kinds of scrolling, automatic and manual:

Automatic scrolling
Occurs without your application being involved. The text edit system senses when the mouse is being dragged (or the caret is moved) outside of the view rectangle and scrolls the text appropriately (also called auto-scrolling).

Manual scrolling
Takes place when your application calls xvt_tx_scroll_hor or xvt_tx_scroll_vert. If you want, you can call these when the user operates the scrollbars at the sides of the document window that contains the text edit object. (You probably won't want to use the scrollbars this way if the window contains more than one text edit object.)

If the window does have scrollbars, you will want to update the thumb position whenever the text scrolls, even if by automatic scrolling. You can establish a scroll callback function with xvt_tx_set_scroll_callback. The text edit system calls this function whenever the scrollbars have to be updated.

The primary information passed to your scroll callback function is the location of the first character in the *view* relative to the start of the text document in the text edit object. If this is non-zero, it means that the text is scrolled up and/or to the left in the view. You can also get this location without waiting for a scroll callback by calling xvt_tx_get_origin.

See Also: For more information on how XVT controls and GUI components handle scrolling, see Chapter 13, *Scrolling*.

Clearing Text and Deleting Text Edit Objects

Tip: To clear all text from a text edit object:

Call xvt_tx_clear.
(Note: The empty object still exists.)

Tip: To entirely eliminate a text edit object:

Call xvt_tx_destroy.

See Also: For more information about manipulating text edit objects, see the xvt_tx_* functions in the *XVT Portability Toolkit Reference*.

8.3.14.4. Text Edit Size Limits

A number of limits constrain the size of text edit objects. The first limit is set by the type, unsigned short, used for paragraph, line, and character numbers. Since unsigned short is sixteen bits (on most systems), this limit is 65,535 (64KB - 1). So the number of lines allowed in any one paragraph is theoretically 64KB.

A second limit, however, is the total number of lines allowed in a text edit, which is also 64KB. Thus if an average paragraph has four lines, the number of paragraphs would be limited to 16KB.

In addition, because of the way information is stored for each paragraph, the actual limit for the total number of characters depends on the number of lines in that paragraph (which in turn depends on margin sizes and whether word wrap is on). This value is much smaller than the theoretical limit of 64KB, being approximately (32KB – 2 * number of lines).

With word wrap on, the number of lines increases as the number of characters increases. As a result, there is no easy answer to the question, “How many characters can be put in one paragraph?” Assuming an average line length of 80 characters, the character limit would be somewhere around 32,000.

8.3.15. Treeview Controls

Treeview controls give XVT applications the ability to display hierarchically-oriented information. The classic example of a tree is the explorer window for viewing directories and files.

The Treeview control is a WINDOW of type WC_TREEVIEW.

The treeview is a commonly seen GUI feature and the ability to provide trees within production applications is a powerful feature that extends the utility of an XVT application.

Tree-specific types

```
void *XVT_TREEVIEW_NODE;

typedef enum e_treeview_node_type {

    XVT_TREEVIEW_NODE_TERMINAL, /* leaf - no children */

    XVT_TREEVIEW_NODE_NONTERMINAL, /* branch - may have
children */

} XVT_TREEVIEW_NODE_TYPE;
```

```
typedef XVT_CALLCONV_TYPEDEF(
        BOOLEAN, XVT_TREEVIEW_CALLBACK,
    (WINDOW ctl_win, XVT_TREEVIEW_NODE node) );
```

Treeview Attribute Constants

```
#define TREEVIEW_DEFAULT …/* default -- implies *_STYLE_NONE,
                                *_EXPAND_ONE, and *_SELECT_NONE */
#define TREEVIEW_NO_BORDER …/* no border around control */
#define TREEVIEW_SHOW_ROOT_NODE…/* draw root node */
#define TREEVIEW_STYLE_NONE …/* no tree lines drawn */
#define TREEVIEW_STYLE_ORTHGONAL …/* squared connections */
#define TREEVIEW_STYLE_SLANT …/* slanted connections - ***NOT
IMPLEMENTED
AT THIS TIME*** */
#define TREEVIEW_EXPAND_ONE …/* only expand one node */
#define TREEVIEW_EXPAND_ALL …/* expand node and all subnodes */
#define TREEVIEW_SELECT_NONE …/* no selection allowed */
#define TREEVIEW_SELECT_ONE …/* one selection allowed */
#define TREEVIEW_SELECT_MANY …/* many slections allowed */
```

Treeview API Library Functions

The following functions are available in the Treeview API:

xvt_treeview_add_child_node

Add a child treeview node to existing treeview node

xvt_treeview_collapse_node

Collapses node

xvt_treeview_create

Creates treeview control

xvt_treeview_create_node

Creates a treeview node

xvt_treeview_destroy_node

Destroys treeview node

xvt_treeview_expand_node

Expands node

xvt_treeview_get_attributes

Get the attributes for the treeview control

xvt_treeview_get_child_node

Get a child node from a parent node

xvt_treeview_get_line_height

Get line height of treeview node

xvt_treeview_get_node_callback

Get the node call back function

xvt_treeview_get_node_data

Get the node data

xvt_treeview_get_node_image_collapsed

Get the collapsed image for a node

xvt_treeview_get_node_image_expanded

Get the expanded image for a node

xvt_treeview_get_node_image_item

Get the item image for a node

xvt_treeview_get_node_num_children

Get the number of child nodes for a node

xvt_treeview_get_node_num_vis_children

Get the number of visible child nodes for a node

xvt_treeview_get_node_string

Get the item text for a node

xvt_treeview_get_node_type

Get node type

xvt_treeview_get_parent_node

Get parent node

xvt_treeview_get_root_node

Get root node form treeview control

xvt_treeview_node_selected

Get node selection state

xvt_treeview_remove_child_node

Remove child node from list

xvt_treeview_resume

Resume updating of treeview control

xvt_treeview_set_attributes

Set the attributes for treeview control

xvt_treeview_set_line_height

Set line height for a node

xvt_treeview_set_node_callback

Set the node call back function

xvt_treeview_set_node_data

Set the node data

xvt_treeview_set_node_image_collapsed

Set the collapsed image for a node

xvt_treeview_set_node_image_expanded

Set the expanded image for a node

xvt_treeview_set_node_image_item

Set the item image for a node

xvt_treeview_set_node_string

Set the item text for a node

xvt_treeview_set_node_type

Set node type

xvt_treeview_suspend

Suspend updating of treeview control

xvt_treeview_update

Force update of treeview control

Detailed information on the xvt_treeview_* functions can be found in the *XVT Portability Toolkit Reference*.

8.4. Control Attributes

XVT allows the application to get or set any control's state or change its attributes. This section discusses two attributes that are apparent to end users—font and color.

8.4.1. Control Fonts

A control font is the XVT logical font used for all text in a control. Applications can set and query XVT logical fonts for all XVT Portability Toolkit (PTK) controls that contain text.

Note: Fonts for window titles and menu items cannot be set with XVT functions. Text edit objects have their own functions for setting logical fonts.

An XVT control may inherit its logical font from its immediate parent window or dialog, or from the application, or if no control font is set at any of these levels, from the native windowing system. The font in which control text is rendered depends on the most specific font set by the application (in order):

- Font set for the specific control
- Default font set for all controls of the container window or dialog
- Default font set for all controls of the application
- Default font set for all controls of the system set by the native platform

The control font can be set at control creation time. When creating a window and its contents, use an array of WIN_DEF structures and xvt_res_get_win_def or xvt_res_get_dlg_def (see section 3.3.2 on page 3-7). Alternatively, control fonts can be set after a control is created (see section 15.4 on page 15-10).

Caution: You may need to increase the font cache size for applications that set and use a large number of control fonts.

See Also: For more information on how to increase the font cache size, refer to the description of ATTR_FONT_CACHE_SIZE in the attributes portion of the *XVT Portability Toolkit Reference*.

8.4.1.1. Setting Fonts on Individual Controls

Tip: To set the logical font for a single control:

Call xvt_ctl_set_font.

Tip: To obtain the current logical font for a single control:

Call xvt_ctl_get_font.

Unless a control font is specifically set for a control, xvt_ctl_get_font returns NULL_FNTID.

Note: Even if the application sets the font of a control, no E_SIZE event is issued. The control does not automatically resize. If the font was enlarged, that control's label(s) may be clipped.

8.4.1.2. Setting Default Container Control Fonts

Tip: To set the default logical font for all controls in a window or dialog:

Call xvt_win_set_ctl_font.

Tip: To obtain the default logical font for all controls in a window or dialog:

Call xvt_win_get_ctl_font.

Note: Unless a default control font is specifically set for the container, xvt_win_ctl_get_font will return NULL_FNTID.

8.4.1.3. Setting the Default Application Control Font

The attribute ATTR_APP_CTL_FONT_RID can be used to set the resource ID of the application's default control font (if not set, then XVT uses the native platform default control font). If you choose to set this attribute, do so prior to calling xvt_app_create.

Example: The following code fragments demonstrate how ATTR_APP_CTL_FONT_RID is used to set the default application control font.

In the application header file:

```
#define MY_APP_CTL_FONT 10
```

In the application XRC file:

```
font MY_APP_CTL_FONT "helvetica" 12 bold italic
```

In the application source code (before calling xvt_app_create):

```
xvt_vobj_set_attr(NULL_WIN, ATTR_APP_CTL_FONT_RID,
                  (long) MY_APP_CTL_FONT);
```

8.4.2. Control Colors

A control color is the color XVT selects for a component of a control, such as foreground, background, border, etc. Applications can set and query XVT colors for the most significant XVT Portability Toolkit (PTK) control components; for more details, refer to section 8.4.2.1, next.

Note: Although you can set the colors of individual controls inside a window or dialog, XVT does not provide functions that can be used to set the color of a window or dialog's window decorations.

An XVT control inherits its component colors from its parent window or dialog, or from the application, or if no control component colors are set at any of these levels, from the native windowing system. A single control inherits each component color separately. The colors in which control components are rendered depend on the most specific component colors set by the application (in order):

- Colors set for the specific control
- Default colors set for all controls of the container window or dialog
- Default colors set for all controls of the application
- Default colors set for all controls of the system set by the native platform

Implementation Note: Users may set default control component colors through native mechanisms, such as Motif application defaults files, Win32 control panels, and Macintosh default color tables.

Control colors can be set at control creation time. When creating a window and its contents, use an array of WIN_DEF structures and xvt_res_get_win_def or xvt_res_get_dlg_def (see section 3.3.2 on page 3-7). Alternatively, control colors can be set after a control is created.

8.4.2.1. Control Component Colors

Each native platform supports different control component colors, but there is much overlap. XVT *portably* supports the most significant component colors, even though some component colors are not supported natively on all platforms. For details, refer to Figure 8.7.

Note: In some cases, an XVT component name does not correspond directly to the name used by the native platform for the same component. However, if you set the color value for an XVT control component, you *will* get the appropriate behavior on all platforms.

Color components of XVT controls are set with data storedin arrays of XVT_COLOR_COMPONENT data structures:

```
typedef s_xvt_color_component {
      XVT_COLOR_TYPE type;          /* color component */
      COLOR color;                  /* XVT color value */
} XVT_COLOR_COMPONENT;
```

The following are valid XVT control color components defined for XVT_COLOR_TYPE:

XVT_COLOR_FOREGROUND	Control text and the arrows on scrollbars
XVT_COLOR_BACKGROUND	Fill color of rectangular region occupied by control
XVT_COLOR_BLEND	Secondary background for some controls so they blend into their container window's background without visual indication of a border
XVT_COLOR_HIGHLIGHT	Visual indication that a control has keyboard focus
XVT_COLOR_BORDER	Outside edge of control (rectangular)
XVT_COLOR_TROUGH	Slider area behind scrollbar thumb
XVT_COLOR_SELECT	Indication that a control has been selected
XVT_COLOR_NULL	Value indicating last element of XVT_COLOR_COMPONENT array

You may set as many or as few of these component values as are needed for your application.

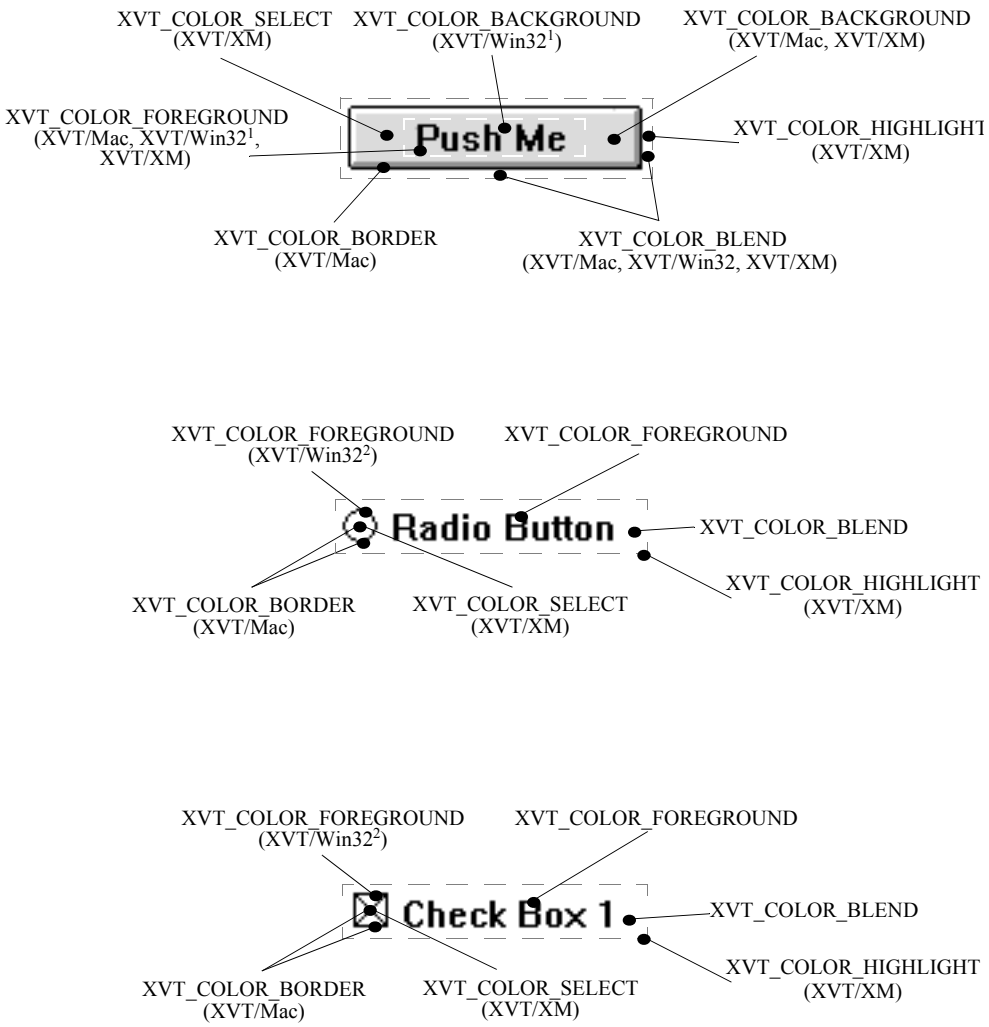


Figure 8.7. XVT control component colors (part 1 of 4)

[1] Unavailable with Windows 95.

[2] Unavailable on some Win32 platforms.

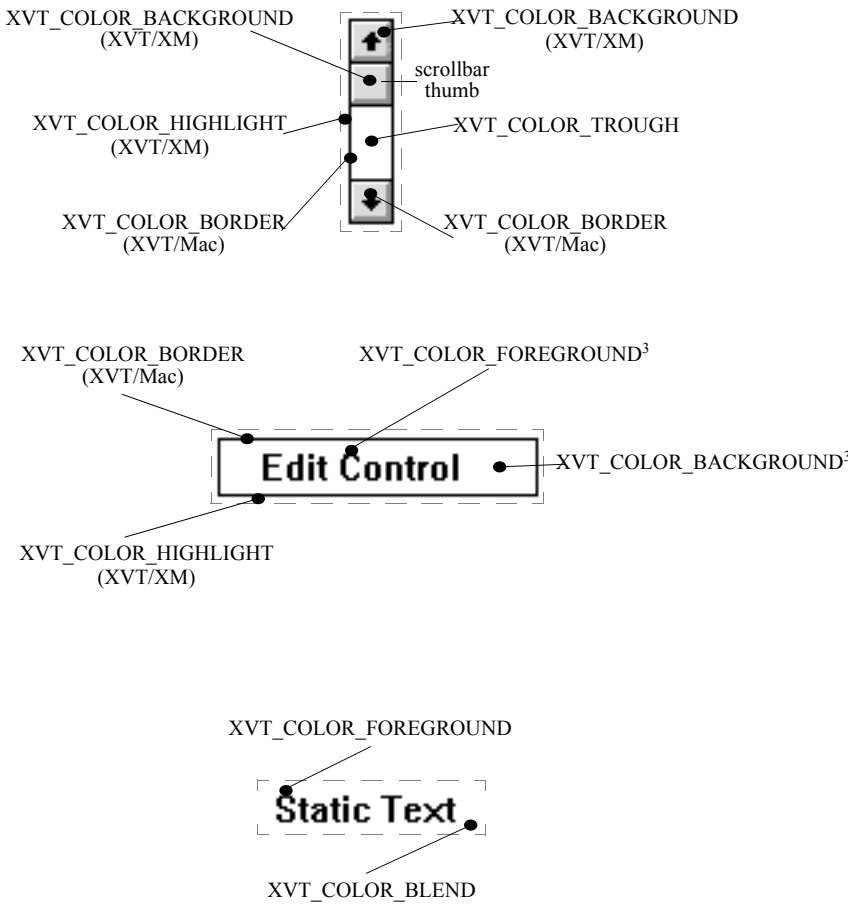


Figure 8.7. XVT control component colors (part 2 of 4)

[3] For XVT/XM, foreground and background colors are reversed on selection. For XVT/Mac monochrome systems, foreground and background colors are reversed on selection. For XVT/Mac color systems, set selected text color on the Color Control Panel.

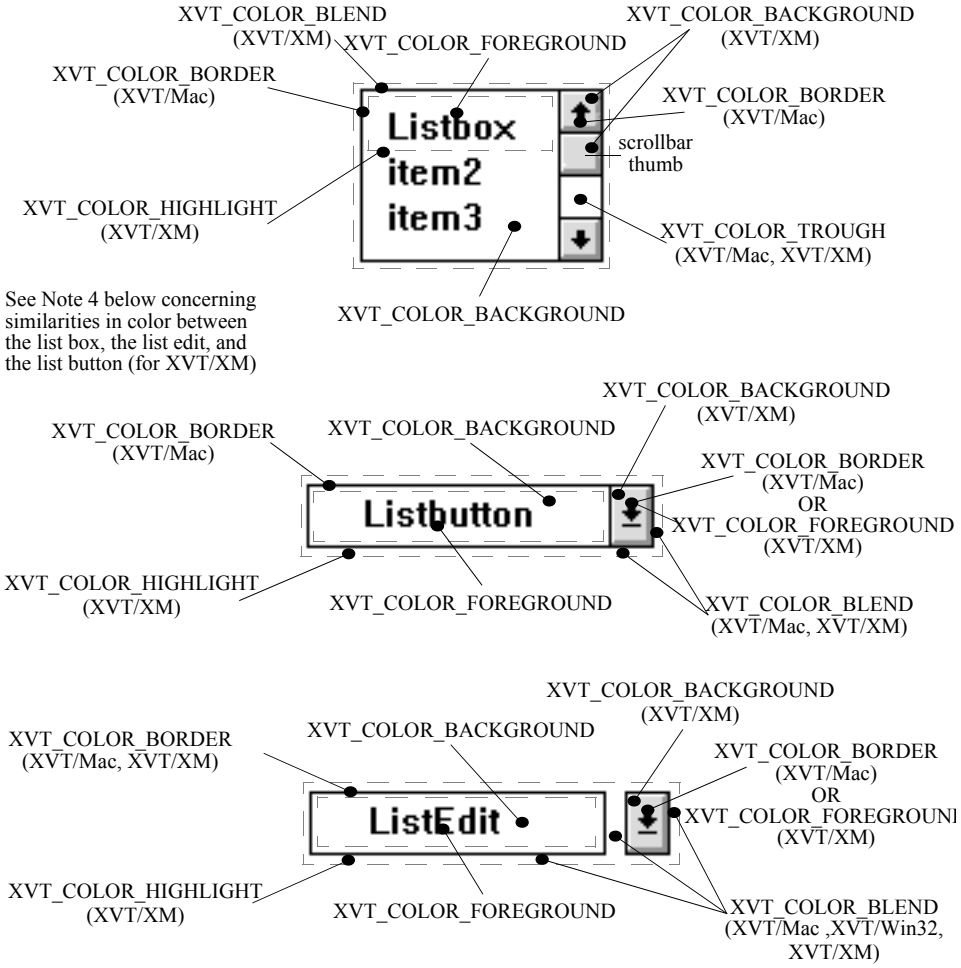


Figure 8.7. XVT control component colors (part 3 of 4)

[4] On XVT/XM only, the colors in the dropdown list (of the list button and the list edit) are the same as for the list box itself.

[5] For XVT/Mac monochrome systems, foreground and background colors are reversed on selection. For XVT/Mac color systems, set selected text color on the Color Control Panel.

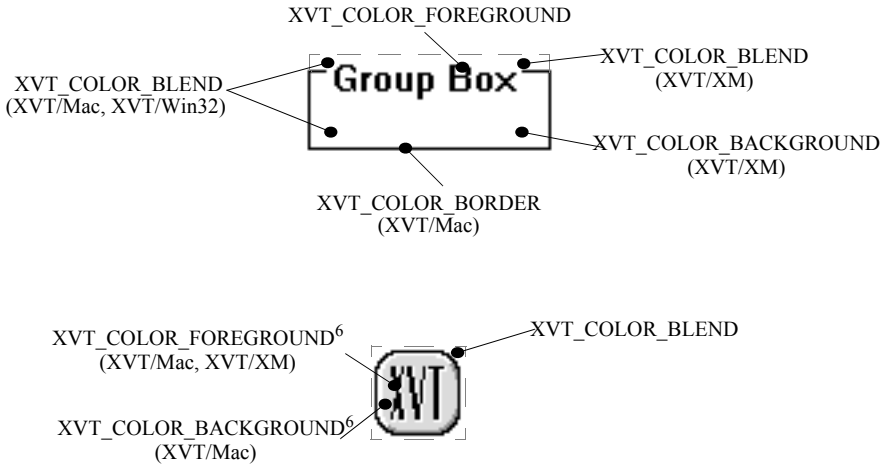


Figure 8.7. XVT control component colors (part 4 of 4)

[6] Available for icons with one bit-plane only.

[7] Set text edit colors (illustrated here for completeness) with xvt_ctl_set_colors .

8.4.2.2. Setting Colors on Individual Controls

Tip: To set the component colors for a single control:

Call xvt_ctl_set_colors.

Tip: To obtain the current component colors for a single control:

Call xvt_ctl_get_colors.

Note: Unless a control color component is specifically set for a control, xvt_ctl_get_colors returns NULL.

8.4.2.3. Setting Default Container Control Colors

Tip: To set the default component colors for all controls in a window or dialog:

Call xvt_win_set_ctl_colors.

Tip: To obtain the default component colors for all controls in a window or dialog:

Call xvt_win_get_ctl_colors.

Note: Unless a control color component is specifically set for a container, xvt_win_get_ctl_colors returns NULL.

8.4.2.4. Setting Default Application Control Component Colors

The attribute ATTR_APP_CTL_COLORS can be used to set the application's default control component colors (if not set, then XVT uses the native platform's default control colors). The attribute is set to the address of an XVT_COLOR_COMPONENT array. If you choose to set this attribute, do so prior to calling xvt_app_create and do not change it again once set.

Example: This code demonstrates how ATTR_APP_CTL_COLORS is used to set default application control component colors (before calling xvt_app_create):

```
static XVT_COLOR_COMPONENT ctl_colors[] = {
        {XVT_COLOR_FOREGROUND, COLOR_BLACK},
        {XVT_COLOR_BLEND, COLOR_WHITE},
        {XVT_COLOR_BACKGROUND, COLOR_BLUE},
        {XVT_COLOR_NULL, 0}
};
...
xvt_vobj_set_attr(NULL_WIN, ATTR_APP_CTL_COLORS,
                                (long) ctl_colors);
```

8.5. Control Mnemonics

A mnemonic character is a character in the title of a control or menu item used (by the application's end user) to select or invoke the control or menu item via keyboard input. Mnemonic characters may be typed by application users in lieu of mouse selection and thus provide one form of keyboard navigation. The mnemonic character (and the keyboard key that is pressed to select the item) is typically indicated by an underlined character in the control or menu item text:

Help OK Extract

Implementation Note: You should refer to the look-and-feel guidelines for specific native platforms to determine the best mnemonic characters for your application.

8.5.1. Setting Control Mnemonics

The mnemonic character is immediately preceded by a tilde (~) in the title text of the control:

~Help ~OK E~xtract

The methods for setting a mnemonic character in the title of non-editable control (pushbutton, check box, radio button, static text, or group box) in a window or dialog are:

- Set the mnemonic character in the title parameter of the xvt_ctl_create function
- Set mnemonic characters in the text fields of the array of WIN_DEF structures passed to xvt_win_create_def, xvt_dlg_create_def, or xvt_ctl_create_def
- Set the mnemonic character in the text field of an XRC control statement
- Set the mnemonic character in the title parameter of the xvt_vobj_set_title function

Implementation Note: Control mnemonics are not implemented on XVT/XM and XVT/Mac. Mnemonic characters on controls are not supported by the native look-and-feel of these platforms. When a mnemonic character is set for a control on these platforms, the character is retained internally by the toolkit, but not displayed.

8.5.2. Getting Control Mnemonics

You cannot directly extract a mnemonic character from a control title. You can, however, obtain a title text string, with any embedded mnemonic character, using either of the following approaches:

- Get the mnemonic character from the text fields in the array of WIN_DEF data structure generated from resources by a call to xvt_res_get_win_def or xvt_res_get_dlg_def
- Get the mnemonic character from the title returned by a call to xvt_vobj_get_title

Implementation Note: All set mnemonic characters are returned in the titles by these functions on XVT/XM and XVT/Mac, even though mnemonic characters are not displayed in controls on these platforms.

8.5.3. Processing Mnemonic Characters

Dialogs

The processing of mnemonic characters in dialog controls is handled automatically by XVT and the native platforms. No special processing of characters is required (character events generally are not sent to dialog event handlers).

Windows

When a control in a window has focus and the user types characters, characters not processed internally by the control (both mnemonic and non-mnemonic) are passed as E_CHAR character events to the control's parent (container) window. Your application event handler then must process these characters for the desired behavior (focus change, selection, etc.).

Keyboard navigation is not automatic in XVT windows. You may use the XVT navigation object (see section 6.6 on page 6-14) to handle E_CHAR events for keyboard navigation in windows, or you may implement your own navigation mechanism.

When a particular control does not have focus and the user types characters, E_CHAR events are delivered to the window that has focus. The action of typing characters (mnemonic or not) does not affect window focus. The attribute ATTR_PROPAGATE_NAV_CHARS controls the propagation of characters from controls to the parent window. To insure that sufficient character events occur for keyboard navigation, this attribute should be set to a value of TRUE.

See Also: For more information on E_CHAR events, refer to section 4.5.1 on page 4-16.
For more information on the propagation of character events to window event handlers, see ATTR_PROPAGATE_NAV_CHARS in the *XVT Portability Toolkit Reference*.

Example: This code demonstrates how to process character events to trap mnemonic characters:

```
XVTV_CHAR mbc[XVT_MAX_MB_SIZE + 1];
XVTV_CHAR mbs[XVT_MAX_MB_SIZE * 255];
XVT_CHAR *mbs_ptr;
SLIST slist;
SLIST_ELT slist_elt;
WINDOW child;
WIN_TYPE wtype;

switch(ep->type) {
        case E_CHAR:

        /* only platforms that support mnemonics */
#if (XVTWS == WIN32WS)

        /* convert the character */
        memset(mbc, 0, XVT_MAX_MB_SIZE + 1);
        xvt_str_convert_wc_to_mb(mbc, ep->v.chr.ch);

        /* do not process if length is greater than 1 */
        if(xvt_str_get_byte_count(mbc) == 1) &&
                xvt_str_is_alnum(mbc)) {
                /* convert character for processing and
                        get the children of win */

                xvt_str_convert_to_upper(mbc, mbc, 1);
                slist = xvt_win_list_wins(win, 0L);
                if(!slist)
                        break;

                slist_elt = xvt_slist_get_first(slist);

                /* process the children until a window
                        with the mnemonic is found */
                while(slist_elt) {
                        child = (WINDOW)
                                (*xvt_slist_get_data(slist_elt));
                        /* only process controls with titles
                                that accept mnemonic characters */
                        wtype = xvt_vobj_get_type(child);
```

```
switch(wtype) {
        case WC_PUSHBUTTON:
        case WC_RADIOBUTTON:
        case WC_CHECKBOX:
        case WC_TEXT:
        case WC_GROUPBOX:
                if (xvt_vobj_get_title(child, mbs,
                                        XVT_MAX_MB_SIZE * 255))
                        break;
        default:
                slist_elt = xvt_slist_get_next(slist,
                                        slist_elt);
                continue;
} /* end switch(wtype) */

/* find mnemonic character in the title */
mbs_ptr = xvt_str_find_first_char(mbs,
                        "~");
if (mbs_ptr) {
        mbs_ptr = xvt_str_get_next_char(mbs_ptr);
        xvt_str_convert_to_upper(mbs_ptr, mbs_ptr, 1);
        if (!xvt_str_compare_n_char(mbs_ptr, mbc, 1)) {
                EVENT ep;
                long flags;

                /* get the first control after
                        a static object */
                if ((wtype == WC_TEXT) ||
                        (wtype == WC_GROUPBOX)) {
                        slist_elt =
                                xvt_slist_get_next(slist,
                                        slist_elt);

                        if (!slist_elt)
                                break;

                        child = (WINDOW)
                                (*xvt_slist_get_data
                                        (slist_elt));
                } /* end if (wtype) */

                flags = xvt_vobj_get_flags(child)

                if (!(flags & CTL_FLAG_INVISIBLE) &&
                        !(flags & CTL_FLAG_DISABLED)) {
                        memset(&ep, 0, sizeof(EVENT));
                        ep.type = E_CONTROL;
                        ep.v.ctl.id = xvt_clt_get_id(child);
                        ep.v.ctl.ci.type = wtype;
                        ep.v.ctl.ci.win = child;
```

```
                                        switch(wtype) {
                                                case WC_EDIT:
                                                        if (xvt_scr_get_focus_vobj()
                                                                != child) {
                                                ep.v.ctl.ci.v.edit.focus_change = TRUE;
                                                ep.v.ctl.ci.v.edit.active = TRUE;
                                                        }
                                                        break;
                                                case WC_LISTEDIT:
                                                        if (xvt_scr_get_focus_vobj()
                                                                != child) {
                                                ep.v.ctl.ci.v.listedit.focus_change = TRUE;
                                                ep.v.ctl.ci.v.listedit.active = TRUE;
                                                        }
                                                        break;
                                        } /* end switch(wtype) */

                                        xvt_scr_set_focus_vobj(child);
                                        xvt_win_dispatch_event(win, &ep);

                                } /* end if (!(flags...) */
                        } /* end if (!xvt_str_compare_n_char ...) */
                } /* end if (mbs_ptr) */
                slist_elt = xvt_slist_get_next (slist, slist_elt)
                /* end while (slist_elt) */

        /* Not found so beep */
        xvt_scr_beep( );
        } /* end if (xvt_str_get_byte_count...) */
...
} /* end switch(ep->type) */
```

9

MENUS

XVT-Design lets you arrange the menubar and its items. You can specify resource identifiers, along with traits such as whether an item shows a check mark when selected, or whether it is initially disabled. This chapter contains background information about menus. If you create menus in XVT-Design, you won't need most of the detail in this chapter.

A menu presents a set of possible selections that allow a user to control the application. XVT menus are optionally attached to task and top-level windows. (You cannot attach menus to modal windows or dialogs.)

Menus are composed of horizontally arranged items. When the user selects an item, one of two things happens:

- An E_COMMAND event is generated
- A subsidiary (or cascading) menu drops down

The same things happen when the user selects an item on a subsidiary menu. XVT menus are hierarchical—a given menu can have many levels of nested menus within it (see Figure 9.1).

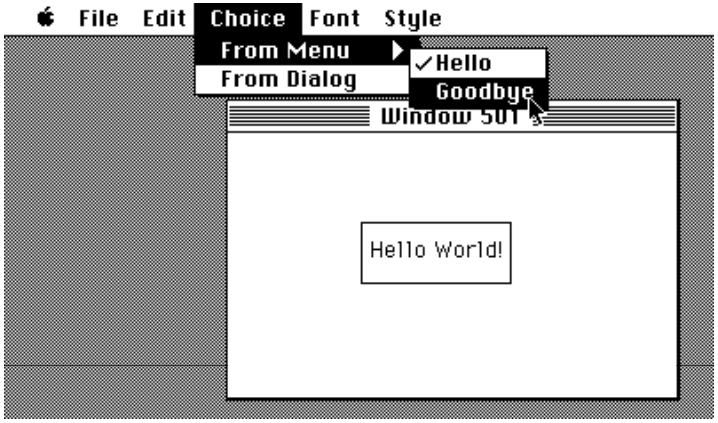


Figure 9.1. Hierarchical menus on the Mac (on the Mac, Help appears under the Apple menu)

Menus are usually defined before program execution as numbered XRC resources. You can define new menus within your XVT application, and assign them to windows during program execution. You can convert resource-based menus to in-memory data structures during program execution. XVT also provides functions to modify existing menus, which is often necessary to reflect changing program states, user selections, or modes.

If you specify the flag WSF_NO_MENUBAR when creating a window, the window won't have a menu and can't have one assigned to it.

XVT provides platform-specific versions of the following menus: File, Edit, Font/Style, and Help. Using these standard menus in your XRC menu descriptions, as needed, ensures that your menus will be portable.

See Also: For more information about using and compiling XRC resources, see Chapter 5, *Resources and XRC*.
For more information about the menubar and menu XRC statements, refer to the *XVT Portability Toolkit Reference*.

9.1. Menu Definitions

The following key definitions can help you understand the capabilities of menubars and hierarchical menus in XVT:

Hierarchical menu
A menu that has one or more submenus. This menu/submenu arrangement is hierarchical because each level includes the next.

Menubar
A menubar is the "root" of the menu hierarchy tree. To design menus, you must first start with a menubar. A menubar is visually represented by a horizontal list of items across the top of a screen or window. A menubar consists of a list of pull-down menus.

Menu, **pull-down menu**
Menus appear horizontally across a menubar. Clicking on a menu, or selecting it with a keyboard mnemonic, "pulls down" a vertical list of items to choose from. A menu can also contain submenus.

Submenu
A submenu is just like a menu, except that it can appear anywhere in the menu hierarchy. When a submenu appears as an item on a menu or submenu, some graphical indication (such as an arrow) shows that the hierarchy extends below this item. When the user pulls down a menu and moves the mouse to a submenu, the list of menu items for that submenu appears.

Menu item
Menu items appear on a menu or submenu. A menu item can either be a "leaf" of the menu tree, in which case it causes an E_COMMAND event to be delivered to the application, or it can be another submenu whose contents are displayed when the user drags the mouse to this item.

Pop-up menu
A pop-up menu is a temporary menu displayed at a specified location over a window. Unlike other types of menus, a pop-up menu is not associated with a menubar. Instead, this type of menu exists only within the context of the function that created it, xvt_menu_popup.

9.2. Menu Events

In XVT, menu selections generate either E_COMMAND events (for most selections), or E_FONT events (for selections from the Font/Style menu or the Font Selection dialog). These events are sent to the event handler of the associated window. Each menu item is assigned a unique numeric tag, usually represented by a constant in your program. This tag is reported to the event handler, by means of the event structure, when the E_COMMAND event is sent. Menu tags are limited to the range from 1 to MAX_MENU_TAG (32,000).

Note: Because E_FONT events can be generated from Font/Style menu selections, they are mentioned in this section. However, E_FONT event information is very different from E_COMMAND event information. In the case of the E_COMMAND, the menu item tag is the only information passed to the event handler. However, in the case of the E_FONT, more complex information is passed.

See Also: For more information about E_FONT events, see E_FONT in the *XVT Portability Toolkit Reference*. Also refer to section 4.5.8 in Chapter 4, *Events*.

9.3. Defining Menus

You can define menus in two ways:

- In the XVT Resource Compiler (XRC)
- In MENU_ITEM data structures

9.3.1. XRC Menubar Definitions

You can define menus and other GUI components as resources in XVT's Resource Compiler (XRC). The functions xvt_win_create and xvt_win_create_res require a menu resource ID.

XVT-Design can create menubar definitions in XVT's platform-independent XVT Resource Compiler (XRC). This is the simplest way to create menus.

Caution: You can edit the XRC file that XVT-Design creates as part of your project. However, XVT discourages this, because it results in two different pictures of your menu resources: one in your XVT-Design project and one in the XRC file.

See Also: For information on how to define menus and other GUI components in XRC, see section 3.3.1 on page 3-4.

9.3.2. MENU_ITEM Data Structures

You can also define menus in XVT by using MENU_ITEM data structures. This approach works only with the function xvt_win_create_def. xvt_menu_popup, the function that creates pop-up menus, creates a new hierarchy inside the MENU_ITEM tree.

Additionally, xvt_menu_set_tree can use in-memory data structures to replace existing menus. (This is the only way to replace menus.)

Here is the MENU_ITEM data structure:

```
typedef struct s_mitem {
        MENU_TAG tag;
        char *text;
        short mkey;
        unsigned enabled:1;
        unsigned checked:1;
        unsigned checkable:1;
        unsigned separator:1;
        struct s_mitem *child;

        /* non-portable fields */
        ...
} MENU_ITEM;
```

When you create MENU_ITEM-based menu definitions, remember that each MENU_ITEM structure represents a single menu item (either in the menubar, or in one of the pull-down or pop-up menus).

An array of MENU_ITEMS represents each menubar and each pull-down menu. These arrays are terminated by an extra MENU_ITEM structure at the end of the array whose tag field is set to zero. Also, if a MENU_ITEM structure tag refers to a subsidiary menu, then the child field points to the first element of the subsidiarymenu's MENU_ITEM array.

The tag field of the MENU_ITEM structure contains the unsigned integer ID number that you can use in your application program to refer to that menu item.

Tip: To allocate in-memory menu definitions:

Call xvt_mem_alloc.

Tip: To recursively free an in-memory menu definition:

Call xvt_res_free_menu_tree.

Of course, you shouldn't call xvt_res_free_menu_tree on any statically allocated data.

See Also: For more information on menu tags, see MENU_TAG in the *XVT Portability Toolkit Reference*. For a complete description of the rest of the fields in the MENU_ITEM structure, also see MENU_ITEM.

Converting XRC Menu Definitions to MENU_ITEM Definitions

You can automatically convert an XRC-based menu definition to an in-memory, MENU_ITEM-based definition. This allows you to define all of your menus in XRC, then read them into your program as in-memory structures to be used with functions that take a MENU_ITEM argument, such as xvt_win_create_def and xvt_menu_set_tree.

Tip: To convert an XRC-based menu definition:

Call xvt_res_get_menu.

Tip: To free in-memory structures when you're done with them:

Call xvt_res_free_menu_tree.

9.4. Managing Menus and Menu Attributes

This section discusses functions that manipulate menus as well as the various menu attributes. Most of these functions refer to menus by the window that owns them, and reference the individual menu items by their tags.

9.4.1. Creating a Menu Hierarchy without Resources

If you do not have a menu in a resource description and need to create it from scratch, construct a MENU_ITEM tree yourself by either statically or dynamically creating it, then properly linking MENU_ITEM arrays in your code.

Tip: Remember that each MENU_ITEM array must end with an extra item with the tag field set to zero.

9.4.2. Modifying Menus

Tip: To modify an existing menu:

1. Call xvt_menu_get_tree.
   This function retrieves a window's menu and places it into in-memory MENU_ITEM data structures.

2. Change the data structure contents.
3. Call xvt_menu_set_tree to replace a window's menu with the modified menu.

9.4.3. Menu Item Strings and Menu Mnemonics

Tip: To change the text of a menu item during program execution:

Call xvt_menu_set_item_title.

The text for a menu item can contain a tilde (~) character to designate the mnemonic character of the menu item. The tilde character should immediately precede the character used for the mnemonic in the text string in a call to xvt_menu_set_item_title. When the menu item is displayed, the mnemonic character is automatically enabled and shown to the user as an underlined character.

Implementation Note: On the Macintosh platform, which does not support mnemonic characters, XVT/Mac removes the tilde character before displaying the menu item (without any underlining) and saves it for retrieval.

9.4.4. Checking Menu Items

When you define individual menu items (either in XRC or as MENU_ITEM structures), you can specify whether the item is checkable and, if so, whether it is checked.

Tip: To check/uncheck menu items to reflect their current state:

Call xvt_menu_set_item_checked.

Since pop-up menus only exist within the context of the function xvt_menu_popup, no other menu functions can affect them. If you want to have a pop-up menu item checked as a default initial selection, you need to mark it checked in the MENU_ITEM tree.

Note: When the user selects a checkable menu item, XVT does not automatically check it. An E_COMMAND event notifies you of the selection, and you then need to check the menu item by using xvt_menu_set_item_checked.

9.4.5. Enabled or Disabled Menu Items

Menu items can be shown as either enabled or disabled. You specify the initial state when defining the menu in XRC.

Tip: To enable/disable individual items and subsidiary menus:

Call xvt_menu_set_item_enabled.

Note: You cannot enable or disable individual items on the Font/Style menu. To set check marks that correspond to an application font_id, use xvt_menu_set_font_sel. You can enable or disable the entire Font/Style menu with xvt_menu_set_item_enabled by specifying the tag FONT_MENU_TAG.

9.4.6. Separators

You can define the separator menu item only when you create a menu. The separator appears as a platform-specific decoration in drop-down menus. The menu item is not selectable and no other attributes or functions can apply to it.

9.5. Pop-up Menus

A pop-up menu is a temporary menu displayed at a specified location over a window (only windows that can receive mouse events may be specified). Pop-up menus are created from a MENU_ITEM tree.

Generally, applications should invoke a pop-up menu only in response to an E_MOUSE_DOWN event. When a user selects a menu item, an E_COMMAND event is sent to the specified window's event handler. This event should be processed just like the E_COMMAND event generated from a menubar menu item. In the case of pop-up menus, a child window (W_PLAIN or W_NO_BORDER) may receive E_COMMAND events.

Tip: To display a pop-up menu:

Call xvt_menu_popup.

The following enumeration type is used to position a pop-up menu with respect to a given window's coordinate system:

```
typedef enum e_xvt_alignment {
        XVT_POPUP_CENTER,          /* Centered below specified
                                        position */
        XVT_POPUP_LEFT_ALIGN       /* Left-aligned below
                                        specified position */
        XVT_POPUP_RIGHT_ALIGN      /* Right-aligned below
                                        specified position */
        XVT_POPUP_OVER_ITEM        /* Centered with respect to
                                        a specific menu item */
} XVT_POPUP_ALIGNMENT;
```

Implementation Note: On XVT/Win32, and XVT/Mac, xvt_menu_popup returns only after the user has selected a menu item or has dismissed the pop-up menu without making a selection. On XVT/Win32, the event is posted to the native queue. On XVT/Mac, E_COMMAND events are sent prior to xvt_menu_popup returning and only if the user makes a selection. On XVT/XM, xvt_menu_popup returns immediately after being called. E_COMMAND events are sent some time after xvt_menu_popup returns and only if the user selects a menu item. As you write your application, consider these differences in event delivery time.

Example: This example demonstrates how to create different pop-up menus when a user clicks on different regions of a window:

```
long XVT_CALLCONV1 win_eh(WINDOW win, EVENT *ep)
{
        static MENU_ITEM *popup_menus;
        PNT pnt;
        ...
        switch (ep->type) {
                ...
                case E_CREATE:
                /* obtain array of popup menus from resources */
                        popup_menus = xvt_res_get_menu (POPUP_MENUS);
                        ...
                        break;
                 case E_MOUSE_DOWN:
                        pnt = event->v.mouse.where;

                        if (pnt.h > 100 && pnt.h < 150 &&
                                pnt.v > 100 && pnt.v < 150)
                                xvt_popup_menu(popup_menus[0]->child, win,
                                        pnt, XVT_POPUP_LEFT_ALIGNED, 0);
                        else if (pnt.h > 150 && pnt.h < 200 &&
                                pnt.v > 150 && pnt.v < 200)
                                xvt_popup_menu (popup_menus[1]->child, win,
                                        pnt, XVT_POPUP_CENTERED, 0);
                        else
                                /* position menu, centering over menu item
                                        with USER_ITEM_TAG id */
                                xvt_popup_menu (popup_menus[2]->child, win,
                                        pnt, XVT_POPUP_OVER_ITEM,
                                        USER_ITEM_TAG);
                        break;
                ...
        }
}
```

10

COORDINATE SYSTEMS

XVT always expresses coordinates in pixels. “Pixel” is a shorthand term for “picture elements.” Pixelsare the individual dots making up the image on a screen or printer.

The point (0, 0) is at the upper-left corner, and the positive directions are to the right and down. This is true no matter what the underlying window system uses for its origin; if necessary, XVT translates.

This chapter discusses the following topics that relate to coordinate systems:

- How coordinates are defined relative to containers (SCREEN_WIN and TASK_WIN)
- Where the client (drawing) area of a window is located
- How text is drawn relative to coordinates
- How to use points (PNTs) and rectangles (RCTs) as coordinates
- How to get information about display and system metrics

10.1. SCREEN_WIN and TASK_WIN

XVT defines all coordinates relative to their container’s coordinate system. The SCREEN_WIN is at the top of the XVT window/dialog hierarchy; it constitutesthe coordinate system of the physical screen. TASK_WIN is then defined relative to SCREEN_WIN.

XVT dialogs are defined relative to SCREEN_WIN; top-level windows are defined relative to TASK_WIN. Child windows and controls are defined relative to their parent window (a dialog, a top-level window, or another child window).

SCREEN_WIN and TASK_WIN relate in varying ways, depending on the platform:

- XVT/XM — TASK_WIN is the same as SCREEN_WIN
- XVT/Win32 — TASK_WIN is a physical window that is defined relative to its SCREEN_WIN container
- XVT/Mac — SCREEN_WIN and TASK_WIN have the same origin, but this origin starts just below the fixed menubar

The differences in screen and task windows between the various XVT-supported platforms are shown in Figure 10.1.

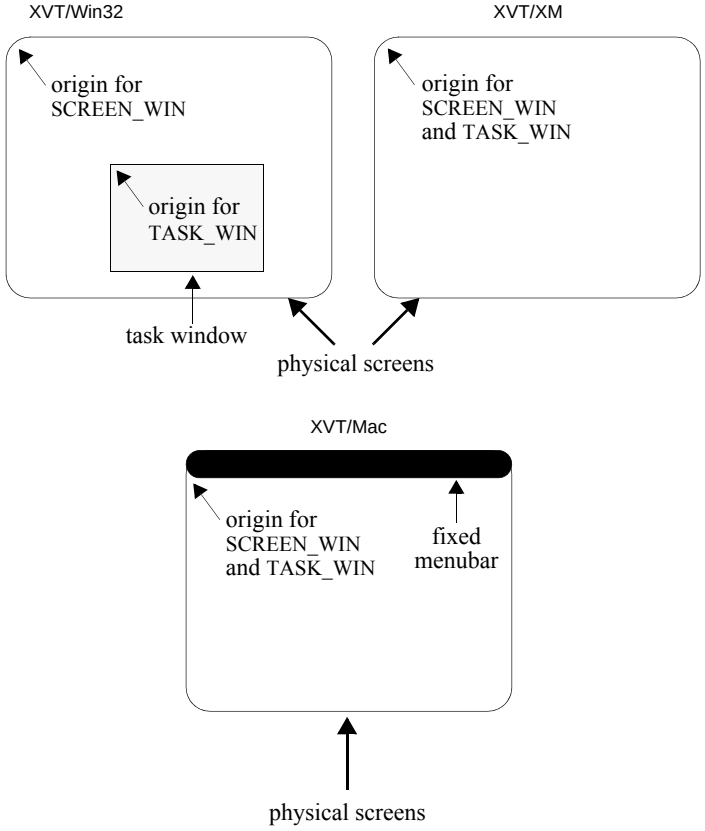


Figure 10.1. SCREEN_WIN and TASK_WIN relationships and origins on different supported platforms

It's sometimes necessary to translate a point or group of points from the coordinate system of one WINDOW to that of another. You can convert points among SCREEN_WIN, TASK_WIN, windows, dialogs, and controls.

Tip: To convert points across coordinate systems:

Call xvt_vobj_translate_points.

10.2. Client Area Location

The client area of a window is the part of the window in which you can draw. The client area starts just inside the window frame. The pixel located at (0,0) is the highest and farthest left point that you can draw on (in other words, it is the location of the upper-left-most pixel that can be turned on or off). The location of the (0,0) coordinate is shown in Figure 10.2.

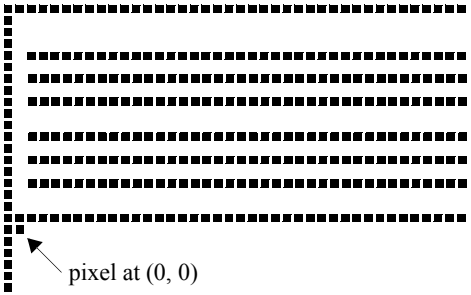


Figure 10.2. Enlarged view of the upper-left corner of a window's client area

See Also: For more information about windows, see Chapter 6, *Windows*. For more information about drawing in the client areas of windows, see Chapter 11, *Drawing and Pictures*.

10.3. Coordinates for Drawing Text

When you call xvt_dwin_draw_text at a starting point, the text starts at the x-coordinate and writes on a "baseline" located at the y-coordinate. Letters with descenders dip below the baseline.

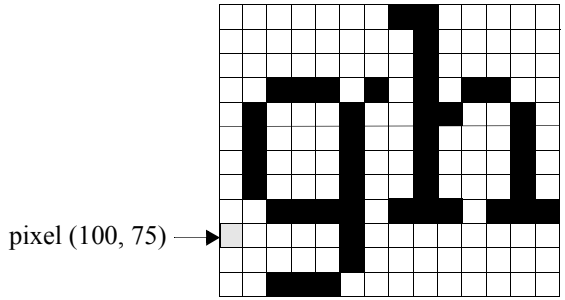


Figure 10.3. Text drawn at (100, 75)

Example: When you draw text at the point (100,75), the text begins to the right of the x-coordinate imaginary line (value 100), and writes above the y-coordinate imaginary line (value 75). The placement of the letters in the drawn text is shown in Figure 10.3.

See Also: For more information about drawing text, see Chapter 15, *Fonts and Text*.

10.4. Points and Rectangles

Because points and rectangles are so widely used, XVT defines two data types for them, PNT for points and RCT for rectangles:

```
typedef struct {            /* mathematical point */
        short v;            /* vertical (y) coordinate */
        short h;            /* horizontal (x) coordinate */
} PNT;

typedef struct {            /* mathematical rectangle */
        short top;          /* top coordinate */
        short left;         /* left coordinate */
        short bottom;       /* bottom coordinate */
        short right;        /* right coordinate */
} RCT;
```

Points and rectangles are simply mathematical entities; they don't appear on the screen. You can use them for screen or window-relative coordinates, and any other purpose for which they're convenient. The placement ofboth a single point and a rectangle (as they might be drawn by an application) are shown in Figure 10.4.

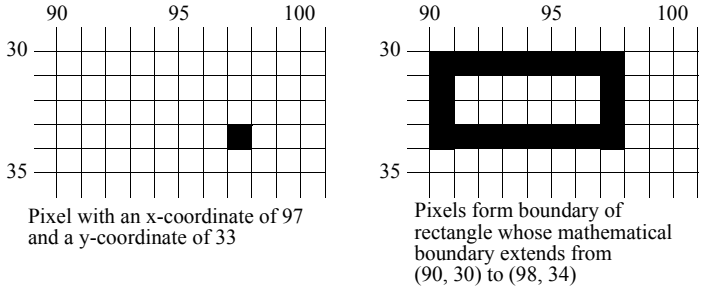


Figure 10.4. A window's coordinate system

XVT provides several functions for manipulating PNTs and RCTs.

Tip: To set the height of a rectangle:

Call xvt_rect_set_height.

Tip: To get the height of a rectangle:

Call xvt_rect_get_height.

Tip: To set the position of a rectangle:

Call xvt_rect_set_pos.

Tip: To get the position of a rectangle:

Call xvt_rect_get_pos.

Tip: To set the width of a rectangle:

Call xvt_rect_set_width.

Tip: To get the width of a rectangle:

Call xvt_rect_get_width.

Tip: To determine if a rectangle is empty:

Call xvt_rect_is_empty.

Tip: To set a rectangle to empty:

Call xvt_rect_set_empty.

Tip: To set a rectangle's coordinates to specific values:

Call xvt_rect_set.

Tip: To offset a rectangle horizontally and/or vertically:

Call xvt_rect_offset.

Tip: To test whether a point is in a rectangle:

Call xvt_rect_has_point.

Tip: To determine whether two rectangles intersect and, if so, to get the intersection:

Call xvt_rect_intersect.

Note: A point located exactly on the right or bottom boundary of a rectangle is *not* in it, whereas a point located exactly on the left or upper boundary *is* in the rectangle.

See Also: For more information about rectangle fills, see section 11.1.3.1 on page 11-13.
For more information on the xvt_rect_* functions, see the *XVT Portability Toolkit Reference*.

10.5. Display and System Metrics

When your application draws in a window, you can't assume anything about the size of the screen, nor can you assume that pixels are square. For instance, a rectangle 100 pixels wide and 100 pixels high may not look like a square to the user. Also, the actual thickness of a two-pixel-wide line depends on the number of pixels per inch of the device on which the line is drawn.

Tip: It is possible for you to draw in a logical coordinate system of your own design that's appropriate for your application. Then, when you output to a specific window, translate those coordinates to pixels.

To help, XVT provides the following system attributes that give some information about the display.

Display Metrics Attributes:	**Description:**
ATTR_SCREEN_HEIGHT	Height of screen
ATTR_SCREEN_WIDTH	Width of screen
ATTR_SCREEN_HRES	Horizontal resolution of screen in dpi
ATTR_SCREEN_VRES	Vertical resolution of screen in dpi
ATTR_PRINTER_HEIGHT	Height of printer page (for default printer)
ATTR_PRINTER_WIDTH	Width of printer page (for default printer)
ATTR_PRINTER_HRES	Horizontal resolution of printer in dpi (for default printer)
ATTR_PRINTER_VRES	Vertical resolution of printer in dpi (for default printer)
ATTR_DOC_STAGGER_HORZ	Recommended horizontal document stagger
ATTR_DOC_STAGGER_VERT	Recommended vertical document stagger

You can use these attributes with xvt_vobj_get_attr to obtain the corresponding value.

See Also: For more information, see the "Attributes" portion of the *XVT Portability Toolkit Reference*.

11

DRAWING AND PICTURES

Drawing refers to graphical operations performed in a window. *Pictures* are collections of drawing functions.

XVT provides a set of portable graphics drawing primitives and graphical attributes (or drawing tools) to allow applications to output graphics in windows. XVT supports the following graphics primitives: line, circle/oval, polyline (multiple-segmented line), arc, polygon, pie segment, rectangle, font-based text, icon, arrow head, and rounded rectangle.

XVT PICTUREs

XVT provides an abstraction called a PICTURE for capturing graphics drawn into a window, for passing them to other applications via the clipboard, for archival, or for later redisplay. Essentially, PICTUREs are one of the set of XVT graphics primitives, but they are more powerful.

These encapsulated PICTUREs can be saved and/or retrieved from files, scaled, and drawn into windows with just one function call. Even though their physical representation differs on each platform, their logical and programmatic representation is the same; thus, their use within an XVT application is portable.

See Also: For more information about PICTUREs, see section 11.2 on page 11-16.

Portable Images

XVT also provides a portable images feature, which manipulates, displays, and prints bitmapped graphic images (XVT_IMAGEs and XVT_PIXMAPs). You can use portable images in many ways, for example as graphics within windows, as labels for Toggle/Picture Buttons (a type of XVT Custom Component), or as icons on a Toolbar (another XVT Custom Component).

See Also: For more information about portable images, see Chapter 12, *Portable Images*.

11.1. Drawing

In XVT, all graphical operations are window-specific. XVT graphical operations include the drawing of graphics or text, and the specification of graphical or textual attributes. Each window maintains its own set of graphical attributes (color, pens, brushes, font, etc.).

Note: You cannot draw to dialogs or controls, however, you can draw to modal windows.

11.1.1. Color

You can specify colors for the outlines of shapes, for their interiors, and for text. This section focuses on the concept of color.

The next section, section 11.1.2 on page 11-4, gives details about how you can set the color of four things in XVT: the pen, the brush, the foreground (used for text and icons), and the background (used for the spaces between the hatch marks of brushes and the characters in text).

The RGB Model

XVT uses a 24-bit number to specify a color. The 24 bits are divided into three 8-bit values for the red, green, and blue components. This color model is referred to as the RGB model

For each component, a value of 0 means no color and a value of 255 means the maximum color. So, pure white would be 0xFFFFFF, pure black would be 0x000000, pure red would be 0xFF0000, and so on. Equal values for the three components produce shades of gray; a medium gray would be 0x808080. (From this perspective, pure white is the lightest shade of gray and pure black is the darkest shade of gray.)

Predefined Colors

For your convenience, XVT provides symbols for eleven colors:

```
#define COLOR_RED ...
#define COLOR_GREEN ...
#define COLOR_BLUE ...
#define COLOR_CYAN ...
#define COLOR_MAGENTA ...
#define COLOR_YELLOW ...
#define COLOR_BLACK ...
#define COLOR_DKGRAY ...
#define COLOR_GRAY ...
#define COLOR_LTGRAY ...
#define COLOR_WHITE ...
```

You can use other colors if you wish. However, if your target platform doesn't flexibly support color, XVT recommends that you use one of the eleven listed colors.

Using Colors on Monochrome Screens

Regardless of which XVT implementation and target hardware you're using, you can always use colors. XVT maps all requested colors (other than black and white) drawn with a solid pen or brush pattern to a grayscale pattern.

For monochrome screens, XVT supports a minimum of five levels of "gray," corresponding to XVT's COLOR_WHITE, COLOR_BLACK, COLOR_LTGRAY, COLOR_GRAY, and COLOR_DKGRAY. XVT represents shades of gray with intermittent dots or stipple patterns. The stipple patterns for light grays can cause some lines (typically vertical lines) to appear narrower than requested.

Color Guidelines

Since white-on-white and black-on-black makes whatever you're drawing invisible, be careful about using more colors than the target hardware can support. Here are some guidelines:

- If color is optional, but isn't necessary to convey information, go ahead and use it freely. Use deep colors on a white background to prevent a light color from being mapped to very light gray on a monochrome or grayscale monitor. Test your application to make sure that everything is visible.
- If color is required, and your application requires at least a grayscale monitor, stick to the grays to be safe.
- If color is required, make sure your users know this and run your application on proper equipment.

- If you must code differently depending on whether color is available, you can check the color capabilities of your target hardware by using the ATTR_DISPLAY_TYPE attribute.

See Also: For information about setting color for pens, brushes, foreground, and background, see section 11.1.2, next.

11.1.1.1. Allowing Users to Choose Colors

Calling xvt_dm_post_color_sel brings up a dialog that allows the user to select a color. The dialog, shown in Figure 11.1 below returns the color the user has chosen.

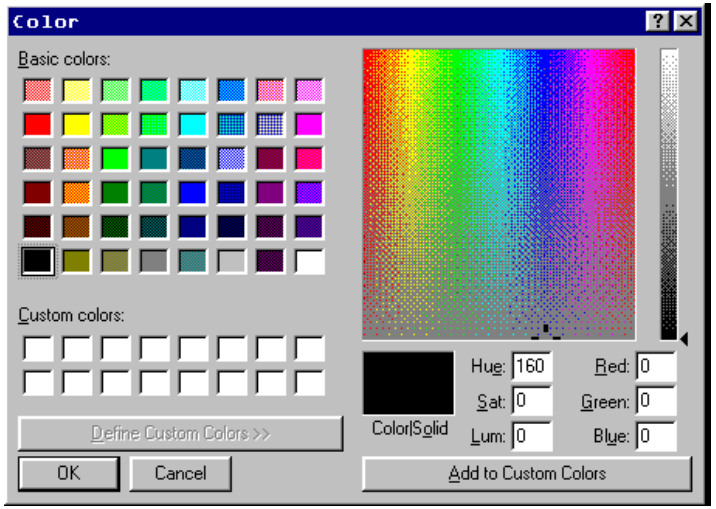


Figure 11.1. The color selection dialog

11.1.2. Drawing Tools

Each XVT window (but not dialog box or control) has a collection of associated drawing tools. The structure DRAW_CTOOLS records these tools:

```
typedef struct s_drawct {              /* set of drawing tools */
        CPEN pen;                      /* color pen */
        CBRUSH brush;                  /* color brush */
        DRAW_MODE mode;                /* drawing mode */
        COLOR fore_color;              /* foreground color */
        COLOR back_color;              /* background color */
        BOOLEAN opaque_text;           /* is text drawn
                                                opaquely? */
} DRAW_CTOOLS;
```

The sections that follow discuss each member of this structure and their types.

11.1.2.1. Pens

A pen draws lines and the outline of closed shapes, while a brush is used for the interior of closed shapes. Figure 11.1 shows which shapes have a pen or a brush.

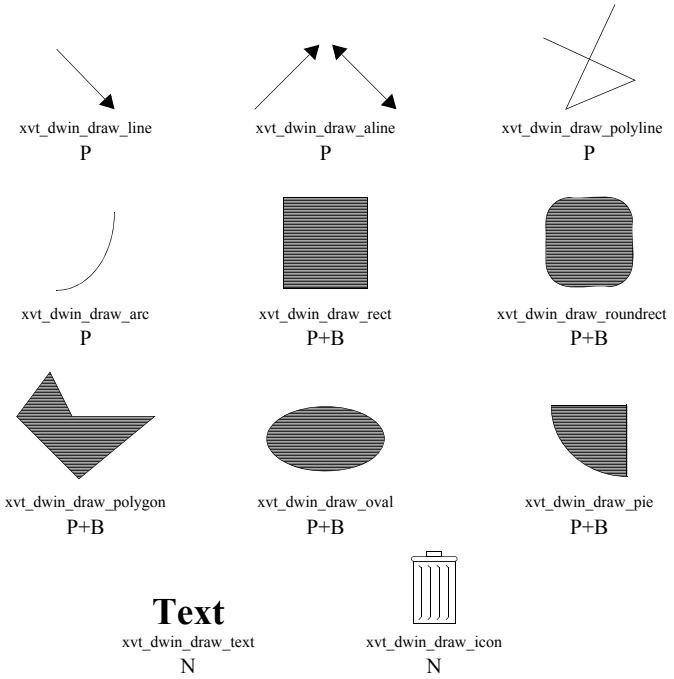


Figure 11.1. XVT drawing functions that use a pen (P), brush(B), or neither (N)

Tip: To specify characteristics of a pen:

Use a CPEN structure:

```
typedef struct s_cpen {
      short width;              /* width */
      PAT_STYLE pat;            /* pattern */
      PEN_STYLE style;          /* style (P_SOLID) */
      COLOR color;              /* color */
} CPEN;
```

The width member specifies the width of the pen stroke in pixels. The pat member specifies the pen pattern. pat has the following allowable values:

PAT_SOLID
: A normal pen that draws a solid line in the specified width with the specified color.

PAT_HOLLOW
: No pen at all. When this is set, shapes made with a brush (e.g., an oval) do not have a border around them, and shapes with only a pen aren't seen at all.

PAT_RUBBER
: A grayish or dotted line useful when rubberbanding. This is the technique of allowing the user to stretch a rectangle or other shape as the mouse is moved (Figure 11.2). Drawing is always done with a mode of M_XOR.

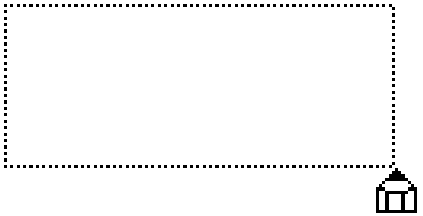


Figure 11.2. Stretching out a rubberband by dragging the mouse

PAT_STYLE Structure

The following is the PAT_STYLE structure. Values other than the three listed above can't be used for pens, but the next section uses them for brushes. One value, PAT_NONE, is for XVT's internal use and should never be used by an application.

```
typedef enum {
        PAT_NONE,           /* no pattern */
        PAT_HOLLOW,         /* hollow */
        PAT_SOLID,          /* solid fill */
        PAT_HORZ,           /* horizontal lines */
        PAT_VERT,           /* vertical lines */
        PAT_FDIAG,          /* diagonal lines—top-left to
                               bottom-right */
        PAT_BDIAG,          /* diagonal lines—top-right to
                               bottom-left */
        PAT_CROSS,          /* horizontal and vertical
                               crossing lines */
        PAT_DIAGCROSS,      /* diagonal crossing lines */
        PAT_RUBBER,         /* rubberbanding */
        PAT_SPECIAL
} PAT_STYLE;
```

Pen Styles

The following enumeration shows the pen styles used in the style field of the CPEN object. These pen styles are meaningful only when the pen pattern is PAT_SOLID. On some platforms, the native toolkit might not support these styles; thus XVT might not support them.

```
typedef enum e_pen_style {              /* pen style */
        P_SOLID,                        /* solid */
        P_DOT,                          /* dotted line */
        P_DASH                          /* dashed line */
} PEN_STYLE;
```

Finally, the color specifies the RGB color, as explained above.

Tip: To set a window's pen:

Call xvt_dwin_set_cpen.

Tip: To explicitly assign a value to the current pen:

Call xvt_dwin_set_std_cpen. The choices are:

- TL_PEN_BLACK
- TL_PEN_HOLLOW
- TL_PEN_RUBBER
- TL_PEN_WHITE

Tip: To set the pen (and all of the other drawing tools):

Call xvt_dwin_set_draw_ctools.

Tip: To find out the current pen:

Call xvt_dwin_get_draw_ctools.

11.1.2.2. Brushes and Background Colors

Tip: To specify characteristics of a brush:

Use a CBRUSH structure:

```
typedef struct s_cbrush {
        PAT_STYLE pat;
        COLOR color;
} CBRUSH;
```

The pat is the pattern; the color is the color of the patterns ink. Figure 11.3 shows the patterned PAT_STYLEs that you can use for brushes (see the previous section for the PAT_STYLE structure).

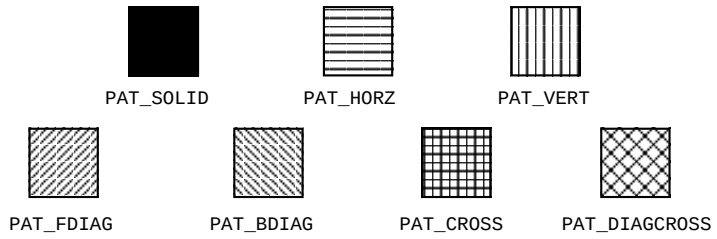


Figure 11.3. XVT's predefined CBRUSH patterns

The hatches drawn by a brush are always in the brush color (member color of the current CBRUSH); the spaces between the hatches are drawn in the current background color (Figure 11.4). The background color is not part of a CBRUSH, but is set separately with xvt_dwin_set_back_color (or xvt_dwin_set_draw_ctools). To draw with a solid color, set the pattern to PAT_SOLID and the color to whatever RGB value you want.

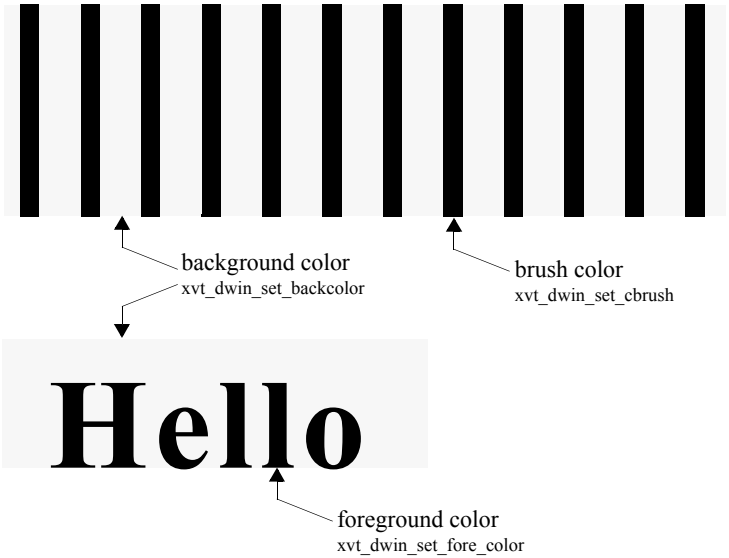


Figure 11.4. A hatched brush (PAT_VERT, here) uses the brush color for the hatching and the background color for the open spaces and opaque text. The ink color for text is determined by the foreground color.

Tip: To set the current brush for a window:

Call xvt_dwin_set_cbrush.

Tip: To set the brush (and all of the other drawing tools):

Call xvt_dwin_set_draw_ctools.

Tip: To find out the current brush:

Call xvt_dwin_get_draw_ctools.

Tip: To set a window's brush to a standard value:

Call xvt_dwin_set_std_cbrush. The choices are:

- TL_BRUSH_BLACK
- TL_BRUSH_WHITE

11.1.2.3. Foreground Colors—Opaque and Transparent Text

Tip: To set the color for text:

Call xvt_dwin_set_fore_color.

This function affects only the ink; the spaces between the letters are either not drawn (whatever was already there shows through), or are drawn opaquely with the current background color.

Tip: To determine whether text is drawn with an opaque background:

Set the opaque_text member of a DRAW_CTOOLS structure and call xvt_dwin_set_draw_ctools.

Note: There is no function for setting only the opacity of the background for text.

On certain platforms, the foreground and background colors are also used when drawing icons (with xvt_dwin_draw_icon). For others, icons are always drawn in black and white. The only way to guarantee that icons are portably drawn in certain colors is to specify a black foreground and a white background, which is the XVT default when a new window is created.

11.1.2.4. Drawing Mode

Tip: To set the current draw mode:

Call xvt_dwin_set_draw_mode or xvt_dwin_set_draw_ctools.

The mode member of the DRAW_CTOOLS structure determines how pixels to be drawn are combined with pixels already on the screen. These modes are allowed:

```
typedef enum {
        M_COPY,
        M_OR,
        M_XOR,
        M_CLEAR,
        M_NOT_COPY,
        M_NOT_OR,
        M_NOT_XOR,
        M_NOT_CLEAR
} DRAW_MODE;
```

The common modes are M_COPY (the default), and M_XOR. The six other drawing modes are used only in painting programs.

M_COPY

M_COPY ignores what's on the screen and copies drawn pixels. Use M_COPY for printing, as some print drivers can't handle any other method of transferring pixels to paper.

M_XOR

The purpose of M_XOR is to temporarily show something on the screen. It does this by toggling what's drawn on the screen, such that a second identical drawing operation restores what was present before the first drawing. This is important because you may not know (or care) what was there before, and because it's difficult to update the screen with the old contents rapidly enough to keep up with a moving mouse.

M_XOR combines new and old pixels so that these rules are obeyed:

- Drawing the same shape twice with an M_XOR mode is guaranteed to leave the screen as it was
- Drawing a shape once with an M_XOR mode allows you to see the shape if at all possible

All XVT implementations obey the first rule at all times. The second rule is always obeyed when you are drawing in black and white, but can be violated if your current color palette is loaded with very unfavorable colors.

Figure 11.2 on page 11-6 shows rubberbanding with an M_XOR mode. To see an example of a C function that draws a rubberband rectangle using M_XOR mode, refer to page 4-52.

See Also: Another common example of rubberbanding is selecting text. For more information about selecting text, see section 15.8.2 in Chapter 15, "*Fonts and Text*."

11.1.2.5. Manipulating Drawing Tools

Before drawing anything in a window, make sure that the window has the correct drawing tool settings by calling xvt_dwin_set_cpen, xvt_dwin_set_cbrush, xvt_dwin_set_draw_mode, and xvt_dwin_set_font as necessary. You can also set all the tools at once with a call to xvt_dwin_set_draw_ctools. You normally call these functions only to change a tool, since settings are saved.

Tip: To make a temporary change without affecting the current setting:

> Bracket your tool changes and associated drawing with calls to xvt_dwin_get_draw_ctools and xvt_dwin_set_draw_ctools.

Tip: To get a set of normal tools:

> Call xvt_app_get_default_ctools and use the result in a call to xvt_dwin_set_draw_ctools.

The normal tools are also the default tools for a newly created window:

- A one pixel wide, solid, black pen (TL_PEN_BLACK)
- A solid, white brush (TL_BRUSH_WHITE)
- Foreground and background colors of black and white, respectively
- A font considered to be a "system font" (this is usually a proportionally spaced font about 10 points in size)
- An M_COPY drawing mode
- A transparent text background (the opaque_text member of the DRAW_CTOOLS structure set to FALSE)

In addition to ensuring that the current window is specified and that the tools in that window are set, you should also ensure that the clipping rectangle is set. Clipping is not recorded in the DRAW_CTOOLS structure, so merely setting the tools won't affect the clipping.

Tip: To turn off clipping, use this function call:

```
xvt_dwin_set_clip(win, NULL);
```

11.1.2.6. Allowing Users to Change Drawing Tools

XVT also has the ability to let the user choose the different components of the DRAW_CTOOLS structure. The dialog shown in Figure 11.5 below has four tabs, and any or all of these tabs may be presented to the user by choosing various combinations of the XVT_CTOOLS* constants.

See Also: For more information on the DRAW_CTOOLS struct and the xvt_dm_post_ctools_sel dialog, see the *XVT Portability Toolkit Reference.*

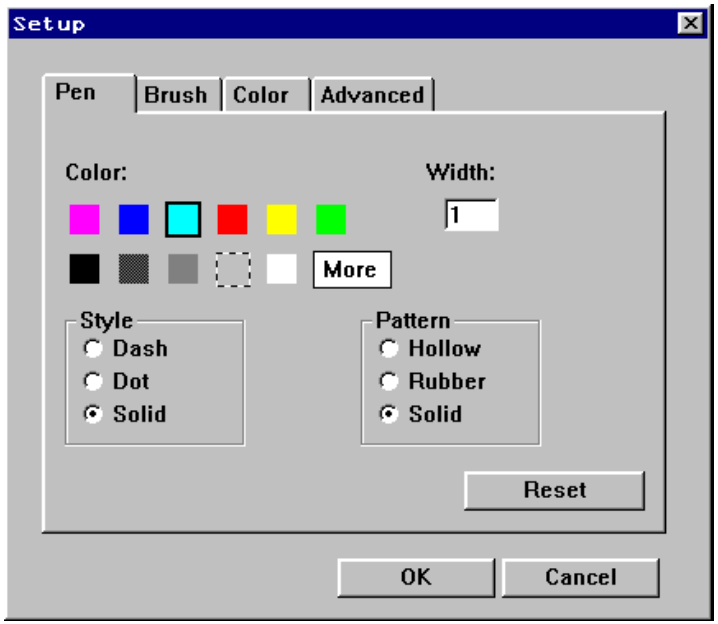


Figure 11.5. The drawing tools selection dialog

11.1.3. Graphic Shapes, Text, and Pictures

Tip: To draw shapes:

Use one of the following functions:

```
xvt_dwin_draw_aline
xvt_dwin_draw_arc
```

xvt_dwin_draw_image
xvt_dwin_draw_line (along with xvt_dwin_draw_set_pos)
xvt_dwin_draw_oval
xvt_dwin_draw_pie
xvt_dwin_draw_pmap
xvt_dwin_draw_polygon
xvt_dwin_draw_polyline
xvt_dwin_draw_roundrect.

See Also: For information about drawing portable images and pixmaps, see Chapter 12, *Portable Images*.

Tip: To draw text:

Call xvt_dwin_draw_text.

You must specify the starting x- and y-coordinates (relative to the client area of the window) and the text itself. The y-coordinate is that of the baseline, so in general the text extends above it and below it.

Tip: To draw an encapsulated picture:

Call xvt_dwin_draw_pict.

None of the XVT color-setting functions affects the result of xvt_dwin_draw_pict. However, you can capture color drawing in the picture when it is created.

All drawing is expressed in pixelcoordinates, which vary according to the device. In particular, the screen and the printer use different coordinate systems.

See Also: For more information about encapsulated pictures, see section 11.2 on page 11-16.
For more information about coordinate systems, see Chapter 10, *Coordinate Systems*.

11.1.3.1. Rectangle Fills

XVT uses an “exclusive model” to fill all rectangle-based areas. This applies to client and outer rectangles for windows, pixmaps, pictures, and images, as well as for clip rectangles, fills for drawn rectangles, oval arcs, and pie shapes. It also applies to rectangles for xvt_dwin_scroll_rect.

Under the exclusive model, the point specifying the rectangle’s bottom-right corner lies outside the rectangle. The point specifying the rectangle’s upper-left corner lies inside the rectangle, as shown in Figure 11.6.

The pixel size of a rectangle is computed like this:

```
width = rct.right - rct.left;
height = rct.bottom - rct.top;
```

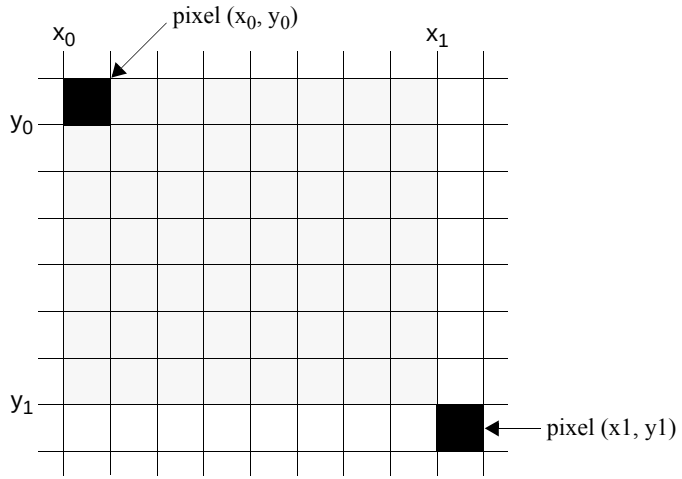


Figure 11.6. Exclusive model for rectangle fills

Calling xvt_vobj_get_outer_rect for a window or control returns a rectangle that would exactly cover the object and its decorations.

Implementation Note: Decorations and decoration sizes forcontrols, such as default button outlines or focus indicators, vary between platforms.

Note: A rectangle with a width or height of zero is illegal for xvt_dwin_draw_* functions.

11.1.3.2. Rectangle Outlines

The line outlining a rectangle centers around the perimeter of the "exclusive" fill rectangle. This applies to rectangles, round rectangles, ovals, arcs, and pie shapes. This means that a single pixel-width outline goes through the points (rct.left, rct.top) and (rct.right-1, rct.bottom-1).

Outlining lines with a width greater than one pixel center on this perimeter. Lines with an even width cannot be centered; the extra pixel width falls to the bottom and right of the perimeter. (This is called "southeast gravity.")

11.1.3.3. Lines, Polylines, and Polygons

Lines, polylines, and polygons connect specified points. This differs from rectangle specifications as discussed in the previous section. Recall that for rectangles, the exclusive model specifies that the bottom-right point lies outside the drawn shape.

Tip: To exactly overlay a rectangle on a line-based drawing:

Increase the rectangle's bottom-right point coordinates by one in each direction.

Zero-pen-width lines
XVT interprets lines of zero pen width as thin lines. These lines draw using the hardware acceleration available on the native platform, which may not follow the rules for lines of width equal to one.

Wide lines
A wide line centers on the thin line connecting its specified points. Lines with an even width cannot be centered; the extra pixel width falls to the bottom and right of the perimeter.

PAT_HOLLOW lines
XVT ignores the line width for lines with style PAT_HOLLOW. For filled shapes such as rectangles and polygons, the outline becomes a part of the fill; the brush color and pattern apply to them. The effect is the same as interpreting PAT_HOLLOW as zero line width.

Implementation Note: Some platforms may deviate from the PAT_HOLLOW rule just mentioned, in the case of polygon fills using a brush pattern other than PAT_SOLID. If your application requires exact brush fills, use rectangular shapes.

11.1.3.4. Line Caps and Joints

XVT's drawing model assumes a round pen of a prescribed width. This round pen approximation is not exact. For example, the Macintosh drawing model uses a rectangular pen; in this case emulating the round pen would yield unacceptable drawing performance.

Line caps
The terminations at the end of lines. Following the round pen model, the line caps are round. Line caps are centered around the line's end points. This implies that a line specified by an identical point for both end points will be represented bya circle centered at the point.

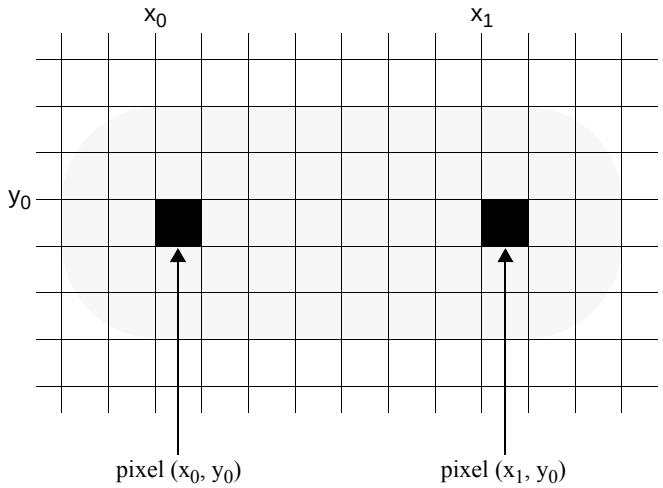


Figure 11.7. The round pen model, showing rounded line caps for a line (of width 5) drawn from x0,y0 to x1,y0

Line joints

The connections between line segments. The line joints for lines with width greater than one use the closest approximation of the round pen model on each platform. This may be a round, mitered, or other platform-specific rendering model. Applications sensitive to exact thick line joint rendering, or applications requiring straight line caps, should use filled shapes such as rectangles or arcs instead of wide lines.

11.2. Pictures

XVT provides an abstraction called a picture, which combines a sequence of drawing operations—tool changes, shapes, and text—into a standard encapsulated form that other, non-XVT applications can interpret.

With a picture, you can capture graphics drawn into a window, then pass them to other applications via the clipboard, archive them, or redisplay them. You can also bring encapsulated pictures into an XVT application. Once a picture is encapsulated, you can't operate on its individual components, but you can scale it by changing the size of its framing rectangle.

Essentially, pictures are one of the set of XVT graphics primitives, but they are more powerful. With just one function call, you can save

and/or retrieved encapsulated pictures from files, scale them, and draw them into windows. Even though their physical representation differs on each platform, their logical and programmatic representation is the same; thus, you can use them portably within any XVT-based application.

An encapsulated picture is referred to by a 32-bit descriptor of type PICTURE, the internals of which are hidden from XVT applications:

```
typedef long PICTURE;
```

Tip: Design your XVT application to be insensitive to the exact form of encapsulated pictures. The representations that XVT uses were chosen because they are used by most painting programs. However, they could change in future releases.

See Also: For more information, see the "PICTURE" data type in the *XVT Portability Toolkit Reference*.

11.2.1. Creating and Accessing Pictures

Tip: To create a picture:

1. Call xvt_dwin_open_pict with a frame rectangle, relative to the current window, that delimits the part to be encapsulated.
2. Draw in the current window. Instead of showing up on the screen, your actions are recorded.
3. To return the PICTURE object when you're done, call xvt_dwin_close_pict.

Tip: To release storage when you no longer need a picture:

Call xvt_pict_destroy.

Tip: To draw a PICTURE in the current window:

Call xvt_dwin_draw_pict.

XVT stretches or shrinks the picture to fit the new frame rectangle you supply.

Tip: To put a PICTURE onto the clipboard:

Call xvt_cb_put_data.

11.2.2. Saving and Retrieving Pictures From Files

If you want to save a PICTURE to a file, you must flatten it into a sequence of bytes.

Tip: To save a picture to a file:

1. Call xvt_pict_lock, which returns a character pointer and the number of bytes it points to.
2. Write the data byte-by-byte with a call to the standard C functions fwrite or write.
3. When you're done with the pointer, call xvt_pict_unlock.

When you read in sequential bytes that were originally accessed via xvt_pict_lock (with fread or read), you can reconstruct a PICTURE by calling xvt_pict_create. It returns the frame rectangle to you so you can use it in a call to xvt_dwin_draw_pict.

Tip: To transform a flattened clipboard picture to a real picture:

1. Call xvt_cb_get_data to get a flattened PICTURE from the clipboard.
2. Call xvt_pict_create to transform it into a genuine PICTURE.

Implementation Note: On XVT/Mac, a PICTURE is a PICT (which is general enough to include bitmaps). On XVT/Win32, it's a bitmap. However, don't write your applications to assume that these are the formats, since the bitmap representations are likely to change with future XVT releases.

12

PORTABLE IMAGES

XVT's portable images feature lets your application manipulate, display, and print bitmapped graphic images—in several different file formats—on all XVT Portability Toolkit platforms. You can create images on the platform(s) of your choice and easily move them to other platforms.

The portable images feature includes these key elements:

- You can use three different color formats: monochrome, 8-bit indexed color, and 24-bit RGB color
- Color formats are independent of display hardware, so your application can manipulate images internally without hardware restrictions
- XVT provides file I/O functions for several different image file formats, including Win32 **BMP**, X Window System **xpm** and **xbm**, and Macintosh **PICT**
- You can copy rectangular regions from one image to another
- You can scale, translate, and stretch rectangular regions
- Color palettes and color mapping let your application display images on devices with limited color capabilities

Because the native graphics systems of the XVT-supported platforms use different standard formats, XVT has established the Win32 **BMP** format as the portable image file format for all XVT products. The XVT Portability Toolkit can display Win32 **BMP** images on any supported platform, automatically making any necessary conversions.

Using Portable Images

You can use portable images in many ways:

- As graphics within windows in your application
- As labels for Toggle/Picture Buttons (a type of XVT Custom Control)
- As icons on a toolbar or palette (by labeling Toggle/Picture Buttons within a Toolbar, which is another XVT Custom Control)

To obtain an image that is portable to all platforms, you create the image only once on any of the supported platforms.

12.1. Image Terminology

This section defines three important terms that appear throughout this chapter: pixel, image, and pixmap.

Pixel

A pixel (from "picture element") is a single dot in a graphic image. In black-and-white images, pixels can be represented by single bits. In color images, pixels are represented by two or more bits.

Image

An image is a rectangular array of pixels that exists in memory addressable by your application. Images are independent of output devices (CRT displays or printers), and hence are not drawn directly on the screen. You cannot use XVT drawing commands to draw into an image. However, your application can directly manipulate the pixels of an image.

Pixmap

A pixmap is a device-dependent array of pixels that can be displayed in a window. You can use XVT drawing commands to draw into pixmaps. However, your application cannot directly modify the pixels of a pixmap.

12.2. Color

All XVT graphics, image, and pixmap operations use a red-green-blue (RGB) color model. The COLOR data type provides eight-bit level resolution for each color component.

Images and pixmaps use one of three color formats:

- Monochrome format
- Indexed format
- True color format

The formats differ in the number of colors displayable and in the amount of memory used by one pixel. The following table summarizes the three formats:

Format:	Displayable Colors:	Memory per Pixel:
Monochrome	Two (black and white)	One bit
Indexed	Up to 256	One byte
True Color	Up to 16 million (32 bits; eight are not currently used)	One COLOR value

12.2.1. Color Look-Up Tables

The indexed color mode uses a color look-up table (or "CLUT"). Each pixel uses one byte as anindex into the look-up table. A CLUT contains up to 256 COLOR entries. As a result, an image or pixmap that uses indexed color can contain up to 256 different colors simultaneously.

See Also: For more information about CLUTs, see section 12.5.1.4 on page 12-7.

12.2.2. Color Mapping

If a region is transferred between images or pixmaps with different color formats, some colors in the region may be changed, or mapped, to different colors. Since different image color formats can represent different numbers of colors, colors in the source image or pixmap must be mapped onto colors in the destination.

When the destination uses true color, the mapping is trivial and no colors are changed, since any color in the source can be represented in the destination. In all other cases, the mapping operation finds the

closest-matching color in the destination for each color in the source. The "closest" color is found by minimizing the sum of the differences between the red, green, and blue components of the source and destination colors.

Color mapping is performed either by the XVT image- and pixmap-transfer functions, or by the underlying window system. Your application has no direct control over how colors are mapped. However, when an indexed color image is the destination, your application can choose the colors in the destination image by filling its color look-up table before transferring the region.

See Also: For more information about color, see Chapter 11, *Drawing and Pictures*.

12.3. Palettes

While the XVT Portability Toolkit provides three different color formats, not all XVT platforms include hardware capable of supporting all three modes. To accommodate hardware with different color formats (for instance, 16-color or gray-scale displays), XVT uses palettes to translate image and pixmap colors into colors displayable by the physical device.

Palettes let your application display color images on devices with less than full-color features. When displaying an image in a window, the colors in the image are mapped to the closest-matching available colors in the window's palette. Your application can construct its own palettes to optimize the appearance of its images on different display devices.

Note: XVT ignores color palettes for any display supporting more than 256 colors. As this may change in future XVT releases, use the default palette for systems with more than 256 colors.

12.4. Portable File I/O

Your applications can use images created on different GUI environments because XVT Portability Toolkit functions can read several different image file formats. XVT supports the following formats:

- MS-Windows and OS/2 **BMP**
- Macintosh **PICT**
- X Window System **xpm** (level 3 only)
- X Window System **xbm**

Your application does not have to provide special code for each image format. When the files are read, they are converted (in memory) into an XVT image. XVT's stored images are compatible with the MS-Windows **BMP** format.

Note: Reading and writing of Macintosh **PICT** formats is supported on the Macintosh only. For X, level 1 and 2 pixmaps are not supported. Other formats are supported across all XVT platforms.

12.5. Working with Portable Images

12.5.1. Images

Memory-based images are central to the XVT Portability Toolkit's portable image feature. Images translate between different file formats and different display hardware. Images can be drawn into pixmaps, windows, and printers.

12.5.1.1. Image Data Types

The XVT Portability Toolkit defines one data type for handling images and an enumerated type for describing their color formats.

XVT_IMAGE

An object of type XVT_IMAGE represents an image. Because XVT_IMAGE is an opaque data type, your application cannot access its internals directly. Instead, you must use XVT functionsto access and modify images.

XVT_IMAGE_FORMAT

The XVT_IMAGE_FORMAT enumerated type defines three values for the color formats available for images:

XVT_IMAGE_* values:	Format:
XVT_IMAGE_NONE	None
XVT_IMAGE_CL8	Indexed color
XVT_IMAGE_RGB	True color
XVT_IMAGE_MONO	Monochrome

12.5.1.2. Creating and Destroying Images

Tip: To create a new image:

Call xvt_image_create.
-OR-
Call xvt_image_read_*.

With xvt_image_create, you must specify the image's color format, width, and height (in pixels). The contents of the image are not initialized.

Tip: To fill the image with a solid color:

Call xvt_image_fill_rect.

Tip: To remove an image from memory when your application no longer needs it:

Call xvt_image_destroy.

Caution: Do not use free or xvt_mem_free to destroy an image.

See Also: For more information about creating images by reading image files, see section 12.5.5 on page 12-15.

12.5.1.3. Manipulating Images

Images are independent of the window system—they are the same regardless of which platform your application runs on. Because images have no direct association with windows, you cannot use XVT Portability Toolkit drawing functions on images.

However, you can copy some or all of an image to a pixmap, use drawing tools and functions to draw into the pixmap, then copy the pixmap back to the image. Or, your application can manipulate images in several other ways.

Tip: To copy images to and from pixmaps:

Call xvt_dwin_draw_image and xvt_image_get_from_pmap.

Tip: To copy some or all of one image to another image:

Call xvt_image_transfer.

Tip: To change the pixel values directly:

Call xvt_image_get_pixel and xvt_image_set_pixel.

Tip: To retrieve a pointer to a complete horizontal row of pixels:

Call xvt_image_get_scanline.

12.5.1.4. Color Look-up Tables for Indexed-color Images

The color look-up table contained by images with the XVT_IMAGE_CL8 color format determines how the following functions convert pixel values to colors, when operating on these images:

xvt_dwin_draw_image
xvt_image_get_from_pmap
xvt_image_get_pixel
xvt_image_set_pixel
xvt_image_transfer

A color look-up table can contain fewer than 256 colors. When your application creates an image, the image's color table contains two entries: black and white.

Tip: To query the number of entries in the color look-up table:

Call xvt_image_get_ncolors.

Tip: To set the number of entries in the color look-up table:

Call xvt_image_set_ncolors.

The number of colors in the look-up table affects how the image's colors are mapped when image-transfer operations are executed.

Tip: To retrieve color values from a color look-up table:

Call xvt_image_get_clut.

Tip: To change a color value in a color look-up table:

Call xvt_image_set_clut.

Note: Changing an entry in the color look-up table does not change the values of the pixels of the image. It only changes how those values are interpreted by image-transfer and pixel-manipulation functions.

See Also: For more information about color mapping, see section 12.2.2 on page 12-3.

12.5.1.5. Drawing Images

Tip: To draw an image:

Call xvt_dwin_draw_image.

This function draws images in windows, pixmaps, and print windows. Colors in the image are mapped to the closest available colors in the destination window or pixmap. Images can be scaled while being drawn.

See Also: For more information about scaling images, see section 12.5.4 on page 12-14.

12.5.2. Pixmaps

Pixmaps are essentially XVT WINDOWs with no visible screen representation. For most graphics operations, pixmaps are equivalent to WINDOWs. You can copy pixmaps into images, windows, and other pixmaps.

12.5.2.1. Pixmap Data Types

An object of type XVT_PIXMAP represents a pixmap. Because XVT_PIXMAP is an opaque data type, you cannot access its internals directly. You must use XVT Portability Toolkit functions to access and modify pixmaps.

12.5.2.2. Creating and Destroying Pixmaps

Tip: To create a new pixmap:

Call xvt_pmap_create.

You must specify the pixmap's parent window, format (which currently must be XVT_PIXMAP_DEFAULT), width, and height (in pixels). The contents of the pixmap are not initialized.

Tip: To initialize a pixmap:

Call xvt_dwin_clear.

Tip: To destroy a pixmap:

Call xvt_pmap_destroy.

Caution: Do not use free or xvt_mem_free to destroy a pixmap.

You should destroy a pixmap (XVT_PIXMAP) when your application no longer needs it. XVT also destroys pixmaps automatically when their parent windows are destroyed. Because pixmaps do not have event handlers, there is no notification that a pixmap is being destroyed. This has two important implications for pixmaps:

You should free any pixmap application data memory during the E_DESTROY event of the parent window. (Unlike controls, XVT destroys child pixmaps *after* their parent.)

Tip: To get the application data associated with a pixmap:

Call xvt_vobj_get_data (with a valid XVT_PIXMAP as the argument).

If a copy of the pixmap is required, your application should copy the pixmap to a portable XVT image (XVT_IMAGE) using xvt_image_get_from_pmap, or to another XVT pixmap using xvt_dwin_draw_pmap (where the destination window is a valid XVT_PIXMAP).

12.5.2.3. Manipulating Pixmaps

Tip: To use a pixmap as a destination for a drawing function:

Pass the XVT_PIXMAP as the window argument of the function.

Most drawing functions can operate on pixmaps as well as windows. The following functions accept pixmaps and windows:

xvt_dwin_clear
xvt_dwin_draw_aline
xvt_dwin_draw_arc
xvt_dwin_draw_icon
xvt_dwin_draw_image
xvt_dwin_draw_line
xvt_dwin_draw_oval
xvt_dwin_draw_pic
xvt_dwin_draw_pie
xvt_dwin_draw_pmap
xvt_dwin_draw_polygon
xvt_dwin_draw_polyline
xvt_dwin_draw_rect
xvt_dwin_draw_roundrect
xvt_dwin_draw_set_pos
xvt_dwin_draw_text
xvt_dwin_get_draw_ctools
xvt_dwin_get_font_metrics
xvt_dwin_get_text_width
xvt_dwin_scroll_rect
xvt_dwin_set_back_color
xvt_dwin_set_cbrush
xvt_dwin_set_clip
xvt_dwin_set_cpen
xvt_dwin_set_draw_ctools
xvt_dwin_set_draw_mode
xvt_dwin_set_font
xvt_dwin_set_fore_color
xvt_dwin_set_std_cbrush
xvt_dwin_set_std_cpen
xvt_vobj_get_client_rect
xvt_vobj_get_data
xvt_vobj_get_outer_rect
xvt_vobj_get_parent
xvt_vobj_get_type
xvt_vobj_set_data

12.5.2.4. Drawing Pixmaps

Tip: To draw a pixmap:

Call xvt_dwin_draw_pmap.

This function draws pixmapsin windows or otherpixmaps. Because xvt_dwin_draw_pmap does not perform any color mapping, you should draw pixmaps only into windows or pixmaps with matching color palettes. Pixmaps can be scaled while being drawn.

Note: Both images and pixmaps can be drawn in print windows.

See Also: For information about scaling pixmaps, see section 12.5.4 on page 12-14.

12.5.3. Color Palettes

Color palettes let your application map the colors in an image onto the colors of the display hardware. Without color palettes, all image colors would be mapped onto the hardware's default colors, which would prevent complex images from being rendered correctly.

At runtime, a default color palette is associated with the screen window. As the application creates windows and pixmaps, their parent window's color palette is inherited; that is, the parent's palette is shared by the newly created window or pixmap. (Usually a new window's parent is the screen or task window.) Child windows inherit a palette from their parent window. This allows your application to manipulate one color palette for a hierarchy of drawable objects.

Color palettes have no effect on true-color windows and pixmaps.

12.5.3.1. XVT_PALETTE Data Type

Objects of type XVT_PALETTE represent color palettes. Because XVT_PALETTE is an opaque data type, your application can access and modify palettes only by using XVT-provided functions.

Palette Types

XVT provides several defined color palette types, which are enumerated by XVT_PALETTE_TYPE. Your application cannot modify palettes of any type except XVT_PALETTE_USER.

XVT_PALETTE_STOCK
: Contains platform-specific colors that are intended to be compatible with the platform's default color scheme. XVT uses this palette type for the initial default palette.

XVT_PALETTE_CURRENT
: Contains the color values currently used by the system's display color palette. This palette type minimizes color flashes and other undesirable effects produced when switching between different windows and applications on one display. The number of colors and their values varies depending on the system's current display palette.

XVT_PALETTE_CUBE16
: Contains 16 basic color values. It is primarily intended for systems limited to 16-color displays.

XVT_PALETTE_CUBE256
: Contains 256 evenly distributed color values, including 16 shades of grey and a uniform set of color hues and saturations.

XVT_PALETTE_USER
: When created, initially contains enough basic system color values to ensure that menus and window decorations can be rendered. There are no more than 32 of these pre-allocated colors for 256-color systems, and no more than two for 16-color systems. Your application can freely modify palettes of this type.

Implementation Note: X has a "first come, first served" approach to color. Some X applications "request" many colors from the current color table, while other applications use only a small number of colors. Also, you have no way of knowing how many applications have started prior to your XVT application, and how many colors they have collectively requested. Although this behavior interferes with your ability to create customized color palettes using XVT/XM, it is normal X behavior, and cannot be avoided. If your application is

using custom color palettes, it should check to see if it actually received the number of colors it specified in its XVT_PALETTE_USER, and provide a contingency plan for those times it does not get as many colors as it requested (e.g., post a "lack of color" message to the user and then terminate).

12.5.3.2. Creating Color Palettes

Tip: To create a new palette:

Call xvt_palet_create.

A newly created palette is not associated with any window or pixmap.

Tip: To assign a palette to a window or pixmap:

Call xvt_vobj_set_palet.

Tip: To destroy a palette when you no longer need it:

Call xvt_palet_destroy.

12.5.3.3. Adding Colors to a Palette

Tip: To add specific colors to a palette:

Call xvt_palet_add_colors or xvt_palet_add_colors_from_image.

When you call xvt_palet_add_colors, all windows associated with the palette receive an E_UPDATE event to update their contents, if necessary.

You call xvt_palet_add_colors_from_image when your application needs to display an image where the actual colors are not known (for instance, an image generated with a scanner or painting program). This function adds colors to the palette, based on the color content of the image, to give the image the best possible screen appearance.

Color Tolerance Attribute

The two xvt_palet_add_colors* functions add colors to palettes according to a color tolerance attribute. A color is added only if it differs from all colors currently in the palette by an amount greater than the palette's color tolerance. This "difference" between two colors is defined as the maximum of the differences between their corresponding red, green, and blue components.

Tip: To set a palette's color tolerance:

Call xvt_palet_set_tolerance.

Tip: To retrieve a palette's color tolerance:

Call xvt_palet_get_tolerance.

12.5.4. Transfer Operations

All pixmap and image transfer functions (xvt_dwin_draw_pmap, xvt_image_get_from_pmap, xvt_dwin_draw_image, and xvt_image_transfer) have two parameters of type RCT.
These parameters specify the bounding rectangles of the region transferred from one image or pixmap to the other. The relative sizes and locations of these rectangles affect how the region is drawn in the destination:

- If these rectangles have different locations (relative to their respective images' origins) the region is translated appropriately
- If the rectangles have different sizes, the region is scaled to fit in the destination rectangle

Note: On X platforms, scaling an image or pixmap in both dimensions (horizontal and vertical) may take significantly longer than transferring with no scaling, or with scaling in only one dimension.

12.5.5. File Operations

The XVT Portability Toolkit (PTK) provides portable functions for reading several different common image file formats. Each of these functions returns an XVT_IMAGE, which your application can manipulate and display.

Tip: To read and display images created on different platforms:

Call one of the following functions:

To Read this File Type:	Use this Function:
Win32 **BMP**	xvt_image_read_bmp
Macintosh **PICT**	xvt_image_read_macpict
X Window System **xbm**	xvt_image_read_xbm
X Window System **xpm**	xvt_image_read_xpm

Note: Only XVT/Mac can readMacintosh **PICT** files. On XVT/Mac only, you can write the **PICT** format to a file.

Tip: To read images without specifying the file type:

Call xvt_image_read.

The xvt_image_read function examines the first few bytes of the file, then calls the appropriate xvt_image_read_* function.

Tip: To save image files:

Call xvt_image_write_bmp_to_iostr.

The XVT Portability Toolkit stores images in a file format compatible with the Win32 **BMP** format.

Note: Currently, the **BMP** format is the only one that the XVT Portability Toolkit can write (with the exception of the **PICT** format for Macintosh).

Tip: To save an image in Macintosh **PICT** format (XVT/Mac only):

Call xvt_image_write_macpict_to_iostr.

13

SCROLLING

To help you understand how to implement text scrolling in an XVT window, this chapter presents three sample algorithms for handling basic scrolling tasks:

- Setting the range, thumb position, and thumb proportion parameters
- Responding to scrollbar activity
- Shifting the view in a window

The algorithms focus on scrolling text, because that is usually more difficult than scrolling a generic graphical window. Although they address several issues specific to handling text, you can easily adapt the algorithms for scrolling graphics or other data. The algorithms demonstrate several important scrolling functions:

- Showing an integral number of lines
- Preventing the user from scrolling past the end of the text
- Adjusting for the effects of E_FONT events
- Using lines instead of pixels as units
- Auto-scrolling
- Aligning patterns

In addition to the sample scrolling algorithms, this chapter discusses some key scrolling terms and concepts, and gives a brief overview of XVT-provided scrolling functions. After the algorithms, the chapter discusses some special situations for which you might want to create customized scrolling functions.

13.1. Basic Scrolling Concepts

To understand what happens during scrolling, you should understand some basic scrolling terminology and concepts:

- Scrollbar range (vertical and horizontal)
- Document origin
- Thumb position
- Thumb proportion
- Auto-scrolling

The following sections discuss each term.

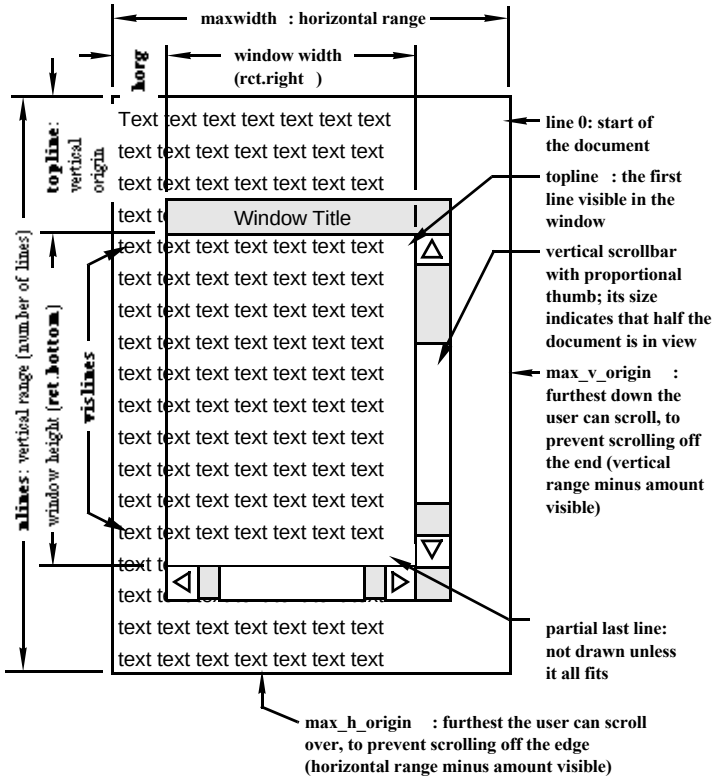


Figure 13.1. Scrollbar terms

13.1.1. Scrollbar Range

The scrollbar range is the allowable set of values (or positions) the scrollbar thumb can have, both horizontally and vertically. Operating a horizontal scrollbar generates E_HSCROLL events; operating a vertical scrollbar generates E_VSCROLL events.

See Also: For more information about the E_HSCROLL and E_VSCROLL events, see Chapter 4, *Events*.
Also see the "Events" section of the *XVT Portability Toolkit Reference*.

Horizontal Range

All horizontal scrolling is done in terms of pixels. The sample algorithms in this chapter use the length of the longest line, measured in pixels, for the horizontal range.

Vertical Range

Text applications commonly set the vertical range to be the number of lines of text in the document that is to be displayed in the window.

For documents with a large number of lines, this creates a problem because the E_HSCROLL and E_VSCROLL events return a short integer for the thumb position for SC_THUMB and SC_THUMBTRACK event types, even though the xvt_sbar_set_range and xvt_sbar_set_pos functions take integer arguments. The scrollbar range is therefore limited to SHORT_MIN through SHORT_MAX values.

XVT did not arbitrarily decide on this limit; it is implicit in some of the window systems on which XVT is implemented. For example, SHORT_MAX is the largest value the Macintosh gives for a scrollbar position, and the internal functions to set scrollbar values take a short argument.

Mapping from Lines to a Scrollbar Range

In order to accommodate documents with more than SHORT_MAX lines, an application must also provide a mapping from lines to an artificial logical scrollbar range when it actually sets the vertical scrollbar's range, thumb position, and proportion.

Tip: The range should be large enough to ensure that not too many lines map to the same scrollbar position. Once the range is larger than the number of positions in which the thumb can be drawn, limited by pixel resolution, some mapping occurs anyway. The range should also be large enough that rounding error is insignificant.

See Also: To see how a sample algorithm handles the scrollbar range, see section 13.3.1.2 on page 13-9.

13.1.2. Document Origin

The document origin is the horizontal and vertical distance from the beginning of the document to the beginning of the part of the document currently in view in the window (see Figure 13.1). The sample algorithms in this chapter internally keep track of the document's vertical origin and scrolling in terms of lines.

13.1.3. Thumb Position

The thumb position indicates which part of the document is in view relative to the entire document:

- If the beginning of the document is in view, the thumb position is at the top of the scrollbar range
- If the middle of the document is in view, the thumb is in the middle of the scrollbar range
- If the end of the document is in view, the thumb position is at the bottom of the scrollbar range

As the user scrolls through the document, the thumb position changes to reflect the current location.

13.1.4. Thumb Proportion

The thumb proportion sets the size of the scrolbar thumb relative to the range. It should indicate to the user how much of the document is currently visible in a window.

Example: If half the text lines in the document are visible in the window, the thumb should take up half of the scrollbar range. If all the text lines are visible, the thumb proportion equals the scrollbar range and the thumb fills the scrollbar. In this case, all the data is displayed, and there is no room to move the thumb.

Note: When the thumb fills the scrollbar, some window systems make the thumb or the entire scrollbar disappear.

Range versus Thumb Proportion Size

The usable part of the scrollbar range decreases by the size of the thumb proportion. This produces exactly the desired effect when implementing text scrolling—it confines the view to the bounds of the text, preventing the user from scrolling past the end of the text.

Example: Consider a document with 100 lines (numbered 0-99), displayed in a window large enough to view 20 lines at a time. If the vertical scrollbar range is 0 to 100 and the thumb proportion is 20, the effective or usable range is 80. That is, the user can scroll vertically 80 units. Initially, the first 20 lines of the document are displayed.

If the user scrolls the thumb all 80 units to the end of the scrollbar, then line 80 is the first visible line in the window. Since the window can show 20 lines, the remaining lines in the document (lines 80 to 99) are displayed, and the last line of the document is at the bottom of the window.

Implementation Note: Some systems, such as Motif, physically display a proportional thumb. On other platforms, including Macintosh, the thumb does not physically show the proportion but does behave logically as if it were proportional.

13.1.5. Auto-scrolling

Auto-scrolling lets the user scroll the contents of a window without explicitly operating the scrollbars. To facilitate auto-scrolling, you should separate the code that calculates the amount to scroll the view from the code that actually shifts the view.

Auto-scrolling happens in response to E_MOUSE_MOVE events that occur when the mouse is dragged outside of the window's client area usually while the user is selecting text. To get mouse events outside of a window's client area, you must trap the mouse.

When the mouse is trapped, the application receives E_MOUSE_MOVE events as long as a mouse button is pressed, even if the mouse is not being moved. XVT generates E_MOUSE_MOVE events under these conditions specifically to permit the user to auto-scroll.

See Also: For a sample auto-scrolling function, see section 13.1.5 on page 13-5.
For more information on trapping the mouse, see xvt_win_trap_pointer and xvt_win_release_pointer in the *XVT Portability Toolkit Reference*.

13.2. XVT-provided Scrolling Functions

XVT provides functions for setting the range, thumb position, and thumb proportion on a scrollbar:

- xvt_sbar_set_range
- xvt_sbar_set_pos
- xvt_sbar_set_proportion

XVT also provides functions for immediate processing of pending updates (xvt_dwin_update) and scrolling the contents of a window (xvt_dwin_scroll_rect).

See Also: For more information about specific functions, see the *XVT Portability Toolkit Reference*.

13.3. Sample Scrolling Algorithms

This section outlines three sample algorithms, which accomplish three basic tasks to enable scrolling:

Task 1: scroll_sync
: Maintains the window's range, thumb proportion, and thumb position parameters as the view in the window changes; the user can change the view by resizing, scrolling, or changing fonts.

Task 2: do_scroll
: Calculates how much to scroll when a scrollbar is operated.

Task 3: shift_view
: Shifts the view in the window appropriately.

Before looking at the algorithms themselves, review Figure 13.1 to understand the components of the data and the window's view that they use.

Tip: XVT recommends assigning a separate function to each of the three scrolling tasks. A single function could handle tasks 2 and 3, but separating them facilitates auto-scrolling.

Note: The algorithms assume that all lines of text are displayed using the same line height.

See Also: For a sample auto-scrolling function, see section 13.1.5 on page 13-5.

13.3.1. Task 1: Maintaining the Scrollbar Settings (scroll_sync)

A sample function called scroll_sync handles the first task, maintaining the scrollbar settings. This function is called during three of a window's events: E_CREATE, E_SIZE, and E_FONT.

Event:	Effect:
E_CREATE	Initialize all three scrollbar parameters.
E_SIZE	The new window size allows a different amount of the data to be shown in the window. Adjust the thumb proportion and position accordingly.
E_FONT	The size and therefore the amount of text displayed in the window changes. Adjust the thumb proportion and position accordingly.

The bottom line and right edge of the document should stay fixed to the end/edge of the window when it is resized larger, as long as there is text to scroll in from the top or left; see Figure 13.2.

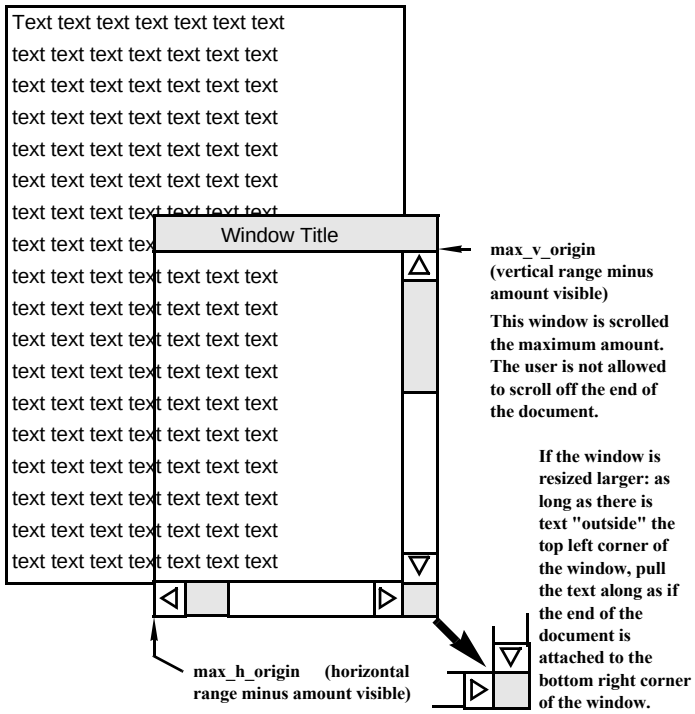


Figure 13.2. Window at the end of a document

13.3.1.1. Required Information for scroll_sync

The algorithm for scroll_sync assumes that at least the following information is available. (For each document, this information is usually carried in a data structureattached to the window displaying the document.)

nlines
: The total number of lines of text.

line_height
: The height of a line of text, in pixels.

maxlines
: The maximum width of the text, which is the length of the longest line in pixels because horizontal scrolling is done in pixel units (vertical scrolling is done in line units).

topline
: The first line visible in the window’s view, which is actually the vertical component of the origin.

horg
The horizontal origin, which in this case is in pixels.

See Also: For more information on attaching application data to a window, see xvt_vobj_set_data in the *XVT Portability Toolkit Reference*.

13.3.1.2. Vertical Range and the VRANGE Macros

scroll_sync uses a vertical range of 0 to 10,000. Using zero as a lower bound simplifies computation. The number 10,000 is large enough to ensure that not too many lines map to the same scrollbar position, and that the rounding error is insignificant.

To map to an arbitrary scrollbar range, the upper bound of the vertical range (10,000) is #defined in a constant, VRANGE, which can be changedeasily. Two macros for scalingline numbers into this range are defined as follows:

```
#define LinesToRange(x)(int)(((float)(x)/nlines) * VRANGE)

#define RangeToLines(x)(int)(((float)(x)/VRANGE) * nlines)
```

These macros apply to the arguments of any function manipulating the vertical scrollbar range, thumb position, or proportion. Essentially, they cause the scrollbar to work in its own logical coordinate system.

13.3.1.3. The scroll_sync Algorithm

```
1)    get the client rectangle for the window (in rct)
2)    calculate visible_lines as rct.bottom / line_height

/* The scrollbar ranges are set as described above - horizontal to the width of the longest line in
pixels; vertical to an arbitrary logical range. The horizontal dimension (maxwidth) must be
updated anytime something changes the document width (for instance, whenever the font
changes). */

3)    set the vertical scrollbar's range to [0, VRANGE]
4)    set the horizontal scrollbar's range to [0, maxwidth]

/* Steps 5 - 8 keep the bottom line and right edge of the document 'glued' to end/edge of the
window when it is resized larger, as long as there is text to scroll in from the top or left
(see Figure 13.2). This can occur when the data is scrolled
to the end and then the window is resized. */

5)    if (topline + visible_lines >= nlines)
6)          shift the view so the last line of text stays at the
                  bottom of the window
7)    if (horg + rct.right >= maxwidth)
8)          shift the view so the right edge of the text stays at the
                  right edge of the window

/* The scrollbar proportion is set to the amount of the data currently in view, but it must not exceed
the maximum dimensions of the data (the scrollbar range). This could happen in the case that the
window is larger than the dimension of the data. The min in steps 9 and 10 prevents this from
happening. */
```

```
9)    calculate vert_proportion as min(visible_lines, nlines)
10)   calculate horz_proportion as min(rct.right, maxwidth)
11)   set the vertical scroll proportion to
            LinesToRange(vert_proportion)
12)   set the horizontal scroll proportion to horz_proportion

/* The vertical or horizontal thumb position is set to the vertical or horizontal component of the
document origin, but must not be outside the usable range of the scrollbar. Since the thumb
proportion decreases the usable range of the scrollbar, the usable range is (vertically) nlines -
vert_proportion. */

13)   set the vertical thumb position to LinesToRange(min(topline,
            nlines - vert_proportion))
14)   set the horizontal scrollbar thumb position to min(horg,
            maxwidth - horz_proportion)
```

13.3.2. Task 2: Calculating the Amount to Scroll (do_scroll)

A sample function called do_scroll handles the second task, calculating the amount to scroll.

When the user operates a scrollbar, generating an E_HSCROLL or E_VSCROLL event, the do_scroll function decides how much to scroll the view in the window. The do_scroll function calculates the amount by which to scroll based on what scrollbar activity occurred. It then calls shift_view (described in section 13.3.3).

In do_scroll, the application defines what it means by a line and a page. Neither XVT nor the underlying window system can define a line or a page, since neither keeps track of what data is being displayed in the client area.

In a text application, you would naturally map the SC_LINE_UP and SC_LINE_DOWN events to move the view by one line of text, and the SC_PAGE_UP and SC_PAGE_DOWN events to move the view by one window's contents (the number of lines currently visible in the window). During these scroll events, XVT does not set the pos field of the scroll event structure. This field has a meaningful value only when the event is an SC_THUMB or SC_THUMBTRACK type.

Note: The do_scroll algorithm presented here does not respond to SC_THUMBTRACK events. To make it do so, add SC_THUMBTRACK cases that execute the same code as the SC_THUMB cases. If the application takes too long to update the window in response to these events, then you might not be able to provide real-time response to dragging the thumb.

13.3.2.1. Required Information for do_scroll

The algorithm for do_scroll requires the information needed by the previous algorithm, scroll_sync, plus the following additional information:

HINTERVAL
: The distance (in pixels) to scroll as a “line” horizontally.

The distance scrolled vertically by LINE_UP or LINE_DOWN is the height of a text line. The distance scrolled either horizontally or vertically by PAGE_UP or PAGE_DOWN is the current width or height of the window.

13.3.2.2. The do_scroll Algorithm

```
1)      get the client rectangle for the window (in rct)
2)      calculate visible_lines as rct.bottom / line_height

/* Next, we have the usual switches on the scrollbar activity. Each case calculates the new origin.
*/

3)      switch (type) {
4)      case HSCROLL:
5)              calculate the maximum allowable amount to scroll
                        horizontally (max_h_origin) as max(0,
                        maxwidth - rct.right)
6)              switch (what) {
7)              case SC_LINE_UP:
8)                      set new origin to max(0, horg - HINTERVAL)
9)              case SC_LINE_DOWN:

/* If the horizontal origin is not less than the maximum allowable horizontal origin, then there is
no room to scroll further. The edge of the document is already in view. However, the user could
resize the window larger, leaving white space past the edge of the text in the view. If the mouse
is used on the LINE_DOWN arrow, simply resetting the origin (without the if statement below)
would cause the text to jump to the right - the opposite of what the user expects. Instead, the if in
step 10 below prevents the origin from changing at all when it is already at the end
(See Figure 13.2). */
```

```
10)                 if (horg < max_h_origin)
11)                         set new origin to min(max_h_origin, horg +
                                HINTERVAL)
12)         case SC_PAGE_UP:
13)                 set new origin to max(0, horg - rct.right)
14)         case SC_PAGE_DOWN:
15)                 set new origin to min(max_h_origin, horg + rct.right)
16)         case SC_THUMB:
17)                 set new origin to the new thumb position in the event
                        structure
18)         }
19)         shift the view horizontally by horg - new origin
20)   case VSCROLL:
21)         calculate the maximum allowable amount to scroll
                vertically (max_v_origin) as
                max(0, nlines - visible_lines)
22)         switch (what) {
23)         case SC_LINE_UP:
24)                 set new origin to max(0, topline - 1)
25)         case SC_LINE_DOWN:
```

/* If the vertical origin is not less than the maximum allowable vertical origin, then there is no room to scroll further. The end of the document is already in view. However, the user could resize the window larger, leaving white space past the end of the text in the view. If the mouse is used on the LINE_DOWN arrow, simply resetting the origin (without the if statement below) would cause the text to jump down - the opposite of what the user expects. Instead, the if in step 26 below prevents the origin from changing at all when it is already at the end (See Figure 13.2). */

```
26)                 if (topline < max_v_origin)
27)                         set new origin to min(max_v_origin, topline + 1)
28)         case SC_PAGE_UP:
29)                 set new origin to max(0, topline - visible_lines)
30)         case SC_PAGE_DOWN:
31)                 set new origin to min(max_v_origin, topline +
                        visible_lines)
```

/* The vertical thumb position must be mapped into a line number because it is in the logical scrollbar range. */

```
32)         case SC_THUMB:
33)                 set new origin to RangeToLines(new thumb position in
                        the event structure)
34)         }
35)         shift the view vertically by topline - new origin
36)   }
```

13.3.3. Task 3: Scrolling the View Window (shift_view)

A sample function called shift_view handles the final task, scrolling the view in the window in response to scrollbar operation.

The shift_view algorithm scrolls the view in the window. It first adjusts the document origin to reflect the amount scrolled, then calls xvt_dwin_scroll_rect to scroll the view. It also sets the new thumb position and, in text scrolling, forces an update for the bottom of the client area if a partial line was left.

13.3.3.1. Required Information for shift_view

The algorithm for shift_view requires the following information:

dh and dv
: The horizontal and vertical distances to shift the view.

max_h_origin and max_v_origin
: The maximum allowable horizontal and vertical origins (see Figure 13.2).

13.3.3.2. The shift_view Algorithm

```
1)    if both dh and dv are zero then return because there is no
          scrolling to be done
2)    get the client rectangle for the window (in rct)

/* Make sure all pending updates are processed before changing the origin. If update events that
were generated before the scrolling occurred are still pending in the native system's event queue,
and have not yet been processed by the application, they must be processed while the document's
origin is the same as it was when the update event was generated. If these events aren't processed,
the scrolling code will change the document's origin and when the updates eventually are
processed the text will be drawn in the wrong place. */

3)    call xvt_dwin_update to process any pending update events

/* If there is vertical scrolling to be done, the vertical origin must be adjusted and the scrollbar's
thumb position reset. The bottom of the rectangle is then reduced to match the bottom of the last
line of text. This scrolls only the full lines of text in the view, not a partial line of white space. */

4)    if (dv != 0) {
5)          set vertical origin (topline) to topline - dv
6)          set the vertical thumb position to
                LinesToRange(min(max_v_origin, topline))
7)          save the current rct.bottom (in bottom)
8)          reset rct.bottom to an integral line boundary
9)    }

/* If there is horizontal scrolling to be done, the horizontal origin must be adjusted and that
scrollbar's thumb position reset. */

10)   if (dh != 0) {
11)         set horizontal origin (horg) to horg - dh
12)         set the horizontal scrollbar thumb position to
                min(max_h_origin, horg)
13)   }
```

```
/* Here, the algorithm actually shifts all the pixels in the view by the designated amount. */

14)   scroll the view in the window (rct) by (dh, dv*line_height)
            with xvt_dwin_scroll_rect

/* If there was a partial last line at the bottom of the view, force an update to clear it and redraw
in case a whole line now
fits. */

15)   if (dv != 0 && bottom != rct.bottom){
16)         rct.top = rct.bottom
17)         rct.bottom = bottom
18)         force an update on the rct with xvt_dwin_invalidate_rect
19)   }
```

13.3.3.3. A Sample Function for Auto-scrolling

Auto-scrolling lets the user scroll the contents of a window without explicitly operating the scrollbars. For instance, if a user drags the mouse outside the bottom of a window, the view of the data in the window shifts to bring lines below the bottom up into view.

To implement auto-scrolling, you follow these general steps:

- Translate the mouse's physical coordinates to the logical space in which the application model is working
- Calculate what direction(s) and distance the mouse lies outside the top, bottom, left, or right of the window
- Once you have this information, call the shift_view function (shown below) to move the data that is logically outside of the window into view from that direction

Tip: If you use the mouse-to-window distance to control the scrolling amount, the user can autoscroll faster or slower by moving the mouse further from or closer to the window border.

Note: You must write the shift_view function to allow both horizontal and vertical scrolling to occur simultaneously. This effect can only be achieved during auto-scrolling, not when scrolling with the scrollbars, since only one scrollbar can be operated at a time.

Sample shift_view Function for Auto-scrolling

Below is a sample shift_view function. Note that this function works for an arbitrary window, whether text or graphics.

```
void XVT_CALLCONV1 shift_view(WINDOW win, int dx, int dy,
        PNT *pntp)
{
        WINDOW_INFO *wip = get_win_info(win);
        RCT rct;
                /* Align on an 8-pixel boundary */
        dx = (dx >= 0 ? 1 : -1) * ((abs(dx) + 7) / 8) * 8;
        dy = (dy >= 0 ? 1 : -1) * ((abs(dy) + 7) / 8) * 8;
        xvt_dwin_update(win);
        if (dx != 0) {
                if (dx > 0)
                        dx = min(dx, wip->origin.h);
                else
                        dx = max(dx, wip->origin.h - wip->range.h);
                wip->origin.h -= dx;
                xvt_sbar_set_pos(win, HSCROLL, wip->origin.h);
        }
        if (dy != 0) {
                if (dy > 0)
                        dy = min(dy, wip->origin.v);
                else
                        dy = max(dy, wip->origin.v - wip->range.v);
                wip->origin.v -= dy;
                xvt_sbar_set_pos(win, VSCROLL, wip->origin.v);
        }
        xvt_vobj_get_client_rect(win, &rct);
        xvt_dwin_scroll_rect(win, &rct, dx, dy);
        if (pntp != NULL) {
                pntp->h += dx;
                pntp->v += dy;
        }
}
```

13.3.3.4. Aligning Patterns

The shift_view code in the previous section rounds the horizontal and vertical scrolling distances (dx and dy) to the nearest multiple of 8. This is done when scrolling graphical data that might contain background or fill patterns because native window systems always align patterns on certain pixel boundaries.

Eight-by-eight patterns are almost universal. In many cases the pattern drawing is done at the hardware level, which means that eight-by-eight boundaries are quite deeply rooted.

13.4. Special Scrolling Situations

The algorithms in this chapter have dealt with scrolling text in a window. You can easily modify them to scroll graphics in a window. However, in other situations you must customize your own scrolling methods:

Scrolling Columns in a Spreadsheet

When the user selects one of the arrows on the horizontal scrollbar, you might want to scroll the spreadsheet by a full column. In this case, your application must keep track of the widths of the columns. This width becomes the amount that the spreadsheet is scrolled.

Dynamic Text Windows

Your application might include a window with non-static information. For example, a word processing program might bring in only one page of text at a time. While the user scrolls within that page, the vertical scrollbar range must reflect the entire size of the document (not just the page). In this situation, your application can keep track of its data in terms of pages. With each new page, add the number of lines represented by a page to the range, and re-calculate the scrollbar's attributes

14

CURSORS AND CARETS

A cursor is the pointer or other shape that indicates the current mouse position. A caret is a blinking vertical line that indicates where the next typed character will appear. In addition to discussing cursors and carets, this chapter tells how to trap the mouse while the user is dragging it.

14.1. Cursors

The cursor indicates the current mouse position, with a pointer or other shape. Each XVT window has a current cursor that you can set to one of five standard shapes, or to a shape that's defined as a resource.

Tip: To set the cursor symbol:

Call xvt_win_set_cursor.

xvt_win_set_cursor can change the cursor (or mouse pointing symbol) *immediately* to one of the following standard cursors:

CURSOR_ARROW	Arrow (default)
CURSOR_CROSS	Crosshair (positioning)
CURSOR_HELP	Help symbol (question mark)
CURSOR_IBEAM	I-beam (character sweep)
CURSOR_PLUS	Plus sign
CURSOR_USER	User-defined symbol
CURSOR_WAIT	Wait symbol (platform-specific)

Tip: To find the current shape:

Call xvt_win_get_cursor.

14.1.1. The Waiting Cursor

When you want to indicate to the user that an operation will take a long time, you can easily set the cursor to the waiting shape (a wristwatch or hourglass).

Tip: To set the waiting cursor:

Call xvt_scr_set_busy_cursor.

As soon as the next event occurs, the cursor automatically switches back to the window's current cursor shape.

Implementation Note: Some XVT platforms don't support a waiting cursor. On these platforms, xvt_scr_set_busy_cursor has no effect.

14.1.2. Hiding the Cursor

While the user is typing, you may want to hide the mouse cursor to get it out of the way.

Tip: To hide the cursor:

Call xvt_scr_hide_cursor.

As soon as the user moves the mouse, the cursor reappears automatically.

14.2. Trapping the Mouse

Occasionally, such as when the user is dragging the mouse, you don't want the cursor shape to change, even if the mouse leaves the client rectangle of the current window. You also might not want the mouse to be used to perform any other activity, such as issuing menu commands, closing the window, or activating another application. In these cases, you can trap the mouse.

Tip: To trap the mouse:

Call xvt_win_trap_pointer.

As soon as dragging stops, you must remember to free the mouse with xvt_win_release_pointer.

Another benefit of trapping the mouse is that your application is guaranteed to get an E_MOUSE_UP event, even if it occurs while the cursor is outside of the window in which the previous E_MOUSE_DOWN event occurred.

While the mouse is trapped, E_MOUSE_MOVE events are generated continuously, even if the mouse isn't physically moved. This allows you to implement auto-scrolling. When the user moves the mouse out of the window or other designated area, scroll the data an appropriate amount in the desired direction.

See Also: For more information about E_MOUSE_UP events, see section 4.5.14 on page 4-52.
For more information about scrolling, see Chapter 13, *Scrolling*.

14.3. Carets

A caret is a blinking vertical line that indicates where the next typed character will appear. Typically, applications use a caret when a window is in "text insertion" mode.

14.3.1. Logical vs. Physical Carets

Each regular (non-dialog) window in XVT possesses a "logical caret." The xvt_dwin_set_caret_visible function turns the logical caret on and off. A window that has its logical caret turned on displays the physical caret when the window has the focus, but does not display the caret when the window loses focus.

Your application can turn on the logical caret for multiple windows simultaneously. XVT then manages the caret as a visual cue to indicate which window has focus. In addition, XVT automatically hides a window's physical caret when one of the following happens:

- The window is processing an E_UPDATE event
- You're drawing text into the window with xvt_dwin_draw_text
- You're scrolling the window's contents with xvt_dwin_scroll_rect

14.3.2. Hiding the Caret

In most cases, you won't hide a caret for a window in "text insertion" mode, because XVT takes care of that for you as appropriate. However, you'll want to hide the caret by using xvt_dwin_set_caret_visible(win, FALSE) in the following situations:

- Your window leaves text insertion mode (such as when a text selection is made).
- *Outside* an E_UPDATE, you draw graphical shapes other than text, which may overlap the caret. This can happen, for

example, if the user drags a graphical shape, and you track the dragging in real time.

Tip: To hide the caret while the user is typing:

Call xvt_scr_hide_cursor.
When the mouse moves, the cursor reappears automatically.

14.3.3. Positioning and Sizing the Caret

If you are using the caret to track the insertion point for typing (the usual case), you should position the caret so that it aligns with the baseline of the text being drawn beside the caret.

XVT automatically sets the caret size to match the height of the window's current physical font. However, XVT does not automatically move the caret when the user types—your application must do that manually.

Tip: To reposition the caret:

Call xvt_win_set_caret_pos with the new position.
(You don't have to call xvt_win_set_caret_visible.)

If you are displaying multiple physical fonts and styles, as is the case in a word processor, then XVT's calculation of the default caret height based on the current physical font will not be adequate.

Tip: To set the caret size manually:

Call xvt_win_set_caret_size.

If you set caret size manually, position the caret so that it aligns with the bottom of the physical font's descender (as opposed to the baseline), and set the caret height to be equal to the ascent+ descent+leading of the physical font displayed beside the caret.

15

FONTS AND TEXT

The XVT Portability Toolkit features an encapsulated font model. Under this font model, an opaque object of type XVT_FNTID identifies a logical font. XVT defines a logical font as a description of a desired physical font—a particular implementation of a font installed on a window system.

When your application needs to draw text, for instance in a window, the Toolkit's font mapping controller maps (i.e., matches) the specified logical font to the closest available physical font. The physical font is then used for rendering the text.

XVT's encapsulated font model includes these key elements:

- Native physical fonts are supported
- You can create logical fonts explicitly or define them as resources
- API functions let you modify and query attributes of logical fonts (i.e., create, copy, get/set)
- You can write your own font mappers for applications, or include font mapping in resource (XRC) specifications
- You can retrieve mapped attributes once logical fonts are mapped to physical fonts
- Application users can select physical fonts through Font Selection dialogs on all platforms
- You can write your own Font Selection dialog or use the XVT Font Selection dialog
- Your applications can read or write logical fonts to and from files, using serializing/deserializing functions

Figure 15.1 shows typical steps you would take when using a logical font in an application.

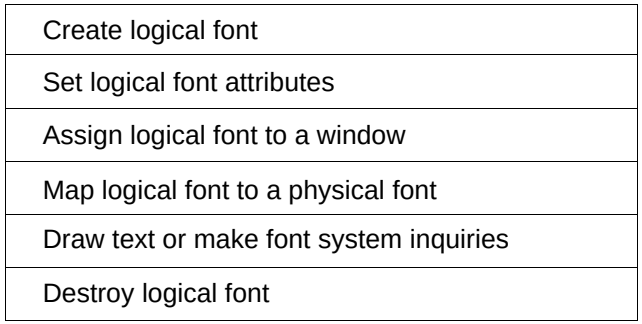


Figure 15.1. Using a logical font

The rest of this introductory section introduces you to some fundamental definitions and concepts of XVT's encapsulated font model. The sections that follow discuss the concepts in greater detail.

See Also: For sample usage of the XVT encapsulated font model, see the Font Mapper (**samples/design/fontmap**) example in the PTK example set.

15.1. Font Terminology

To help you understand how XVT's encapsulated font model works, this chapter uses the following XVT-defined terms.

Font
: A set of graphic shapes with a unified design, used to represent characters on an output device. The design itself is called a typeface. A set of typefaces designed to work together is called a typeface family, or family. For example, the Times family contains several typefaces: Times Roman, Times Bold, and Times Italic.

Typeface
: The underlying design of a font (without consideration of the font size or weight).

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.

Font mapping
The process of matching a logical font to the physical font that most closely resembles it. Once this match has been made, the logical font is "mapped."

Font mapper
A function or method that performs font mapping. Some font mappers are included in the XVT Portability Toolkit, while others are supplied by applications.

XVT font mapping controller
The XVT function, invoked by xvt_font_map, which directs a sequential series of multi-level font mappers that perform the mapping.

Font model
An approach to defining and manipulating logical and physical fonts. In an encapsulated font model, such as the one the XVT Portability Toolkit currently uses, the logical font's internals are opaque. In an exposed font model, they are open or visible.

15.2. Basic Font Concepts

This section briefly describes the following basic concepts of the encapsulated font model: logical font attributes, logical font functions, font mappers, Font Selection dialogs, and Font/Style menus. All are explained in greater detail in later sections of this chapter.

15.2.1. Logical Font Attributes

Logical font attributes describe a desired physical font. All the attributes except the native descriptor are portable:

- Application data
- Family
- Size
- Style mask
- Window
- Metrics
- Is mapped

- Is scalable
- Is valid
- Is printable
- Has valid native descriptor
- Native descriptor (non-portable)

See Also: For more information about logical font attributes, see section 15.3.1 on page 15-7.

15.2.2. Logical Font Functions

XVT provides many functions that manipulate logical fonts. Basically, you can perform the following operations on a logical font:

- **Creating** (taking the default XVT attributes)
- **Setting or inquiring attributes** (setting or getting attributes or mapped attributes)
- **Assigning** (assigning a logical font to a window for drawing)
- **Copying** (including copying of selected attributes set)
- **Mapping** (mapping the logical font attributes into a native font descriptor)
- **Unmapping** (canceling and freeing any physical font mapping contained within a logical font)
- **Serializing and deserializing** (reading and writing logical fonts to and from files)
- **Destroying** (unmapping and freeing the logical font)

See Also: For information about creating, assigning, destroying, and copying logical fonts, and setting or inquiring their attributes, see section 15.4 on page 15-10.
For information about mapping or unmapping logical fonts, or setting or inquiring mapped attributes, see section 15.5.5 on page 15-22.
For information about serializing/deserializing logical fonts, see section 15.8.3 on page 15-37.

15.2.3. Font Mappers

A font mapper matches a logical font with an equivalent physical font. XVT's encapsulated font model supports a multi-level font mapping strategy—you can write your own mapper, provide mappings for the XRC font mapper, or simply use the default XVT mapper.

An XVT-supplied font mapping controller determines which method is used. If an application-supplied font mapper or XRC mappings are not present, or are present but don't successfully map the logical font, the default XVT mapper is *guaranteed* to produce a valid mapping, even if it must use a system font to do so.

Figure 15.2 shows the multi-level mapping strategy used by the font mapping controller:

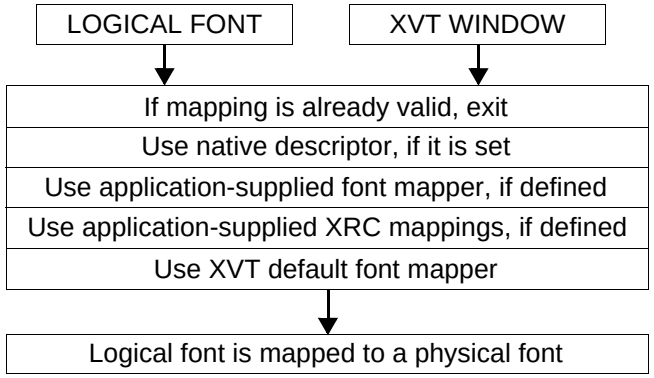


Figure 15.2. How the font mapping controller works

See Also: For more information about multi-level mapping, see section 15.5.2 on page 15-20.

15.2.4. Font Selection Dialogs

XVT's encapsulated font model lets application users interactively set logical font attributes in a dialog. XVT provides a Font Selection dialog that shows all the physical fonts on a given system (the dialog conforms to native look-and-feel). Or, you can design customized Font Selection dialogs for your applications, using either XVT font mapper inquiry functions or native font inquiry functions.

See Also: For more information about Font Selection dialogs, see section 15.6 on page 15-30.

15.2.5. Font/Style Menus

On some systems, users can also interactively set logical font attributes by means of a Font/Style menu. Applications don't supply their own Font/Style menus, because these are inherently non-portable. Instead, XVT provides appropriate standard menus. Font/Style menus are available only on the XVT/XMand XVT/Mac platforms.

Note: On platforms that don't have Font/Style menus, users can select physical fonts from the Font Selection dialog. On XVT/Win32, XVT provides a Font Selection dialog instead of a menu because it better conforms to native look-and-feel.

See Also: For more information about Font/Style menus, see section 15.7 on page 15-33.

15.3. Logical Fonts

A logical font is a description of a desired physical font. The description is composed of attributes such as the typeface family name, size, and style. The attributes describe the physical font that the application eventually wants to be mapped onto the logical font. All but one of these attributes (native descriptor) are portable.

15.3.1. Logical Font Attributes

The tables in this section show logical font attributes: portable and non-portable. The third column in each table indicates whether you can set or get each attribute, and also whether you can get the mapped value of the attribute (i.e., the attribute as realized in the physical font that the logical font was mapped to).

15.3.1.1. Portable Attributes

The following table shows the portable attributes that you can use to describe a logical font:

Attribute:	Data Type:	Access:
application_data	void*	Set/Get
family	char*	Set/Get, get mapped
size	long	Set/Get, get mapped
style_mask	XVT_FONT_STYLE_MASK	Set/Get, get mapped
window	WINDOW	Get
metrics	int	Get

application_data
: Application-specific information. This field is primarily useful for communicating information to an application-provided font mapper. XVT ignores this data. When you copy a logical font with xvt_font_copy, only a pointer to the application data is copied, not the data itself.

family
: Specifies the logical font family as a string.

size
: Logical font size in points.

style_mask
: A bit mask of flags controlling individual styles such as bold

or italics. A set of application-specific style flags has been reserved for use with an application-supplied font mapper.

window
: The window associated with amapped logical font. XVT's font mapping controller assigns this attribute when an application requests that a logical font be mapped. For all non-mapped logical fonts, it is NULL_WIN.

metrics
: The font's leading, ascent, and descent. This information is available only for mapped logical fonts.

See Also: For more information about setting and getting the portable attributes for a logical font, see section 15.4.4 on page 15-12.

15.3.1.2. Non-portable Native Descriptor Attribute

The following native descriptor attribute is filled in when a logical font is mapped, either by the application (with an application-supplied font mapper) or by the XVT Portability Toolkit (with the XRC mapper or the XVT default font mapper):

Attribute:	Data Type:	Access:
native_descriptor	char*	Set/Get

native_descriptor
: A string that describes a physical font. The contents of this string are platform-specific, and the string can contain multibyte characters. When this attribute is set and the physical font it describes actually exists, the font mapping controller uses this attribute to map the logical font.

See Also: For more information about native descriptors, see section 15.4.4.1 on page 15-15.
For details about the platform-specific contents of this string, see the "Native Font Descriptors" section in the *XVT Platform-Specific Books*.

15.3.1.3. Mapped Logical Font Inquiry Attributes

The attributes in the following table represent inquiries you can make about mapped logical fonts:

Attribute:	Data Type:	Access:
is_mapped	BOOLEAN	Get
is_print	BOOLEAN	Get mapped
is_scalable	BOOLEAN	Get mapped

is_mapped
: Indicates that the logical font has been mapped into a physical font. Once a logical font has been mapped into a physical font, the mapped attribute inquiries return values from the physical font.

is_print
: Indicates that the fonthas been mapped into a print window and that the mapped logical font is a printer font.

is_scalable
: Indicates that the mapped logical font can be scaled to arbitrary size.

Note: Any attempt to access “mapped” attributes on an unmapped logical font results in an error.

15.3.1.4. Default Logical Font Attribute Values

The default logical font attributes are:

Attribute:	Default Value:
family	System
style_mask	XVT_FS_NONE
size	12
application_data	NULL
window	NULL_WIN
native_descriptor	NULL

15.3.2. XVT_FNTID

A logical font is identified by an object of type XVT_FNTID, which is opaque to applications. Because the XVT_FNTID is opaque, you cannot directly access its internals. Instead, XVT provides access functions that you can use to get and set its attributes.

See Also: For more information about XVT_FNTID, see the *XVT Portability Toolkit Reference*.

15.4. Working with Logical Fonts

This section describes some of the actions you can perform on logical fonts. The next section, section 15.5, describes font mapping functions.

See Also: For detailed information about each function mentioned in the following sections, see their individual descriptions in the *XVT Portability Toolkit Reference*.

15.4.1. Creating and Destroying Logical Fonts

To use a new logical font in an application, your application must allocate it in one of several ways:

- Create the logical font by calling xvt_font_create
- Define the logical font in a resource file, then get it with xvt_res_get_font
- Create the logical font by calling the inquiry functions xvt_dwin_get_font, xvt_menu_get_font_sel, or xvt_ctl_get_font

A logical font allocated by these functions has certain attributes with either default or explicitly set values. When an application program creates a logical font, that application "owns" the logical font and must destroy it later when it is no longer needed (see section 15.4.3). Allocating a logical font with a call to any of the functions listed above requires a corresponding call to xvt_font_destroy.

Tip: To create a logical font:

Call xvt_font_create.

Calling xvt_font_create allocates a logical font with default values (see list of default values in section 15.3.1.4 on page 15-9), and returns the XVT_FNTID to the calling function.

Tip: To change logical font values from their defaults:

Use the font attribute setting functions (for more details, see section 15.4.4 on page 15-12).

Tip: To free the space used by an application-owned logical font:

Call xvt_font_destroy.

15.4.2. Using Logical Fonts from Resource Files

Instead of creating logical fonts with xvt_font_create, you can define them in resource files. For example, if your application knows in advance that it needs several logical fonts with known attributes, you could define them all as resources.

Tip: To use logical fonts from a resource file:

1. Define the logical fonts in an XRC file, using font or font_map statements.
2. Call xvt_res_get_font to allocate the logical fonts.

See Also: For more information about font and font_map statements, see section 15.5.7 on page 15-25.
For more information about creating logical fonts, see section 15.4.1 on page 15-10.

15.4.3. Logical Font Ownership

An application owns any logical fonts it creates. This applies to logical fonts created by these functions:xvt_font_create, xvt_res_get_font, xvt_dwin_get_font, xvt_menu_get_font_sel, and xvt_tx_get_font.

The XVT Portability Toolkit manages the creation and destruction of its own internal logical fonts. Your application must concern itself only with the creation and destruction of the logical fonts that it owns.

Logical font ownership is important because the logical font's owner must remember to destroy the logical font when it is no longer needed. Otherwise, the logical font unnecessarily consumes resources.

XVT EVENTs and WIN_DEFs can contain logical fonts. Your application should not attempt to destroy logical fonts contained in WIN_DEFs and EVENTs when the Toolkit creates those structures. Of course, when your application creates these structures, it is responsible for freeing them and their contained logical fonts.

When an XVT_FNTID appears in a structure, such as E_FONT, the owner of that structure also owns that internal logical font. Font IDs in EVENTs have exactly the same lifespan as the EVENT structure itself.

Caution: You might be tempted to save an XVT_FNTID from a Toolkit-generated EVENT and use it after the EVENT has disappeared. This does *not* work. Doing so inevitably leaves the application with a font_id that refers to a deleted internal font. To properly copy the font_id from an EVENT, you should call xvt_font_copy.

15.4.4. Setting and Getting Logical Font Attributes

If your application needs to set or get logical font attributes, it can call the logical font attribute access functions. Setting any attribute to a new and different value invalidates any previous font mapping.

For example, if the application sets the family, style_mask, size, application_data, or native_descriptor for a mapped logical font, and if the new value of the attribute differs from the old value, the XVT Portability Toolkit automatically calls xvt_font_unmap to unmap the logical font.

Example: This example shows how you can construct a 14-point, bold, italic Helvetica logical font:

```
XVT_FNTID font_id = xvt_font_create();
xvt_font_set_family(font_id, XVT_FFN_HELVETICA);
xvt_font_set_style(font_id, XVT_FS_BOLD | XVT_FS_ITALIC);
xvt_font_set_size(font_id, 14);
```

Tip: To set logical font attributes:

Call xvt_font_set_app_data, xvt_font_set_family, xvt_font_set_native_desc, xvt_font_set_size, or xvt_font_set_style.

When you set multiple logical font attributes, you must set the native_descriptor last, because any setting of family, size, style_mask, or application_data could destroy the native font descriptor and thus unmap the logical font.

See Also: For more information about setting the native font descriptor, see section 15.4.4.1 on page 15-15.

Tip: To set logical font attributes for a font associated with a window:

Call xvt_dwin_set_font_app_data,
xvt_dwin_set_font_family,
xvt_dwin_set_font_native_desc,

xvt_dwin_set_font_size, or
xvt_dwin_set_font_style.

Tip: To get portable logical font attributes:

Call xvt_font_get_app_data,
xvt_font_get_family,
xvt_font_get_size, or
xvt_font_get_style.

Tip: To get portable logical font attributes for a font associated with a window:

Call xvt_dwin_get_font,
xvt_dwin_get_font_app_data,
xvt_dwin_get_font_family,
xvt_dwin_get_font_size, or
xvt_dwin_get_font_style.

Tip: To get mapped logical font attributes:

Call xvt_font_get_family_mapped,
xvt_font_get_metrics,
xvt_font_get_native_desc,
xvt_font_get_size_mapped, or xvt_font_get_style_mapped.

Tip: To get mapped logical font attributes for a font associated with a window:

Call xvt_dwin_get_font_family_mapped, xvt_dwin_get_font_metrics,
xvt_dwin_get_font_native_desc, xvt_dwin_get_font_size_mapped, or
xvt_dwin_get_font_style_mapped.

Example: The following code creates a logical font and sets the values of its portable attributes. It then maps the font, assigns it to a window, and draws text in the window.

```
/* create and initialize font */
XVT_FNTID font_id = xvt_font_create();
xvt_font_set_family(font_id, "new century schoolbook");
xvt_font_set_style(font_id, XVT_FS_BOLD | XVT_FS_ITALIC);
xvt_font_set_size(font_id, 14);

/* Map the font */
xvt_font_map(font_id, window);

xvt_dwin_set_font(win, font_id);
xvt_dwin_draw_text(win, x, y, “hello”, -1);
...
xvt_font_destroy(font_id);
```

See Also: For details about each attribute discussed in this section, see section 15.3.1 on page 15-7.
For more information about how a font is mapped, see section 15.5 on page 15-19.
For details about functions that retrieve currently mapped font attributes, see section 15.5.5 on page 15-22.

15.4.4.1. Setting Native Font Descriptors

When a logical font is mapped, its portable attributes are matched to the closest available physical font. If you want to exactly specify a particular physical font, your application must use a native font descriptor. This is usually done inside an application-supplied font mapper.

You can use a native font descriptor in two ways:

- As a parameter to xvt_font_set_native_desc, when writing your own font mapper
- As part of an XRC font or font_map statement, for the XRC font mapper (see section 15.5.7)

Native Font Descriptor String

A native descriptor is a string of data fields that textually describe a physical font that resides on your system. The fields in the descriptor represent the internal, or native, font selection attributes present on each platform (for example, lfWeight or lfCharSet on XVT/Win32).

The native font descriptor string contains the following data:

- The native window system and version of the XVT encapsulated font model (the current version is "01").
- A platform-specific string that the XVT Portability Toolkit can decode and use to locate and identify a physical font. The platform-specific string is composed of separate fields that describe the attributes of a physical font. Each field in the string is separated by a slash, "/".

The native font descriptor, then, has this format:

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

System and Version

The system and version identifier in the native font descriptor must be one of the following:

Identifier:	Platform:
X11<vers>	X Window System
MAC<vers>	Macintosh
NT_<vers>	MS-Windows (32-bit versions)

Note: For this release, the version number of the font model system is 01.

Platform-specific Parameters

In the native font descriptor, you can provide parameters for the following platforms: XVT/Win32, XVT/Mac, and XVT/XM. You can also provide parameters for PostScript printing on XVT/XM.

See Also: For details about native font descriptor parameters for a particular platform, see its *XVT Platform-Specific Book*.

Setting and Verifying Native Font Descriptors

Tip: To set the native font descriptor for a physical font:

Select a physical font from a Font Selection dialog.
-OR-
Call xvt_font_set_native_desc or xvt_dwin_set_font_native_desc.

If the font was previously mapped, these functions unmap it, leaving the logical font in an "unmapped" state.

Note: You should call xvt_font_set_native_desc only from inside an application-supplied font mapper (registered with XVT by means of the ATTR_FONT_MAPPER attribute). If you call it outside such a mapper, the logical font's window attribute will probably be set incorrectly, which produces an invalid mapping.

Tip: When you set multiple logical font attributes, you must set the native_descriptor last, because any setting of app_data, family, size, or style_mask to a new value destroys the native font descriptor.

Tip: To determine if a native description is valid:

Call xvt_font_has_valid_native_desc.

A native description is valid if the font mapper recognizes it as an accurate description of a physical font.

See Also: For more information about font attributes, see section 15.3.1 on page 15-7.

15.4.5. Assigning Logical Fonts to Controls and Windows

15.4.5.1. Controls

Tip: To assign a logical font to a control:

Call xvt_ctl_set_font.

The xvt_ctl_set_font function copies the contents of the application-owned logical font into the drawing font of the control. The window attribute of the logical font is not copied.

Tip: To get logical font information for a control:

Call xvt_ctl_get_font.

Note: The xvt_ctl_get_font function creates a new logical font, which it returns to the application, just like the function xvt_dwin_get_font.

15.4.5.2. Windows

Tip: To assign a logical font to a window:

Call xvt_dwin_set_font.

The xvt_dwin_set_font function copies the contents of the application-owned logical font into the drawing font of the window. The window attribute of the logical font is not copied.

Tip: To get logical font information for a window:

Call xvt_dwin_get_font.

Note: The xvt_dwin_get_font function creates a new logical font, which it returns to the application that contains a copy of the window's drawing font. The application owns the returned logical font, and it is responsible for destroying the logical font when it is no longer needed.

Example: The sample code below creates a logical font and uses it to set the drawing font for a window. The following code creates a logical font, sets family, style, and size attributes for it, assigns it to a window, and destroys it. Text drawn into the window is then rendered using the specified logical font attributes:

```
XVT_FNTID font_id = xvt_font_create();
xvt_font_set_family(font_id, "new century schoolbook");
xvt_font_set_style(font_id, XVT_FS_BOLD | XVT_FS_ITALIC);
xvt_font_set_size(font_id, 14);
xvt_dwin_set_font(win, font_id);
xvt_font_destroy(font_id);
xvt_dwin_draw_text(win, "hello");
```

Calling xvt_dwin_draw_text ensures that the logical font is mapped before the text is drawn. Ifthe logical font has already beenmapped, it is not remapped.

The following code has the identical consequences as the previous example. However, this code uses the xvt_dwin_set_font_* functions:

```
xvt_dwin_set_font_family(win, "new century schoolbook");
xvt_dwin_set_font_style(win, XVT_FS_BOLD | XVT_FS_ITALIC);
xvt_dwin_set_font_size(win, 14);
xvt_dwin_set_draw_text(win, “hello”);
```

Note: Changing the attributes of font_id *after* using it in the call to xvt_dwin_set_font or xvt_ctl_set_font has no effect on the window’s font.

See Also: For more information about an application’s ownership of logical fonts that it creates, see section 15.4.3 on page 15-11.
For more information about font mapping, see section 15.5 on page 15-19.
For more information about how to sets fonts (and colors) in controls, see section 8.4 on page 8-56.

15.4.6. Copying Logical Fonts

You can copy logical font values from a source font to a destination font. You need not copy the entire logical font. A font attribute mask in the copying function tells which portions of the logical font to copy.

Tip: To copy a logical font:

Call xvt_font_copy.

The xvt_font_copy function does not create orallocate any new logical fonts. Both the source and destination logical fonts must have been previously created by the application.

15.4.7. Verifying a Font ID

Tip: To determine if a font ID has been defined:

Call xvt_font_is_valid.

Tip: To identify a NULL font ID:

Use the following predefined macro:

```
#define NULL_FNTID ((XVT_FNTID)NULL)
```

For example:

```
if(font_id == NULL_FNTID)
    xvt_dm_post_error("NULL font");
```

15.5. Font Mapping and the Font Mapping Controller

Before the XVT Toolkit can use any logical font—for drawing text or answering inquiries about its mapped attributes—it must be mapped into some available physical font. An XVT function called the font mapping controller, which is invoked by xvt_font_map, manages this conversion. The font mapping controller directs a sequence of font mappers that perform the mapping.

The font mapping controller uses underlying font mappers to map a logical font to a physical font. The font mappers do this by filling in the native_descriptor attribute for the XVT_FNTID and marking the logical font as "mapped." If the logical font already has a valid native descriptor, the font mapping controller uses it for mapping.

15.5.1. Font Mapping in an Encapsulated Font Model

The encapsulated font model can use several different mappers. If more than one mapper is available, the font mapping controller tries them in a predetermined order, until mapping succeeds.

The mapping always succeeds, although the resulting match may not be exact. If the application never explicitly maps a logical font (by calling xvt_font_map or xvt_font_map_using_default), the Toolkit implicitly performs the mapping when the physical font is actually needed to draw text or to answer a physical-font-related inquiry.

Tip: To list all logical font families supported by the font mapping controller:

Call xvt_fmap_get_families.

This function lists all logical font families supported by the font mapping controller, exclusive of any application-supplied font mappers.

Tip: To get attributes for supported logical fonts:

Call xvt_fmap_get_family_sizes, xvt_fmap_get_familysize_styles, xvt_fmap_get_familystyle_sizes, or xvt_fmap_get_family_styles.

These functions get attributes for logical fonts supported by the font mapping controller, exclusive of any application-supplied font mappers.

See Also: For more information on the xvt_fmap_* functions, see the *XVT Portability Toolkit Reference*.

15.5.2. The Multi-Level Mapping Process

When XVT code automatically calls for mapping, or when you manually call xvt_font_map, the font mapping controller proceeds to map the logical font. It does this by trying the following multi-level methods in the order shown:

- If the logical font has already been mapped, use the mapped physical font
- If the logical font is not already mapped, but the native font descriptor has been set (either by calling xvt_font_set_native_desc or by selecting attributes from a Font Selection dialog), use this descriptor to map the logical font
- If the logical font is not mapped and the ATTR_FONT_MAPPER attribute has been set, use the application-supplied font mapper
- If the logical font is still not mapped and application-supplied XRC mappings have been provided, use them
- If the logical font is still not mapped, use the default XVT mapper, xvt_font_map_using_default (guaranteed to succeed)

As soon as the logical font is successfully mapped, the controller stops. In other words, once a successful mapping occurs, the rest of the methods will not be tried.

The mapping is performed in the context of a window, even though on some platforms a screen mapping may be the same for all windows. Mappers must support both screen and print windows.

15.5.3. Types of Mappers

The multi-level mapping approach embodied in the font mapping controller can use four types of mappers:

Native description mapper
If the font mapping controller detects that the logical font already has a valid native descriptor, it invokes the native description mapper to locate the physical font that matches that native descriptor. If no native descriptor exists, the font mapping controller tries the application-supplied font mapper.

Application-supplied font mapper
When a specific application requires customized mapping strategies, you can write your own mapper. To use it, you'll register your mapper with the ATTR_FONT_MAPPER attribute. If your application-supplied mapping does not map the logical font, XVT's font mapping controller calls the XRC font mapper as a backup. An application-supplied font mapper might not map a particular logical font for several reasons. For example:

- The mapper might be designed only to map logical fonts of certain families or styles
- The physical font that the application-supplied font mapper tries to use might not exist on the window system

XRC font mapper
The XRC font mapper tries to map the logical font using optional application-supplied XRC font and font_map statements. If the application doesn't provide any font or font_map statements, or if those provided don't describe the specific logical font being mapped, the XRC mapper returns without mapping (i.e., the logical font is left in an "unmapped" state). If the XRC mapper does not map the logical font, the mapping controller next tries the XVT default mapper.

XVT default font mapper
The XVT default font mapper is the "last chance" mapper. As part of the XVT Portability Toolkit, it is guaranteed to succeed in mapping the logical font to a physical font. The XVT default mapper attempts to find a physical font that closely matches the logical font. If no such match can be found, this mapper maps the logical font onto a default system physical font.

See Also: For more information about the last three mappers, see sections 15.5.6, 15.5.7, and 15.5.8. For a sample application-supplied font mapper, see the Font Mapper (**samples/design/fontmap**)

sample in the PTK example set. You can use this Font Mapper sample to generate sample XRC font mappings.

15.5.4. When Mapping Occurs

The font mapping controller is called in two situations, depending on whether the logical font needs to be mapped automatically or whether mapping has been manually requested:

Automatic (implicit) mapping
Whenever the XVT Portability Toolkit needs to draw text or acquire text metrics, it calls the mapping controller (xvt_font_map) to ensure that the font mapping is current. This whole process occurs automatically whenever you call xvt_dwin_draw_text, xvt_dwin_get_font_metrics, xvt_font_get_metrics, or xvt_dwin_get_text_width.

Manual (explicit) mapping
When your application needs a runtime mapping for a logical font, you can explicitly call either xvt_font_map (the font mapping controller) or xvt_font_map_using_default. You call one of these functions if you intend to draw with this logical font in several windows (with xvt_dwin_set_font) or if you want to call any of the mapped logical font attribute inquiry functions (xvt_font_get_*_mapped).

In either case, a logical font is ultimately "mapped" when an application calls either xvt_font_map (automatically or manually) or xvt_font_map_using_default.

15.5.5. Mapping and Unmapping Logical Fonts

As explained in the previous sections, mapping occurs automatically for previously unmapped logical fonts whenever XVT needs to draw text or acquire text width or text metrics. But you can also map logical fonts manually, for example in an application-supplied font mapper.

You can also call different functions to inquire about mappings and mapped attributes, or to unmap a logical font. Unmapping a logical font does not affect any of the portable attributes or the native descriptor, but it does release any physical font resources.

Tip: To manually map a logical font:

Call xvt_font_map.
-OR-

Call xvt_font_map_using_default
(to invoke the XVT default font mapper).

Tip: To unmap a logical font:

Call xvt_font_unmap.

Tip: To determine if a logical font is mapped:

Call xvt_font_is_mapped.

Tip: To determine if a logical font is mapped to a print font:

Call xvt_font_is_print.

Tip: To determine if a mapped logical font can be scaled:

Call xvt_font_is_scalable.

Tip: To get mapped logical font attributes:

Call xvt_font_get_family_mapped, xvt_font_get_native_desc, xvt_font_get_style_mapped, or xvt_font_get_size_mapped.

15.5.6. Application-Supplied Font Mappers

If you wish, you can create an application-supplied font mapper for your application. You then register the application-supplied font mapper by using xvt_vobj_set_attr to set the XVT_FONT_MAPPER attribute. You can retrieve current application font mapper function pointers with xvt_vobj_get_attr.

Tip: To use an application-supplied font mapper:

1. Create your application-supplied mapper.
2. Set the ATTR_FONT_MAPPER attribute. For example:

```
xvt_vobj_set_attr(NULL_WIN, ATTR_FONT_MAPPER, (long)
      my_font_mapper);
```

where my_font_mapper is an application function.

The window argument for the set or get function is not used for this attribute. The argument to the xvt_vobj_set_attr function is the font mapper pointer. The type definition for the font mapper is as follows:

```
typedef void (*XVT_FONT_MAPPER)(XVT_FNTID font_id)
```

The application-supplied font mapper is called by the font mapping controller whenever mapping is needed; or, in other words, whenever xvt_font_map is invoked by the application or when font mapping occurs automatically.

The application-supplied font mapper can map a logical font in two ways:

Portable method
With the portable method, the application-supplied font mapper relies on XVT mappers (default and XRC-based) to perform the mapping, but under the application-supplied mapper's control. The application mapper calls the XVT mappers (usually xvt_font_map_using_default), but if not satisfied with the result it changes the *portable* attributes and tries mapping again. When it is finally satisfied with mapping, it exits.

Native method
With the native method, the application-supplied font mapper uses native font inquiries to build the native font descriptor.

With either method, an application-supplied font mapper must pay attention to the logical font's window type, because a different mapping may be required for print and screen windows. The window is embedded in the logical font, and can be gotten with xvt_font_get_win.

Note: After the application-supplied font mapper modifies the logical font attributes, it *must* call either xvt_font_map or xvt_font_map_using_default to complete the mapping.

Example: Here is an example of a simple application-supplied font mapper for XVT/XM, which uses the native method described above:

```
BOOLEAN XVT_CALLCONV1 my_font_mapper(XVT_FNTID font_id)
{
        char native_desc[256]; /*XLFD font name */
        char* family = xvt_font_get_family(font_id);
        XVT_FONT_STYLE_MASK style = xvt_font_get_style(font_id);
        long size = xvt_font_get_size(font_id);
        BOOLEAN found = FALSE;
        char size_string[10];
        WINDOW win = xvt_font_get_win(font_id);
        xvt_str_copy (native_desc, "X1101/");
        if (xvt_str_compare_ignoring_case(family,
                "new century schoolbook") == 0) {
                found = TRUE;
                xvt_str_concat (native_desc,
                                        "adobe/new century schoolbook/");
                if (style & XVT_FS_BOLD)
                        xvt_str_concat (native_desc, "bold/");
                else
                        xvt_str_concat (native_desc, "medium/");
                if (style & XVT_FS_ITALIC)
                        xvt_str_concat (native_desc, "i/");
                else
                        xvt_str_concat (native_desc, "r/");
                xvt_str_concat (native_desc, "normal//*/");
                xvt_str_sprintf (size_string, "%d", size * 10);
                xvt_str_concat (native_desc, size_string);
                xvt_str_concat (native_desc, "/*/*/*/*/*/*");
        }
        xvt_mem_free(family);
        if (found) {
                xvt_font_set_native_desc (font_id, native_desc);
                xvt_font_map_using_default (font_id);
                return TRUE;
        else
                return FALSE;
}
```

See Also: For a sample application-supplied font mapper that uses the portable method, see the Font Mapper (**samples/design/fontmap**) sample in the PTK sample set.

15.5.7. XRC Font Mapper

You can place font definition and font mapping statements in an application's XRC file for two purposes:

- To provide customized extensions to the XVT default font mapper. XRC font or font_map statements, which are used by the built-in XRC font mapper, extend the default font mapping but do not replace it.
- To define logical fonts as resources, as an alternative to creating them with xvt_font_create. You can then allocate the logical fonts by calling xvt_res_get_font.

To provide mappings for the XRC font mapper, you can define a series of logical fonts and corresponding mappings in the XRC file. When the application calls a function that forces an explicit (xvt_font_map) or implicit (xvt_dwin_draw_text) font mapping, the mapper attempts to match each definition in the series, in ascending numeric order (by font or font_map resource ID). If it finds a match, it then maps the logical font to the corresponding specification string.

Note: Although the native descriptor portion of XRC font statements is platform-specific, the built-in XRC font mapper that processes them operates generically across all platforms.

See Also: For more information about using logical fonts from resource files, see section 15.4.2 on page 15-11.
For a sample customized font selection dialog that uses the portable method mentioned above, see the Font Mapper (**samples/design/fontmap**) sample in the PTK example set.

15.5.7.1. XRC Font Resource Types

To provide font mapping extensions or define logical fonts in the XRC file, you'll specify two XRC resource types: font and font_map.

Since the native descriptors in the XRC font statement are strings, you can easily write and read them from files. You can also manipulate them by using the portable API functions xvt_font_get_native_desc and xvt_font_set_native_desc.

font Resource Type

font *id family size* [*style*] [**map** *native_desc*]

In the font resource, *family*, *size*, and [*style*] contain the XVT portable attributes (family, size, style). For the *size* and [*style*] fields, you can use the wildcard "any". If the XRC font mapper encounters a wildcard, it allows any size or style, respectively, to be mapped to the specified native descriptor.

The [**map** *native_desc*] portion of the font statement is *optional*. See Tip, below.

Example: Here is an example of how you would define "MYFONT101" in your XRC file:

```
#define MYFONT101 1
font MYFONT101 "helvetica" 12 bold italic
```

Tip: You can avoid using the font_map statement altogether by appending the native descriptor to the end of the font statement, preceded by the keyword map, like this:

```
#define MYFONT101 1
font MYFONT101 "helvetica" 12 bold italic map "X1101\
adobe/helvetica/..."
```

font_map Resource Type

font_map *id native_desc*

The *native_desc* portion of the font_map statement is a native descriptor string. It is the same as the optional *native_desc* portion of the XRC font statement. It has the format required by the function xvt_font_set_native_desc, and returned by the function xvt_font_get_native_desc.

You can use wildcards in *native_desc*. If a native descriptor contains wildcards, the corresponding portable attributes of the specific logical font being mapped are used as the value of the native attribute.

Example: Here is an example of how you would define a native mapping for "MYFONT101" on XVT/XM:

```
#define MYFONT101 1
font_map MYFONT101 "X1101/adobe/helvetica/bold/i/\
*//*/120/*/*/*/*/*/*"
```

See Also: For information about the contents of a native descriptor string, see section 15.4.4.1 on page 15-15 plus a section dedicated to this subject in each *XVT Platform-Specific Book*.

15.5.7.2. Using Multiple Resources for a Logical Font

You can use multiple XRC font resources for the same logical font, varying only the native descriptor used as the value of the map keyword. The XRC font mapper tries to use the mappings defined in the font statements, in numerically increasing order of font resource *id*. Here are multiple XRC font resources you might use for a Helvetica bold logical font:

```
font 1 Helvetica any Bold map "native_desc_1"
font 2 Helvetica any Bold map "native_desc_2"
font 3 Helvetica any Bold map "native_desc_3"
...
```

In this example, the XRC font mapper first attempts to use the mappings defined in font resource 1. If the logical font attributes there do not match those of the font being mapped, *or* if they do match but the physical font described by *native_desc_1* does not exist, the XRC font mapper tries the mapping defined in font statement 2.

If that fails, the mapper tries the rest of the font statements until one succeeds or until no more font resources remain.

Example: The following XRC code example shows how you could define five logical fonts along with five mappings for them on XVT/XM:

```
font 1 "lucida" 10 bold italic
font 2 "lucida" 24 italic
font 3 "lucida" any any
font 4 "lucidabright" 12 bold
font 5 "lucidabright" 14 any

font_map 1 "X1101/b&h/lucida/bold/i/normal/sans/10/100/75\
/75/p/67/iso8859/1"
font_map 2 "X1101/b&h/lucida/bold/r/normal/sans/24/240/75\
/75/p/152/iso8859/1"
font_map 3 "X1101/b&h/lucida/*/*/normal/sans/*/*/*/*/*/*\
iso8859/1"
font_map 4 "X1101/b&h/lucidabright/demibold/r/normal//12\
120/75/75/p/71/iso8859/1"
font_map 5 "X1101/b&h/lucidabright/*/*/normal//14/140/*/*\
p/*/iso8859/1"
```

This is how mapping would work with the above XRC resources. Suppose that an application created an XVT_FNTID and set its family, style, and size to Lucida, bold, and 18, respectively. The mapper would try to match this with font resource 1, which would match in family but not in style or size. It would then try font resource 2, which also would not match exactly because the size is wrong.

However, font resource 3 would match, because it matches the family and specifies "any" for style and size. The logical font would then be mapped to the specification in the font_map 3 string. According to this string, the mapped logical font would be Lucida bold, since the font statement wildcards the style. For size, 18 points would be used, because the font_map statement contains a wildcard in that field.

If these XRC specifications did not produce a successful match, the font mapping controller would invoke the default XVT mapper to perform the mapping.

Note: The mapping comparison for family is case-insensitive (i.e., "Lucida," "lucida," and "LUCIDA" all match).

15.5.8. XVT Default Font Mapper

The XVT default font mapper is the "last chance" mapper. It includes mapping for at least the four standard logical font families that XVT guarantees to support: System, Fixed, Times, and Helvetica. Logical fonts with these family names are guaranteed to map well across all XVT-supported platforms.

No such guarantee exists for less common logical font families such as "Calligraphic" or "Avant Garde." However, the XVT default font

mapper may provide other useful mappings for logical fonts that it knows about.

Implementation Note: XVT's default mapper uses whatever method is appropriate for a particular platform. The XVT Portability Toolkits for various platforms initialize their font mappers differently. XVT/XM initializes from XRC strings; XVT/Mac, and XVT/Win32 have a function that contains mapping logic.

A logical font is capable of being mapped as soon as it has been created with xvt_font_create. XVT's default mapper makes no assumptions about what attributes the application has set in the logical font. XVT's default font mapper interprets the logical font attributes like this:

native_descriptor
: If the native_descriptor attribute is set and is valid, it takes precedence over any portable attributes below. XVT's default font mapper parses this specification and attempts to load the specified physical font.

family
: If the native_descriptor is not set, the mapper attempts to map the logical font family attribute onto a physical font that has the same family. If no direct mapping is possible, a default font family is used in the mapping.

style
: If the native_descriptor is not set, XVT's default font mapper attempts to use the logical font style in the mapping. The style is always set: a value of 0 corresponds to the macro XVT_FS_NONE, which is itself a valid style. XVT's default font mapper attempts to find a physical font that has the same style as the logical font being mapped. If no matching physical font style is available, the mapper attempts to determine a "best fit" for style within the family.

size
: If the native_descriptor is not set, XVT's default font mapper attempts to match the logical font size as closely as possible using a "best fit" algorithm.

15.6. Font Selection Dialogs

Application users can interactively set logical font attributes from a dialog. XVT provides a Font Selection dialog for this purpose, with native look-and-feel, or you can provide a customized one. In the Font Selection dialog, users can select the logical font attributes they want to use from the full range of physical fonts available on the system.

Once a user selects the desired attributes, an E_FONT event is generated and the corresponding attributes (family, size, style, and native_descriptor) are set for the logical font. The logical font is not mapped at this point. When the logical font is mapped later for any reason, the user's selected attributes will be used.

Tip: To get the current state ofthe Font Selection dialog or the Font/Style menu:

Call xvt_menu_get_font_sel.

Tip: To set the default logical font for a dialog or the Font/Style menu:

Call xvt_menu_set_font_sel.

See Also: For more information about what happens when E_FONT events are generated, see section 15.7.2 on page 15-34 and section 4.5.8 on page 4-31.
Also see the description of xvt_dm_post_font_sel in the *XVT Portability Toolkit Reference.*

15.6.1. Implementing a Font Selection Dialog

XVT supplies a Font Selection dialog with a native look-and-feel. This dialog is accessible from any menubar that uses the DEFAULT_FONT_MENU. With it, your application can give users a native look-and-feel dialog from which they can choose a physical font and set the logical font attributes to be used in an XVT_FNTID.

Tip: To use XVT's Font Selection dialog:

Call xvt_dm_post_font_sel.

-OR-

From the application's Font/Style menu, select the menu item that invokes the dialog. The default style dialog opens automatically.

Note: The method of invoking the dialog from the Font/Style menu varies among platforms, conforming to native look-and-feel.

15.6.2. Customized Font Selection Dialogs

You can write customized Font Selection dialogs for your applications. When you do this, you have two basic options:

Native method

You can write the dialog natively. If you do this, you must rely on native inquiries about the availability of physical fonts and their attributes.

Portable method

You can write the dialog portably, using XVT portable font attributes instead of native font attributes. If you do this, you can use XVT inquiry functions to find out what logical fonts are available and what logical attributes can be set. The information returned by these functions comes from XVT's default and XRC-based font mappers.

15.6.2.1. Implementing a Customized Font Selection Dialog

After you create a customized Font Selection dialog, you then register the customized font dialog by using xvt_vobj_set_attr to set its attribute.

Tip: To use a customized font dialog:

Set the ATTR_FONT_DIALOG attribute. For example:

```
xvt_vobj_set_attr(NULL_WIN, ATTR_FONT_DIALOG,
      (long) my_font_dialog);
```

where my_font_dialog is an application function.

When creating a customized selection dialog, you can use the following inquiry functions:

```
xvt_fmap_get_families
xvt_fmap_get_family_sizes
xvt_fmap_get_familysize_styles
xvt_fmap_get_familystyle_sizes
xvt_fmap_get_family_styles
```

The following section provides some guidelines for you to follow when creating your dialog.

15.6.2.2. Guidelines for Creating Customized Dialogs

When creating a customized selection dialog, you should follow these guidelines:

1. The user should be allowed to cancel the dialog without changing the logical font.
2. The user should be allowed to change numerous logical font attributes before dismissing the dialog.
3. The Font Selection dialog should be modal.
4. The dialog should send an E_FONT event to the window whose WINDOW is passed into the function if the user makes a selection from the dialog. If the calling function doesn't want an E_FONT event to be sent, it should pass NULL_WIN as the window parameter to xvt_dm_post_font_sel. The font dialog event handler passes the updated default_font_id font as part of the E_FONT event.
5. If the default_font_id is modified, the function should return TRUE. If the default_font_id is *not* modified, the function should return FALSE.
6. The dialog should test to see that the passed-in window is not NULL_WIN before generating an E_FONT event. Similarly, the

dialog should confirm that the default_font_id is still valid before modifying it.

7. If the window passed in is NULL_WIN, or if the user cancels the dialog instead of exiting normally, no event is generated.

8. If the user makes a selection from the dialog and exits normally, the modified default_font_id should contain a reasonable set of logical font attributes that correspond closely to the dialog selection. This includes the family, style, and size. If the dialog is designed to select a specific physical font, the native descriptor should also be set. Sometimes no exact match exists between the selected physical attributes and the set of XVT portable attributes; this would occur if the physical font contains some attributes that don't correspond to any XVT portable attributes, or if the physical font has style settings that don't exist in XVT. In these cases, this dialog should attempt to specify a "best fit" with the XVT portable attributes. The default_font_id is modified independently of whether an E_FONT event is generated.

See Also: For a sample customized font selection dialog that uses the portable, usable method described above, see the Font Mapper (**samples/design/fontmap**) sample in the PTK sample set.

15.7. Font/Style Menus

On some platforms (XVT/XM and XVT/Mac), application users can set logical font attributes from a Font/Style menu. You can add this menu to your application by using DEFAULT_FONT_MENU in its XRC file.

15.7.1. Implementing a Font/Style Menu

From the Font/Style menu, users can select a logical font style, family, or size. Selecting attributes from the Font/Style menu generates an E_FONT event. The XVT_FNTID member of the E_FONT event represents the user's selection, including family, size, and/or style, but *not* native_descriptor.

The E_FONT event simply notifies the application that a Font/Style menu selection has been made; it does not automatically set check marks on the menu.

Tip: To get the state of check marks on the Font/Style menus:

Call xvt_menu_get_font_sel.

Tip: To set the font selection and check marks for the Font/Style menu:

Call xvt_menu_set_font_sel.

15.7.2. Responding to User Font Changes

Applications can use the Font/Style menu or Font Selection dialog to select the attributes of a logical font. Such a selection causes the Toolkit to notify the application that changes have been made to the logical font attributes. When that happens, a normal E_COMMAND event can't be passed to your application, because the E_COMMAND event doesn't contain enough information to communicate the logical font selection.

Instead, the application gets an E_FONT event. The XVT_FNTID member of the E_FONT event is a logical font that reflects the user's modifications to the logical font attributes, which could be a change of style, family, and/or size. (If the E_FONT event was generated by a Font Selection dialog, the native_descriptor attribute of the logical font may also be set.)

If your application has been keeping track of the current logical font with an XVT_FNTID variable, it can process this event by copying the font_id in the event, using the xvt_font_copy function. If you want to draw text with the logical font returned in the E_FONT event, call xvt_dwin_set_font.

When the application wants to tell the user what attributes are current, by updating the check marks on the Font/Style menu or changing the default physical font setting in the Font Selection dialog, it calls xvt_menu_set_font_sel.

After this call is made with a particular font_id as an argument, subsequent E_FONT events return a logical font with these same attributes as modified by the user's menu or dialog choices.

See Also: For more information about E_FONT events, see section 4.5.8 in Chapter 4, *Events*.
For more information about what happens when E_FONT events are generated, see section 15.6 on page 15-30 and section 4.5.8 on page 4-31.
Also see the "DEFAULT_*_MENU Values" section of the *XVT Portability Toolkit Reference*.

15.8. Working with Text

This section contains information about: 1) how to determine text width and font metrics for mapped logical fonts, 2) how to show highlighted text selections, and 3) a method for storing logical font information in a file.

15.8.1. Text Width and Font Metrics

To determine the width of a text string and metrics for a mapped logical font, you'll use these functions:

- xvt_dwin_get_font_metrics
- xvt_dwin_get_text_width
- xvt_font_get_metrics

If the logical font passed in these functions (or the one belonging to the window) is not mapped, the functions force an automatic mapping to occur. Metrics are available only for mapped logical fonts.

Figure 15.3 illustrates font metrics, which consist of the ascent, the descent, and the leading.

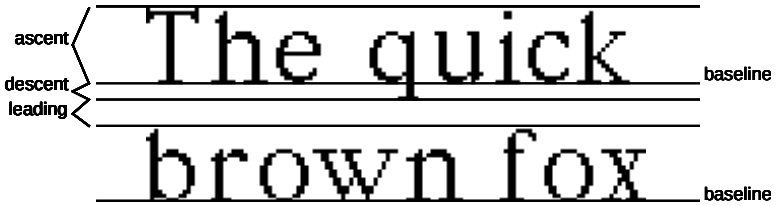


Figure 15.3. Font metrics

Tip: To find the width of a string in the specified window's current logical font:

Call xvt_dwin_get_text_width.
(It returns the width in device pixels.)

Tip: To find out the leading, ascent, and descent of amapped logical font:

Call xvt_font_get_metrics or
xvt_dwin_get_font_metrics.

15.8.2. Showing Text Selections

Text editing applications often must show a text selection as the user drags the mouse across a paragraph of text with the button down, as shown in Figure 15.4.

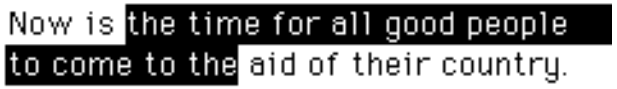


Figure 15.4. Selected text

To show a selection dynamically as the user drags the mouse, you must first display the text so the user can see it. You'll also need a data structure that contains the location (in pixels) of each character.

Tip: To show text selections within a single line:

1. On an E_MOUSE_DOWN event, round the horizontal and vertical coordinates of the mouse position to the nearest character starting position. Save that point in two variables, p and s1. Trap the mouse with trap_mouse. To remove the caret if there is one, call xvt_win_set_caret_visible(win, FALSE).
2. On an E_MOUSE_MOVE event (with the mouse down), round the mouse position to the nearest character starting position, and store it in q. That character, the one to the right of this point, is not part of the selection (yet).
3. Using xvt_rect_set, construct a rectangle from p (from step 1) to q (from step 2). The height of this rectangle is the height of the line, which your application determines. In simple cases it is the ascent plus descent plus leading that you obtained with xvt_font_get_metrics.
4. Set the pen to hollow, the brush to PAT_SOLID and COLOR_BLACK, and the draw mode to M_XOR. Then call xvt_dwin_draw_rect to draw the rectangle calculated in step 3.
5. Replace the value of p (first set in step 1) with the value of q.
6. Keep performing steps 2 through 5 as long as E_MOUSE_MOVE events occur. On the next E_MOUSE_UP, round off the mouse point as before and store it in the variable s2. Call xvt_win_release_pointer. At this point the selection is from s1 to s2, and it is already highlighted.

7. If s1 equals s2, the user did not drag the mouse, but merely clicked it. No selection has been shown because, even if an E_MOUSE_MOVE occurred, the rectangle drawn in step 4 was empty. This means that the user merely wanted to set the insertion point, so call xvt_caret_set_pos to position a blinking caret.

Note: To allow the user to drag from line to line, you must modify the algorithm shown above slightly.

Also, you'll usually want to allow a double-click (E_MOUSE_DBL) to select a complete word. Since the E_MOUSE_DBL follows E_MOUSE_DOWN and E_MOUSE_UP events, the algorithm given is executed prior to receiving the E_MOUSE_DBL. Clear the selection by drawing the same rectangle with a mode of M_XOR, then show the appropriate word as being selected.

You might also want to implement dragging after double-clicking (with the button still down) to allow a range of words to be selected.

See Also: For more information about mouse events, see the discussion of dragging in section 4.5.13 on page 4-48.

15.8.3. Transferring Logical Font Information

The XVT Portability Toolkit provides two functions that can serialize a logical font for writing to a file and deserialize it when reading from a file. If an application wants to preserve a permanent copy of a logical font, it must maintain the following font attributes:

- family
- size
- style
- native_descriptor

Tip: To serialize logical font attributes:

Call xvt_font_serialize.

This function serializes the logical font into a stream of bytes in a buffer.

Tip: To deserialize logical font attributes:

Call xvt_font_deserialize.

This function sets the logical font to correspond to the one saved by xvt_font_serialize.

16

CLIPBOARD

This chapter discusses the following topics related to using the clipboard:

- Clipboard formats
- Putting data on the clipboard
- Getting data off the clipboard
- Handling the Cut, Copy, and Paste commands

16.1. Clipboard Formats

XVT supports two predefined clipboard formats, for text and pictures, as well as an unlimited number of application-defined formats. A value of type CB_FORMAT indicates the format:

```
typedef enum {              /* standard clipboard format */
      CB_TEXT,              /* ASCII text */
      CB_PICT,              /* encapsulated picture */
      CB_APPL               /* app's type (name required) */
} CB_FORMAT;
```

The following sections describe the three formats.

16.1.1. CB_TEXT

The CB_TEXT format consists of a sequence of ASCII characters, possibly broken into lines that are terminated with an end-of-line sequence whose value is in the constant EOL_SEQ.

In all cases, the sequence is either a plain carriage return (\r), a plain line feed (\n), or a carriage return followed by a line feed (\r\n). The entire sequence is *not* terminated with a NULL byte. The only way to determine its end is to refer to the size parameter, which always accompanies the data itself.

When breaking CB_TEXT data into lines (such as after calling xvt_cb_get_data), you can avoid having to use EOL_SEQ directly by calling xvt_str_find_eol. However, when building CB_TEXT data, you must concatenate the contents of EOL_SEQ onto each line (with strcat, for example). The last line is not required to end with an end-of-line sequence.

16.1.2. CB_PICT

The CB_PICT format consists of a linear sequence of bytes that represents an encapsulated picture. The internals of this format are undefined, but you can safely pass the bytes from one address space to another (unlike a non-linearized PICTURE):

- If you already have an object of type PICTURE (returned by xvt_dwin_close_pict or xvt_pict_create), you can put it onto the clipboard directly with xvt_cb_put_data—it's not necessary to linearize it first.
- If you get a linearized picture off the clipboard with xvt_cb_get_data, you can turn it into a PICTURE object with xvt_pict_create.

Note: You should not access or delete the PICTURE after passing it into xvt_cb_put_data, since the clipboard "owns" the PICTURE.

16.1.3. CB_APPL

The CB_APPL format lets you put your own data structures onto the clipboard, presumably for use by other applications that know about those data structures. Each format has a name, which consists of 1 to 4 alphabetic and/or numeric characters. When referring to a CB_APPL format, you must also specify the name.

You can put unlimited CB_APPL formats onto the clipboard (along with CB_TEXT and CB_PICT formats, if you like) as long as they have different names.

The only requirement placed on your CB_APPL data structures is that they must be address-space independent, since they can be passed from one application to another. This means that they must not contain pointers, because those pointers would be invalid to the receiving application.

Tip: Another way to think about whether a data structure is valid is to ask yourself this question: if the data structure were written to a file, could another instance of your application read it back in and properly interpret it later?

16.2. Putting Data On the Clipboard

Tip: To put CB_TEXT or CB_APPL data on the clipboard:

1. Allocate clipboard memory with xvt_cb_alloc_data, which returns a pointer, and move your data there. (Note: This is *required.* It's not enough to have allocated memory with malloc, or even with xvt_gmem_alloc. For CB_PICT data, all you need is a PICTURE object; you don't have to call xvt_cb_alloc_data in this case.)
2. Open the clipboard with xvt_cb_open.
3. Put your data onto the clipboard with xvt_cb_put_data.
4. Close the clipboard with xvt_cb_close.
5. If you called xvt_cb_alloc_data, free the clipboard memory with xvt_cb_free_data.

Note: xvt_cb_put_data knows implicitly about the memory allocated with xvt_cb_alloc_data, so you do not pass it the pointer that xvt_cb_alloc_data returned. The correct order in which to call the functions is actually this: xvt_cb_open, xvt_cb_alloc_data, xvt_cb_put_data, xvt_cb_free_data, xvt_cb_close.

Example: The following code shows the steps for putting text onto the clipboard.

```
eol_len = strlen(EOL_SEQ);
size = 0;
for (i = 0; text[i] != NULL; i++)
        size += strlen(text[i]) + eol_len;
if ((p = xvt_cb_alloc_data(size)) == NULL) {
        xvt_dm_post_error("Cannot allocate clipboard memory.");
        return;
}
for (i = 0; text[i] != NULL; i++) {
        for (j = 0; text[i][j] != '\0'; j++)
                *p++ = text[i][j];
        for (j = 0; EOL_SEQ[j] != '\0'; j++)
                *p++ = EOL_SEQ[j];
}
if (!xvt_cb_put_data(CB_TEXT, NULL, size, (PICTURE)NULL))
        xvt_dm_post_error("Error putting text onto clipboard.");
xvt_cb_free_data();
```

Implementation Note: On XVT/Win32, you can put objects larger than 64K onto the clipboard, although your XVT program cannot retrieve them. However, non-XVT applications might be able to retrieve them.

16.3. Getting Data Off the Clipboard

Tip: To get data off the clipboard:

1. Determine if the format you desire is available by calling xvt_cb_has_format (perhaps several times for different formats).
2. If a usable format is present, open the clipboard with xvt_cb_open and retrieve a pointer to and the size of the data with xvt_cb_get_data.
3. If the data is CB_TEXT or CB_APPL, immediately move it into an area of memory that you have allocated (perhaps with malloc or xvt_gmem_alloc) because the data on the clipboard that the pointer addresses might disappear when you close the clipboard.
4. If the data is CB_PICT, capture it by calling xvt_pict_create.
5. As soon as possible, close the clipboard with xvt_cb_close to allow other applications to access it.

Example: The following code segment shows the operations used in getting data off the clipboard. Note that the type of format is checked first.

```
if (!xvt_cb_open(FALSE)) {
        xvt_dm_post_error("Error opening clipboard.");
        return;
}
free_data();
xvt_dwin_invalidate_rect(win, NULL);
data.fmt = format.fmt;
if ((p = xvt_cb_get_data(data.fmt, format.name,
        &data.size)) == NULL) {
        if (user_initiated)
                xvt_dm_post_note("No data in chosen format.");
}
else
        switch (data.fmt) {
        case CB_TEXT:
        case CB_APPL:
                if (data.size > SZ_TEXT) {
                        xvt_dm_post_error
                        ("More than %d bytes of data on clipboard.",
                        SZ_TEXT);
                        break;
                }
                if ((data.ptr = xvt_mem_alloc((unsigned)data.size))
                == NULL) {
                        xvt_dm_post_error("Cannot allocate memory.");
                        break;
                }
                memcpy(data.ptr, p, (int)data.size);
                data.valid = TRUE;
                break;
        case CB_PICT:
                if ((data.pict = xvt_pict_create(p, data.size,
                &data.frame)) == NULL_WIN) {
                        xvt_dm_post_error("Error making picture.");
                        break;
                }
                data.valid = TRUE;
        }
if (!xvt_cb_close())
        xvt_dm_post_error("Error closing clipboard.");
```

16.4. Handling Cut, Copy, and Paste Commands

If possible, every application should implement the Cut, Copy, and Paste items on the standard XVT Edit menu:

Cut and Copy

These are the same as far as the clipboard is concerned; the only difference between them is that Copy simply puts data onto the clipboard, whereas Cut also deletes it from the document. Enable Cut and Copy (with xvt_menu_set_item_enabled) only if the user has selected something that can be put onto the clipboard.

Paste

This command gets data from the clipboard and inserts it into the document. Enable Paste only if the clipboard contains data that can be inserted (determine this by calling xvt_cb_has_format, usually several times for different formats).

Note: You should enable or disable these menu items every time one of your document windows is activated (E_FOCUS event with active set to TRUE), since the clipboard may have changed while another application ran.

Putting Data On the Clipboard

When putting data on the clipboard, you should try to cast the data into both CB_TEXT and CB_PICT formats if possible, or one of them at least. You should also put the data into one or more of your own CB_APPL formats.

Your application should allow the user to paste in CB_TEXT or CB_PICT data, if at all reasonable. The Paste command should first look for a CB_APPL format, then CB_PICT, and then CB_TEXT. The idea is to pass as much structure as possible through the clipboard.

Getting Data Off the Clipboard

When getting data off the clipboard, you generally call xvt_cb_get_data just once because you have already determined what format is available with xvt_cb_has_format. When putting data onto the clipboard you usually call xvt_cb_put_data several times, once for each format to which you can convert the data.

17

FILES

This chapter discusses how to handle files for XVT applications. It covers the following topics:

- Portably referring to filenames, directories, and file types with XVT's FILE_SPEC data type
- Getting and setting file attributes
- Using standard functions for file input and output
- Processing selected files
- Standard file dialogs

You may need to change the way you process file and pathnames. Since file and pathnames can contain multibyte characters, they must be treated like other multibyte strings. All PTK functions and data types that accept file or pathname strings are multibyte capable.

Any application that needs to use wide characters for the purposes of internationalization (i.e., a multibyte-aware application) should use the XVT string processing functions. Also, do not use the XVT R3 function xvt_fsprintf and others like it. For more details, refer to section 19.3.3.1 on page 19-39.

17.1. Portable Filenames, Directories, and Types

17.1.1. SZ_FNAME Constant

The symbol SZ_FNAME defines the length of filenames supported (include '/0').

17.1.2. SZ_LEAFNAME Constant

The XVT constant SZ_LEAFNAME defines the maximum byte length of a single token in a file pathname (a directory name or a filename). This maximum value takes into account any file extensions (including the periods '.') but excludes pathname delimiters ('/', '\' or ':'). The value of this constant is platform-specific and may depend on the character code set used. It is convenient to use this constant with the XVT function xvt_fsys_parse_pathname.

Note: By comparison, the constant SZ_FNAME is defined as the maximum byte length of a full pathname for a particular file system.

17.1.3. FILE_SPEC Data Type

The operating systems underneath the various XVT implementations handle file and directory names quite differently. As a result, XVT must provide an abstract way to refer to files, directories, and file types. It does this with the data type FILE_SPEC:

```
#define SZ_FNAME ...
                                /* max len of name (excl. NULL) */
typedef struct {
        DIRECTORY dir;          /* directory */
        char type[6];           /* file type or extension */
        char name[SZ_FNAME + 1];
                                /* name (or partial path) */
        char creator[6];        /* Mac creator */
} FILE_SPEC;
```

17.1.4. DIRECTORYs

The internals of a DIRECTORY are hidden from XVT applications. Consequently, you must not assume that a directory specification is even a character string. Whenever the user enters a file specification using the standard file dialogs, the application also receives a DIRECTORY. Hence, XVT handles most operations on directories in an abstract way, and a portable application does not need to know what a DIRECTORY actually is.

Several XVT functions let you portably manipulate a DIRECTORY.

Tip: To change the default DIRECTORY to the DIRECTORY that was current when the application started:

Call xvt_fsys_set_dir_startup.

Tip: To get the present current directory:

Call xvt_fsys_get_dir.

Tip: To set the directory to a specific DIRECTORY:

Call xvt_fsys_set_dir.

Tip: To save the current DIRECTORY:

Call xvt_fsys_save_dir.

Tip: To restore the current DIRECTORY:

Call xvt_fsys_restore_dir.

Tip: To convert an abstract DIRECTORY to a local, non-portable string:

Call xvt_fsys_convert_dir_to_str.

Tip: To convert a local, non-portable string to an abstract DIRECTORY:

Call xvt_fsys_convert_str_to_dir.

Tip: xvt_fsys_convert_str_to_dir is particularly useful when you prompt the user for a directory path using a method other than XVT's standard file dialogs, or when the pathname is in a resource file as a string.

Tip: To construct a native (single-byte or multibyte) pathname string from the pathname pieces:

Call xvt_fsys_build_pathname.

Tip: To parse a (single-byte or multibyte) pathname string, breaking it into pathname tokens (volume name, directory path, leaf root name, leaf extension, and leaf version):

Call xvt_fsys_parse_pathname.

See Also: For more information about processing strings, including filenames and pathnames, in a multibyte-aware application, see section 19.2.4 on page 19-24.

Example: This example shows how you might use SZ_LEAFNAME with xvt_fsys_parse_pathname to parse a file pathname:

```
static void parse_pathname(char* buf)
{
        char dir[SZ_FNAME];
        char ext[SZ_LEAFNAME];
        char file[SZ_LEAFNAME];
        char ver[SZ_LEAFNAME];
        char vol[SZ_LEAFNAME];
        char fullname[SZ_FNAME]; /* reconstructed
                                                    pathname */

        xvt_dm_post_note("Original pathname is = \"%s\"",
                                buf);
        if (!xvt_fsys_parse_pathname(buf, vol, dir,
                        file, ext, ver))
                xvt_dm_post_note("Failed to parse pathname");
        else {
                if (xvt_fsys_build_pathname(fullname, vol,
                                dir, file, ext, ver))
                        xvt_dm_post_note(
                                "Reconstructed pathname is \"%s\"",
                                fullname);
                else
                        xvt_dm_post_note(
                                "Fail to rebuild pathname");
        }
}
```

17.1.5. File Types

A file type is a three- or four-letter string by which the application refers to files containing its documents. On some XVT platforms, when the user is at the “desktop” level, he or she can click on such a file, and the corresponding application is started.

Tip: To set the file type:

Call xvt_fsys_set_file_attr
(...,XVT_FILE_ATTR_TYPESTR,...)

For portability, the type string should be in a resource file, not in the program itself.

Implementation Note: On XVT/Mac, this function also sets the creator:
xvt_fsys_set_file_attr(...,XVT_FILE_ATTR_CREATORSTR,...)
On other platforms, this argument is ignored.

17.2. Getting and Setting File Attributes

XVT abstract files encapsulated by the FILE_SPEC data structure have attributes that you can get and set with the XVT functions xvt_fsys_get_file_attr and xvt_fsys_set_file_attr.
You can get all of the following file attributes, but can set only the last two:

Attribute:	Description:	Access:
XVT_FILE_ATTR_EXIST	File existence	Get
XVT_FILE_ATTR_READ	Can file be read	Get
XVT_FILE_ATTR_WRITE	Can file be written	Get
XVT_FILE_ATTR_EXECUTE	Can file be executed	Get
XVT_FILE_ATTR_DIRECTORY	Is file a directory	Get
XVT_FILE_ATTR_NUMLINCKS	Number of links to file	Get
XVT_FILE_ATTR_SIZE	Size of file in bytes	Get
XVT_FILE_ATTR_ATIME	Last time accessed file	Get
XVT_FILE_ATTR_MTIME	Last time modified file	Get
XVT_FILE_ATTR_CTIME	Creation time of file	Get
XVT_FILE_ATTR_DIRSTR	Directory name of file	Get
XVT_FILE_ATTR_FILESTR	Filename	Get
XVT_FILE_ATTR_CREATORSTR	Creator	Get/Set
XVT_FILE_ATTR_TYPESTR	File type	Get/Set

17.3. File Input and Output Using Standard Functions

You can open, create, read, write, and perform other operations on files using standard C functions—XVT doesn't have to provide specific toolkit-independent support for these. However, XVT does provide portable functions for these operations:

- Obtaining a filename from the user (so that the file can be opened)
- Specifying directories and file types in an abstract, portable way
- Handling related operations

If possible, you should use these standard I/O library functions, because they are supported by all compilers that XVT works with:

clearerr	fputs
creat	fread
fclose	freopen
feof	fseek
ferror	ftell
fflush	fwrite
fgetc	open (with 2 arguments only)
fgets	putc
fileno	rewind
fopen	setbuf
fprintf	sprintf
fputc	ungetc

Implementation Note: Because printf is not supported natively on all XVT-supported platforms, you should avoid using it.

17.4. Processing Selected Files

It's possible for the application to be started when the user selects one or more documents. XVT provides functions to get the names of these files.

Tip: To find out how many files were selected, and their names:

1. Call xvt_app_file_count.

   This function also tells you whether the user has selected the files for printing, rather than for opening (it might return zero).

2. Call xvt_app_get_file repeatedly, until it returns NULL.

3. After processing a file, call xvt_app_set_file_processed to indicate that you're done with it.

Tip: To get a list of all files and/or directories in the current directory:

Call xvt_fsys_list_files.

This function also allows you to restrict the list to the files whose names match a wildcard pattern.

17.5. Standard File Dialogs

Each toolkit on which XVT runs has standard file dialogs that applications use to get filenames from the user. Figure 17.1 shows an example of the standard Open dialog, which gets the name of a file to open. Figure 17.2 shows an example of a standard Save dialog.

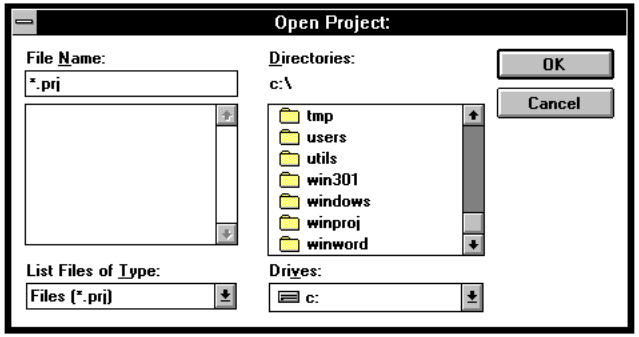


Figure 17.1. Open dialog

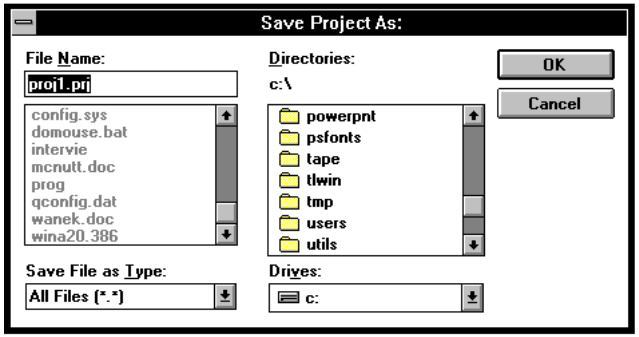


Figure 17.2. Save dialog

Tip: To display the standard Open dialog:

Call xvt_dm_post_file_open.

The files shown in the list box are of the type specified in the FILE_SPEC argument. On a successful return, the DIRECTORY and filename members of FILE_SPEC are set to the user's selection.

Tip: To display the standard Save dialog:

Call xvt_dm_post_file_save.

The initial name for the edit box is supplied in the FILE_SPEC argument, and that argument contains the user's response upon a successful return.

Tip: To display the standard Directory dialog:

Call xvt_dm_post_dir_sel

A list of directories will be displayed for selection. On a successful return, the DIRECTORY member of FILE_SPEC is set to the selected directory.

After getting a FILE_SPEC from any of the functions, open the file in the normal way, for example with fopen. Upon creating a new file, since fopen doesn't handle file types, you must set its type with a call to this function:

```
xvt_fsys_set_file_attr
    (...,XVT_FILE_ATTR_TYPESTR,...)
```

Implementation Note: On XVT/Mac, the following function sets the file's creator, which should be a four-character name unique to the application:

```
xvt_fsys_set_file_attr
    (...,XVT_FILE_ATTR_CREATORSTR,...)
```

18

PRINTING

This chapter contains information about the following printing topics:

- Basic printing steps
- The printing context (print records and print windows)
- Printing to a print window
- Restrictions on printing
- The standard page setup dialog
- Aborting printing
- Initializing and terminating printing
- Working with printer drivers

18.1. Basic Printing Steps

The basic XVT model for a printing cycle involves the following steps:

- Create a printing context
- Create a print thread
- Create a print window
- Iterate on print pages
- Iterate on print bands within the print page
- Call XVT drawing functions
- Destroy print window
- Destroy printing context

See Also: For more information about printing, see the xvt_print_* functions in the *XVT Portability Toolkit Reference*.

18.2. Print Records and Print Windows

XVT uses objects of type PRINT_RCD (print record) to keep track of the printing context, including the page setup. The actual printing is done into a "print window" (a WINDOW of type W_PRINT) similar to drawing into a normal screen window.

18.2.1. Print Records

Most printing functions require information about the printer. This contextual information is provided in a print record. This section gives information on creating, using, and destroying a default print record.

18.2.1.1. PRINT_RCD Data Type

XVT does not expose the actual declaration of a print record. Instead, XVT supplies a fictitious declaration so that applications can declare pointers to a PRINT_RCD:

```
typedef struct {
    ...
} PRINT_RCD;
```

Tip: To create a default PRINT_RCD:

Call xvt_print_create.

Tip: To destroy a print record you no longer need:

Call xvt_print_destroy.

18.2.1.2. Using Print Records

You should save a document's print record for at least the duration of a print cycle.You may opt to save it longer, since it can contain page setup information that the user has set and will expect your application to remember.

The call to xvt_print_create returns the size of the print record, so that it can be written to a file. When the print record is read back in, you must make sure that it is valid for the current chosen printer.

Tip: To determine if the print record is valid for the current printer:

Call xvt_print_is_valid.

Tip: When you design the data structures for saving your documents, allow room for the print record, too.

18.2.2. Print Windows

Printing is an operation very similar to drawing into normal screen windows. This section provides information on creating, using, and destroying a print window.

Tip: To create a print window:

1. Call xvt_print_create to obtain a print record.
2. Pass this print record to xvt_print_create_win. You must supply the name of the document to be printed so the print spooler or network can identify the job.

Tip: To use the print window:

Use the appropriate drawing tools and drawing functions, just as if you were drawing in a normal window. The drawing isn't displayed on the screen, but is instead printed on paper.

Tip: To destroy the print window:

1. When the print job is completed, destroy the print window by calling xvt_vobj_destroy, passing the print window as its parameter.
2. After the print window has been destroyed, you can optionally destroy the print record by calling xvt_print_destroy.

18.3. Printing to a Print Window

This section discusses print pages, print bands, and portable print functions.

18.3.1. Print Pages

In XVT printing, a print job is divided into pages.

Tip: To start a new page of printing:

Call xvt_print_open_page with a valid print record. If this function returns FALSE, terminate your print job.

Tip: To end a page of printing:

Call xvt_print_close_page.

Your application determines how many pages to print.

18.3.2. Print Bands

Each page of a print job is divided into rectangular bands. On most platforms there is only one band per page. However, on platforms in which native memory limitations occur, there are multiple bands. The number of bands is in part determined by the resolution of the printer. Your application is responsible for drawing into each of these bands to fill out the page.

Tip: To print a page of print bands:

1. Call xvt_print_get_next_band repeatedly in a while loop until it returns NULL. For each band, print at least the portion of the page that lies within the rectangle returned by xvt_print_get_next_band. Any drawing outside of the rectangle will be clipped.
2. Use xvt_dwin_is_update_needed to determine if a particular rectangle of interest intersects the print band.

18.3.3. Writing a Portable Printing Function

On some XVT platforms, printing is performed in a separate thread of execution. To ensure portability, the print job should be encapsulated in a single function. This function begins with a call to xvt_print_create_win and ends with a call to xvt_vobj_destroy.

You must define a function similar to this:

```
BOOLEAN XVT_CALLCONV1 print_document(long data)
{
    if ((print_win = xvt_print_create_win
                    (print_rcd, "Doc Title") == NULL_WIN)
    return FALSE;
    ...
    xvt_vobj_destroy(print_win);
}
```

The function must return TRUE if the job succeeds and FALSE otherwise. The function should do nothing other than printing and drawing operations.

Note: To ensure portability, use XVT_CALLCONV1 in all prototypes and headers for XVT callback functions, including print functions.

18.3.3.1. Invoking Your Printing Indirectly

To execute your printing function, call xvt_print_start_thread. Do not call your printing function directly. XVT will call it for you, as shown in this example:

```
case M_FILE_PRINT:
    if (!xvt_print_start_thread(print_document,
            PTR_LONG(&data)))
        xvt_dm_post_error("printing failed");
    break;
```

The long argument of xvt_print_start_thread is passed to the printing function. You can use it for any purpose, but usually it passes data to the print function. The value returned by your print function becomes the value returned by xvt_print_start_thread.

Some XVT implementations return immediately from xvt_print_start_thread, while others do not return until your printing function returns. You must protect your data from unwanted cross-thread corruption.

Caution: To be safe, always return immediately from the window event handler that called xvt_print_start_thread. Even if you know that your print function will execute in a separate thread, do not attempt concurrent processing during printing.

See Also: For more information about xvt_print_start_thread, refer to its description in the *XVT Portability Toolkit Reference*.

18.3.3.2. Print Thread Implementations

On platforms that do not support thread based printing, the xvt_print_start_thread function simply calls your print function and is implemented like this:

```
BOOLEAN xvt_print_start_thread(XVT_PRINT_FUNCTION fcn,
    long data)
{
    return(fcn(data));
}
```

Caution: You cannot test whether you have successfully refrained from calling the prohibited functions on non-threading platforms. On the other hand, calling prohibited functions won't cause a problem except on platforms that support threading.

18.3.4. Calls You Can Make From a Print Function

You can call the following XVT functions from a print function:

xvt_cb_*

xvt_debug_*
xvt_dwin_* (except xvt_dwin_invalidate_rect,
xvt_dwin_scroll_rect, and xvt_dwin_update)
xvt_errid_*
xvt_errmsg_*
xvt_font_*
xvt_fsys_*
xvt_gmem_*
xvt_image_*
xvt_iostr_*
xvt_mem_*
xvt_palet_*
xvt_pict_*
xvt_pmap_*
xvt_print_* (except xvt_print_start_thread,
xvt_print_open and xvt_print_close)
xvt_rect_*
xvt_res_*
xvt_scr_beep
xvt_scr_busy_cursor
xvt_slist_*
xvt_str_*
xvt_vobj_destroy (for print window only)

Caution: A print window does not receive update events, and you must not call xvt_dwin_invalidate_rect or xvt_dwin_update.

18.3.5. Sample Print Function

The following function shows the basic procedure to print a document. Remember, a print function must be called by xvt_print_start_thread, as shown earlier in this section.

```
BOOLEAN XVT_CALLCONV1 print_document(long pages)

{
    int page;
    WINDOW print_win;
    RCT* rct_band;

    if (print_rcd == NULL ||
       (print_win = xvt_print_create_win(print_rcd,
                "Document")) == NULL_WIN)
        return FALSE;               /* user cancelled--no error */

    for (page = 1; page <= (int)pages; page++) {
        if (!xvt_print_open_page(print_rcd))
            break;
        while ((rct_band = xvt_print_get_next_band()) !=
```

```
                    NULL)
                app_draw_page(print_win,rct_band,page);
            if (!xvt_print_close_page(print_rcd))
                break;
        }

        xvt_vobj_destroy(print_win);
        return TRUE;
}
```

18.4. Printing Restrictions

Printing can occur in a separate thread, and this thread canshare data with the main thread. To protect data from corruption, XVT imposes strict restrictions on what you are allowed to do in a printing function:

- In general, do not do anything that can generate an event; in particular, avoid E_UPDATE events
- Do not create or destroy any window, dialog, or control (except, of course, for the print window)
- Do not move a window, bring a window to the front, or perform a similar operation
- Draw only what goes on the printed page

18.5. Printer Page Setup

This section discusses modifying the print context record and querying the printer attributes.

18.5.1. Page Setup Dialog

You can display a standard dialog to let the user adjust the page setup and store the settings in the print record. Figure 18.1 shows a typical page setup dialog.

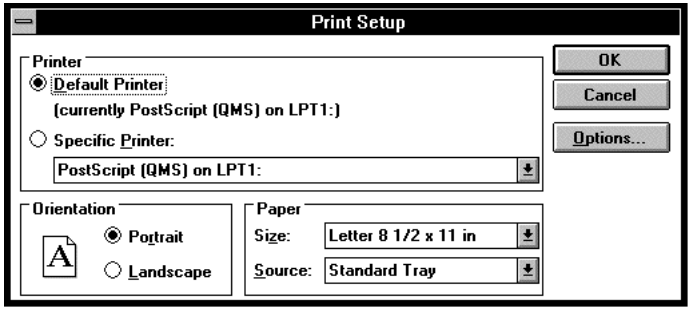


Figure 18.1. Standard Page Setup dialog on MS-Windows

Tip: To display a standard page setup dialog:

Call xvt_dm_post_page_setup.

See Also: For more information about page setup dialogs, see the description of xvt_dm_post_page_setup in the *XVT Portability Toolkit Reference.*

18.5.2. Print Metrics

Tip: To find the total size of a page and the printer's resolution:

Call xvt_app_escape(XVT_ESC_GET_PRINTER_INFO...)
or xvt_vobj_get_attr(ATTR_PRINTER_*).

You must do this before printing because, in general, these print metrics differ from those for the screen.

The following code demonstrates querying printer metrics:

```
        METRICS pmetrics;
#if (XVTWS == MACWS) || (XVTWS == NTWS)
        {
                long height, width, vres, hres;

                xvt_app_escape(XVT_ESC_GET_PRINTER_INFO,
                        print_rcd, &height, &width, &vres, &hres);
                pmetrics.width = (short)width;
                pmetrics.height = (short)height;
                pmetrics.hres = (short)hres;
                pmetrics.vres = (short)vres;
        }
#else
        pmetrics.width = (short)xvt_vobj_get_attr(NULL_WIN,
                ATTR_PRINTER_WIDTH);
        pmetrics.height = (short)xvt_vobj_get_attr(NULL_WIN,
                ATTR_PRINTER_HEIGHT);
        pmetrics.hres   = (short)xvt_vobj_get_attr(NULL_WIN,
                ATTR_PRINTER_HRES);
        pmetrics.vres   = (short)xvt_vobj_get_attr(NULL_WIN,
                ATTR_PRINTER_VRES);
#endif
```

See Also: For details on possible platform-specific print functionality, see the platform-specific books.

18.6. Aborting a Print Job

On most platforms, while a print job is underway, XVT displays a dialog box that lets the user abort the job. On XVT/XM, however, the application does not communicate with the printer. Instead, these platforms create a PostScript file that the application is responsible for sending to the printer. As a result, no dialog box appears.

On other platforms, if the user aborts the print job, the next call to xvt_print_open_page returns FALSE. When your application receives the FALSE return, you should finish the print job with a call to xvt_vobj_destroy.

18.7. Initiating and Terminating Printing

Before calling any printing functions, you must have previously called xvt_print_open. However, XVT's printing functions xvt_print_create and xvt_print_create_win make this call automatically. XVT provides these functions to enable printer attributes to be queried.

A print job is always bracketed between two calls: xvt_print_open, which initializes the printer for output, and xvt_print_close, which terminates printer output.

When you call xvt_print_open, call xvt_print_close as soon as you're done with print functions. Call both functions each time the application needs to print, or allow the xvt_print_create_win and xvt_vobj_destroy functions to do it for you.

Tip: Do not call xvt_print_open only at application initialization and xvt_print_close only at termination, since this can prevent other applications from printing.

18.8. Printer Driver Issues

XVT attempts to take the most general approach to working with printer drivers (on platforms which support them). XVT implements its printing API to the printing specifications of the operating system vendors. Printing with XVT is not guaranteed to work with printer drivers that do not meet these specifications.

Sometimes, printer drivers fail to work properly. Examples of problems that printer drivers may cause include the improper reporting of printer attributes, the improper sizing of page margins, the invalid setting of requested fonts, the incorrect setting of line pen styles, and the premature termination of printing by network printers. XVT attempts to work around such problems known to exist in commonly used drivers.

In writing your application, you should be aware that such problems exist. You may need to make adjustments in your drawing for the drivers you require.

19

MULTIBYTE CHARACTER SETS AND LOCALIZATION

This chapter introduces the terminology, concepts, and methods involved in developing XVT applications that support locales and international languages. The XVT Portability Toolkit (PTK) application programming interface(API) and XVT resource files are internationalized and support the use of multibyte character codesets.

XVT-Design can generate its standard application code and resources in internationalized form and supports easy localization of developer-written code.

See Also: You should refer to the *XVT Platform-Specific Books* and to your native platform documentation for more information on setting locale data for your platforms.

19.1. Around the World with XVT

19.1.1. About Internationalization and Localization

This section highlights some of the general issues involved in adapting applications for international language and locale support.

19.1.1.1. Why and When to Adapt an Application

Adapting an application for a specific locale, or *localization*, involves several issues. You must evaluate the differences in locales to determine which, if any, locale categories are relevant to the application. For example, written language (and character codesets) may be a minor issue to some localization efforts. An application

developed for American users will have only slight differences for British citizens (date formatting, monetary formatting and minor variations in spelling). In other cases, all locale categories may be affected by a localization effort. An Asian language-based application will have significantly different needs than the same English-based application including such categories as character codesets, layout, collation, and monetary formatting. Localization, and to what degree localization isperformed, are strongly dependent on the target locales of the application.

Several important tasks are involved in the localization of an application. Often, the most daunting task is the translation of string literals from English (or another original language; this guide assumes that English is the base language) to the local language. Often this task is outside the scope of the application developer and will require you to obtain specialized expertise. Secondary to this task is proper processing of those strings whether that be sorting, concatenating, parsing, or simply formatting strings for screen layout. Other related tasks include creating locale-specific resource files and setting the environment configuration appropriate to the locale.

Internationalization (I18N) involves modifying application code and resource files so they can be easily localized. Ideally, the result of I18N is that localization efforts can be accomplished without changing application source code and without requiring code recompilation. Applications that target a significant variety of locales are candidates for internationalization. In deciding whether to internationalize your application or not, you must evaluate the pros and cons that arise from supporting several versions of an application, each for a specific locale, or even a singleversion which contains code for multiple locales. Also, it is important to consider which locales you may need to support in the future.

There are several things you need to consider before you internationalize an application. In addition to handling single-byte character strings, your application code must be able to process multibyte or wide character strings depending upon the locales you must support. This includes general processing for collation, parsing, formatting and layout. String literals and other locale-sensitive items should be made external to the application source code so they can be translated and substituted as needed.

See Also: For a specific description of how to internationalize and localize your XVT applications, see the next section, section 19.1.1.2.

19.1.1.2. How to Adapt an Application

This section describes an overall methodology for writing XVT applications that support locales and international languages. Then the section provides specific steps you can follow to implement the methodology.

Additional details are provided for customers who have access to XVT-Design. Paragraphs formatted like this paragraph introduce XVT-Design-specific information.

Remember to debug your application prior to undertaking the localization effort. Resolving localization issues is much easier if your application is working well to begin with. Although this seems to add extra steps to your development process, it actually reduces the total amount of effort by cleanly separating coding problems. As you gain experience in adapting applications, you may begin to find it easier to write localized XVT applications from scratch.

See Also: Applications can be localized to some languages without using wide (multibyte) characters. However, localizing to other languages, such as Japanese, will require "specialcharacters." For more information, refer to sections 19.1.2.1 on page 19-14 and 19.3.3.4 on page 19-41.

Internationalization

Internationalization requires disassociating any locale-sensitive information from your application and encapsulating it in external files such as resource files. Any locale-sensitive processing operations also must be encapsulated and handled in a general manner. Some of the factors you must consider when internationalizing include the following:

- String literals
- Special strings and data that require locale-specific formatting or parsing (i.e., sprintf, textual representation of numbers, date/time formats, proper word- and line-wrap)
- Dialog and window layout
- Graphics (icons, bitmaps)
- Colors
- Font references
- Keyboard modifiers, mnemonics and accelerators
- Help source files

If you are using character codesets that use wide character or multibyte encoding schemes, your application code for manipulation of strings must be modified to handle these character codesets. The following string operations are candidates for modification or replacement:

- Collation
- Parsing
- Incrementing or decrementing character pointers
- Character or string comparison
- Handling upper and lower case (some languages are indifferent)
- Conversion between character codesets

Appropriate text and graphic object positions and dimension data should also be removed from the application and be placed instead in external resource files.

For XVT-Design Users Only

If you are an XVT-Design user, refer to the following list of general steps to internationalize your XVT application. Most of the information in these steps is described in greater detail in later sections of this chapter or in the XVT Platform-Specific Books. Another source of information is the chapter on Internationalization and Localization in the XVT-Design Manual.

1. In the Application Attributes dialog, select **Internationalization**. This selection causes SPCL:I18N_Header, SPCL:I18N_XRC, and SPCL:I18N_Main tags to be created (as described below), as well as inserting special localization macros.
2. In the SPCL:I18N_XRC tag in the ACE, you now see code similar to the following:

   ```
   #ifdef XVT_LOCALIZABLE
   #include “strres.h”
   #endif
   ```

   The XRC include file **strres.h** contains locale-specific strings; it is generated in step 1 (on page 19-7 below) when you run the **strscan** utility program.
3. In the SPCL:I18N_Header tag in the ACE, you now see code similar to the following:

```
#ifdef XVT_LOCALIZABLE
#include "strdef.h"
#endif
```

The include file **strdef.h** will be generated in step 1 (on page 19-7 below). **strdef.h** contains #defines for resource IDs used in **strres.h**.

4. Still in the ACE, replace string literals in your code with calls to the LOCAL_C_STR macro (for details about this macro, see section 19.3.2.2 on page 19-36). Use the XVT-Design **Find** command to help you locate string literals.

5. Using the SPCL:User_Header tag in the ACE, add the following code:

   ```
   #define XVT_LOCALIZABLE
   ```

   Alternatively, you may modify your makefile or makefile templates to define this flag.

6. Use the XVT-Design **Generate Application** command to generate all files.

Now, working outside of XVT Development Solution for C and its visual design tool, XVT-Design, complete these two additional steps:

7. In your external files (those not generated by XVT-Design), replace string literals with calls to the LOCAL_C_STR macro.

Once you have completed the steps presented in the preceding list (steps 1 through 7), your C application is modified to use the LOCAL_C_STR macro and your application is "internationalized." In other words, your application's displayable strings have been processed in a manner that allows them to be easily "localized," that is to say, modified for a specific locale.

This marks the end of XVT-Design-specific information about internationalization.

Localization

Localization is quite straightforward once your application has been internationalized. The biggest part of localization is placing string literals in an external file that can be modified as required by specific locales.

Your application must be localized for each unique environment in which it will operate. The steps vary slightly depending on the application and the selected locales, but generally speaking, plan on the following steps:

1. Decide which character codeset to use for translation depending on which languages you need to support and on which operating systems your application must execute. Different codesets used on the various platforms that XVT supports are listed in section A.2 in Appendix A.
2. Translate string literals to the target language.
3. Set up special strings such as dates and times for formatting.
4. Select the appropriate keyboard modifiers, mnemonics and accelerators.
5. Select fonts appropriate to the character codeset.
6. Provide locale-specific icons and colors.
7. Adjust text and graphic object sizes and positions.
8. Compile locale-specific resource and help files.
9. Establish the proper operating/window system locale-specific environment (set up environment variables, code pages, etc.).
10. Set the application locale environment information (locale information can be bound at application build time or application startup time.

See Also: To see hundreds of examples of international symbols used in various fields of endeavor, refer to *Symbol Sourcebook: An Authoritative Guide to International Graphic Symbols*, by Henry Dreyfuss, published by Van Nostrand Reinhold, New York, N.Y., 1984.

For XVT-Design Users Only

If you are an XVT-Design user, refer to the following list of general steps to localize your XVT application. Most of the information in these steps is described in greater detail in later sections of this chapter or in the XVT Platform-Specific Books. Another source of information is the chapter on Internationalization and Localization in the XVT-Design Manual.

1. Execute the **strscan** utility on all of your ***.c** and ***.xrc** files to generate the include files **strres.h** and **strdef.h**. If you have carefully followed steps 4 through 7 (on page 19-5), **strres.h** now contains all your locale-specific strings. View both files after running the utility.

2. Make copies of **strres.h** and give them names that co-workers will recognize as locale-specific resource files, such as **engres.h** and **gerres.h**. You will want to adopt a file naming convention for your different versions of **strres.h**. Renaming the files protects you in the future when you run **strscan**, since **strres.h** is consistently and predictably overwritten when it already exists. Filenaming conventions are discussed in more detail in section 19.2.1 on page 19-18.

3. Using the SPCL:I18N_XRC tag in the ACE, replace the reference to **strres.h** with references to a file of strings translated into German, **gerres.h**, and another file of English strings, **engres.h**. When the editing in your application resource file is complete, this section of code will resemble the following:

```
#ifdef XVT_LOCALIZABLE
#ifdef LANG_GER_W52
      #include "gerres.h"
#else /* English */
      #include "engres.h"
#endif
#endif
```

   If you are supporting multiple languages in other localized files, modify the above code as needed to reference these files, as well.

4. Translate the strings in the locale-specific resource files, such as **gerres.h**, for the locales you need to support.

5. Consider redefining the way dates or money variables are displayed (to match local practices). Likewise, in your external files (those not generated by XVT-Design), search for all sprintfs

that you wish to format for locale-specific display. For more details and an example, refer to section 19.3.4 on page 19-43.

6. Compile your resources and check the translation of text and the size and position of GUI objects.

7. Adjust the size and positions defined by creation rectangles in **strres.h** to accommodate the increased or decreased lengths of the translated strings.

You do not need to re-translate your entire **strres.h** file when you make changes to your application. Usually it is only necessary to regenerate **strres.h** and **strdef.h** using **strscan**, then identify the strings that have been added or changed and add their translated equivalents to your translated versions of **strres.h**.

Building the locale-specifc executable requires the setting of one or more specific #defines. XVT source code files are "localized" when XVT_LOCALIZABLE is defined, and switch to a specific language based upon other #defines, as well. To build the locale-specifc executable, follow these additional steps:

8. Modify your makefile or makefile templates to build localized versions of your resources. If you wish to build, for example, a German version, you would also define LANG_GER_W52. The various compile constants you can use are listed in Table 19.2 on page 19-19.

   Refer to the example at the end of this section for an example of how to modify a UNIX makefile. Different programmers or organizations have their own personal preferences and different platforms will require slightly different syntax.

   On some platforms, you may need to run **xrc** manually from the command line, as shown in the following XVT/Win32 **xrc** compile statement:

```
xrc -r rcwin -i..\..\include -dLANG_GER_W52
      -dLIBDIR=.\..\..\lib app.xrc
```

   Although the command line shown above is printed on two lines, you should enter a command line as a single line.

   You now have a resource file—if you view it, you will see, in this case, that all strings are now in German.

9. If your makefile did not completely finish the build, you should now complete any unfinished steps in your build process.

Example: This example shows a UNIX makefile that builds a German version of an XVT application:

```
# Define localized options.
# Start a German build.

LOCALIZE_OPTS        = -dLANG_GER_W52
CC_OPTS              = -c $(INC_PATH)
XRC                      = $(XVT_DSC_DIR)/bin/xrc
...

#
# Include the defines in all source code compilations
.c.o
        $(CC) $(CC_OPTS) $(LOCALIZE_OPTS) $<
# Also pass them to xrc
app.uil: app.xrc
                $(XRC) $(XRC_OPTS) $ (LOCALIZE_OPTS) app.xrc
...
```

This marks the end of XVT-Design-specific information about localization.

See Also: For more information about specifying resources with XRC, see Chapter 5, *Resources and XRC*.
For more information about using **xrc**, including a list of **xrc** options, see the *XVT Portability Toolkit Reference*.

19.1.1.3. Terminology

This section introduces the terminology that is used later to describe the details of internationalization and localization.

category

A category is one of several components that, when taken together, describe a locale. The most common categories are listed below:

capitalization and contextual characters

The use of a capitalized character or other forms of a character may vary depending upon its position in a word or message.

character code set (or codeset)

The set of numerical codes that represent encoded characters.

collation algorithm

The scheme by which a list of characters is properly ordered or searched for a given language and culture.

color
The meanings of colors vary between cultures.

icons
Graphic symbols with particular cultural connotations.

language
The characters, words, combinations of words, and punctuation conventions particular to a country, region, culture, or other speech community.

layout
The arrangement of text and GUI objects based on language and cultural norms.

keyboard modifiers, accelerators and mnemonics
The Shift key, Control key, Option key, Alt key, accelerators, and mnemonics have different contextual meanings in different cultures.

monetary, postal address, telephone number, weight, and measure notations
The representation of common cultural information.

numeric representation
The symbols used for numeral notation such as decimal values (‘,’ versus ‘.’) or accounting debit indication (‘$-23.45’ versus ‘($23.45)’).

time/date representation
The order and format of times and dates.

code page
A code page is a platform-specific term for the object that encapsulates character codeset information in the Win32 environment.

character code
A character code is a numeric value representing a character of a particular language.

character code set
A character code set (also known as a *codeset* in other literature) is the set of character codes determined by a particular encoding scheme representing characters in one or more languages. Some commonly used character codesets include:

ASCII
American Standard Code for Information Interchange, the U.S. version of the single-byte (7-bit) encoding scheme for the ISO 646 standard character codeset.

EBCDIC
: Extended Binary-Coded Decimal Interchange Code, the single-byte encoding scheme once used extensively by IBM, particularly in System/370.

Extended ASCII
: The 8-bit version of the single-byte encoding scheme for the ISO 8859 languages.

EUC
: Extended UNIX Code, a multibyte encoding scheme with single-shift encoding used by most UNIX systems.

JIS
: Japanese Industrial Standard, a multibyte encoding scheme for Japanese using shift-sequences.

Shift-JIS
: A multibyte encoding scheme for Japanese using single-shift encoding.

Unicode
: Universal character codeset, a wide character encoding scheme (two bytes per character) that includes all language characters in one character codeset, used by Win32.

Each operating or windowing system may have its own implementation of these character sets.

encoding scheme
: An encoding scheme defines a set of parsing rules for encoding a byte stream representation of a set of characters. Several types of encoding schemes are single-byte (one byte per character), multibyte (variably one or more bytes per character) or wide character (more than one fixed number of bytes per character).

ideograph
: An ideograph is a character that does not represent pronunciation alone, but also encapsulates a word's meaning. Japanese Kanji characters are ideographs.

internationalization (I18N)
: Internationalization (sometimes referred to as I18N, 18 being the number of letters between the first I and the last N) is the process of adapting application code so it can be easily localized. Ideally, after I18N, the localization of an application can be performed without changing or recompiling source code.

input method editor (IME)
: An input method editor is a native window system-specific

mechanism that allows a user to enter multibyte or wide characters from a keyboard that does not support these particular characters. These characters are then dispatched to the application through conventional character events. For example, on some systems, the IME translates English keyboard characters into Kanji and delivers Kanji character events to the application. Other IMEs access a dictionary that allow phonetic spellings to be transposed into their corresponding ideographs (actually its character codeset representation), as in Katakana to Kanji.

invariant character codeset

The invariant character codeset is a subset of characters common to most standard character codesets according to *ISO 646*. These character codes and glyphs remain the same across compliant character codesets. This subset includes:

<space> ! " % & ' ( ) * + , - . / : ; < = > ? _

0 1 2 3 4 5 6 7 8 9

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Note that the following characters in the ASCII codeset are *not* invariant—that is, their character code values may map to different glyphs in other character codesets:

$ @ [\ ] ^ ` { | } ~

International Standards Organization (ISO)

The International Standards Organization has as one of its charters the development of computer standards. One of its documents, *ISO 646*, defines standard character codesets.

Japanese characters

Japanese text may be comprised of characters from four different character systems:

Kanji
: Ideographic characters representing thousands of words.

Kana (Hiragana and Katakana)
: Characters which represent about 80 phonetic sounds, depicted as pictures.

Roman
: Glyphs that represent letters, digits, and punctuation.

locale

A locale is a set of conventions based on some language, culture, or nationality.

localization

Localization is the process of adapting a computer application for a specific language, country, and culture.

multibyte character codeset (MBCS)

Each character in a multibyte character codeset (multibyte character encoding scheme) is encoded variably as one or more bytes per character. Multibyte encoding is accomplished by one of two methods:

The *shift-sequence* method, often called *stateful encoding* or *locking shift*, uses escape codes (a shift-sequence) within a string to switch between one- and two-byte character modes, or between different character codesets. When an encoding shift is encountered in a byte stream, all subsequent characters are interpreted in the new encoding until another shift-sequence occurs.

In the *single-shift* method, character code length is specified in the initial byte of a character. The range of the value of the first byte indicates whether the individual character code consists of one, two or three bytes.

Multibyte encoding schemes must be used to represent languages such as Japanese, Chinese, or Korean due to the large number of characters in these languages. Note that the Shift-JIS character codeset for Japanese uses the single-shift method, hence the name.

single-byte character codeset (SBCS)

A single-byte character requires one and only one byte for its encoding. Single-byte character codesets (single-byte encoding scheme) are used to represent English and most European languages. English is most often encoded by one of two single-byte encoding schemes in either the ASCII or the EBCDIC character codesets.

Although single-byte encoding may be considered a subset of either multibyte or wide character encoding, it is distinguished separately in this book for clarity.

wide character (wchar_t)

A wide character is an encoding scheme in which all characters are of a specific fixed byte length (generally greater than 1 byte per character). Each character has a unique encoding. In the C programming language, wide characters are defined as type wchar_t (two bytes per character). Unicode is another character codeset based on the wide character encoding scheme.

19.1.2. Multibyte Awareness in XVT Applications

This section provides an overview of the XVT Portability Toolkit (PTK) elements that support internationalization and localization and how you use these elements to adapt your XVT applications.

19.1.2.1. Support for Character Codesets

The XVT Portability Toolkit is internationalized to the extent that all of its API functions support both single-byte and multibyte characters (depending upon the value of ATTR_MULTIBYTE_AWARE). Applications can be localized to some languages without switching to a multibyte (MBCS) character codeset. However, localizing to other languages, such as Japanese, requires two levels of planning:

- Placing your application resources in separate, external files so that your application can be easily localized in the future.
- Adapting your application so it can handle multibyte (wide character) character codesets.

The XVT PTK uses only one character codeset at a time. The Portability Toolkit fully supports the ASCII or Extended ASCII character codesets. It allows the manipulation of characters in single-shift MBCS, such as the Extended Unix Code (EUC) or Shift-JIS character codesets on the platforms where they are implemented. Although the supported MBCS vary by platform, you can write portable XVT code that is independent of a specific MBCS.

The PTK also provides support for wide characters. Wide characters permit rapid character pointer arithmetic compared with multibyte strings. Functions for translating wide characters to multibyte and vice-versa enable you to optimize the performance of your string processing.

The PTK does not directly support shift-sequence MBCS encodings, such as JIS (Japanese Industrial Standard), nor does it support Unicode (other than as *wide characters* on MS-Windows Win32) or MIT Compound Strings. You may use these character codesets but you must convert the characters to single-shift multibyte for use in XVT functions. Right-to-left or vertical (top-to-bottom) languages are also not supported.

19.1.2.2. Localized PTK Resources

For your convenience, XVT provides compatible localizations of standard PTK resources and help text; the various codesets used to provide these resources are listed in Table 19.1.

Language:	XVT/Win32:	XVT/Mac:	XVT/XM:
US English	**ANSI** (Windows 1252)	**Mac-Roman**	**ASCII** (ISO 646)
French German Italian	**ANSI** (Windows 1252)	**Mac-Roman**	**ISO-Latin-1** (ISO 8859-1)
Japanese	**Shift-JIS** (Codeset 932)	**Shift-JIS** (Mac-Japanese)	**Shift-JIS, AJEC** (Japanese EUC)

Table 19.1. Localized versions of standard PTK resources and help text predefined for five languages

See Also: Filenames and filenaming conventions for the files listed in Table 19.1 are discussed in section 19.2.1 on page 19-18.

Note: You are not limited to these localizations, but you may want to use these as a basis for localizing your own applications.

The XVT PTK data is externalized in one of three file types for localization by your application:

- Standard XVT resource strings (XRC)
- Standard XVT help strings
- Standard XVT error messages

For convenience, a set of XVT constants is provided to allow the standard XVT resource strings and help text files to be easily included in your applications; these constants are listed in Table 19.2 on page 19-19.

19.1.2.3. Changes to PTK Release 4.0 Functions that Accept Strings

XVT PTK functions that accept strings as arguments also accept multibyte strings for applications executing in multibyte-aware mode. However, for functions that require character index or buffer size parameters, the meanings of these parameters may have changed. In multibyte-aware mode, the following Release 4.0 data types and functions have buffer sizes expressed in number of bytes (versus characters):

FILE_SPEC
SCROLL_CALLBACK
xvt_cb_get_data
xvt_cb_put_data
xvt_dm_post_string_prompt
xvt_dwin_get_font_family
xvt_dwin_get_font_family_mapped
xvt_dwin_get_font_native_desc
xvt_errmsg_get_text
xvt_font_get_family
xvt_font_get_family_mapped
xvt_font_get_native_desc
xvt_font_serialize
xvt_fsys_convert_str_to_dir
xvt_list_get_elt
xvt_list_get_first_sel
xvt_res_get_str
xvt_str_find_eol
xvt_tx_get_line
xvt_vobj_get_title

For multibyte applications, you should increase the buffer sizes appropriately when using these functions.

Regardless of whether an application is operating in multibyte-aware mode, the following Release 4.0 functions have buffer sizes or character indices expressed in number of characters (versus bytes):

xvt_ctl_get_text_sel
xvt_ctl_set_text_sel
xvt_dwin_draw_text
xvt_dwin_get_text_width
xvt_tx_get_num_chars
xvt_tx_get_sel

19.1.2.4. Input Method Editors

Input Method Editors (IMEs) are provided by native windowing systems to allow users to enter multibyte or wide characters through a keyboard that does not support these characters. On each system, certain attributes must be set to allow the user to create composed multibyte characters. These composed characters are defined by specific keyboard values that are typed while an IME is invoked. The method for invoking IMEs is platform-specific.

Generally, the use of IMEs does not result in any requirement for special action by applications. Character events are generated with the appropriate character codes for the composed characters.

See Also: Refer to the *XVT Platform-Specific Books* for more information on invoking and using Input Method Editors.

19.2. How the XVT API Supports Internationalization

The XVT Portability Toolkit is internationalized to the extent that all of its API functions support both single-byte and multibyte characters (depending upon the value of ATTR_MULTIBYTE_AWARE). Furthermore, all internal PTK strings and resources have been externalized so they can be easily localized in your applications.

Several different aspects of the XVT PTK API provide support for internationalization and localization. These API elements include:

- Filenaming conventions
- Portable attributes
- Data types
- Constants
- String functions
- E_CHAR (character) event
- Resource file binding

How each of these API elements supports internationalization and localization is discussed in the sections that follow.

19.2.1. PTK Filenaming Conventions

XVT's PTK uses a set of *conventions* for defining relevant constants and filenames using three character abbreviations for language and three or four character abbreviations for character codeset (see Appendix A for a complete list of these abbreviations):

- Language constant

  LANG_<3 character language>_<3-4 character codeset>

  Default: U.S. English ASCII does not require a language constant

- XRC standard resource strings file

  u<3 character language><3-4 character codeset>**.h**

  Default: **uengasc.h** (U.S. English ASCII)

- XVT standard help text file (included by **xvt_help.csh** to provide help topic text on reserved help topic symbols)

  h<3 character language><3-4 character codeset>**.csh**

  Default: **hengasc.csh** (U.S. English ASCII)

- XVT error code strings file:

  e<3 character language><3-4 character codeset>**.txt**

  Default: **ERRCODES.TXT** (US. English ASCII, the default filename does not adhere to this convention)

Table 19.2 lists the language and character codeset constants and filenames recognized in the XVT Portability Toolkit. XVT resource and help compilers recognize these constants for automatic inclusion of appropriate filenames:

Language:	Compile Constant:	XRC Strings Filename:	Help Text Filename:	Error Messages Filename:
XVT/XM:				
U.S. English	(Default)	**uengasc.h Ð**	**hengasc.csh Ð**	**ERRCODES.TXT Ð**
French	LANG_FRE_IS1	**ufreis1.h Ð**	**hfreis1.csh Ð**	**efreis1.txt**
German	LANG_GER_IS1	**ugeris1.h Ð**	**hgeris1.csh Ð**	**egeris1.txt**
Italian	LANG_ITA_IS1	**uitais1.h Ð**	**hitais1.csh Ð**	**eitais1.txt**
Japanese (SJIS)	LANG_JPN_SJIS	**ujpnsjis.h Ð**	**hjpnsjis.csh Ð**	**ejpnsjis.txt**
Japanese (EUC)	LANG_JPN_UJA	**ujpnuja.h Ð**	**hjpnuja.csh Ð**	**ejpnuja.txt**
Norwegian	LANG_NOR_IS1	**unoris1.h**	**hnoris1.csh**	**enoris1.txt**
Russian	LANG_RUS_IS1	**urusis1.h**	**hrusis1.csh**	**erusis1.txt**
Spanish	LANG_SPA_IS1	**uspais1.h**	**hspais1.csh**	**espais1.txt**
Swedish	LANG_SWE_IS1	**usweis1.h**	**hsweis1.csh**	**esweis1.txt**
XVT/Win32:				
U.S. English	(Default)	**uengasc.h Ð**	**hengasc.csh Ð**	**ERRCODES.TXT Ð**
French	LANG_FRE_W52	**ufrew52.h Ð**	**hfrew52.csh Ð**	**efrew52.txt**
German	LANG_GER_W52	**ugerw52.h Ð**	**hgerw52.csh Ð**	**egerw52.txt**
Italian	LANG_ITA_W52	**uitaw52.h Ð**	**hitaw52.csh Ð**	**eitaw52.txt**
Japanese	LANG_JPN_SJIS	**ujpnsjis.h Ð**	**hjpnsjis.csh Ð**	**ejpnsjis.txt**
Norwegian	LANG_NOR_W52	**unorw52.h**	**hnorw52.csh**	**enorw52.txt**
Russian	LANG_RUS_W51	**urusw51.h**	**hrusw51.csh**	**erusw51.txt**
Spanish	LANG_SPA_W52	**uspaw52.h**	**hspaw52.csh**	**espaw52.txt**
Swedish	LANG_SWE_W52	**uswew52.h**	**hswew52.csh**	**eswew52.txt**

Note: XVT provides only those localized files denoted by **Ð**.

Table 19.2. Language and character codeset constants and filenames recognized in the XVT Portability Toolkit (part 1 of 2)

Language:	Compile Constant:	XRC Strings Filename:	Help Text Filename:	Error Messages Filename:
XVT/Mac:				
U.S. English	(Default)	**uengasc.h Ð**	**hengasc.csh Ð**	**ERRCODES.TXT Ð**
French	LANG_FRE_MRMN	**ufremrmn.h Ð**	**hfremrmn.csh Ð**	**efremrmn.txt**
German	LANG_GER_MRMN	**ugermrmn.h Ð**	**hgermrmn.csh Ð**	**egermrmn.txt**
Italian	LANG_ITA_MRMN	**uitamrmn.h Ð**	**hitamrmn.csh Ð**	**eitamrmn.txt**
Japanese	LANG_JPN_SJIS	**ujpnsjis.h Ð**	**hjpnsjis.csh Ð**	**ejpnsjis.txt**
Norwegian	LANG_NOR_MRMN	**unormrmn.h**	**hnormrmn.csh**	**enormrmn.txt**
Russian	LANG_RUS_MCYR	**urusmcyr.h**	**hrusmcyr.csh**	**erusmcyr.txt**
Spanish	LANG_SPA_MRMN	**uspamrmn.h**	**hspamrmn.csh**	**espamrmn.txt**
Swedish	LANG_SWE_MRMN	**uswemrmn.h**	**hswemrmn.csh**	**eswemrmn.txt**

Note: XVT provides only those localized files denoted by **Ð**.

Table 19.2. Language and character codeset constants and filenames recognized in the XVT Portability Toolkit (part 2 of 2)

The file **xrc.h** has a conditional compile statement for the compile constants defined in the preceding table (such as LANG_JPN_SJIS) that will include the appropriate resource strings file (for example, **ujpnsjis.h**). This constant can be defined on the **xrc** compile line or in your XRC file. The file **xvt_help.csh** has a conditional compile statement which will include the appropriate help strings file (like **hjpnsjis.csh**). The constant can be defined on the **helpc** compile line or in the help file. The file **xvt_help.csh** should be included in your application help source (**.csh**) file if you intend to use the XVT default help topics.

XVT supplies only the localized resources and help text noted on the previous pages (U.S. English, French, German, Italian and Japanese). Use these localizations as a basis for adapting your own application locales. You may also want to add your own language constants.

See Also: For more information on using localized resources with your XVT applications, refer to sections 19.2.7 on page 19-32 and 19.4.7 on page 19-55 and also to the *XVT Platform-Specific Book* for your particular platform.
For a complete list of XVT language and character codeset abbreviations, refer to Appendix A.

19.2.2. XVT Portable Attributes

This section contains information that amends section 2.4.

Several XVT attributes enable internationalization and localization of your applications:

ATTR_APPL_NAME_RID
: The resource ID of a multibyte string, set prior to calling xvt_app_create, to override the XVT_CONFIG application initialization structure appl_name string (document window name prefix).

ATTR_COLLATE_HOOK
: A hook function used for the collating sequence in calls to xvt_str_collate, xvt_str_collate_ignoring_case, and xvt_slist_add_sorted.

ATTR_ERRMSG_FILENAME
: The name of a file used to select the desired localized XVT error system messages.

ATTR_MULTIBYTE_AWARE
: The flag, set prior to calling xvt_app_create, which informs the PTK whether the application may be using a multibyte character codeset. Even if this attribute is set (TRUE) for multibyte aware, single-byte character codesets can be processed. This attribute is also responsible for selecting the appropriate version of an application key hook function, the proper use of E_CHAR fields, and the optimization of string processing for the possible character codeset.

ATTR_RESOURCE_FILENAME
: The name of a file used to select the desired localized resource file, set prior to calling xvt_app_create, to override the XVT_CONFIG application initialization structure base_appl_name string (application base name).

ATTR_TASKWIN_TITLE_RID
: The resource ID of a multibyte string, set prior to calling xvt_app_create, to override the XVT_CONFIG application initialization structure taskwin_title string (task window title).

Note: The XVT help system already provides a mechanism for your applications to specify the directory and filename of a help file.

See Also: For more information about system attributes, refer to section 2.4 in Chapter 2, *About the XVT API*.

19.2.3. XVT Data Types

This section discusses the data types you will need to use in developing internationalized applications.

19.2.3.1. Characters and Strings

Single- and multibyte characters and strings are defined by an ANSI type in the following manners:

- Type char is used for single-byte (ASCII and Extended ASCII) and multibyte characters
- char arrays for single- and multibyte strings
- char* for pointers to single- or multibyte strings.

char values are used by all XVT PTK functions that require characters or strings, regardless of whether your application is multibyte or single-byte. Because each character may be a different size, char* pointers should not be used for pointer arithmetic, particularly if multibyte characters sets are to be supported.

XVT wide characters and strings are encapsulated by an XVT type in the following manner:

- Type XVT_WCHAR is used for wide characters
- XVT_WCHAR arrays for wide character strings
- XVT_WCHAR* pointers to wide character strings

XVT_WCHAR is generally equivalent to the ANSI wchar_t type, which is not supported by all compilers. XVT_WCHAR should be used in place of wchar_t in XVT applications.

Wide characters must be converted, as appropriate, to single-byte or multibyte characters before use in some XVT PTK functions. XVT provides functions for converting from wide characters to multibyte characters (and reverse). In converting *individual* wide characters to single-byte, a simple cast is sufficient unless the high byte of the wide character is significant (as is the case for the virtual key characters delivered in an E_CHAR event). XVT_WCHAR wide characters are delivered by the XVT PTK in E_CHAR events. In single-byte mode, the high byte of this character indicates whether or not the character represents a virtual key.

There are two options for processing individual characters of a string:

- Use XVT_WCHAR* pointers for performing pointer arithmetic.
- Use XVT convenience functions for incrementing and decrementing a character pointer in a multibyte string.

The first option may be faster for processing long strings because XVT_WCHAR characters are a fixed length and are easier to process.

19.2.3.2. Byte Streams

XVT byte streams are defined by pointers to types XVT_BYTE and XVT_UBYTE. These types should be used in place of char or unsigned char, respectively, when the data is to be used for raw data processing. These data types should not be used for defining characters or strings. A pointer to XVT_BYTE is equivalent to the XVT type DATA_PTR.

See Also: For more information about XVT data types, see the "Data Types" portion of the *XVT Portability Toolkit Reference*.

19.2.4. XVT Constants

This section discusses the constants you will need to use in developing internationalized applications.

19.2.4.1. Filename Sizes

The XVT constant SZ_LEAFNAME defines the maximum *byte* length of a single token in a file pathname (a directory name or a filename). This maximum value takes into account any file extensions (including the periods '.') but excludes pathname delimiters ('/', '\' or ':'). The value of this constant is platform-specific and may depend on the character codeset used.

On the other hand, the XVT constant SZ_FNAME defines the maximum *byte* length of an entire pathname including file extensions and delimiters.

See Also: For more information on SZ_LEAFNAME and pathname functions, see section 17.1.2 in Chapter 17, *Files*.

19.2.4.2. Character Sizes

The XVT constant XVT_MAX_MB_SIZE defines the maximum byte size of the largest multibyte character on a specific platform. Use this constant to allocate memory for multibyte character arrays.

Example: The following code shows how to convert a wide character string to the equivalent multibyte string using XVT_MAX_MB_SIZE to size the multibyte string array:

```
int size;
char mbs[XVT_MAX_MB_SIZE * 100];
XVT_WCHAR wcs[100];
...
size = xvt_str_convert_wcs_to_mbs(mbs, wcs);
mbs[size] = (char)0;
```

See Also: For more information about XVT constants, see the "Constants" portion of the *XVT Portability Toolkit Reference*.

19.2.5. XVT String Functions

Some compilers do not support the processing of multibyte or wide characters in their ANSI C Libraries. XVT supplies functions which encapsulate this functionality on platforms that do and implements it on platforms that do not. These functions should be used in place of the ANSI functions wherever appropriate in XVT applications.

Note: If a particular XVT string function is not appropriate for a multibyte character codeset, then it operates without modifying the contents of the passed string. For example, xvt_str_convert_to_upper is not appropriate for Japanese, so it simply returns the passed string unconverted.

19.2.5.1. Character Set Conversions

The following XVT functions support the conversion of characters or strings between wide characters and multibyte or single-byte characters:

xvt_str_convert_mb_to_wc
: Converts the first character of a single-byte or multibyte string to a wide character.

xvt_str_convert_mbs_to_wcs
: Converts a single-byte or multibyte string to a wide character string.

xvt_str_convert_wc_to_mb
: Converts a wide character to a single-byte or multibyte character.

xvt_str_convert_wcs_to_mbs
: Converts a wide character string to a single-byte or multibyte string.

19.2.5.2. String Processing

The following convenience functions are the only wide character processing functions provided by XVT:

xvt_str_convert_wchar_to_lower
: Converts a wide character to a lowercase wide character.

xvt_str_convert_wchar_to_upper
: Converts a wide character to an uppercase wide character.

The following functions encapsulate ANSI C string operations and may be used for single-byte or multibyte character codesets (remember, you cannot mix single-byte and multibyte codesets within a single XVT application):

xvt_str_collate
: Collates two strings (collation algorithm depends on the ATTR_COLLATE_HOOK function).

xvt_str_collate_ignoring_case
: Same as xvt_str_collate except case is ignored (collation algorithm depends on the ATTR_COLLATE_HOOK function with uppercase and lowercase treated the same).

xvt_str_compare
: Compares two strings.

xvt_str_compare_ignoring_case
: Same as xvt_str_compare except case is ignored.

xvt_str_compare_n_char
: Compares n *characters* of two strings.

xvt_str_concat
: Appends a copy of one string to the end of another string.

xvt_str_concat_n_char
: Appends a copy of n *characters* from one string to the end of another string.

xvt_str_convert_to_lower
: Converts the first n *bytes* in one string to lowercase and copies them to another (or the same) string.

xvt_str_convert_to_upper
: Converts the first n *bytes* in one string to uppercase and copies them to another (or the same) string.

xvt_str_copy
: Copies one string into another string.

xvt_str_copy_n_char
: Copies n *characters* from one string into another string.

xvt_str_copy_n_size
: Copies n *bytes* from one string into another string.

xvt_str_duplicate
: Makes a copy of a string.

xvt_str_find_char_set
: Searches one string for the first occurrence of a character that is also in a second string.

xvt_str_find_eol
: Searches the first n *bytes* of a string for an end-of-line character (newline, carriage return, or NULL).

xvt_str_find_first_char
: Searches a string for the first occurrence of a specified character.

xvt_str_find_last_char
: Searches a string for the last occurrence of a specified character.

xvt_str_find_not_char_set
: Searches a string for the first occurrence of a character that is *not* in a second string.

xvt_str_find_substring
: Searches a string for the first occurrence of a second string.

xvt_str_find_token
: Finds string tokens.

xvt_str_get_byte_count
: Counts the number of *bytes* in a string.

xvt_str_get_char_count
: Counts the number of *characters* in a string.

xvt_str_get_char_size
: Counts the number of *bytes* in a character.

xvt_str_get_n_char_count
: Counts the number of complete *characters* in the first n *bytes* of a string.

xvt_str_get_n_char_size
: Counts the number of *bytes* in the first n *characters* of a string.

xvt_str_get_next_char
: Increments a char pointer to the next *character* in a string.

xvt_str_get_prev_char
: Decrements a char pointer to the previous *character* in a string.

xvt_str_is_alnum
: Determines if the first *character* of a string is in the invariant alphanumeric character codeset (a-z, A-Z, 0-9).

xvt_str_is_alpha
: Determines if the first *character* of a string is in the invariant alphabetic character codeset (a-z, A-Z).

xvt_str_is_digit
: Determines if the first *character* of a string is in the invariant decimal character codeset (0-9).

xvt_str_is_equal
: Determines if two strings are equal (*character* for *character*) and of the same length.

xvt_str_is_invariant
: Determines if the first *character* in a string is in the *ISO 646* invariant character codeset. See section 19.1.1.3 on page 19-9 for a list of invariant characters.

xvt_str_is_lower
: Determines if the first *character* in a string is in the invariant lowercase alphabetic character codeset (a-z).

xvt_str_is_space
: Determines if the first *character* in a string is a standard whitespace character: space (‘ ’), form feed (‘\f’), new-line (‘\n’), carriage return (‘\r’), horizontal tab (‘\t’), or vertical tab (‘\v’).

xvt_str_is_upper
: Determines if the first *character* in a string is in the invariant uppercase alphabetic character codeset (A-Z).

xvt_str_is_xdigit
: Determines if the first *character* in a string is a hexadecimal digit (0-9, A-F, a-f).

xvt_str_match
: Performs simple pattern matching.

xvt_str_parse_double
: Converts a string to a double-precision floating point value. This functions skips over any whitespace characters at the beginning of the string and stops converting when it reaches a character that can’t be part of a number (including characters).

xvt_str_parse_long
: Converts a string to a long integer value in the indicated numeric base. This function skips over any whitespace characters at the beginning of the string and stops converting when it reaches a character that can’t be part of a number (including characters).

xvt_str_parse_ulong
: Converts a string to an unsigned long integer value in the indicated numeric base. This function skips over any whitespace

characters at the beginning of the string and stops converting when it reaches a character that can't be part of a number (including characters).

xvt_str_sprintf
xvt_str_vsprintf

These functions process formats according to their equivalents in the ANSI C specification, and also allow you to specify argument order. See section 19.3.4 on page 19-43 for an example of xvt_str_sprintf.

See Also: For detailed information about how multibyte strings are processed by XVT, see section 19.3.3 on page 19-39.

19.2.6. E_CHAR Events

E_CHAR Event Structure

```
...
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
                                       m    odifiers */
} chr;
...
```

XVT sends an E_CHAR event to the event handler for a WINDOW when the user types a character or virtual key code into a window. The E_CHAR event is delivered only to the event handler of the window which has the keyboard focus, and for which a control has not absorbed the character event for its own use.

19.2.6.1. Processing Characters

The EVENT substructure chr sent to a window contains the character code field (ch) which is an XVT_WCHAR. XVT_WCHAR is an encapsulation of the ANSI wchar_t type, although this implementation may vary depending on the amount of support supplied by native ANSI C libraries. Applications should not make assumptions about the size of the ch field.

Multibyte-aware applications must call the XVT function xvt_str_convert_wc_to_mb before assigning a wide character to a multibyte string array or processing the character with other XVT functions. In a switch statement test of a wide character, a

multibyte application must also compare the ch character to a wide character constant.

It is recommended, though not required, that single-byte applications also call xvt_str_convert_wc_mb. However, single-byte applications can always cast XVT_WCHAR characters to char as long as the character is not a virtual key (and does not rely on the virtual key—high byte—portion of the XVT_WCHAR). In multibyte applications, this method does not work because the high byte portion is necessary for representing normal character keys.

19.2.6.2. Virtual Keys

XVT virtual key values are the K_* values (F1, Home key, etc.) defined in the **xvt_defs.h** header file. Virtual keys in character events may be detected in a variety of ways.

For the ASCII character codeset only, values of the ch field greater than UCHAR_MAX indicate a virtual key (except for K_DEL which is less than UCHAR_MAX).

The virtual_key member of the chr substructure is also set to TRUE to distinguish virtual key characters. In multibyte applications, virtual key codes may conflict with some multibyte character encodings. Therefore, the virtual_key field must be validated for multibyte applications.

Alternately, the most general means for testing for a virtual key (regardless of character codeset) is to pass the EVENT structure to the xvt_event_is_virtual_key utility function which determines if the character in a E_CHAR event is a virtual key.

See Also: For more information on how virtual keys are processed, refer to section 4.5.1 in Chapter 4, *Events*.

19.2.6.3. Key Hook Attribute

You can change the mapping of raw key codes (as generated by the keyboard) to XVT virtual key codes, or add new codes, by changing the default key hook function. This is done with the function xvt_vobj_set_attr and the attribute ATTR_KEY_HOOK.

The parameters passed to a key hook function vary depending upon whether your XVT application is capable of processing multibyte characters (ATTR_MULTIBYTE_AWARE is set to TRUE). Parameters also vary between platforms. In single-byte mode, hook functions receive only platform-specific data. In multibyte-aware mode, though, key hook functions on all platforms receive a pointer to the

EVENT structure (E_CHAR event) in addition to platform-specific information. This is necessary because only the hook function knows if it is mapping a passed character to a virtual key in a multibyte-aware environment and can set the virtual_key member properly. Note that the interface for multibyte hook functions is called only if ATTR_MULTIBYTE_AWARE is set to TRUE, otherwise the single-byte (default) interface is used.

See Also: For more detailed information on the E_CHAR event, refer to section 4.5.1 in Chapter 4, *Events*.

19.2.7. Resource File Binding

This section contains information that supplements Chapter 5, *Resources and XRC*.

In addition to allowing external XVT resources (see section 5.2 on page 5-3), the XVT Portability Toolkit allows resources to be bound to the application at application startup time. Executing any localized application will, of course, require that you correctly install the appropriate operating system, windowing system, and fonts for the target locale.

Versions of your XRC files may be adapted to target locales and languages. Multiple XRC files are compiled into separate binary resource files using **xrc** and any appropriate native resource compilers. The application may statically bind a base locale resource to the application. Then, as the application is invoked, a new locale-specific resource may be selected to override the default. Depending on the platform, the selection may be made as early as before the application is invoked, or as late as just before the call to xvt_app_create. The selection method is platform-specific following native guidelines:

XVT/XM

Prior to calling the xvt_app_create function, XVT/XM applications may select the UID resource file using the XVT attribute ATTR_RESOURCE_FILENAME. If this attribute is not set, XVT/XM uses the UIDPATH environment variable for determining resource location, or otherwise, the basename of the application and the current directory will be used. Follow the guidelines for each UNIX platform for specifying locales (including setting the LANG_* environment variable).

XVT/Mac

Prior to calling the xvt_app_create function, XVT/Mac applications set a path to the resource file through the XVT attribute ATTR_RESOURCE_FILENAME. If no path is specified, the resources in the default application resource fork will be used. The application may retrieve the language or locale information from a file such as a "preferences" file, or can query the Macintosh system to get the default or current script system.

XVT/Win32

Prior to calling the xvt_app_create function, applications can set a path to the resource file through the XVT attribute, ATTR_RESOURCE_FILENAME. The application may retrieve the language or locale information from a file such as an application initialization (**.ini**) file or registry. The locale resource files are bound in separate DLLs.

19.2.7.1. Configuration Attributes

Two attributes, ATTR_APPL_NAME_RID and ATTR_TASKWIN_TITLE_RID, may be used to localize the strings used for appl_name and taskwin_title in the XVT_CONFIG initialization structure. The application cannot load the strings from resources before calling xvt_app_create. Instead, these attributes set the string resource ID to override the XVT_CONFIG values *after* xvt_app_create has loaded the resource file.

Note: The base_appl_name member of XVT_CONFIG does not have a corresponding attribute. This string is used in loading files such as the resource or help file, and thus it cannot be easily localized.

19.3. Internationalizing XVT Applications

This section describes the adaptations required to write an internationalized XVT application:

- Using XVT resources for internationalization
- Extracting string literals
- Processing strings
- Formatting locale-specific strings
- Handling character events
- Extracting graphics and colors
- Loading fonts
- Determining GUI object size and position

These adaptations result in a general XVT application which can be executed in both single-byte and multibyte environments (assuming the proper localizations have been made).

19.3.1. Using the XVT Resource Compiler (XRC)

In order to avoid recompiling your application for each locale you need to support, place locale-sensitive data external to your application source code. For XVT applications, the easiest way to do this is with the XVT Resource Compiler (XRC). XRC resources allow you to externally define the layout, contents, and data for your GUI objects. These resources may be compiled separately from your application and then later bound at application startup time. XVT provides the xvt_res_* functions for dynamically accessing this resource data while the application is executing.

See Also: For more information on XRC, refer to Chapter 5, *Resources and XRC*.
For more on the dynamic binding of resources, refer to sections 19.2.7 on page 19-32 and 19.4.7 on page 19-55.

19.3.2. Extracting String Literals

Any string literals in your application code that will be seen by users should be moved to an external resource file so they can be translated outside of the program. These string literals should be replaced with calls to xvt_res_get_str to obtain the strings from a resource file. Use the XRC string statement to define string literals in resources. XVT recommends placing string literals in resource files (as opposed to #define'd in header files) so they can be easily localized without recompiling your application.

See Also: For more information on the XRC string statement, refer to the *XVT Portability Tookit Reference*.

19.3.2.1. Character Codeset Issues

Although most multibyte character encoding schemes contain the subset of ASCII characters, only those characters included in the *ISO 646* character codeset are guaranteed to be invariant. This occasionally may cause problems for some special characters you need to use (such as control characters or delimiters). Defining these characters in your source code may present problems when porting to other character sets.

Tip: One technique that you can use to avoid difficulties with non-portable special characters is to define these characters in resource strings classified by their purpose or usage. For example, it might be better to associate the character '\' with the identifier ID_STR_DOS_DIR_DELIM (indicating that the character is a file pathname delimiter) instead of just calling it ID_STR_BACKSLASH. It is unlikely that you would ever need to change this character, but if you do, it would be external to your source code.

19.3.2.2. strscan Utility and String Literal Convenience Macros

XVT-Design generates source code and resource files with all locale-specific information defined in a few special macros. The XVT utility program, **strscan***, may be used to scan these files to search for the special macros.*

The **strscan** utility program scans XVT-Design-generated source code and XRC resource files for references to special macros. **strscan** uses the arguments passed to these macros to generate a source code include file that contains resource IDs (**strdef.h**) and a resource include file that contain resources for string literals (**strres.h**)—this file can be easily localized.

LOCAL_C_STR is the most commonly used of the XVT-Design-generated macros. You can also use it in your own application source code. LOCAL_C_STR is defined as follows:

```
#ifdef XVT_LOCALIZABLE
#define LOCAL_C_STR(rid, literal, buf, size) \
        xvt_res_get_str(rid, buf, size)
#else
#define LOCAL_C_STR(rid, literal, buf, size) \
        literal
#endif
```

The expansion of the special XVT-Design-generated macros depends upon whether the XVT_LOCALIZABLE macro is defined. You can define XVT_LOCALIZABLE directly in your application header. Alternatively, you can define it on the compile line of a source code or resource compiler.

Caution: **strscan** does not replace string literals in your application code, but merely scans the code for calls to LOCAL_C_STR and other localization macros to generate header and resource files which contain the string literal values. Because #ifdefs are compile-time pre-processor directives, and **strscan** is an independent executable that runs prior to code compilation, **strscan** cannot properly process your #ifdefs. Do not call LOCAL_C_STR in #ifdef'd code using the same string resource ID with different strings (see example below).

XVT-Design uses the macro window IDs as a basis for creating resource IDs for its generated calls to LOCAL_C_STR. If you are using ***strscan*** *with XVT-Design, and you have defined the macro XVT_LOCALIZABLE, then you should not create any macro window identifiers whose name lengths exceed 26 characters.*

Example: The following code fragments show several lines of non-internationalized application code followed by its adaptation using LOCAL_C_STR and the corresponding header and resource code generated by the **strscan** utility:

Non-internationalized source code:

```
...
#if XVTWS == MACWS
        xvt_menu_set_item_title(win, USER_MENU_EXIT_TAG,
                "Quit");
#else
        xvt_menu_set_item_title(win, USER_MENU_EXIT_TAG,
                "Exit");
#endif
...
```

Internationalized source code:

```
...
#define BUF_SIZE 1024
static char buffer[BUF_SIZE];
...
#if XVTWS == MACWS
        xvt_menu_set_item_title(win, USER_MENU_EXIT_TAG,
                LOCAL_C_STR(LS_QUIT, "Quit",
                        buffer, BUF_SIZE));
#else
        xvt_menu_set_item_title(win, USER_MENU_EXIT_TAG,
                LOCAL_C_STR(LS_EXIT, "Exit",
                        buffer, BUF_SIZE));
#endif
...
```

strscan-*generated header file* (**strdef.h**):

```
...
#define LS_QUIT 1006
#define LS_EXIT 1005
...
```

strscan-*generated resource file* (**strres.h**):

```
...
STRING LS_QUIT "Quit"
STRING LS_EXIT "Exit"
...
```

You need to include the **strscan**-generated header file in any module that uses strings, and include the **strscan**-generated XRC and header files in your application's XRC file.

19.3.2.3. Renaming and Changing strscan-generated Files

By default **, strscan** generates two files: 1) **strres.h** (resource include file), and 2) **strdef.h** (source code include file). You may override these default filenames and use names of your own.

Note: **strscan** also allows you to specify the starting range and increment of resource IDs as well as macro replacement values.

19.3.3. Processing Characters and Strings

The three main aspects of character and string processing in internationalized XVT applications are:

- Replacing ANSI string functions with XVT portable string functions
- Handling string pointers
- Adjusting string buffer sizes

19.3.3.1. Replacement ANSI String Functions

The standard C libraries are not multibyte-aware on some platforms supported by XVT. The XVT Portability Toolkit (PTK) provides portable multibyte-aware replacements for most ANSI C library functions that take strings as parameters. XVT string functions are portable and work on all platforms supported.

The following list shows standard ANSI C library functions and their XVT API replacements.

ANSI Function:	XVT Function:
atof	xvt_str_parse_double
atoi	xvt_str_parse_long
atol	xvt_str_parse_long
isalnum	xvt_str_is_alnum
isalpha	xvt_str_is_alpha
isdigit	xvt_str_is_digit
islower	xvt_str_is_lower
isspace	xvt_str_is_space
isupper	xvt_str_is_upper
isxdigi	xvt_str_is_xdigit
mblen	xvt_str_get_char_size
mbtowc	xvt_str_convert_mb_to_wc
mbstowcs	xvt_str_convert_mbs_to_wcs
sprintf	xvt_str_sprintf
strcat	xvt_str_concat
strchr	xvt_str_find_first_char
strcmp	xvt_str_compare
strcmpi	xvt_str_compare_ignoring_case
strcoll	xvt_str_collate
strcpy	xvt_str_copy
strcspn	xvt_str_find_char_set
strlen	xvt_str_get_char_count
strlen	xvt_str_get_byte_count
strncat	xvt_str_concat_n_char
strncmp	xvt_str_compare_n_char
strncpy	xvt_str_copy_n_char
strncpy	xvt_str_copy_n_size
strrchr	xvt_str_find_last_char

strtpbrk	xvt_str_find_char_set
strspn	xvt_str_find_not_char_set
strstr	xvt_str_find_substring
strtod	xvt_str_parse_double
strtol	xvt_str_parse_long
strtoul	xvt_str_parse_ulong
strtok	xvt_str_find_token
tolower	xvt_str_convert_to_lower
toupper	xvt_str_convert_to_upper
vsprintf	xvt_str_vsprintf
wcstombs	xvt_str_convert_wcs_to_mbs
wctomb	xvt_str_convert_wc_to_mb

Note: If you decide to use standard ANSI functions, check your compiler documentation to verify that they are fully ANSI-compliant and multibyte-aware. Some functions, for example, strftime (not included in the list above) may not be internationalized. Some ANSI functions (for example, strcat, strcmp, strcpy, strlen, strstr) work with both single-byte and multibyte character codesets. In general, it is safer to use the XVT-supplied functions although they may be slower than the C library functions because they do more error checking.

See Also: For more information on XVT string processing functions, see section 19.2.5 on page 19-25.

19.3.3.2. Other XVT String Functions

The following are additional XVT-supplied functions that do not have corresponding ANSI C library functions but that you might want to use when coding your internationalized application:

xvt_str_collate_ignoring_case
xvt_str_convert_wchar_to_lower
xvt_str_convert_wchar_to_upper
xvt_str_duplicate
xvt_str_get_next_char
xvt_str_get_prev_char
xvt_str_find_eol
xvt_str_get_n_char_count
xvt_str_get_n_char_size
xvt_str_is_equal
xvt_str_is_invariant
xvt_str_match

See Also: For more information on XVT string processing functions, see section 19.2.5 on page 19-25.

19.3.3.3. Manipulating String Pointers

Since multibyte characters can be one or more bytes long, single-byte character pointer arithmetic is inadequate for multibyte-aware XVT applications. Use the XVT-provided functions xvt_str_get_next_char and xvt_str_get_prev_char for incrementing and decrementing, respectively, character pointers. These functions are appropriate for both single-byte and multibyte-aware applications.

Example: In this example, single-byte pointer arithmetic code is adapted for a multibyte-aware application:

Non-internationalized code:

```
char *s;
...
for (s = buf; *s; s++) {
        if (*s == '\n')
                break;
}
```

However, in a multibyte application, the character pointer s is incremented to the next byte rather than the next character. It would be quite possible for the second byte of a two-byte character to have the same value as '\n'. The non-internationalized code may be rewritten as follows:

Internationalized code:

```
char *s;
size_t len;
...
for (s = buf; *s; s += len) {
        len = xvt_str_get_char_size(s);
        if ((len == 1) && (*s == '\n'))
                break;
}
```

Note: To comply with ANSI C requirements, no multibyte character contains a second or other subsequent byte with a value of zero, so there is no potential problem with '\0' as there is with '\n'.

19.3.3.4. Wide Characters

XVT wide characters (XVT_WCHAR) provide another way to manipulate multibyte character strings. The Portability Toolkit (PTK) provides the following functions for converting single characters or strings between multibyte encodings and wide character encodings:

```
xvt_str_convert_mb_to_wc
xvt_str_convert_wc_to_mb
xvt_str_convert_mbs_to_wcs
xvt_str_convert_wcs_to_mbs
```

You may find arrays of wide characters easier to manipulate than multibyte strings since indexing is more straightforward. If your XVT application needs to manipulate many strings, it may be worthwhile to maintain the strings as wide character arrays and convert to and from multibyte characters only when necessary.

See Also: For more information on XVT string processing functions, see section 19.2.5 on page 19-25.

Example: The code in the previous example could also be written as follows for wide character encoding:

```
char mbcs[MAX_BUF_SIZE];
XVT_WCHAR wcs[MAX_BUF_SIZE];
int i, nchars;
...
nchars = xvt_str_convert_mbs_to_wcs(wcs,
                    mbcs, MAX_BUF_SIZE);
...
for (i = 0; i < nchars; i++) {
        if (wcs[i] == L'\n')
                break;
}
```

This code uses the standard C notation L'\n' to enforce the use of a wide character literal that corresponds to the indicated single-byte character value '\n'. For most compilers, the character literals L'\n' and '\n' are the same numerically, but you cannot assume that this equivalence is supported by all compilers. As a result of this limitation, use the PTK function xvt_str_convert_mb_to_wc to convert a single-byte character to a wide character. Alternatively, you may want to save commonly-used character literals in an external file where they can be easily localized.

19.3.3.5. String Buffer Sizes

When working with multibyte character strings, it is often important to differentiate between the number of bytes and the number of characters in a string or substring. The PTK provides the following functions for counting bytes and characters:

```
xvt_str_get_byte_count
xvt_str_get_char_count
xvt_str_get_char_size
xvt_str_get_n_char_count
xvt_str_get_n_char_size
```

For PTK functions that accept both a string and a numeric value as parameters, you need to know whether the numeric value specifies the number of bytes or characters. This distinction is important if you are adapting existing code for international support.

See Also: To see lists of functions that require information about number of bytes and number of characters, refer to section 19.1.2.3 on page 19-16.

Example: In this example, strlen is replaced with multibyte-aware code.

Non-internationalized code:

```
int nchars, nbytes;
char * string;
...
nbytes = nchars = strlen(string);
```

Internationalized code:

```
int nchars, nbytes;
char * string;
...
nbytes = xvt_str_get_byte_count(string);
nchars = xvt_str_get_char_count(string);
```

19.3.3.6. Filenames and Pathnames

Since file and pathnames may contain multibyte characters, they must be treated like other multibyte strings. All PTK functions and data types that accept file or pathname strings are multibyte capable.

See Also: For more information on processing filenames, see Chapter 17, *Files*.

19.3.4. Formatting Locale-specific Strings

In some cases, moving string literals to a resource file for translation may not provide full string internationalization. For example, the month and date fields of a date/time stamp created via sprintf may need to be swapped for different locales. The XVT Portability Toolkit functions xvt_str_sprintf and xvt_str_vsprintf work similar to ANSI sprintf and vsprintf. These XVT functions provide an additional %<digit>$ format specifier where <digit> indicates a parameter's position in the resulting formatted string.

Example: In this example, an ANSI C style string format specifier is replaced by the corresponding XVT string format specifier:

ANSI style:

```
sprintf(buf, "Date: %d/%d/%d", month, day, year);
```

The month value is substitued for the first %d, the day for the second %d, and the year for the third%d. In many locales, this is not the preferred order and the result will be confusing to the user.

XVT style:

```
int month=5, day=24, year=95;
...
xvt_str_sprintf(buf, “Date: %1$d/%2$d/%3$d”, month,
        day, year);
```

Result: “Date: 5/24/95”

In this case, the result is the same. The month value is substituted for %1$d, the day issubstituted for %2$d, and the year is substituted for %3$d. However, if this format string is moved to the resource file, it can be modified to suit the locale. For example, you could easily change the format specification for German:

```
int month=5, day=24, year=95;
...
xvt_str_sprintf(buf, “Heute: %2$d.%1$d.%3$d”, month,
        day, year);
```

Result: “Heute: 24.5.95”

You will need to extract from your code any strings used to construct dates, time, numbers, or currency values for display to users. Since the formatting of these strings varies with locale(as demonstrated above), you will want them in resource files for easy localization. For example:

```
xvt_str_sprintf(buf,
        LOCAL_C_STR(LS_date,
            “Date: %1$d/%2$d/%3$d”,
            xdStrBuf1, BUFSIZE),
        month, day, year);
```

19.3.5. Handling Character Events

The EVENT substructure chr sent to a window for an E_CHAR character event contains a character code field (ch) defined as an XVT wide character type XVT_WCHAR. Multibyte-capable applications use the XVT function xvt_str_convert_wc_to_mb to convert this wide character toa multibyte characterbefore processing it with other XVT functions. In a switch statement test of a wide character, a multibyte application must also compare the ch character to a wide character constant.

Note: It is recommended, but not required, that single-byte applications also use xvt_str_convert_wc_to_mb (because you may need to support multibyte applications in the future). However, single-byte applications can always cast XVT_WCHAR characters to char as long as the character is not a virtual key (and must rely on the virtual key, high byte, portion of the ch field).

See Also: For more multibyte-oriented information on the E_CHAR character event, refer to section 19.2.6 on page 19-29.

Example: This code demonstrates the processing of a multibyte character delivered in an E_CHAR event:

```
long XVT_CALLCONV1 win_eh(WINDOW win, EVENT *ep)
{
    char mbc[XVT_MAX_MB_SIZE];
    int len, width;
    ...
    switch (ep->type) {
        ...
        case E_CHAR:
            ...
            if (!xvt_event_is_virtual_key(ep)) {
                len = xvt_str_convert_wc_to_mb(mbc,
                        ep->v.chr.ch);
                width = xvt_dwin_get_text_width(win,
                        mbc, 1);
            }
            ...
            break;
        ...
    }
}
```

19.3.6. Extracting Graphics and Colors

You may need to place graphics and color information in an external file when coding internationalized applications.

19.3.6.1. Icon Controls

Objects such as control icons and XVT images can be placed in an external resource file. Use the icon XRC statement to define an external bitmap for an icon control.

See Also: For information on icon controls in XRC, see the icon XRC statement in the *XVT Portability Toolkit Reference* and the *XVT Platform-Specific Books*.

19.3.6.2. Drawn Images

To retrieve a bitmap for drawing into a window, use the image XRC statement to define an external bitmap, and in your source code, use xvt_res_get_image to get the image from resources.

See Also: For more information on portable images defined in XRC, see the image XRC statement in the *XVT Portability Toolkit Reference*.

19.3.6.3. Colors

XVT does not support a color statement in XRC; however, you may use userdata statements to define external color references.

See Also: For more on XRC user data, see the userdata XRC statement in the *XVT Portability Toolkit Reference*.

Example: This example demonstrates how three XVT colors may be stored externally and obtained from a single window resource userdata statement (implemented for a single-byte character codeset):

XRC file:

```
...
window COLOR_WIN_101 XRC_RECT(61,169,339,380) \
        "Color Test" doc size \
        userdata "0x00FF0000 0x0000FF00 0x000000FF"
...
```

These colors now may be modified easily for other locales.

Source code file:

```
...
char* udata, end;
COLOR local_red, local_green, local_blue;
...
udata = xvt_res_get_win_data(COLOR_WIN_101, 0, 0);
local_red = xvt_str_parse_ulong(udata, &end, 16);
local_green = xvt_str_parse_ulong(end, &end, 16);
local_blue = xvt_str_parse_ulong(end, &end, 16);
xvt_mem_free(udata);
...
```

19.3.7. Loading Fonts

You need to make sure that the fonts used by your application are appropriate for the character codeset. For example, an application localized for Japanese must use a Japanese font. XVT provides several mechanisms for mapping fonts. The most straightforward method is to use XRC font mapping. Using XRC font or font_map statements, you can easily define fonts external to your application. Use calls to xvt_res_get_font to load fonts in your application at runtime.

See Also: For details on XVT logical fonts, see section 15.3 in Chapter 15, *Fonts and Text*.
For more on the font and font_map XRC statements, refer to the *XVT Portability Toolkit Reference*.

19.3.8. Generalizing GUI Objects Positions and Sizes

If you are using XRC to define your GUI objects, much of the work involved in externalizing your size and position data (XVT rectangle type RCT and point type PNT) is done. However, for data that is not associated directly with a particular GUI object, which must be used after an object is created, or which is used for drawn objects, you may additionally need to provide algorithms for modifying sizes and positions at runtime.

XVT does not support a specific statement for location or dimension data in XRC (other than window, dialog, and control creation rectangles); however, you may use userdata statements to define external size and position references.

See Also: For more information on rectangles and points, see Chapter 10, *Coordinate Systems*.
For more information on XRC user data, see the userdata XRC statement in the *XVT Portability Toolkit Reference*.

Example: This example demonstrates how an XVT rectangle may be stored externally, obtained from a single window resource userdata statement, and used to set the size of a window (implemented for a single-byte character codeset):

XRC file:

```
...
window POSITION_WIN_101 XRC_RECT(50,50,100,200) \
      "Position Test" doc size \
      userdata "100 200 150 350" /* rectangle data */
...
```

These rectangles now may be modified easily for other locales.

Source code file:

```
...
WINDOW win;
char* udata, end;
RCT local_rct;
...
udata = xvt_res_get_win_data(POSITION_WIN_101, 0, 0);
local_rct.top = xvt_str_parse_ulong(udata, &end, 10);
local_rct.left = xvt_str_parse_ulong(end, &end, 10);
local_rct.bottom = xvt_str_parse_ulong(end, &end, 10);
local_rct.right = xvt_str_parse_ulong(end, &end, 10);
xvt_mem_free(udata);
xvt_vobj_move(win, &local_rct);
...
```

19.4. Localizing XVT Applications

Once you have internationalized your XVT application, localizing it is very straightforward. There are several tasks involved in localizing your XVT application:

- Setting the operating/windowing system environment and the character codesets for the target locale
- Translating literal strings to the locale using appropriate character codesets
- Substituting colors, images, and icons
- Setting positions and sizes for GUI objects
- Initializing the application to the proper locale
- Compiling localized resources and help text

19.4.1. Selecting the Environment

Before adapting your resource and help files, you must select a character codeset that supports the target language. In making this decision, evaluate the language characters that must be represented, the fonts that support these characters, and the relative availability of these character codesets and fonts on your target windowing and operating systems.

Caution: Make sure that you have set the proper configurations of the operating and windowing systems for the locales you need to support.

See Also: Refer to your native platform documentation for more information on system setup for locale.
Different codesets used on the various platforms that XVT supports are listed in section A.2 in Appendix A.

19.4.2. Translating Strings

Any strings that may be displayed to your users are candidates for language translation. These include, but are not limited to, menu item titles, keyboard accelerators and mnemonics, window titles, dialog titles, control titles, text and mnemonics, error messages, and help topics and text. If you have internationalized your XVT application (as described in section 19.1.1.2 on page 19-3 and again in section 19.3 on page 19-34), then this text should exist in one of several types of files which will require translation:

- Application resource files, or

- Application include files (which contain only specific localization data to be included in a resource file),
- Application help source text files, and
- XVT resource files, default help text files, and error message file (for locales not translated and provided by XVT)

Note: Note that menu and control mnemonics and menu accelerators cannot include multibyte characters.

To translate locale-specific text, you must invoke a text editor which supports the character codeset and corresponding fonts that will be used by your XVT application. (Some vendors offer editors which simultaneously support multiple character codesets.)

You will need to maintain a copy of each file for each locale you need to support. Alternatively, you may be able to keep a single copy of each file with `#ifdef`'d resources for each locale. However, this alternative is only possible if your target locales are supported by the same character codeset.

Translating strings directly in XVT-Design only works in target environments where XVT-Design is compatible with the character codeset and can properly display translated strings.

If you have used the **strscan** *utility with XVT-Design, then translating resource strings requires modification to the file* **strres.h**.

19.4.2.1. Setting Special Format Strings

If you have any locale-specific format strings, you must review the parameters for these strings (as described in section 19.3.4 on page 19-43), and adjust the format in your resource files to match the requirements of your target locale.

19.4.2.2. Using Standard XVT Resource, Default Help and Error Message Files

XVT provides localized versions of its standard resource text and help source text for U.S. English, German, French, Italian, and Japanese (see section 19.1.2.2 on page 19-15). These localizations are encapsulated in include files referenced by XVT XRC and help source text files. You may control the inclusion of these files by defining a LANG_* constant on the command line for **xrc** or **helpc**, or by defining the constant in your source files.

XVT recommends using the U.S. English files as the basis for your translation. All of the text characters used in English are in the *ISO 646* invariant character codeset, and your translation will merely require a replacement of the XVT text strings with strings for your target language. Characters outside of the invariant set may not be properly displayed in the editor you use for translation.

If XVT has not provided files for your target locale, follow these steps for localizing standard XVT files:

1. Translate all strings in the file **uengasc.h** (located in the **...include** directory) to your target locale. The standared XRC resources file **uengasc.h** contains data used internally by the PTK. (Remember to adjust any object sizes and positions to reflect changes in size of the translated text—for details, see section 19.4.4 on page 19-51).
2. If you are using the online hypertext help system, translate the standard XVT help text in file **hengasc.csh**.
3. If your users will be exposed to any error messages from the error handling facility, translate the text in file **ERRCODES.TXT** to your target locale.
4. Rename your files according to the naming conventions described in section 19.2.1 on page 19-18.
5. Following the conventions in section 19.2.1, add code to the file **xrc.h** to handle the LANG_* variable for your target locale. For example, add the lines:

```
#elif defined(LANG_DAN_IS1)          /* Danish XVT/XM */
#include "udanis1.h"
#elif defined(LANG_DAN_MRMN)         /* Danish XVT/Mac */
#include "udanmrmn.h"
#elif defined(LANG_DAN_W52)          /* Danish XVT/Win32 */
#include "udanw52.h"
```

before the code that includes the U.S. English default file:

```
#else
#include "uengasc.h"            /* Default English (ASCII) */
#endif
```

See Also: Refer to Appendix A for a complete list of language and character code abbreviations.

19.4.3. Replacing Colors and Graphics

Be sure to replace any locale-sensitive colors and graphics (drawn, bitmaps, icons, or images) with ones appropriate to your target locale.

See Also: For more information on externalizing colors and graphics (to simplify the process you use when localizing your XVT applications), see section 19.3.6 on page 19-45.
To see hundreds of examples of international symbols used in various fields of endeavor, refer to *Symbol Sourcebook: An Authoritative Guide to International Graphic Symbols*, by Henry Dreyfuss, published by Van Nostrand Reinhold, New York, N.Y., 1984.

19.4.4. Adjusting Object Sizes and Positions

One of the most tedious aspects of adapting your internationalized application to a particular locale can be adjusting the sizes and positions of GUI objects and drawn objects. This includes adjusting the creation rectangles for windows, dialogs and controls. Unfortunately, there are no simple solutions for adjusting these creation rectangles in your XRC files. Correcting the sizes and positions probably will require several iterations of compiling the XRC file, executing the application in the required environment to see how graphic objects look in relation to translated strings and to each other, and then editing the XRC file to adjust the sizes.

One alternative is to calculate sizes and position at runtime based on the font size, text width (in pixels, not characters or bytes) and other characteristics of the localized object.

See Also: For more information on extracting position and size data from your source code, see section 19.3.8 on page 19-47.

19.4.5. Using XVT's Utility Programs to Write Localized Applications

XRC files and the help source text files are compiled (via **xrc** and **helpc** respectively) into binary resource files. And, although the methods differ, each of these binary files can be associated with your XVT application at startup time or runtime. This means that a program executable can be the same in all environments and only the binary resource files have to be customized for a target locale.

When running **xrc** or **helpc**, you will need to make sure that the correct header files for your locale are available and that you have defined a LANG_* constant in the source file, or on the command line, to include the localized header file version for your target locale.

Caution: To produce output consistent with the target locale's character codeset, **xrc** and **helpc** should be executed on platforms where the character codeset is properly installed. This is especially necessary when the target platform is using a multibyte character codeset.

19.4.5.1. xrc

Tip: To run the resource compiler and include German default XVT resources for an XVT/Win32 application, use a command line similar to the following:

```
xrc -r rcwin -I..\..\include -DLANG_GER_W52
    -DLIBDIR=.\..\..\lib sample.xrc
```

Using this command line will cause the file **ugerw52.h** (which is the XVT-supplied German translation of the default resource file) to be included in your resources.

Implementation Note: XRC resource (source) files are portable between XVT-supported platforms if the target character codesets are compatible. Specifically, characters used in the resource files must come from the invariant character codeset. U.S. English and Japanese Shift-JIS are generally the only two languages and character codesets for which this is true.

See Also: For a complete list of XRC resource compiler options, refer to the *XVT Portability Toolkit Reference*.
For a list of LANG_* constants supported by XVT, see section 19.2.1 on page 19-18.

19.4.5.2. helpc

Tip: To run the help compiler and include German default help topics for an XVT/Win32 application, use a command line similar to the following:

```
helpc -f win -I..\..\include -DLANG_GER_W52
      sample.csh
```

Using this command line will cause the file **hgerw52.h** (which is the XVT-supplied German translation of the default help topics file) to be included in your help source.

Implementation Note: Help source text files and the binary help files generated by **helpc** are portable between XVT-supported platforms if the target character codesets are compatible. Specifically, characters used in the help source files must come from the invariant character codeset.) U.S. English and Japanese Shift-JIS are generally the only two languages and character codesets for which this is true.

See Also: For a complete list of **helpc** compiler options, refer to the *XVT Portability Toolkit Reference*.
For a definition of the invariant character codeset and other localization terminology, see section 19.1.1.3 on page 19-9.

For a list of LANG_* compile constants supported by XVT, see Table 19.2 on page 19-19.

19.4.6. Localizing the XVT Portable Help Viewer

To use the *standalone* version of the XVT portable help viewer (**helpview**) with a localized XVT application, follow these steps:

1. Localize the standard XVT resource and help text files as described in section 19.4.2 on page 19-48. These standard files are included by **helpview.xrc**, the help viewer resource file.
2. Compile the help viewer resource file **helpview.xrc** in the **...src\helpview** directory. For example:

   ```
   xrc -r win -i..\..\include -dLANG_DAN_W52
       helpview.xrc
   ```

   Following the instructions in the *XVT Platform-Specific Books*, create a native resource file for the help viewer on your platform. Name the resource file according to the convention:

   helpv<3 character language>.<extension>

   A complete list of the <3 character language> abbreviations may be found in Appendix A. The <extension> is platform specific. For example (with Danish):

   XVT/XM: **helpvdan.uid**

   XVT/Mac: **helpvdan.rsrc**

   XVT/Win32: **helpvdan.dll**
3. In the directory which contains your application executable, create a help viewer language configuration file **helpview.lng**. This file contains a single word of text—the name of the language of your target locale. The language name may be any language from the list in Appendix A.
4. When your application invokes **helpview**, the PTK uses the language name defined in **helpview.lng** to determinethe correct resource file to bind with **helpview**.

Caution: If any of the following conditions are not met, the PTK uses the default U.S. English resource file:

- the file **helpview.lng** is present in the directory that contains **helpview**
- the language name is recognized

- the corresponding resource file is present in the directory required for resources by your platform (see *Installing XVT Development Solution for C* for your platform for details)

Note: The bound version of the XVT help viewer and native help viewers do not require these special steps.

19.4.7. Selecting the Environment and Initializing the Application

The startup procedure for a localized XVT application is very similar to a single-locale XVT application. However, in addition to other attributes, your localized application must notify the PTK that it is multibyte aware. The application does this by setting the value of the attribute ATTR_MULTIBYTE_AWARE to TRUE.

Select the resource file to be used by setting the value of the attribute ATTR_RESOURCE_FILENAME. The application name and the task window title are localized by setting the attributes ATTR_APPL_NAME_RID and ATTR_TASKWIN_TITLE_RID.

The application name and task window title in the XVT_CONFIG structure passed to xvt_app_create are overridden by localized strings obtained from resources. ATTR_APPL_NAME_RID and ATTR_TASKWIN_TITLE_RID set the resources IDs from which to obtain these strings.

If you have created a locale-specific error file (**ERRCODES.TXT**), use the ATTR_ERRMSG_FILENAME attribute (as you would the ATTR_RESOURCE_FILENAME attribute), to override the default filename before xvt_app_create is called.

See Also: For more information on binding resource files to your application, see section 19.2.7 on page 19-32.
For more information on locale-specific names for the error file, see section 19.2.1 on page 19-18.

Example: This example code demonstrates how to select locale files and set configuration strings at application startup. In this example, the locale name is read from an external file (for portability with XVT/Mac applications). You also may want to consider allowing your users to set the locale from the command line at application startup (by processing the values of argc and argv).

Source code:

```
#include “sample.h”
#include “xvt.h”

BOOLEAN XVT_CALLCONV1 select_resources(void)
{
        FILE* file;
        char buffer[128];

        file = fopen(“sample.opt”, “r”);

        if (!file)
         return FALSE;

        memset(buffer, 0, 128);

        if (fgets(buffer, 127, file) == NULL) {
         fclose(file);
         return FALSE;
         }

        /* determine locale and set resource file */
        if (xvt_str_find_substring(buffer,
                                “JAPANESE”)) {
         xvt_vobj_set_attr(NULL_WIN,
                ATTR_MULTIBYTE_AWARE, TRUE);

         xvt_vobj_set_attr(NULL_WIN,
                ATTR_RESOURCE_FILENAME,
                (long) JAPANESE_RESOURCE_FILE);
        }
        else if (xvt_str_find_substring(buffer,
                                “USENGLISH”)) {
         xvt_vobj_set_attr(NULL_WIN,
                ATTR_RESOURCE_FILENAME,
                (long) USENGLISH_RESOURCE_FILE);
        }
        else {
         /* no locale found */
         fclose(file);
         return FALSE;
         }

        /* set localized application name */
        xvt_vobj_set_attr(NULL_WIN,
                ATTR_APPL_NAME_RID,
                SAMPLE_APPL_NAME_RID);
```

```
        /* set localized task window name */
        xvt_vobj_set_attr(NULL_WIN,
                ATTR_TASKWIN_TITLE_RID,
                SAMPLE_TASKWIN_TITLE_RID);

        fclose(file);
        return TRUE;

} /* end select_resources */


int XVT_CALLCONV1 main(int argc, char *argv[])
{
        static XVT_CONFIG config;
        ...
        if (select_resources()) {
         /* application name and task window name
                will be obtained from resources */
         config.appl_name = "\0";
         config.taskwin_title = "\0";
         }
         else {
         config.appl_name = "sample";
         config.taskwin_title = "Sample Program";
         }
         config.base_appl_name = "sample";
         config.menu_bar_ID = SAMPLE_MENUBAR;

         xvt_app_create(argc, argv, 0L, task_eh, &config);
         ...
}
```

Application header file (**sample.h**):

```
#include "xvt.h"
...
#if (XVTWS == WIN32WS) /* XVT Win32 platforms */
        #define USENGLISH_RESOURCE_FILE "useng.dll"
        #define JAPANESE_RESOURCE_FILE "japanese.dll"
#elsif (XVTWS == MACWS) /* XVT/Mac */
        #define USENGLISH_RESOURCE_FILE "useng.rsrc"
        #define JAPANESE_RESOURCE_FILE "japanese.rsrc"
#elsif (XVTWS == MTFWS) /* XVT/XM */
        #define USENGLISH_RESOURCE_FILE "useng.uid"
        #define JAPANESE_RESOURCE_FILE "japanese.uid"
#endif
...
#define SAMPLE_APPL_NAME_RID 170
#define SAMPLE_TASKWIN_TITLE_RID 171
```

U.S. English XRC resource file:

```
#include "sample.h"
...
string SAMPLE_APPL_NAME_RID "sample"
string SAMPLE_TASKWIN_TITLE_RID "U.S. English Sample"
```

Japanese XRC resource file:

```
#include “sample.h”
...
string SAMPLE_APPL_NAME_RID “...”
string SAMPLE_TASKWIN_TITLE_RID “...”
```

20

MEMORY ALLOCATION

This chapter contains information about the following memory allocation topics:

- Application and global heaps
- XVT substitutes for malloc, realloc, and free
- Allocating memory on the global heap
- XVT's portable ATTR_MEMORY_MANAGER attribute
- Resource memory allocation

20.1. Application and Global Heaps

XVT portably provides access to two types of memory heaps: application and global. The application heap is similar to memory allocations done through the standard C functions malloc, realloc, and free. However, XVT provides and recommends alternatives to those functions. The Toolkit uses these alternatives for its memory allocations: xvt_mem_alloc, xvt_mem_realloc, xvt_mem_free. (See section 20.2, next.)

20.2. XVT Substitutes for malloc, realloc, and free

Internally, the XVT libraries never call the standard C functions malloc, realloc, or free directly. Instead, they call xvt_mem_alloc, xvt_mem_realloc, or xvt_mem_free.
A main purpose of the xvt_mem_* functions is to control precisely how XVT libraries allocate memory internally.

For most XVT implementations, the XVT memory functions are identical to the standard C functions. That is, xvt_mem_alloc is a synonym for malloc.

20.3. Allocating Memory on the Global Heap

Tip: To allocate memory on the global heap:

Call xvt_gmem_alloc or xvt_gmem_realloc.

Unlike their counterparts, xvt_mem_alloc and xvt_mem_realloc, these XVT functions return an object of type GHANDLE. Such an object remains valid while your program is running, although the allocated storage to which it refers can be moved around by the memory management system, if it needs to.

For portability reasons, you should not pass a GHANDLE to another process (task), or thread, nor should you write it to a file for use by a later invocation of the program that wrote it (as with all dynamic allocations).

Tip: To access the memory referred to by a GHANDLE:

Lock it with xvt_gmem_lock, which returns a pointer.

Tip: To unlock memory when you're done using it:

Call xvt_gmem_unlock.

This call invalidates the xvt_gmem_lock pointer, so you must make sure you don't use it (for safety, set the pointer variable you've been using to NULL). When you call xvt_gmem_lock again, you might get a different pointer.

Tip: It's best to keep global memory blocks locked only for brief intervals, in order to give the system maximum flexibility to manage memory.

Tip: To free a global memory block:

Call xvt_gmem_free.

Tip: To obtain the size of a global memory block:

Call xvt_gmem_get_size.

20.4. ATTR_MEMORY_MANAGER Attribute

You can write customized memory management functions for your applications. To use them, you register the functions by means of XVT's portable ATTR_MEMORY_MANAGER attribute.

The ATTR_MEMORY_MANAGER attribute contains the addresses of the system-wide memory management functions that are called

when the application invokes xvt_mem_alloc, xvt_mem_free, xvt_mem_realloc, and xvt_mem_zalloc. This attribute is the address of a structure of type XVT_MEM (defined in **xvt_type.h**).

Example: The following code sets the memory management functions, which must be done before xvt_app_create:

```
XVT_MEM my_functions = {my_alloc, my_free, my_realloc,
        my_zmalloc};
xvt_vobj_set_attr(NULL_WIN, ATTR_MEMORY_MANAGER,
        (long)&my_functions);
```

If your application does not set the ATTR_MEMORY_MANAGER attribute, the system automatically sets it to default values at system initialization time.

Note: You should remember to use the XVT_CALLCONV1 macro in the prototypes and headers for your memory management functions.

See Also: For more information about ATTR_MEMORY_MANAGER, see the *XVT Portability Toolkit Reference*.

20.5. Resource Memory Allocation

Many XVT objects or resources have their own dedicated memory allocation functions. For example, xvt_cb_alloc_data and xvt_cb_free_data respectively allocate and free data for the clipboard. If objects provide memory functions, you should use these functions, and only these functions, to allocate and free memory.

Some XVT objects or resources allocate memory as a side effect of the creation of the object. For example, xvt_image_create and xvt_palet_create must be accompanied by calls to xvt_image_destroy and xvt_palet_destroy, respectively.
The application is responsible for cleaning up its memory.

Caution: On some native platforms supported by XVT, memory cleanup on application termination is an important consideration. You should *not* assume that terminating
an application automatically returns to the heap any of the memory consumed by that application. Make sure that you free all memory and resources requested by the application.

21

DIAGNOSTICS AND DEBUGGING

This chapter discusses diagnostic and debugging aids that can help you identify and handle code errors. XVT provides the following tools for dealing with errors:

- General error checking techniques
- Error signaling with error codes
- Error handler functions
- An **errscan** tool
- Error definition and message files
- Error dialogs for reporting errors and warnings

See Also: For additional information about the **errscan** tool, refer to section 21.4 on page 21-6.

21.1. XVT Error Checking Techniques

To help you program reliably and defensively, XVT uses these general techniques for checking errors:

- Validating function input arguments
- Indicating errors in function return values
- Calling error handlers (special functions called whenever an error is encountered)

21.1.1. Arguments and Return Values

The XVT API validates all function arguments, and rejects any improperly requested operation. In addition, XVT often indicates that an error occurred by using a function's return value, such as returning a NULL pointer or a BOOLEAN value to show that an operation failed.

21.1.2. Error Handlers

If XVT detects an error during argument validation or while processing a request, it reports the problem to one or more error handlers. Error handlers post and/or record error messages and tell the program that an error occurred. Sometimes they even perform the cleanup necessitated by an unsuccessful operation.

XVT provides a default “last chance” error handler. However, the application can register other error handlers that are called before this one.

See Also: For more information about error handlers, see section 21.3 on page 21-4.

21.2. XVT Error Signaling

Whenever XVT encounters an error, it signals this error to one or more error handlers. Except in the case of fatal errors, XVT continues after the signal, and typically also returns some indication of failure, such as a NULL pointer. XVT supplies two function-like macros to signal errors:

- xvt_errmsg_sig
- xvt_errmsg_sig_if

You can use these functions to produce arbitrary, application-generated messages. You can then use the **errscan** tool to extract such messages from the your source code files and generate both an error message file and a file containing error ID #defines.

See Also: For more information about xvt_errmsg_sig and xvt_errmsg_sig_if, see the *XVT Portability Toolkit Reference*.
For more information about **errscan**, see section 21.4 on page 21-6.

21.2.1. Error Codes (XVT_ERRID)

When XVT signals an error, it identifies the error with an error code (of type XVT_ERRID). This error code is used to retrieve an appropriate error message from the error message file. It also allows the error handler to deal with messages selectively. Each error code has three components:

- Major category
- Minor category
- Message number (within the minor category)

When you develop an application, you can insert the following xvt_errmsg_def_* calls in the application to define categories:

xvt_errmsg_def_mjr
: Defines new major error categories. To create a major category, add a suffix to the "ERR_" string: "ERR_xxx".

xvt_errmsg_def_cat
: Defines minor categories. To create a minor category, add a suffix to the major category: "ERR_xxx_yyy".

xvt_errmsg_def_std
: Defines undivided standard error messages. To create a special error message, add a suffix to the minor category: "ERR_xxx_yyy_zzz". For example: ERR_ARG_VALUE_TOOLOW.

When the **errscan** tool processes the xvt_errmsg_def_* and xvt_errmsg_sig functions, it generates error codes automatically. XVT provides several xvt_errid_* macros to define error codes and to access and compare their components.

21.2.2. Types of Errors

Error signaling functions, such as xvt_errmsg_sig, classify errors by their level of severity. Error message handlers can then use these severity classifications to decide how to handle a particular error signal:

SEV_WARNING
: The requested operation can be performed, but some corrective action is required.

SEV_ERROR
: The requested operation cannot be performed, but the application can continue.

SEV_FATAL
: Execution cannot continue; the application must terminate.

Note: A SEV_ERROR often means that the operation has been skipped, which can adversely affect application behavior.

21.2.3. Error Message Objects

Each error message signaling call (xvt_errmsg_sig) creates an error message object (of type XVT_ERRMSG), which is passed to individual error handlers. This object exists only during the duration of the call to xvt_errmsg_sig. When that function terminates, the error object disappears.

Error handlers can make inquiries about the object, to find out its signal severity, error code, source filename and line number, or the message associated with the error category (major, minor, or complete error code).

See Also: For more information, see the xvt_errmsg_* functions in the *XVT Portability Toolkit Reference*.

21.3. Error Handlers

Whenever xvt_errmsg_sig signals an abnormal condition (that is, an error) the XVT error messaging system calls an error handler function. An error handler is defined by a typedef XVT_ERRMSG_HANDLER. Its first argument is an error message object (of type XVT_ERRMSG) encapsulating all information about an error.

An error handler can do one of two things:

- Intercept the error signal. The error handler returns TRUE, indicating that it has processed the error. The error messaging system does not try the next error handler in the stack.
- Pass the signal to another handler. The error handler returns FALSE, indicating that it has chosen not to handle the error. The error messaging system tries the next error handler in the stack.

21.3.1. Error Handler Hierarchies

Your application can establish a hierarchy of error handlers:

- Stacked error handlers specific to a window event handler
- A permanent application event handler
- The XVT-supplied “last chance” event handler

At each level of this hierarchy, an error handler can either intercept the error message (i.e., post a dialog, write the message to a file, or simply hide the error), or pass the error message to the next handler in the hierarchy.

To prevent infinite recursion, any errors signaled from within the error handler (for example when calling the XVT API) are delivered only to remaining handlers in the hierarchy.

21.3.1.1. Stacked Error Handlers

Stacked error handlers have a scope limited to a particular window event handler call. You must explicitly push them onto the error handler stack (with xvt_errmsg_push_handler) and pop them off it (with xvt_errmsg_pop_handler).

If you fail to remove (pop) such handlers from the stack before returning from the window event handler, you'll receive a warning, and the handlers will be removed automatically.

See Also: For more information about xvt_errmsg_pop_handler and xvt_errmsg_push_handler, see the *XVT Portability Toolkit Reference*.

21.3.1.2. Application-supplied Error Handlers

XVT allows your application to register an application-supplied, permanent error handler that is called immediately *before* the XVT-supplied "last chance" error handler. You might do this for the following reasons:

- To post customized error dialogs
- To hide (intercept) errors having a particular severity or class, or certain specific error messages
- To hide (intercept) all errors that slip through previously "pushed" error handlers
- To perform application-specific shutdown on fatal errors

Tip: To register a permanent application error handler:

Set the ATTR_ERRMSG_HANDLER attribute.

Tip: When you write an error handler, keep in mind that SEV_FATAL errors require application termination. The error messaging system terminates the application if the handler returns TRUE. Also, your error handler must pay attention to E_UPDATE event processing. Many XVT operations (such as posting error dialogs) are illegal during an E_UPDATE, because they can cause endless recursive calls to the error handler.

See Also: For more information on ATTR_ERRMSG_HANDLER, see the *XVT Portability Toolkit Reference*.
For more information on E_UPDATE, see section 4.3.3 on page 4-10.

21.3.1.3. The XVT Error Handler

The XVT-supplied "last chance" error handler appends all messages to the debug file (if present), and posts an appropriate dialog. The dialog provides the following information:

- Error message
- Name of the XVT API call
- Source filename and line signaling an error

See Also: For more information about the debug file, see section 21.5.3 on page 21-8.
For more information about error dialogs, see section 21.6 on page 21-8.

21.4. XVT's errscan Tool

XVT supplies an **errscan** tool, which can examine your application code and perform the following operations:

- Find all instances of error signaling (xvt_errmsg_sig and xvt_errmsg_sig_if calls)
- Find all predefined error messages (defined with xvt_errmsg_def_* macros)
- Generate the error message text file **ERRCODES.TXT**
- Generate the error codes definition file **xvt_perr.h**

The benefit of using the **errscan** tool is that you don't have to manually collect and maintain a list of error codes and associated messages.

The **errscan** tool uses the message suffix and number supplied by each xvt_errmsg_sig or xvt_errmsg_def_* macro to build an error code #define. Consequently, the suffix and number must be unique within a given message category.

errscan does not process multibyte characters, but its output can be localized. In other words, you must run **errscan** on single-byte character strings and then localize the output to the correct locale.

The **errscan** tool warns you of any duplication and/or syntax errors. However, its syntax checking is not as sophisticated as a compiler's.

In particular, **errscan** does not use the **cpp** pre-processor, and it makes several assumptions about the error signaling call.

You can build **errscan** either as a command line utility or as an interactive application. XVT provides **errscan** both as a source file and an executable, in the directories **...src/errscan/errscan.c** and **...bin/errscan**.

Note: Source customers can also run **errscan** on XVT source code.

See Also: For information about using the **errscan** tool, refer to the *XVT Portability Toolkit Reference*.
To learn how to build **errscan**, see the *XVT Platform-Specific Book* for your particular platform.
For information about xvt_errmsg_sig and xvt_errmsg_sig_if, refer to section 21.2 on page 21-2.

21.5. Error Files

XVT provides two error definition header files and an error message file. In addition, your application can write error tracing information into a temporary debug file.

21.5.1. Error Header Files

XVT provides header files that contain error definitions.

xvt_perr.h
Defines a #define constant for each XVT_ERRID used by a particular (that is, platform-specific) XVT implementation. The **errscan** tool generates this file, along with the error message (text) file.

xvt_msgs.h
Defines a basic set of error categories and messages. This file has no executable meaning; it merely serves as the message definition source for the **errscan** tool. Source customers can use this file to add new standard errors.

21.5.2. XVT Error Message File

All XVT-signaled messages are collected in an error message file, **ERRCODES.TXT**. The error messaging system retrieves message text from this file. This allows you to localize messages by substituting translated text in the file.

If XVT can't find an error message file, or if a particular message is missing in the file, XVT uses a standard, hardcoded (English)

message. Hardcoded message text is available only for a limited set of basic error categories and messages.

If no message file or hardcoded messages exist, individual messages are represented by amessage number, such as "MSG (7/3)3556". In this encoding, (7/3) indicates the message category number (major/minor), followed by the message number in the major/minor category.

You can ship the XVT-supplied file **ERRCODES.TXT** with your application. Or, if your customers might not understand the XVT error messages, you can provide a subset or translation of this file.

Tip: To redefine the location of the error message file:

Set the ATTR_ERRMSG_FILENAME attribute.

21.5.3. Debug File for Error Tracing

To help with debugging, you can write error tracing information into a temporary debug file. XVT supplies a function and a macro to append information to the debug file:

- xvt_debug_printf (function)
- xvt_debug (macro)

The XVT-supplied “last chance” error handler also writes error messages to this same debug file (if the file is present).

Tip: To redefine the location of the temporary debug file:

Set the ATTR_DEBUG_FILENAME attribute.

See Also: For more information about xvt_debug and xvt_debug_printf, see the *XVT Portability Toolkit Reference*.

21.6. Error Dialogs

When a program encounters an error, it can either perform some recovery, or tell the user about the error (if the error resulted from incorrect use of the program). XVT provides standard dialogs for reporting warnings, errors, and fatal errors. You can call them with these functions:

xvt_dm_post_warning
xvt_dm_post_error
xvt_dm_post_fatal_exit

You can also use other dialogs or xvt_scr_beep to notify the user of an error.

See Also: For more information about xvt_dm_post_warning, xvt_dm_post_error, xvt_dm_post_fatal_exit, or xvt_scr_beep, see the *XVT Portability Toolkit Reference*.

22

HYPERTEXT ONLINE HELP

XVT's online help feature provides a powerful, flexible, hypertext-based help system for your applications. The online help feature includes these key elements:

- A hypertext viewer, integrated with the XVT Portability Toolkit
- Complete text-formatting features, including multiple fonts and styles
- Easy association between all GUI objects and specific help topics
- Bitmap images embedded in help text
- Support for native help display facilities

22.1. Help System Components

XVT's help system contains the following software components:

Help Text Source File(s)
One or more plain-text files containing the help information itself, along with formatting commands, hypertext links, and other information that defines how the help information is presented to your application's user.

Help Compiler (helpc)
A compiler that converts help text source files into several other file formats, including a portable, XVT-defined format, and several platform-specific formats.

Portable Binary Help File
A portable, machine-readable file containing the help information. The XVT help viewer displays this file during application execution.

XVT Help Engine
A portion of the XVT Portability Toolkit that handles help-related event processing, manages context information, and invokes the help viewer(s) to display the help text.

XVT Help Viewer
A viewer that presents the help information to the user. This viewer is available on all XVT platforms. The help viewer can either be added to your application, or used as a standalone application.

The relationship of the various components of the XVT help system is shown in Figure 22.1.

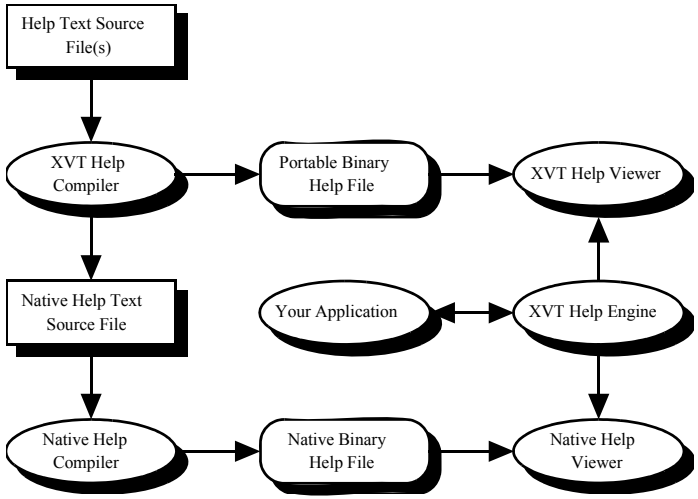


Figure 22.1. Help system components

Some GUI platforms provide their own help viewers. On these platforms, the XVT help system includes these additional components:

Native Help Text Source File
A platform-specific file that contains the help information itself, as well as formatting commands and hypertext links. Usually you won't have to use these files directly.

Native Help Compiler
A compiler that converts native help text source files into files readable by the native help viewer. This utility program is supplied with your native development tools.

Native Binary Help File
A platform-specific, machine-readable file containing the help information. This file is displayed by one of the native help viewers supported by the XVT help system.

Native Help Viewer
A viewer that presents help information to application users, using the native window system's help viewer.

Note: Currently, XVT supports the native help systems for Win32.

22.2. XVT Help Viewer

The help viewer serves as the interface between users and the available help information. In other words, the help viewer is the part of the help system that users see. This section describes how the XVT help viewer operates, from the application user's point of view.

When users invoke help, they can see help information displayed in two kinds of windows. Users can browse among topics, search for a specific topic, or navigate along a topic thread. They can also insert a bookmark into a topic to mark it, and copy or print topics.

See Also: For information about invoking help, see section 22.3 on page 22-9.

22.2.1. Help Windows

The online help system displays information in two kinds of windows:

- Topic windows
- Pop-up windows

Both window types can display help text and bitmap graphics. However, the two types differ in how the user interacts with them and in their appearance. Topic windows offer more extensive navigation control and decorations, while pop-up windows have minimal user-interface features.

22.2.1.1. Topic Windows

Topic windows display primary help topic information—that is, the main help text itself. Topic windows have the following features:

- Text and graphics display
- On-screen controls and keyboard commands for moving through the topic information

- Browsing controls for moving between logically related topics, like pages in a book
- Embedded hypertext links and hot buttons in the topic text, which have a different appearance from plain text (and each other)
- A search facility for locating other topics
- A dynamically maintained topic thread, a list of topics viewed by the user
- Navigation controls for moving forward and backward through the list of topics in the topic thread
- Bookmarks that can be set by the user, and controls for navigating to these marked topics
- Copying of topics to the system clipboard
- Printing of topics

Once a topic window has been opened, the user must explicitly dismiss it.

22.2.1.2. Pop-up Windows

Pop-up windows display short-term information, such as glossary definitions and hot button topics. Since pop-up windows quickly display short help topics, they have a limited number of features as compared to topic windows:

- Limited user input—no navigation controls
- Text and graphics display (although on some platforms the time required to display a graphic image can reduce the convenience of pop-up windows)

Unlike topic windows, which the user must explicitly dismiss, pop-up windows are dismissed when any key event or mouse click occurs.

22.2.2. Navigation

Help topic windows contain navigation controls to enable application users to move from one help topic to another, as shown in Figure 22.2.

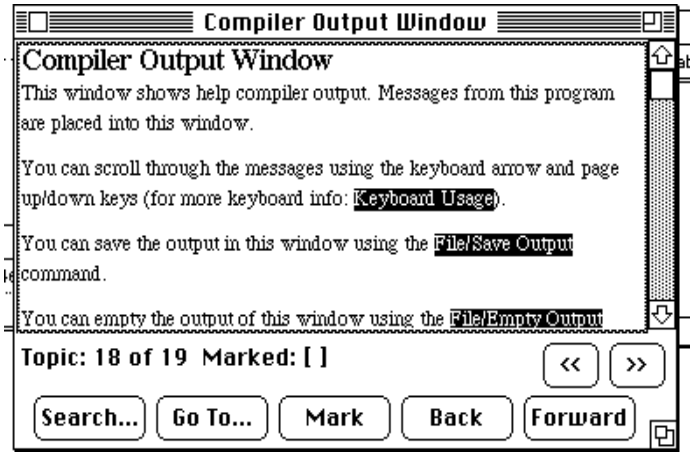


Figure 22.2. Navigation controls in a topic window

The topic window contains a scrolling text pane that displays the help topic text. The user can move and resize the topic window using standard native document-window border decorations.

The topic window contains several push buttons for navigating from topic to topic, and invoking other help system dialogs:

Search
: Invokes the Search dialog, with which the user can search for help information by topic name or by keyword (see below).

Go To
: Invokes the Go To dialog, with which the user can navigate to the help index, glossary, table of contents, or marked topics (see below).

Mark
: Marks the current topic, and adds its name to the list of Bookmarks in the Go To dialog. If the current topic is already marked, clicking this button removes the mark.

Back
: The help system maintains a topic thread, a list of topics that have been displayed in the topic window. Clicking this button

moves to the previously viewed topic in the list, and displays it in the topic window.

Forward

Moves to the next topic in the help system's list of displayed topics, and displays the topic in the topic window. This button is disabled until the Back button has been clicked at least once.

>>

Moves to the next logically related topic (the next topic in the browse sequence), and displays it in the topic window.

<<

Moves to the previous topic in the browse sequence, and displays it in the topic window.

22.2.2.1. Go To Dialog

Clicking the Go To button in a topic window opens a modal dialog, as shown in Figure 22.3.

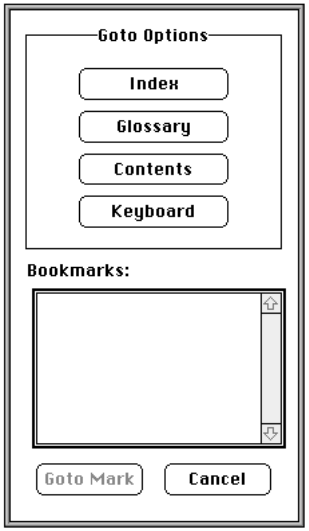


Figure 22.3. Modal Go To dialog

The buttons near the top of the dialog open a corresponding help topic when clicked. If you areusing the "application-bound" viewer configuration, you can change the button names to suit your application. The **xvt_help.xrc** resource file defines the Go To dialog and button names.

The Bookmarks list box contains a list of topics that the user has marked by clicking the Mark button in the topic window.

Tip: To return to a previously marked topic:

1. Select the topic name in the Bookmarks list.
2. Click Go to Mark.
   -*OR*-
   Double-click the topic name.

Tip: To dismiss the Go To dialog without changing topics:

Click Cancel.

See Also: For a comparison of application-bound and standalone help viewer configurations, see section 22.4.1 on page 22-12.

22.2.3. Searching

An application user can search for help topics with the Search dialog; this dialog is shown in Figure 22.4. To invoke this dialog, the user can either click the Search button in a topic window, or choose Search from the Help menu.

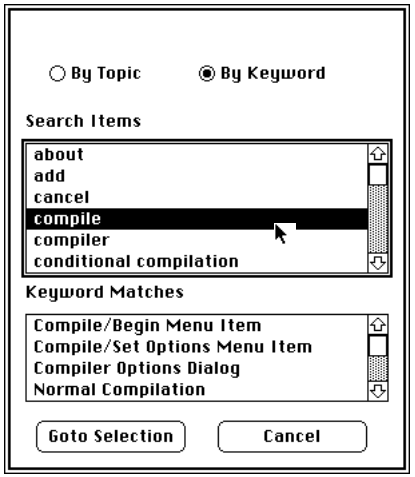


Figure 22.4. Search dialog

Help topics are identified either by topic names or by keywords (optional). Two radio buttons in the Search dialog determine which of these appears in the Search Items list box, and thus how help topics are displayed:

Topic Name
When this radio button is checked, the Search Items list box displays all the topics in the help file. Double-clicking on a topic name dismisses the dialog and opens the appropriate topic window.

Keyword
When this radio button is checked, the Search Items list box displays all keywords defined in the help file. Selecting one of these keywords displays a list in the Matched Items list box of all topics that contain this keyword.

22.3. Invoking Help

Application users can ask for help information in several ways. They can get two kinds of context-sensitive help: spot or object-click help. Or they can choose help topics from the Help menu. In addition, your application can display help information programmatically.

22.3.1. Spot Help

Spot help is context-sensitive. It presents help information associated with the GUI object that has focus. To invoke spot help, the user presses a special key (often F1 or Help), or chooses On Context from the Help menu.

Note: Some platforms cannot give focus to all control types; as a result, spot help is not available for some controls on these platforms.

22.3.2. Object-click Help

Object-click help is also context-sensitive. It is similar to spot help, but works in reverse—the user requests help by choosing Object Click from the Help menu, then selecting the desired GUI object. Help information is displayed for that object.

See Also: For more information on object-click help, refer to the *XVT Platform-Specific Book* for your platform. (On many platforms, object-click help is not consistent with the window manager's look-and-feel.)

22.3.3. Menu Help

Menu help simply lets the user choose one of several help topics from the Help menu. Some or all of following topics are placed on the Help menu by default:

- On Context (for invoking spot help)
- Index
- Contents
- Tutorial
- Keyboard
- Object Click (for invoking the object-click method)
- Search (for invoking the search dialog)
- Using Help
- Version

The actual default menu items vary from platform to platform.

See Also: For the definitions of each platform's Help menu, see the platform-specific books and the **xvt_help.xrc** file.

22.3.4. Invoking Help Programmatically

The XVT help system automatically handles the previously described methods of invoking help. However, you can also display help information under control of your application.

For instance, when an error occurs, your application can display a help topic describing the error and suggestions for resolving it. Your application can easily display any help topic.

Tip: To display a help topic:

Call xvt_help_display_topic.

22.4. Adding Online Help to an Application

XVT-Design lets you quickly associate help topics with GUI objects. In each attribute dialog for the GUI Objects you create with XVT-Design, you can specify the appropriate help topic in the dialog's "Help Topic" list button.

This section describes how to add online help to your XVT-based application. To add online help to your application, you would follow these general steps:

1. Write a help text source file, a plain-text file that contains the text for the online help information.
2. Add formatting commands to your help source file, using the help system markup language commands.
3. Place definitions for symbolic names for the help topics in a header file.
4. Compile your help source file with **helpc**, the help-text compiler.
5. Add a Help menu to your application's menu resource definitions.
6. Add code to your application to open and close the compiled help file.
7. Add code to your application to associate help topics with GUI objects.
   -*OR*-
   Create a help topic association file.

XVT-Design performs steps 5, 6, and 7 for you.

The following sections explain how to accomplish each of these steps. They cover the following topics:

- Help viewers
- Header files
- Resource files
- Creating a help menu
- Opening a help file

- Associating topics with objects
- Disassociating topics from objects
- Event handling
- Displaying help topics
- Handling object-click help
- Modal dialogs and help

See Also: For detailed descriptions of all functions mentioned in this section, see the *XVT Portability Toolkit Reference*.

22.4.1. Help Viewers

The XVT help system provides two different kinds of help viewers:

Application-bound viewers
An application-bound viewer has all its executable code and resources integrated with your application's executable file.

Standalone viewers
An application that is separate from your application's executable file.

From the application user's point of view, both kinds of viewers have the same appearance and behavior. Both use the same API functions; the same application code handles either one. You determine which viewer your application uses at link time, by linking one of two object libraries with your application.

Note: The Win32 native viewer is a standalone viewer.

See Also: For information on libraries and linking for your particular development platform(s), see the *XVT Platform-Specific Books*.

22.4.2. Header Files

Individual help topics are referred to by integer identifiers, which you can put into a header file.

Tip: To define symbolic names for help topic identifiers:

Use `#define` statements in a header file.

Include this header file in any source-code file that refers to specific help topics.

22.4.3. Resource Files

The **xvt_help.xrc** file contains XRC definitions of all resources used by the help system. This file is included automatically when you include **xrc.h** in your source files. If your application does not use the help system, you can omit the help system's resources from your application.

Tip: To include the help system's resources in your application:

#include the standard XVT resource header file, **xrc.h**.

Tip: To omit the help system's resources from your application:

Define the symbol NO_HELP_RESOURCES before including **xrc.h**, like this:

```
#define NO_HELP_RESOURCES
```

Note: If your application uses a standalone viewer, you should omit the help system's resources from your application. The application does not need these resources, since they are present in the standalone help viewer.

22.4.4. Creating a Help Menu

XVT-Design's Menubar Editor includes the default help menu as a standard menu that you can add to any menubar in your application.

The help system provides a default help menu that conforms to the native style guidelines for eachGUI platform. This menu's resource ID is DEFAULT_HELP_MENU. It is included in the default menubar.

Tip: To add the help menu to your resource file:

Use the predefined menu identifier DEFAULT_HELP_MENU in your menubar. For example:

```
MENUBAR MY_MENUBAR
MENU MY_MENUBAR
      DEFAULT_FILE_MENU
      DEFAULT_EDIT_MENU
      ...                          /* your menus */
      DEFAULT_HELP_MENU
```

Tip: To omit the help menu from the default menubar:

Define the symbol NO_STD_HELP_MENU before including **xrc.h**, like this:

```
#define NO_STD_HELP_MENU
```

22.4.5. Opening a Help File

Before your application can display any help topic information, you must open a help file.

Tip: To open a help file:

Call xvt_help_open_helpfile while handling the E_CREATE event for your application's task window.

This function returns a help file descriptor of type XVT_HELP_INFO. You should pass the value of this descriptor to all help functions that require a XVT_HELP_INFO parameter.

Most applications only need to open the help file while handling the E_CREATE event for the task window and close it whilehandling the E_DESTROY event. If necessary, however, you can open and close help files freely during the execution of you program. You can also have more than one help file open at a time. Use multiple XVT_HELP_INFO variables to distinguish the different files.

Tip: To close a help file when it is no longer needed:

Call xvt_help_close_helpfile.

Caution: You must close all open help files before your application terminates.

See Also: For information on defining the help search path, see the description of xvt_help_open_helpfile in the *XVT Portability Toolkit Reference*.

22.4.6. Associating Topics with Objects

If your application uses the context-sensitive help invocation methods (spot help and object-click help), you must associate specific help topics with the appropriate GUI object.

You do not have to create helptopics andassociations forall objects. In some cases, such an association is not even desirable. For instance, there is no point inassociating a helptopic with a WINDOW that cannot be manipulated by the user.

Note: There are restrictions on the type of topics that can be associated with a GUI object. Only topics defined with the HTOPIC format can be associated, portably, with objects.

See Also: For more information on the HTOPIC format, refer to the *XVT Portability Toolkit Reference*.

In XVT-Design, you can associate topics with any WINDOW or MENU_TAG when you set the object's attributes. XVT-Design automatically inserts into the source code the correct function calls to associate the topics with the GUI objects.

Tip: To programmatically associate a help topic with a WINDOW (window, dialog, or control):

Call xvt_help_set_win_assoc.

Tip: To associate a help topic with a menu item:

Call xvt_help_set_menu_assoc.

Tip: To associate a help topic with any object:

Create an association table (see section 22.4.6.1).

22.4.6.1. Association Tables

As an alternative to calling xvt_help_set_win_assoc or xvt_help_set_menu_assoc, you can create associations between help topics and objects in file-based association tables. An association table is a text file that lists resource ID values and their associated help topic ID. If your application uses more than one help file, each file has its own association table.

Association tables require less application code, and are easier to maintain since all topic/object associations are kept in one location. However, you must remember to ship the association table file(s) with your application.

22.4.6.2. Association Table File Format

Association table files must be located in the same directory as the help topic files. They must have the same name as the corresponding help topic file, with the extension **.csa**.

You can place comments freely in association files. Any line beginning with a space, Tab, or '#' character is ignored.

Each line in the association table associates one help topic with one GUI object. The lines have the following format:

<object type> <object id> <parent id> <topic id>

<object type>
: One character that indicates the type of the GUI object: M for menu, W for window, D for dialog, or C for control.

<object id>
: The resource ID of the object.

<parent id>
: The resource ID of the object's parent window. If this field is zero, the help topic is associated with all objects of the specified object type and resource ID, regardless of their parent windows.

<topic id>
: The identifier for the help topic to be associated with the object.

Note: All ID fields must be integers.

See Also: For more information on using the **helpc** compiler, refer to section 22.6 on page 22-23.

22.4.7. Disassociating Topics from Objects

During program execution, your application may need to change which help topics are associated with GUI elements. For example, when an error occurs, you might wish to remove the usual help topic and replace it with a help topic that offers suggestions for recovering from the error.

Tip: To remove the association between a GUI object and a help topic:

Call xvt_help_set_win_assoc, passing NULL_TID for the topic identifier parameter.

Tip: To remove the help associations for a container and all the objects it contains:

Call xvt_help_disassoc_all.

In most instances, disassociation occurs automatically. If you close a window that contains child windows and/or controls, all help topics for the container and its offspring are disassociated automatically.

However, if you destroy a control without destroying its container, you must explicitly remove its help-topic association, using xvt_help_set_win_assoc.

22.4.8. Event Handling

The help system handles help-related user events automatically. In most situations, your event handlers will never receive an E_HELP event. XVT's help system intercepts and processes these events before your event handlers are called. This automatic handling of events does not take place until after your application has opened at least one help topic file.

22.4.9. Displaying Help Topics

In addition to letting the help system automatically handle help requests, your application can explicitly display help topics.

Tip: To display a help topic:

Call xvt_help_display_topic.

Note: To be portable, the topic ID passed to this function must correspond to an HTOPIC.

See Also: For more information on the HTOPIC format, refer to the *XVT Portability Toolkit Reference*.

22.4.10. Handling Object-Click Help

Normally, the application user invokes object-click help. However, your application can also invoke it directly. Once object-click help is invoked, the mouse pointer is trapped and events are consumed by the help system. Object-click mode terminates when the user clicks an object, or when your application explicitly terminates it.

Tip: To invoke object-click help mode:

Call xvt_help_begin_objclick.

Tip: To terminate object-click help mode:

Call xvt_help_end_objclick.

22.4.11. Modal Dialogs and Help

If your application uses the application-bound help viewer, you cannot display help topic windows when a modal dialog is active. Also, when a modal dialog is active, the application user cannot interact with any help topic windows.

22.5. Help Source File Format

Help source files contain the following elements:

- Comments (optional)
- Preprocessor commands (optional)
- Header section
- Help text body
- Predefined help topic information

Each help file element is discussed in the following sections. To provide portability across all XVT development environments, help source files use a plain-text markup language for formatting.

Tip: To create your help source files:

Use any text editor or word processor capable of generating plain text files.

See Also: For detailed information about the help source file format language, refer to the *XVT Portability Toolkit Reference*.

22.5.1. How the Help System Applies Formatting Commands

Most of the formatting commands take effect at runtime, rather than compile time. Together, the compiler and runtime parser format the help text following these rules:

- A single newline between lines of text in the help source file translates into a space in the compiled file.
- Two adjacent newlines in the help source file translate into a single newline in the compiled file.
- The parser word-wraps all topic text from the compiled help file. Wrapping stops when a newline is encountered. The newline starts a new line in the display window. The No Word Wrap command overrides this.
- A \A in the compiled topic body instructs the parser to start a new paragraph. One and one-half blank lines (in the current font) separates paragraphs on-screen. The first line of a new paragraph is not indented; to indent the first line, place a Tab immediately after a \A.
- Tab stops are fixed to approximately the width of four "W" characters in the default font.
- A default system font is automatically selected until the first font change. Every topic always starts with the default font.
- An embedded font change command changes the current font, until another font command changes it again.
- Text is reformatted when the display window is resized.
- Pictures are always output at the current X coordinate of the display window (i.e., the X position of the previous text character or the left margin if there is no text),and are clipped to the right edge.
- Left margins remain in effect until reset, or another topic starts—each topic begins with a zero left margin.
- Indentation settings are reset to zero at the beginning of each paragraph, and after two newlines.

22.5.2. Predefined Help Topic Information

XVT provides some reserved topic identifiers, as well as pre-written help topics for several of them.

22.5.2.1. Reserved Help Identifiers

The following symbols are reserved topic identifiers, which correspond to the items on the predefined Help menu:

Topic ID:	Corresponding Item:
XVT_TPC_HELPONHELP	Information about the help system
XVT_TPC_KEYBOARD	Information about special keys
XVT_TPC_INDEX	Help index
XVT_TPC_CONTENTS	Help table of contents
XVT_TPC_TUTORIAL	Application tutorial information
XVT_TPC_ONVERSION	Application version information
XVT_TPC_GLOSSARY	Glossary of terms

The **xvt_help.h** file defines these reserved topic identifiers. It may also contain additional identifiers that are not listed here.

If you use the standard Help menu, you must provide help text for each of these topics. You can either write the text yourself, or use pre-written text for some of the topics.

Note: Since the topics XVT_TPC_INDEX, XVT_TPC_CONTENTS, and XVT_TPC_TUTORIAL are necessarily dependent on your application, XVT provides no pre-written text for them.

The following symbols are reserved topic identifiers, which correspond to predefined XVT dialogs:

Topic ID:	Corresponding Dialog:
XVT_TPC_FILE_OPEN	xvt_dm_post_file_open
XVT_TPC_FILE_SAVE	xvt_dm_post_file_save
XVT_TPC_ASK	xvt_dm_post_ask
XVT_TPC_NOTE	xvt_dm_post_note
XVT_TPC_ERROR	xvt_dm_post_error
XVT_TPC_WARNING	xvt_dm_post_warning
XVT_TPC_STRING_PROMPT	xvt_dm_post_string_prompt
XVT_TPC_FONT_SEL	xvt_dm_post_font_sel
XVT_TPC_PAGE_SETUP	xvt_dm_post_page_setup
XVT_TPC_MESSAGE	xvt_dm_post_message

When the user requests help while an XVT predefined dialog is active, XVT sends an E_HELP event with the corresponding topic ID to the task event handler. The tid member of the help event structure is set to one of the predefined IDs above. If the help file contains a topic pertaining to that ID, the help viewer displays it.

22.5.2.2. Predefined Help Topics

XVT provides help topic text for several of the reserved topic symbols, including XVT_TPC_HELPONHELP, XVT_TPC_KEYBOARD, and others. The **xvt_help.csh** file contains these topics.

You can include this file in your help source file to provide default help information for the reserved topic symbols. However, the predefined help topic text in the **xvt_help.csh** header is incomplete, so you will most likely want to override these topics in your own help source file.

Tip: To include all of the XVT-provided topics in **xvt_help.csh**:

Add this line to the end of your help source file:

```
#include "xvt_help.csh"
```

Tip: To include some, but not all, of the topics in **xvt_help.csh**:

1. Provide your own help topic text for the topics you wish to customize, in your help source file.
2. At the bottom of your help source file, undefine all reserved topic identifiers whose XVT-provided topic text you want to omit, and redefine them as –1. The compiler skips any help topics that have a topic identifier of –1.
3. Add this line to the end of your help source file:

```
#include "xvt_help.csh"
```

Note: If your code includes the file **xvt_help.csh**, you must place the "#scan "xvt_help.h"" statement in the order shown in the following example.

The **xvt_help.h** file defines the topic identifiers referenced by **xvt_help.csh**.

Example: Suppose you want to provide custom help text for the XVT_TPC_KEYBOARD topic, but use the XVT-provided text for all other reserved topics. Your help source file would contain the following text:

```
HTOPIC XVT_TPC_KEYBOARD "Special Keyboard Commands"
' your help text
…
' at the end of the file:
#scan "xvt_help.h"
#undef XVT_TPC_KEYBOARD
#define XVT_TPC_KEYBOARD -1
#include "xvt_help.csh"
```

22.6. The Help Compiler

XVT's help compiler, **helpc**, compiles your help source files into a compact, binary format. This file format allows the help system to rapidly access your help text while your application executes. A compiled help file is portable across all XVT platforms; you can use one file on each platform without separate recompilation.

The help compiler operates in essentially the same manner, and uses the same command-line options, on all XVT platforms.

See Also: For information about using the **helpc** compiler, refer to the *XVT Portability Toolkit Reference*.

22.6.1. Manifest Constants

The help compiler always predefines the following symbols before compiling the help source:

__helpc__
: Defined when the complier is running. When the compiler was built, this symbol was set to the value of XVT_HELP_VERSION. You can conditionally compile your header files based on the existence of this symbol.

HELP_FMT_XVT

HELP_FMT_WIN
: One of these is defined by the compiler based on the value of the -f command line option. This allows you to conditionally compile your help source based on the format of the output that the help compiler is generating. These symbols are defined as follows:

Symbol Defined:	-f Option:	Description:
HELP_FMT_XVT	XVT	XVT portable help file format
HELP_FMT_WIN	WIN	Win32 file format

Example: This code conditionally compiles a topic for the native MS-Windows help viewer, **Winhelp**:

```
#ifdef HELP_FMT_WIN
HTOPIC NativeWinhelpTopic "Native Topic"
This topic will only be compiled for the native
      MS-Windows help viewer.
#endif
```

22.6.2. Help Source File Text Limitations

Help source files are constrained by the following limitations:

- The text for each topic must be no more than 64K bytes
- Individual bitmaps must be no more than 32K bytes in size
- Bitmaps can be black and white, 16-color or 256-color
- Bitmaps must be in MS-Windows **BMP** format, and should have a resolution of 96 dots per inch
- The total size of the help source file must be no more than 99,999,999 bytes
- Each topic can have no more than 16 keywords
- Tokens in the help file must be no more than 256 bytes each (a token is delimited by white space or punctuation)
- Help files that use different languages and character codesets may need to be stored in separate files (and be compiled on the platform where they are to be used)—see Version 4.5 note below

Help source text files and the binary help files generated by **helpc** are portable between XVT-supported platforms if the target character codesets are compatible. Specifically, characters used in the help source files must come from the invariant character codeset.) For more details, see section 19.4.5 on page 19-52.

A

LANGUAGES AND CODESETS

This appendix lists XVT abbreviations for languages and character codesets. However, XVT does not directly support all these languages and character codesets. The five languages that are fully supported at this time are:

English	Portugese (Brazil)
German	Japanese
Spanish (Spain)	Korean
Italian	Simplified Chinese
French	Traditional Chinese

Because XVT string resources are now stored in a separate file, your application can be programmed in any language, but the five languages listed above are the only languages for which pre-translated resources are shipped with the XVT Portability Toolkits. If you need to support any other language, you must translate many standard resources yourself, such as the strings that are displayed in XVT's predefined dialogs.

Remember that bi-directional languages are not supported. However, for your convenience, all recognizable language abbreviations are listed below—both bi-directional and left-to-right. You need to know the language abbreviation, because you must use it as part of the filename for the file that contains your internationalized resources. (For more information on filenaming conventions in internationalized applications, see section 19.2.1 on page 19-18.)

All listed languages are uni-directional, left-to-right unless specified otherwise.

A.1. Language Abbreviations

XVT <3 character language code> abbreviations are as follows:

Abbrev:	Lanquage:	Direction:
afr	Afrikaans	
alb	Albanian	
amh	Amharic	
ara	Arabic	bidirect
arm	Armenian	
asm	Assamese	
aze	Azerbaijani bidirect	
bah	Bahasa Indonesia	
bal	Baluchi	
bel	Belorussian	
ben	Bengal	
bih	Bihari	
bul	Bulgarian	
bur	Burmese	
cat	Catalon	
che	Chewa	
chi	Chinese	
chu	Chuang	
cop	Coptic	
cro	Croatian	
cyr	Cyrillic	
cze	Czech	
dan	Danish	
dar	Dari Persian	bidirect
dut	Dutch	
dzo	Dzongkha	
eng	English	
est	Estonian	
ewe	Ewe	
fae	Faeoese	
far	Farsi	bidirect
fij	Fijian	
fin	Finnish	
fle	Flemish	
fre	French	
ful	Fulani	

Abbrev:	Lanquage:	Direction:
gal	Galla	
geo	Georgian	
ger	German	
gre	Greek	
grl	Greenlandic	
guj	Gujarati	
hau	Hausa	
hbr	Hebrew	bidirect
hin	Hindi	
ibo	Ibo	
ice	Icelandic	
iri	Irish Gaelic	
ita	Italian	
jpn	Japanese	(also top/bot)
jav	Javanese	
kan	Kanarese	
kas	Kashmiri	
kaz	Kazakh	
khm	Khmer	
kir	Kirghiz	
kor	Korean (al	so top/bot)
kur	Kurdish	bidirect
kuy	Kuy	
lad	Ladino	
lao	Laotian	
lap	Lappish	
lat	Latin	
ltv	Latvian	
lav	Lavana	
lit	Lithuanian	
lux	Luxembourgian	
mac	Macedonian	
mad	Madurese	
mag	Magyar	
mlg	Malagasy	
mag	Malay	bidirect
mlm	Malayalam	
mld	Maldivian	
mlt	Maltese	
mao	Maori	
mar	Marathi	

Abbrev:	Lanquage:	Direction:
mol	Moldavian	
mon	Mongasque	
mng	Mongolian	top/bot
nau	Nauruan	
nep	Nepali	
nor	Norwegian	
ori	Oriyan	
pal	Pali	
pas	Pashto	bidirect
pid	Pidgin	
pol	Polish	
por	Portuguese	
pun	Punjabi	
rom	Romanian	
rmh	Romansch	
rua	Ruandan	
run	Rundi	
rus	Russian	
sam	Sami	
smn	Samoan	
san	Sango	
snk	Sanskrit	
ser	Serbian	
ses	Sesotho	
set	Setswana	
sho	Shona	
sin	Sindhi	bidirect
snh	Sinhalese	
slo	Slovak	
sln	Slovenian	
som	Somali	
spa	Spanish	
sud	Sudanese	
swa	Swahili	
swz	Swazi	
swe	Swedish	
tad	Tadzhik	
tag	Tagalog	
tak	Taki-Taki	
tam	Tamil	

Abbrev:	Lanquage:	Direction:
tel	Telugu	
tha	Thai	
tib	Tibetan	
tig	Tigre	
tgr	Tigrinya	
ton	Tongan	
tsw	Tswana	
tur	Turkish	
trk	Turkmen	
tuv	Tuvaluan	
ukr	Ukrainian	
urd	Urdu	bidirect
uzb	Uzbek	
ven	Venda	
vie	Vietnamese	
xho	Xhosa	
yid	Yiddish	bidirect
yor	Yoruba	
zul	Zulu	

A.2. Character Codeset Abbreviations

The XVT <3-4 character codeset> abbreviations are one of the following:

Abbrev:	Codeset:	Languages:
General use:		
inv	Invariant ASCII Codeset (ASCII Subset)	
asc	ASCII Codeset (7-bit)	
jis	JIS	Japanese
sjis	Shift-JIS	Japanese
XVT/XM:		
is1	ISO 8859-1 (ISO Latin-1)	Western European (Danish, Dutch, English, Faeroese, Finnish, French, German, Icelandic, Italian, Norwegian, Portuguese, Spanish, Swedish)
is2	ISO 8859-2 (ISO Latin-2)	Eastern European (Albanian, Czechoslovakian, English, German, Hungarian, Polish, Romanian, Serbo-Croatian, Slovak, Slovene)
is3	ISO 8859-3 (ISO Latin-3)	Southeastern Europe (Afrikaans, Catalan, Dutch, English, Esperanto, German, Italian, Maltese, Spanish, Turkish)
is4	ISO 8859-4 (ISO Latin-4)	Northern European (Danish, Estonian, English, Finnish, German, Greenlandic, Lappish, Latvian, Lithuanian, Norwegian, Swedish)

Abbrev:	Codeset:	Languages:
is5	ISO 8859-5 (ISO Cyrillic)	Bulgarian, Belorussian, English, Macedonian, Russian, Serbo-Croatian, Ukrainian
is6	ISO 8859-6 (ISO Arabic)	Arabic, English
is7	ISO 8859-7 (ISO Greek)	English, Greek
is8	ISO 8859-8 (ISO Hebrew)	English, Hebrew
is9	ISO 8859-9 (ISO Latin-5)	Western European (Danish, Dutch, English, Faeroese, Finnish, French, German, Italian, Norwegian, Portuguese, Spanish, Swedish, Turkish)
is10	ISO 8859-10 (ISO Latin-6)	Danish, English, Estonian, Faeroese, Finnish, German, Greenlandic, Icelandic, Lappish, Latvian, Lithuanian, Norwegian, Swedish
uja	EUC-JA	Japanese, English
uctw	EUC-CH_tw	Traditional Chinese, English
uccn	EUC-CH_cn	Simplified Chinese, English
uko	EUC-KO	Korean, English

Abbrev:	Codeset:	Languages:
XVT/Win32:		
big5	Big-5	Traditional Chinese
gbc	GB-Code	Simplified Chinese
cns	CNS	Simplified Chinese
kcs	KCS	Korean
w50	Windows 1250	WINEE
w51	Windows 1251	WINCYR
w52	Windows 1252	ANSI
w53	Windows 1253	WINGREEK
w54	Windows 1254	WINTURK
w55	Windows 1255	WINHEB
w56	Windows 1256	WINARAB
w57	Windows 1257	WINBALT
XVT/Mac:		
mrmn	Mac-Roman	Roman-based languages
mce	Mac-CE	Central and Eastern Europe
mcro	Mac-Croatian	Croatian
mheb	Mac-Hebrew	Hebrew, Ladino, Yiddish
mcyr	Mac-Cyrillic	Belorussian, Bulgarian, Kazakh, Kirghiz, Macedonian, Moldavian, Russian, Serbian, Tadzhik, Turkmen, Ukrainian, Uzbek, Azerbaijani, Mongolian
mtha	Mac-Thai	Thai, Kuy, Lavna, Sanskrit, Pali
mara	Mac-Arabic	Arabic, Baluchi, Dari Persian, Farsi, Kurdish, Pashto, Sindhi, Urdu, Azerbaijani, Kashmiri, Malay
mice	Mac-Icelandic	Icelandic
mgre	Mac-Greek Greek,	Coptic
mtu	Mac-Turkish	Turkish
big5	Big-5	Traditional Chinese
kcs	KCS	Korean

B

UTILITIES

This appendix contains information about the following XVT utilities:

- String list (SLIST) functions
- I/O stream objects
- The NOREF macro

B.1. String List (SLIST) Functions

A string list is a linked list of zero or more NULL-terminated C character strings in some order. XVT refers to string lists with the type SLIST (whose actual definition is hidden). Each string is associated with a long word (32 bits) that can hold any data you want—typically, it holds a pointer. You cannot manipulate SLISTs directly, but XVT supplies several functions to do the manipulation for you.

Note: Multibyte strings can be used for all strings contained in SLISTs. For detailed information about functions that can be used to process strings in a multibyte-aware application, see section 19.2.5 on page 19-25.

Tip: To create an SLIST:

Call xvt_slist_create.

Tip: To dispose of an SLIST (freeing its memory):

Call xvt_slist_destroy.

Tip: To add a string (along with a long word of data) or another SLIST to an existing SLIST:

Call xvt_slist_add_at_elt.

Tip: To insert a new string (along with a word of data) into an existing SLIST in alphabetical order:

Call xvt_slist_add_sorted.

The strings already present in the existing SLIST should be in alphabetical order.

Tip: To add a string or SLIST at a given position:

Call xvt_slist_add_at_pos.

Tip: To remove a string from an SLIST:

Call xvt_slist_rem.

Tip: To count the number of elements in an SLIST:

Call xvt_slist_count.

Tip: To get a textual representation of an SLIST:

Call xvt_slist_debug.

This function writes its output to the same file used by xvt_debug and xvt_debug_printf.

Tip: To find out if you have a pointer that is pointing to an SLIST:

Call xvt_slist_is_valid.

SLIST_ELT Objects

Another kind of object, an SLIST_ELT, holds an element of an SLIST (string plus long word of data). Given an SLIST_ELT you can do the following:

Tip: To retrieve an SLIST_ELT's string and data:

Call xvt_slist_get.

Tip: To retrieve the string and data by numeric index (starting with 1):

Call xvt_slist_get_elt.

Tip: To loop through all elements of an SLIST:

Call xvt_slist_get_first and xvt_slist_get_next.

Tip: To change the data associated with an element:

Call xvt_slist_get_data.

SLISTs are commonly used to add strings to a list box or to retrieve the contents of a list box. They are also returned by the functions xvt_fsys_list_files, xvt_res_get_str_list, and xvt_scr_list_windows.

Note: SLISTs cannot contain carriage returns, line feeds, or new lines.

XVT automatically handles memory allocation and deallocation for SLIST_ELTs and for strings when you call the various functions mentioned above. However, if you use an element's long word to store a pointer, you are responsible for allocating and deallocating the memory to which the pointer is pointing. Also, if you are presented with an SLIST by a function such as xvt_list_get_sel, you are responsible for deallocating it (with xvt_slist_destroy) when you no longer need it.

Tip: Because the implementation of SLISTs involves much linear searching, it's inefficient to usethem for more than about a hundred elements. If you need to efficiently manage hundreds or thousands of elements, you'll have to design your own data structures using techniques such as hashing.

See Also: For more information about SLISTs, see SLIST in the *XVT Portability Toolkit Reference*.

B.2. The I/O Stream Object

The I/O stream object is an abstraction of an arbitrary data input/output stream ("stream" implies that individual bytes are *always* processed sequentially and that there is never any need for direct memory access—the classic example is the sequential text file, sometimes called a "byte stream").

XVT introduced the I/O stream object primarily as a vehicle for processing image information either from a file source or from an in-memory source (i.e., resource files and help system). Since image processing is independent of the data source and requires only sequential data access, XVT implemented a single image read function that gets its data from a single I/O stream source.

Tip: To create an I/O stream object for reading data from a file:

Call xvt_iostr_create_fread.

Tip: To create an I/O stream object for writing data to a file:

Call xvt_iostr_create_fwrite.

Tip: To create an I/O stream object for reading data:

Call xvt_iostr_create_read.

Tip: To create an I/O stream object for writing data:

Call xvt_iostr_create_write.

Tip: To get a pointer to the data stream context of an I/O stream object:

Call xvt_iostr_get_context.

Tip: To destroy an I/O stream object:

Call xvt_iostr_destroy.

B.3. NOREF

Tip: To establish a reference to a function's otherwise unused argument:

Call the NOREF macro.

The NOREF macro generates no code—it simply suppresses the warning given by some compilers when you fail to reference a variable. The NOREF statement must follow all other variable definitions.

GUIDE

INDEX

Symbols

A

D

E

F

I

J

K

L

M

N

O

P

Q

R

S

T

U

X

Y