ViewPoint Application Developer's Guide

XEROX

610E00200
September 1985

Xerox Corporation
Office Systems Division
2100 Geng Road
MS 5827
Palo Alto, California 94303

Copyright © 1985, Xerox Corporation. All rights reserved.
XEROX ®, 8010,and XDE are trademarks of XEROX CORPORATION.

Printed in U.S. A.

TABLE OF CONTENTS

Section I: Introduction to ViewPoint I-1

1.Overview 1-1

1.1 Reference documentation 1-1

1.2 Typographical considerations 1-2

1.3 Programming conventions 1-2

1.3.1 Resource management 1-3

1.3.2 Application termination 1-4

1.3.3 Multilingual considerations 1-4

2.User interface 2-1

2.1 The desktop and icons 2-1

2.2 Windows 2-2

2.3 Pop-up menus 2-4

2.4 Attention window 2-4

2.5 Form windows and property sheets 2-5

2.6 Wastebasket 2-6

2.7 The Directory icon 2-7

2.8 Documents, folders, and file drawers 2-8

Section II. Building an application II-1

3.Getting started 3-1

3.1 Logging on and off 3-1

3.2 The user profile 3-1

3.3 The workstation profile 3-1

4.Running an application 4-1

4.1 The programming cycle 4-1

4.2 CommandCentral 4-2

4.3 The SystemFolder 4-3

4.4 The ApplicationFolder 4-3

4.5 .autorun files 4-4

5.Strings and messages 5-1

5.1 XChar 5-1

5.2 XString 5-1

5.2.1 Readers and ReaderBodys 5-2

5.2.2 Writers and WriterBodys 5-6

5.3 XMessage 5-7

5.4 Attention 5-9

6.TIP 6-1

6.1 TIP tables 6-1

6.2 NotifyProcs 6-3

7.Creating a simple application 7-1

7.1 Attention menu 7-1

7.2 StarWindowShells 7-2

7.2.1 Creating the shell 7-3

7.2.2 Transition procedures 7-3

7.2.3 IsCloseLegalProc 7-4

7.2.4 Body windows 7-4

7.2.5 Displaying information in a window 7-5

7.2.6 Commands and menus 7-6

7.2.7 Displaying windows on the screen 7-7

7.3 Context 7-7

7.4 Complete example 7-9

8. Form windows and property sheets 8-1

8.1 Form window 8-1

8.1.1 Creating a form window 8-1

8.1.2 Creating form items 8-3

8.1.3 MakeCommandItem 8-3

8.1.4 Layout 8-4

8.1.5 Recognizing changes in the form window 8-6

8.1.6 Getting and setting values 8-7

8.1.7 Destroying a form window 8-7

8.2 Property sheets 8-7

8.2.1 Creating a property sheet 8-8

8.2.2 Linked property sheets 8-9

9. Icon applications 9-1

9.1 What is an icon? 9-1

9.2 File types and icon applications 9-1

9.3 Getting icons on the desktop: the Prototype folder 9-2

9.3.1 Opening the Prototype folder; the user's perspective 9-2

9.3.2 Putting icons in the Prototype folder: the programmer's perspective 9-3

9.4 Registering an application with the desktop 9-4

9.4.1 Picture procs 9-4

9.4.2 Generic procs 9-5

9.5 Examples 9-6

10. Selection 10-1

10.1 Some basic guidelines 10-1

10.2 Requestors 10-1

10.2.1 Can you convert the selection? 10-3

10.2.2 Enumerating selections 10-3

10.2.3 Resource allocation and deallocation 10-4

10.3 Managers 10-4

11. Packaging an application 11-1

11.1 Building an application folder 11-1

11.1.1 Identify the components of an application 11-1

11.1.2 Build the data files 11-2

11.1.3 Write an application description file 11-2

11.1.4 Change or add code to access data files 11-3

11.1.5 Integrate the components into an application folder 11-5

11.2 Code samples 11-6

11.2.1 Message files 11-6

11.2.2 Private icon file 11-8

11.2.3 Private TIP file 11-8

11.3 ADF syntax 11-9

Section III. Tools III-1

12. Icon editor 12-1

12.1 Getting started 12-1

12.2 Editing an icon 12-2

13. Message tools 13-1

13.1 Message Master File Creation tool 13-1

13.2 Message Master Editor 13-3

13.2.1 Searching Message Master files 13-3

13.2.2 Search parameters 13-5

13.2.3 Editing 13-5

13.2.4 Closing, saving, and resetting 13-8

13.2.5 Merging message files 13-8

13.2.6 Printing message files 13-9
13.2.7 Displaying message entries 13-9
13.3 Message file property sheet 13-11
13.4 Runtime File Creation tool 13-12

14. Bitmap edit tool 14-1
14.1 Getting started 14-1
14.2 Draw mode 14-2
14.3 Edit mode 14-2
14.3.1 Selection 14-2
14.3.2 Move, copy, and erase 14-3
14.3.3 Magnify 14-3
14.3.4 Loading and storing bitmaps and press files 14-5
14.4 Restrictions 14-5

15. Keyboard tool 15-1
15.1 Getting started 15-1
15.2 Keyboard file subwindow 15-1
15.3 Keyboard and bitmap subwindows 15-2
15.4 Key subwindow 15-5
15.5 How to edit a keyboard 15-7
15.5.1 Getting started 15-7
15.5.2 Editing the bitmap 15-7
15.5.3 Creating a new keyboard file 15-8
15.5.4 Changing the interpretation of the keys 15-8
15.5.5 Using a keyboard file 15-9
15.6 Restrictions 15-9

Appendices:
A. Boot switches A-1
B. Scavenging B-1
C. Maintenance panel codes C-1

Section I. Introduction

This section introduces you to the ViewPoint environment. It explains some basic programming conventions and highlights some of the features of ViewPoint. It also gives you an illustrated, step-by-step introduction to the user interface and office metaphors of ViewPoint.

1. OVERVIEW

This document introduces the basics of ViewPoint programming. It is a condensed version of the material contained in the *ViewPoint Programmer's Manual*. ViewPoint is both a set of applications programs and a set of interfaces that can be used to build new applications programs. Thus, a ViewPoint user can be either an end user or someone who uses the ViewPoint interfaces to create new applications programs. This manual is directed primarily at the latter kind of user: an applications programmer who is interested in adding new programs to the environment.

This document is divided into three sections. The first section discusses the philosophy behind ViewPoint and introduces the basic user interface features. This section will be of use to both end users and programmers. The second section focusses on how to write a new ViewPoint application. The chapters in this section build upon one another, starting with how to create a very simple application and progressing to how to create a polished application. The third section discusses some of the ViewPoint tools (application programs) that are available to help you build new applications.

The appendixes contain useful information that will help you with routine operations. Appenidix A lists switches that are consulted at boot time. Appendix B discusses how to scavenge the volume holding the desktop and the ViewPoint system files, and Appendix C lists codes that may appear on the maintenance panel of your hard disk during routine operations and describes any actions you should take if they appear.

This document assumes that you are familiar with the Mesa Language and the Xerox Development Environment, at least to the degree required to complete the Mesa Course. It does not assume any prior knowledge of ViewPoint. Some experience the Star or ViewPoint user interface would be helpful, although it is not necessary.

1.1 Reference documentation

Here is a list of other documentation that you might want to consult and a brief description of the area that it covers:

XDE User Guide	introduces the user interface used in the development environment and describes the available programming tools.

ViewPoint Series Reference Library	manuals for ViewPoint applications software
Mesa Language Manual	Reference guide for Mesa programming language.
ViewPoint Programmer's Guide	complete reference for ViewPoint programming, including descriptions of ViewPoint interfaces.

1.2 Typographical conventions

In this manual, the Mesa language is represented in **boldface** type. Comment lines within code appear in *italics*, as does input the user is expected to type. Commands that the user selects with the mouse are presented as regular text.

1.3 Programming conventions

This section introduces the system architecture and philosophy underlying ViewPoint. The ViewPoint architecture is open and flexible: ViewPoint makes no assumptions about the programs that will be running above it. In effect, it waits for applications to call it and state that they implement some facility. This is referred to as a plug–in approach: an application "plugs itself in" to the lower-level ViewPoint software.

Because ViewPoint knows nothing about the applications that will be running above it, and the applications know nothing about one another, a set of conventions has developed to facilitate cooperation among applications and to ensure a consistent user interface. Since ViewPoint assumes that all applications are friendly, there is no enforcement of these conventions; you are expected to follow them voluntarily.

The basic premise for writing an application is that the user should be free to interact with any application at any time. For example, the user is in charge of window layout and the current selection. Although it is easy to write an application that changes the selection or the window layout, this is strongly discouraged.

To avoid preempting the user, an application should only perform actions in response to an explicit user request. When a new application is loaded, it waits for ViewPoint to notify it when the user wants the application to do something. This is known as the "Don't call us, we'll call you" principle; it prevents programs from seizing control of the processor while getting user input. Figure 1.1 illustrates a sample ViewPoint application. When this application is loaded, the window appears on the screen, but the program does not actually do anything. Instead, it waits for the user to invoke one of its commands.

Instead of a main procedure that calls subroutines, therefore, each application program contains an initialization procedure and individual command execution routines. Loading an

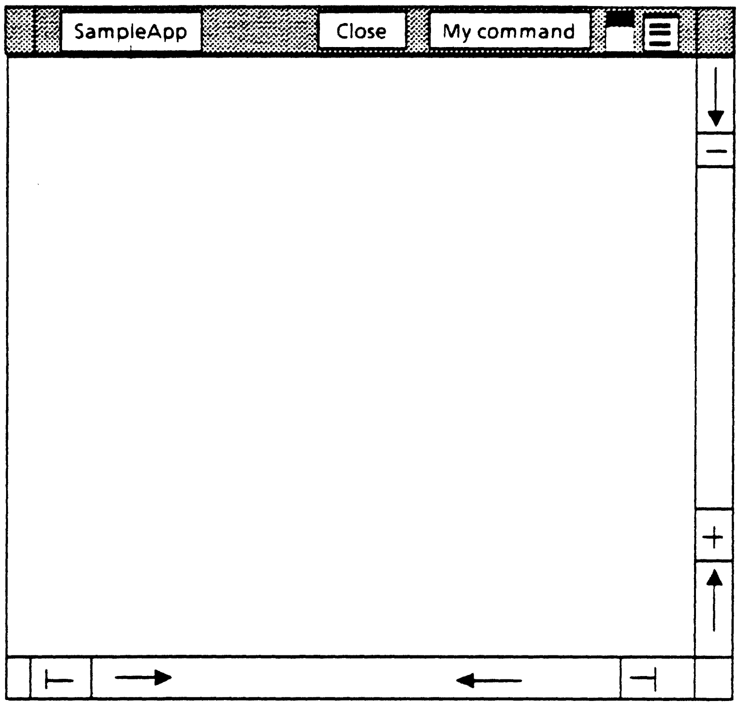


Figure 1.1 A typical window

application program calls its initialization procedure, which registers the available commands with the system and provides *call-back* procedures that are to be called when the user invokes those commands. When the program is fully initialized, control returns to the system. Thus, a application program simply provides a set of functions and arranges for ViewPont to notify it when the user wants it to perform some action. This idea will be discussed more fully in Chapter 7, Creating a Simple Application.

To simplify application development, as well as to encourage a consistent user interface, ViewPoint provides programming interfaces to support its primary user interface characteristics, such as icons, windows, and property sheets. This makes life easier both for the programmer and for the end user, since the programmer doesn't have to write his own user interface code, and the user is presented with a consistent user interface. The user interface is discussed in greater detail in Chapter 2, User Interface.

1.3.1 Resource Management

ViewPoint programs must explicitly manage resources. For example, applications must explicitly allocate and deallocate memory; there is no garbage collector to reclaim unused memory. All programs share the same pool of resources, and there is no scheduler watching for programs using more than their share of execution time, memory, or any other resource.

When interfaces exchange resources, programsmust be very careful about who is responsible for the resource. The program that is responsible for the deallocation of a resource is the *owner* of that resource. One example of a resource is a file handle. If a program passes a file handle to another program, both programs must agree about who owns that file handle. Did the caller transfer ownership by passing the file handle, or is it retaining ownership and only letting the called procedure use the file handle? If there is disagreement between the two programs, either the file will be released twice, or it will never be released at all. All interfaces involving resources must state explicitly whether ownership is transferred.

1.3.2 Application termination

The ViewPoint environment consists of cooperating processes. There are no facilities for cleanly terminating an arbitrary collection of processes. You are expected to design your tools to stop voluntarily when asked to do so.

An application should stop if the user aborts the application. There are two ways to determine if the user has aborted an application. An application's window can have a **TIP.AttentionProc** that will be called as soon as the user presses the STOP key. Or, procedures in the Terminal Interface Package (**TIP**) can check whether a user has aborted an application with the STOP key in the application's window. (Note: The TIP package is responsible for processing user input such as mouse clicks and keystrokes. See Chapter 6, TIP, for details) An application should check for a user abort at frequent intervals and be prepared to stop executing and clean up after itself. Because the application controls when it checks, it can check at points in its execution when its state is easy to clean up. Packages that can be called from several programs should take a procedure parameter that can be called to see whether the user has aborted.

1.3.3 Multilingual considerations

ViewPoint is designed so that applications can easily be translated into other languages.The ViewPoint string package supports the Xerox Character Code Standard, which allows strings in many languages to be intermixed. Other facilities support the translation of user messages into other languages by encouraging the application programmer to put all these messages into a module separate from the rest of the application code. The messages can thus be edited or translated without recompiling the code itself.

Application programmers are strongly encouraged to allow their application to be multilingual. This means that you should use the ViewPoint string facilities, and it also means that you should not make any language assumptions about characters received from the user. An application that expects typing input from the user should be prepared to receive characters from *any* character set. We discuss the string and message facilities more fully in Chapter 5, Strings and Messages.

2. USER INTERFACE

This chapter describes the ViewPoint user interface, which is based on the metaphor of a physical office. The user interface includes symbols for standard components of the business office, such as the desktop, folders, file drawers, baskets for incoming and outgoing mail, and wastebaskets. This metaphor provides a user interface that is consistent, intuitive, and easy to use.

In addition, ViewPoint also provides programming interfaces to support these user interface characteristics. By using these facilities when you write new applications, you ensure that your applications integrate well with existing software. This chapter describes the user interface; the rest of this document describes how to use the ViewPoint interfaces to incorporate these user interface features into a new application.

2.1 The Desktop and icons

The ViewPoint user interface is based on the idea of *icons* that reside on a *desktop*. The desktop represents the typical business office; an icon is a pictorial representation of a ViewPoint object. A typical desktop might include icons that represent various documents, folders, spreadsheets, mail baskets, a printer, and other such objects.

The user accesses the object through the icon, generally by selecting the icon and pressing the OPEN key. The user can also use the MOVE, COPY, and DELETE keys on icons.

When it first appears, the desktop looks like a gray pattern that occupies the entire display. Internally, the surface is organized as an array of 1-inch squares, each of which can contain an icon. Figure 2.1 illustrates a desktop with several different icons and one open document.

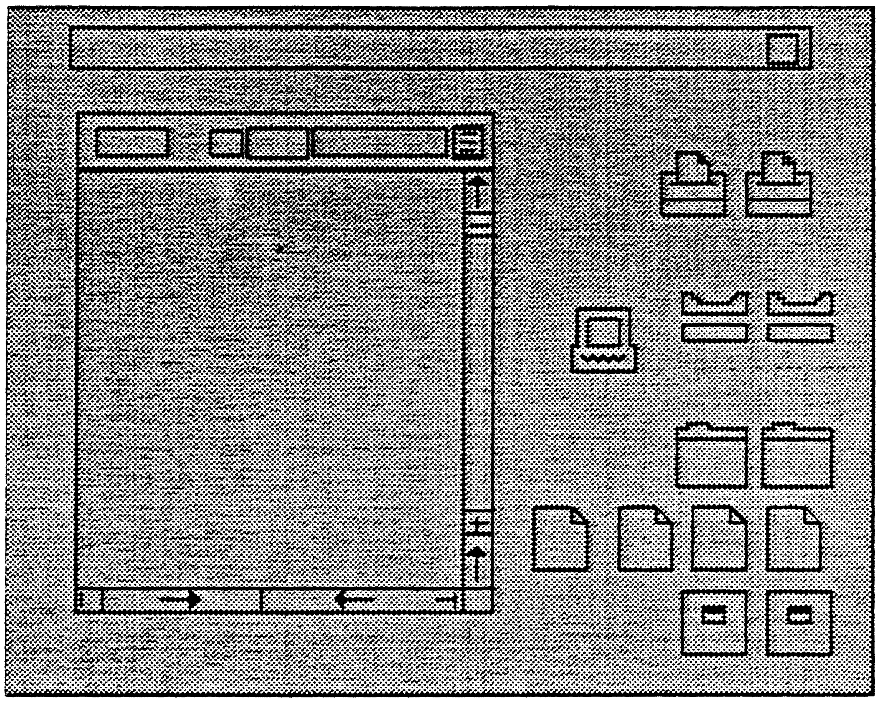


Figure 2.1 Icons on a desktop

The user can move the icons to different positions on the surface, but two icons cannot occupy the same square at the same time. If the user creates more icons than can fit on the screen at one time, a message appears indicating the number of undisplayed icons and the Move Undisplayed Icons command appears in the desktop pop-up menu. The user should delete the icons that are not needed and then select the Move Undisplayed Icons command; the system places the undisplayed icons on the empty spaces.

The use of icons is simple and intuitive. Therefore, you should use icons to represent those applications that the end user will access frequently. However, since associating an icon with an application requires a fair amount of programming overhead and the icon itself uses screen real estate, icons are not a cost-effective or efficient way of representing simple, infrequently used applications. Instead, you can have such applications run from a command in a global pop-up menu, as described in the next section.

For more information on writing applications that use icons, see Chapter 9, Icon Applications.

2.2 Windows

A *window* is a rectangular region of the display screen in which an application can display information to the user. Windows have the following attributes, as illustrated in Figure 2.2:

Borders

Header

Title

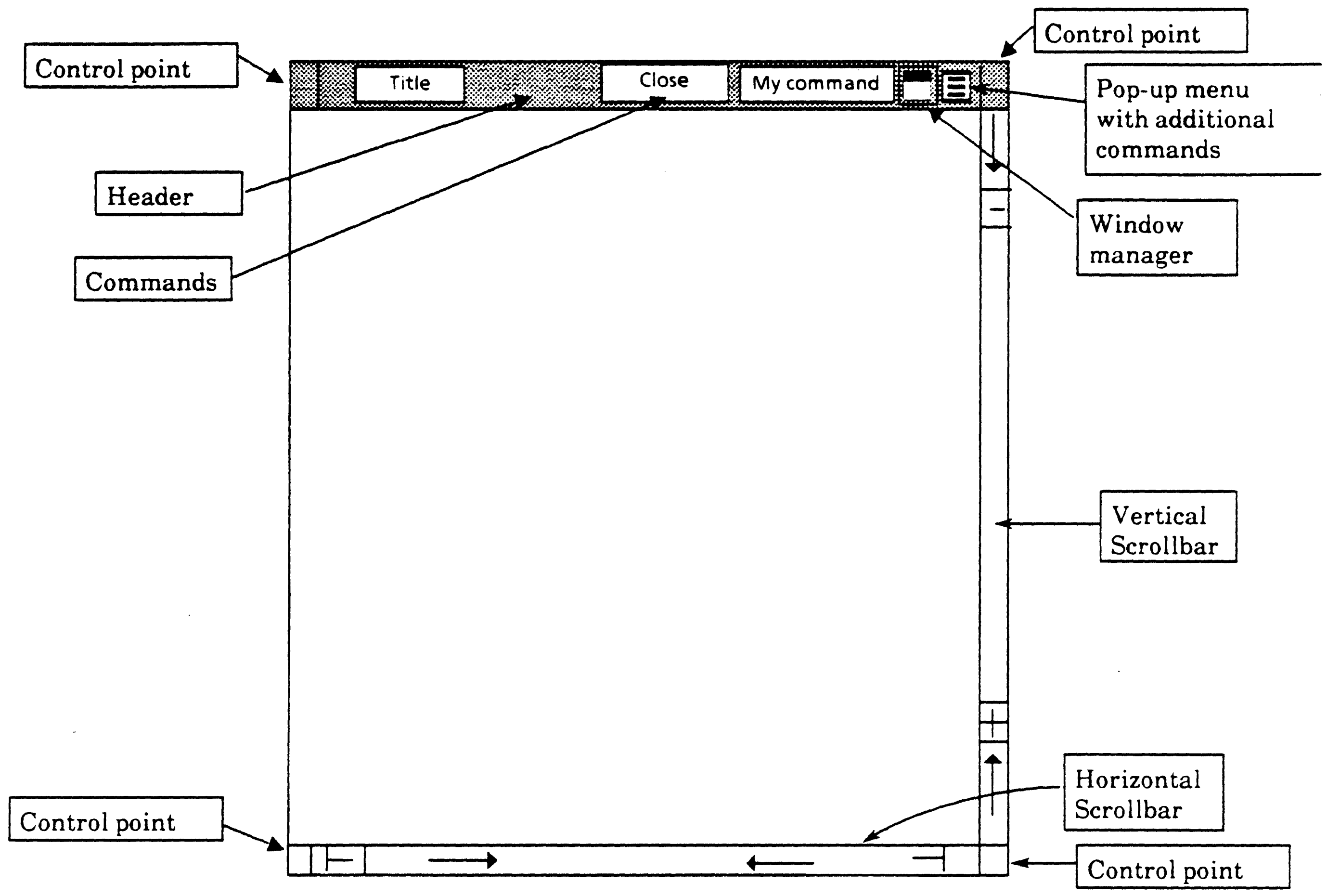


Figure 2.2 A basic window

Commands (visible and in the pop-up menu)

Scroll bars

Window control points

Windows can be in one of two modes: *overlapping* mode or *tiled* mode. In overlapping mode, windows can appear on top of each other; there is no limit to the number of windows that can appear. In tiled mode, each window occupies its own section of the screen, and there is no overlap. Windows must be in one mode or the other; you cannot have some windows in overlapping mode and others in tiled mode.

Initially, windows are in overlapping mode: each window has a single-line header and a control point in each corner. Pressing POINT (the left mouse button) in any control point invokes a Top/Bottom operation. Pressing POINT down in any control point and then moving the mouse moves the entire window. Pressing ADJUST (the right mouse button) in any control point and then moving the mouse resizes the window.

You can specify whether overlapping windows employ *simple offset, repeat offset,* or *don't offset. Simple offset* means that up to six windows can appear at one time, starting at the upper left and going to the lower right. The seventh window appears on top of the first window and the same pattern continues for each succeeding window. If you close a window and then re-open it, the system remembers the window's initial position and redisplays it in that position. *Repeat offset* opens windows in the same way as simple offset. However, if you close and then reopen a window, the system does not remember the initial location of the window, but rather places it in the first available position. With *don't Offset*, there is no rigid ordering; windows can appear anywhere on the screen.

When windows are in tiled mode, no more than six windows can appear on the screen at one time. You cannot move a tiled window on top of another tiled window. You can only move it to an empty space on the screen.

To switch between overlapping and tiled mode and between *simple, repeat,* and *don't offset*, you can either use the Window Management property sheet or edit the User Profile. The Window Management property sheet is available through the Attention Window menu; it specifies whether windows appear overlapping or tiled and with single- or double-line headers.

If you want to change the defaults for these parameters, you can edit the User Profile. (For a complete discussion of the User Profile, see Chapter 3.) Here is an example of a User Profile entry for window characteristics:

```
[Windows]
Arrangement: overlapping   --or tiled
Header Style: single line  -- or double line
Placement: simple offset   -- or repeat or don't offset
```

2.3 Pop-up menus

A *menu* is a list of named commands. A *pop-up menu* is a menu that appears only when the user specifically requests it by holding down the left mouse button over the pop-up menu symbol (≡). Each application generally has a pop-up menu; the author of the application chooses which commands go directly in the header and which go in the pop-up menu. Using pop-up menus conserves screen space while the menus are not in use, but means that the commands are not readily visible and that the user must go through an extra step to access a command.

For more information, see Chapter 7, Creating a Simple Application.

2.4 Attention window

The Attention window is the window that appears across the top of your screen. The Attention window has an associated pop-up menu with a list of system-wide commands. The Attention window also allows applications to display messages to the user. Figure 2.3 illustrates the Attention window and its associated menu. (The Attention window is also shown in Figure 2.1.)

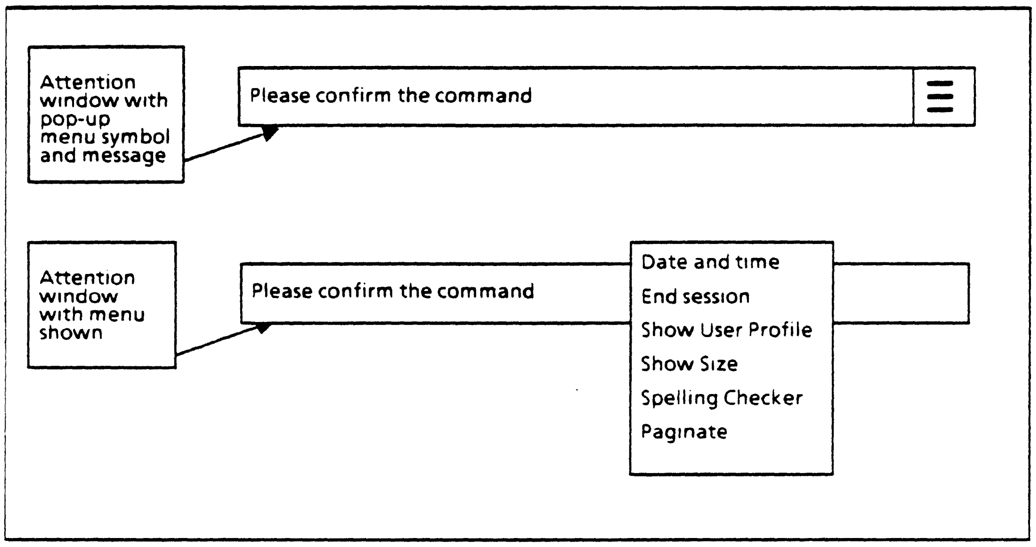

Attention window with pop-up menu symbol and message
Please confirm the command
Attention window with menu shown
Please confirm the command
Date and time
End session
Show User Profile
Show Size
Spelling Checker
Paginate

Figure 2.3 Attention window

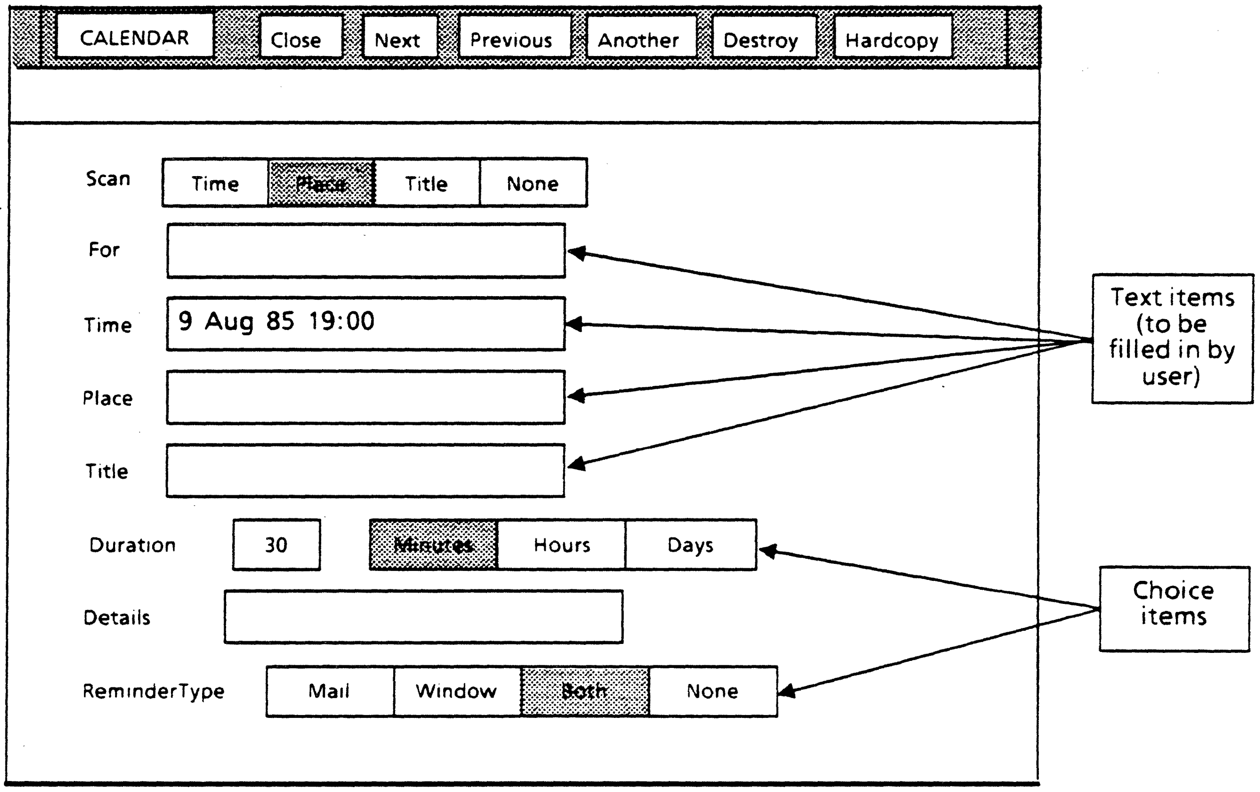
You can access a standard set of commands available from the Attention menu, and applications can add additional commands to this menu. For example, many applications run from a command in the Attention window menu rather than from an icon. Thus, when the user wants an application's window to appear, he invokes the appropriate command from the Attention window menu, instead of selecting an icon and opening it. As a programmer, you get to choose whether your application runs from an icon or from a command in the Attention window. Placing commands in the Attention Window menu conserves screen space, but makes them less accessible than icons.

Note that to prevent possible "scrambling" of messages, only one process can post messages to the Attention window at a given time. For more information on the Notifier or on posting messages to the Attention window, see Chapter 5, Strings and Messages. For more information on writing applications that run from a command in the Attention window menu, see Chapter 7, Creating a Simple Application.

2.5 Form windows and property sheets

A Form window is a window that displays one or more *items*. There are many types of items, the most common of which are boolean, choice (enumerated), and text. The user can observe the current value of each item in the form and change that value if he desires. Figure 2.4 illustrates a form window for a calendar application.

A property sheet is a Form window in which the items control the *properties* of an object. Different objects have different properties; for example, the properties of a paragraph include left and right margins, justification, and line height. The head of a property sheet can contain only the standard commands: ? (Help), Done, Cancel, Apply, and Defaults. Property sheets provide a consistent method of viewing and changing object



Scan: Time [✓ Place] Title None

For:

Time: 9 Aug 85 19:00

Place:

Title:

Duration: 30 [✓ Minutes] Hours Days

Details:

ReminderType: Mail Window [✓ Both] None

Figure 2.4. Sample Form window

attributes for all ViewPoint objects. To see the properties of an object, press the PROPS key.

For more information, see Chapter 8, Form Windows and Property Sheets.

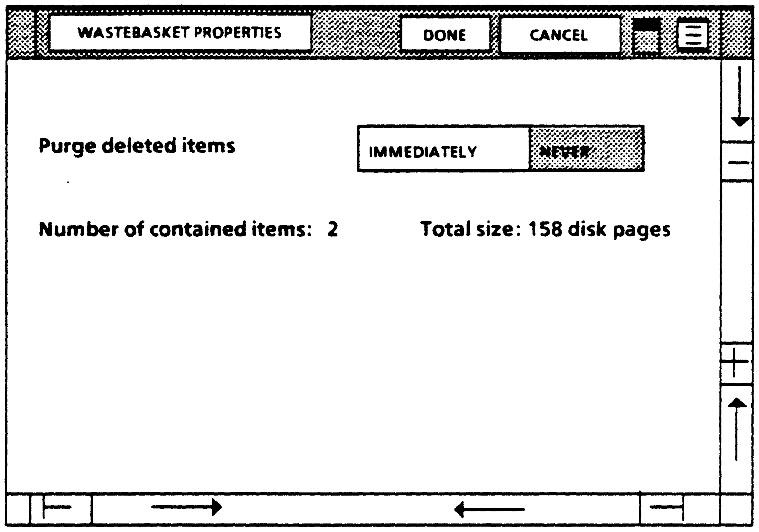
2.6 Wastebasket

The Wastebasket allows the user to delete icons from the desktop by moving the icon and depositing it in the Wastebasket icon.

The Wastebasket icon has an associated property sheet that allows the user to specify whether the deposisted icons are purged immediately or simply stored, as illustrated in Figure 2.5. If the items are stored, the user must invoke the Purge Wastebasket command from the Attention window.

The Wastebasket icon also has an associated window that appears when the user presses the OPEN key. This displays the object name and the time and date that the last object was moved to the Wastebasket, which allows the user to retrieve objects deposited in the wastebasket. If the user specifies immediate purging, however, the Wastebasket window is always empty and retrieval is therefore not possible.

Note that if the user specifies "Never" as the purge mode, any objects he deletes from his mail in-basket will also go to his Wastebasket. Also note that if two or more Wastebaskets are



Purge deleted items IMMEDIATELY [✓ NEVER]

Number of contained items: 2 Total size: 158 disk pages

Figure 2.5 Property sheet for the Wastebasket icon

set up, they all point to the same Wastebasket file inside the directory. A change to the property sheet for one Wastebasket affects all of them.

2.7 The Directory icon

The Directory icon is on every ViewPoint desktop. This icon provides access to various ViewPoint applications and features. Opening the Directory icon provides three choices: Workstation, User, and Network. The Workstation category contains workstation-specific items, such as blank documents, the Converter, and the Loader. (The Converter allows you to convert documents from other formats, such as ASCII, into ViewPoint documents. The Loader, which allows you to load new ViewPoint applications, is discussed more fully in Chapter 4, Running an Application.) The User category contains user-specific items, such as mail in and out baskets, a Wastebasket, and the User Profile. The Network category provides access to icons for remote servers, such as printers and file drawers. You can copy icons out of the Directory as needed.

2.8 Documents, folders, and file drawers

ViewPoint's document handling is based on *documents*, *folders*, and *file drawers*, just as in the typical business office. A *document* is a metaphor for a piece of paper. You can edit documents, create new documents, file documents, and so forth. A *folder* is a metaphor for a physical folder: you can store vartious documents within a folder either on the desktop or in a file drawer. A *file drawer* represents a a physical file drawer: it allows you to access additional storage space on a remote server and organize that space much like a standard file cabinet. For more information on document editing, folder properties, or using file drawers, consult the appropriate sections of the *ViewPoint Series Reference Library* and the *ViewPoint Series Training Guides*.

Section II. Building an Application

This section describes the process of building an application. It details the interaction of your application with the underlying ViewPoint system, introduces you to some of the more common ViewPoint interfaces, illustrates the use of procedures from those interfaces, and explains basic ViewPoint programming concepts. It includes many examples of code to illustrate ViewPoint programming.

3. GETTING STARTED

This chapter presents some of the information you will need to get started using your ViewPoint system. It provides only developer-specific information, which means that it does not cover general topics such as setting up a desktop. If you don't know how to retrieve icons for your initial desktop, consult the appropriate sections of the *ViewPoint Series Reference Library* and *Training Guides*.

3.1 Logging on and off

Before you can access the items on your desktop, you must first log on using the logon window. To get this window on a newly booted system, press the STOP key to stop the bouncing symbol, and the logon window will automatically open. Enter your name and password as registered with the authentication service, and select START or press NEXT. If you already have created a desktop on the machine, you will be logged in to that desktop; if you don't have a desktop, you will be given the opportunity to create a new one.

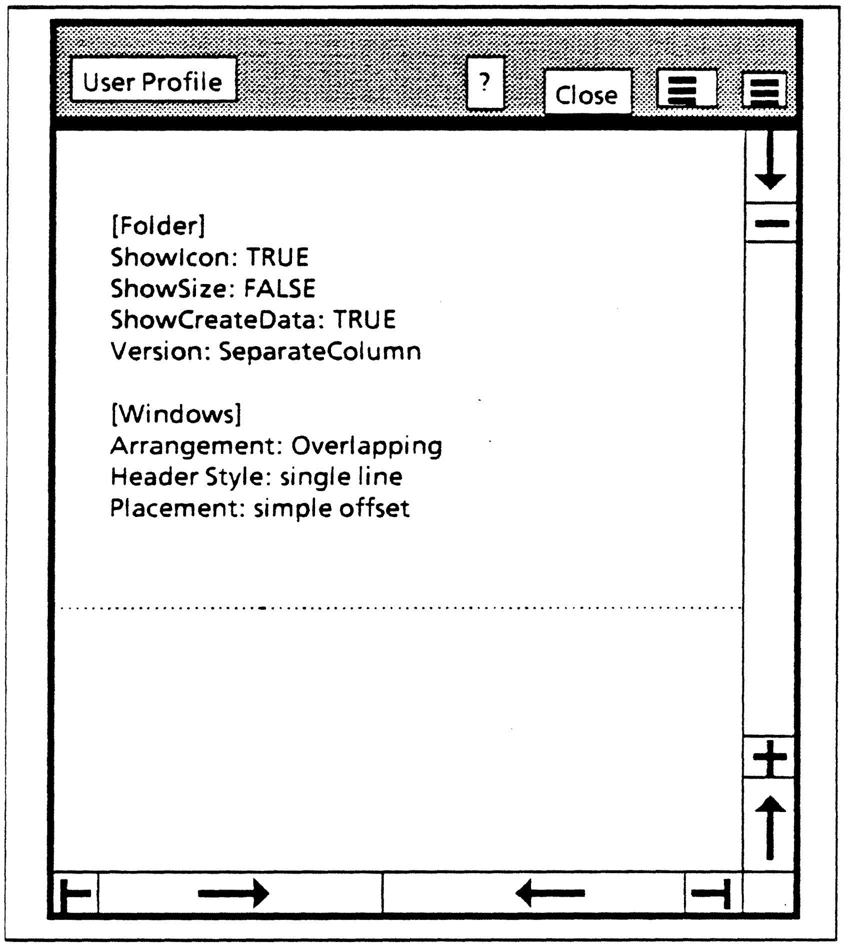
To log off, invoke the End Session command from the Attention window menu. This will bring up a property sheet, from which you can choose whether to delete your desktop, store it on a file server, or retain it locally.

3.2 The User Profile

The User Profile is used to customize the desktop by specifying user options. The information in the User Profile usually represents user-selected default values for specified applications. At start-up, many applications look in the User Profile for a relevant section and initialize designated values accordingly. As a user, you can find out what User Profile information an application recognizes by consulting its documentation. As a programmer, you can write your application so that it reads the User Profile by using the **OptionFile** interface. See the *ViewPoint Programmer's Manual* for more information.

To look at the User Profile, select the Show User Profile command from the Attention window menu. Alternatively you can open the Directory icon, then the User folder, and finally the User Profile. Figure 3.1 is an illustration of a User Profile. (Because of space constraints, the Edit command is not visible in the illustration).

The **SimpleEditor**, which is exactly what its name implies, is available with the User Profile window. To edit, just select the



```
[Folder]
ShowIcon: TRUE
ShowSize: FALSE
ShowCreateData: TRUE
Version: SeparateColumn

[Windows]
Arrangement: Overlapping
Header Style: single line
Placement: simple offset
```

Figure 3.1 A User Profile

Edit command in the header of the window. Here is an example of a User Profile that illustrates the syntax:

> **[Section]**
> **StringEntry: This is a string entry value**
> **BooleanEntry:** TRUE -- *or FALSE*
> **IntegerEntry: 12345**

The section and entry names can have spaces in them. The square brackets around the section name, the colon following the entry name, the double dashes introducing a comment, and the carriage return at the end of each entry are significant. Warning: putting a comment after a section heading causes the entire section to be treated as a comment. For example:

> **[Windows]** -- *This comments out the* **Windows** *section*
> **Arrangement: Overlapping**

3.2 The Workstation Profile

The *Workstation Profile* is similar to the User Profile, except that it is intended for selecting parameters forthe workstation rather than the desktop. There is one Workstation Profile per workstation, whereas there is one User Profile per desktop. The

Workstation Profile is similar to the User Profile in that it has the same syntax, its contents can be modified by opening its window and invoking the Edit command, and it can be programatically accessed via the **OptionFile** interface.

To modify the contents of the Workstation Profile, you must first have the **SystemFolder** application running (see Chapter 4 for an explanation of how to run it). Select the Attention window menu System Folder command to open the System folder. Look for the Workstation Profile in the folder, then select and open it. You can then select the Edit command to modify its contents.

A default Workstation Profile is provided on your desktop when you first install ViewPoint.

4. RUNNING AN APPLICATION

The ViewPoint programming cycle involves both the Xerox Development Environment (XDE) as a development environment and ViewPoint as the target environment, introducing dependencies between the two. In this chapter we discuss those dependencies and the related issue of how to run a ViewPoint application.

4.1 The programming cycle

As a ViewPoint programmer, you will actually spend a fair amount of time in XDE developing and debugging your applications. A typical programming cycle involves the following steps, repeated as many times as necessary:

1. Write, edit and compile a program in XDE
2. Copy the object code to ViewPoint and execute it
3. Debug with the XDE debugger.

You can copy the code from XDE to ViewPoint with XDE's CommandCentral, or you can put the code in a remote file drawer from XDE and then retrieve it from ViewPoint. During development, it is most convenient to use CommandCentral. Once your application is stable, you should put it on a file server so that it can be run directly from ViewPoint. (See the *XDE User's Guide* for more information on CommandCentral, the debugger, the editor, the compiler, and the binder.)

A few words of warning about running ViewPoint applications: generally, there is no visible indication that a program has been started. Since an application can be accessed either from a menu command (see Chapter 7, Creating a Simple Application) or through an icon (see Chapter 9, Icon Applications), starting your program will either make an icon available in a designated folder or generate a menu command. If the application runs from a command in the Attention menu, you need to bring up that menu to see the command. If the application runs from an icon, you need to copy that icon from the Prototypes folder onto your desktop. You access this folder by opening the Directory icon, then the Workstation folder, and then the Basic Documents, Folders, and Record Files folder (which is the Prototypes folder.) Once you have copied an icon onto your desktop, it will remain there. The next time you reboot, you need to run the application to activate the icon, but you won't have to retrieve the icon from the Prototypes folder.

4.2 CommandCentral

XDE's CommandCentral is the inter-volume link that allows you to transfer programs developed in XDE to ViewPoint for testing. Before we explain the details of how this works, we present some important points about ViewPoint's volume and file structure.

The *system volume* is a ViewPoint volume that contains the installed ViewPoint boot file. The *data volume* is a ViewPoint volume that contains the local filing system. (Frequently the system and data volumes are the same, but they need not be.) Residing on the data volume in the *System directory* are data files, which can be anything from executable programs to auxiliary files such as font or icon files. Files in the System directory are accessible through the **SystemFolder** application (described below).

XDE's CommandCentral, in cooperation with the ViewPoint boot file, will copy files over to the System directory of the data volume and then start them. The steps are:

1. In XDE, use the CommandCentral option sheet to set the correct client volume, which is your ViewPoint System volume, or the volume containing the ViewPoint boot file. Also set appropriate client switches at this time.

2. Put the names of your programs and data files, if any, on the Run: line in CommandCentral's form subwindow. Since data files are auxiliary files not meant for execution, follow the names of data files by the */-e* switch, which tells CommandCentral to copy them to the System directory, but not to execute them. (Meaningless if used with /-c; see below.)

3. Invoke the **Run!** command in CommandCentral. ViewPoint will be booted and the specified files will be copied to ViewPoint's System directory; those not accompanied by the */-e* switch will be started before the ViewPoint bouncing keyboard is displayed.

In addition to the switches documented in the CommandCentral chapter of the *XDE User's Guide*, the ViewPoint boot file recognizes the following Run line switches:

/! Causes a return to XDE with the message "Nub: ! switch detected". This switch is useful when you want to install data files such as system icon files, TIP tables, and others and not boot ViewPoint until they have been installed. Since you will probably install ViewPoint with scripts provided by Xerox, it is not likely that you will need this switch.

/c Unconditionally copy the file to the System directory. */-c* means do *not* copy the file. */c* is the default.

/p make parallel loading activity pause while this program is being started. Used in conjunction with *P* boot switch, explained in Appendix A. /-p is the default.

--Example:
Run: FooImplThatImportsBaz/p BazImplThatImportsFoo

/u Copy the file to the ViewPoint volume if it is newer than the version already there, or if there is no such file there. /-u is the default.

/# Copy the file if it is different than the version already there or if there is no such file there. /-# is the default.

If you want to get back to XDE at any time, simultaneously depress both SHIFT keys and the STOP key.

4.3 The SystemFolder

The **SystemFolder** application provides access to all files in the System directory. If you do not run this application, files will be in your System directory, but you will not have any way of accessing them. When run, **SystemFolder** registers the System Folder command in the Attention window menu.

Invoking the System Folder command opens a window showing the contents of the System folder, including object files, TIP files, font files, and icon picture files. (Files are put into the System folder by copying them from XDE using CommandCentral.)

SystemFolder also registers an auxiliary command: Set System Folder Filter. Invoking this command from the Attention menu displays a small property sheet with a single text item. If you enter a wildcard string into the text field and select the Done command, the next time you open the System folder, it will be filtered by the string you typed. For example, if you only want to see a list of object files, enter the string "*.bcd" in the text field of the filter property sheet and then open the System folder via the System Folder menu command. This time you will see only the files that end with the "*.bcd" extension.

The **SystemFolder** application registers a third command, Prototype Folder, which provides easy access to the Prototypes folder.

4.4 The Application Loader

Copying files from XDE via CommandCentral is one way to run an application; the other way is to use the Application Loader to load and start programs directly from ViewPoint. To use the Application Loader, you must have a Loader icon on your desktop. If you don't, open the Directory icon, and then the User folder. Inside the User folder you will find the Loader icon; copy it to your desktop. You can then copy or move object code icons (bcds) or application icons to the Application Loader for subsequent loading and starting, with associated feedback appearing in the Attention window.

Using the Application Loader in conjunction with the **SystemFolder** application makes it very easy to load files that are in the System directory. You just open the System folder, select the desired files, and move or copy them to the Loader icon.

You can also load applications directly from remote file drawers. To do so, just open the file drawer, select the application, and copy it either directly to the Loader or onto your desktop.

Opening the Loader icon will show all the applications on the workstation and their status; that is, whether they are idle or running. An additional way of running a program that is on the desktop but not yet started is to select it from within the open Loader icon and select the Run command in the header of the Loader window.

You should note, however, that the term *application* is a loose one; there is actually a difference between a file of obect code and something called an *application folder*. An application folder is a complete application; it always contains at least one bcd file, but it can also contain other items such as information on the picture that will appear on the icon, messages that the application will post to the user, and other supplementary information. A bcd file is a single file of object code. Application folders represent finished applications; bcds often represent applications that are still under development. Thus, "standard" applications such as the document editor are actually application folders; applications with the extension .bcd are object files. (See Chapter 11, Packaging an Application, for more details on application folders.)

This distinction is important because the Loader looks for the following entry in the Workstation Profile: (See Chapter 3, Getting Started, for more information on the Workstation Profile):

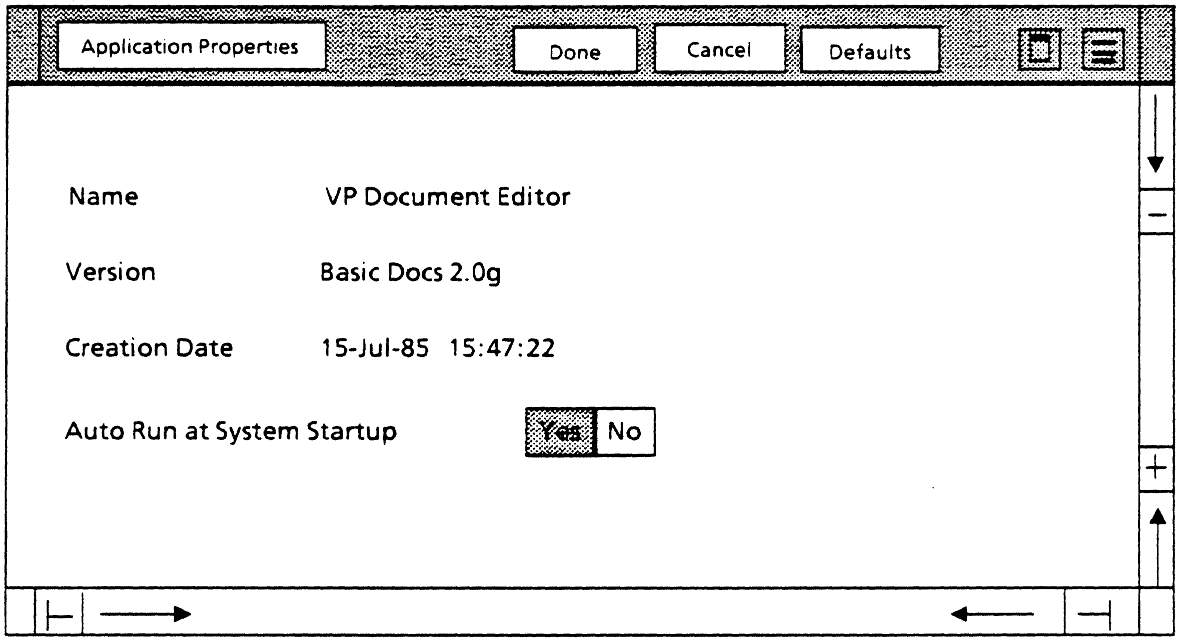
[Application Loader]
Developer: TRUE (or FALSE)

If the **Developer** value is **TRUE**, the opened Loader icon will show application folders and bcds. If **Developer** is **FALSE**, it will only display application folders.

4.5 .autorun files

At boot time, the loader looks in the system catalog for files with an extension of .autorun and automatically loads and starts any files with that extension. Thus, commonly used tools, such as **SystemFolder**, usually have the .autorun extension. To change a file's extension to .autorun, either name it that way in XDE and use CommandCentral to copy it into the System folder, or change its name in ViewPoint by selecting it within the System folder and modifying its name via its property sheet. If you rename the file from XDE and use CommandCentral to copy it to ViewPoint, you must use the /-e client switch in CommandCentral. If you don't, ViewPoint will attempt to start it twice, which will cause problems.

You can also use property sheets to specify that an application should be run automatically. To do this, just open the Loader icon, select the name of the application from the list, and press the PROPS key. This will bring up a property sheet that allows you to specify whether or not the application should be run automatically. For example, if you want to have the ViewPoint document editor run each time you boot ViewPoint, you would set up the property sheet as illustrated in Figure 4.1. (Note: this only works with application folders.)



Name	VP Document Editor
Version	Basic Docs 2.0g
Creation Date	15-Jul-85 15:47:22
Auto Run at System Startup	[✓ Yes] No

Figure 4.1 Property sheet for the document editor

Note also that there are built-in applications that are always run automatically. Such applications are not the same as .autorun applications, because you do not get to choose whether they are run. Such applications are referred to as *invisible applications*, because they appear even when not explicitly run. The Wastebasket and the Directory are examples of invisible applications.

5. STRINGS AND MESSAGES

This chapter introduces some of the underpinnings of ViewPoint programming: how characters are represented, how strings of characters are represented, and how messages are directed to the user.

5.1 XChar

For a system to be truly multilingual, it must have a character representation that allows for a tremendous number of different characters. English-only systems usually represent characters with either a 7-bit code (ASCII), or an 8-bit code (ISO). An 8-bit code allows 256 characters, which is plenty for English and associated special characters, but not nearly enough for multilingual capability.

The Xerox solution to this problem is a character encoding system (The *Xerox Character Code Standard*) that normally conforms to the ASCII and ISO 8-bit character codes, but expands to a 16-bit code when necessary. Defining a character as 16 bits provides 65,536 distinct characters; reserving space for control characters reduces it to 35,532. This 35,532 range is partitioned into 256 blocks (*character sets*) of 256 character codes each. Thus, each character is composed of two 8-bit quantities: a character set and a character code. The character set is optional; it can be omitted when a special character set is not required. When no character set is specified, the character code conforms to the ASCII and ISO codes. This approach thus provides both versatilty and compactness.

The **XChar** interface defines the basic character type and some operations on that character type.

```
XChar.Character: TYPE = WORD;

XChar.CharRep: TYPE = MACHINE DEPENDENT RECORD [
   set, code: Environment.byte];
```

5.2 XString

The **XString** interface provides data structures and operations for strings of characters encoded by the Character Code Standard. Again, multilingual considerations make the **XString** design somewhat different from "standard" string packages. In particular, **XString** declares two kinds of strings: one for reading ("reader") and one for writing ("writer"). The basic idea is that examining and manipulating existing strings is fundamentally different than building and creating new strings, and that most strings do not need to be changed once they are created. Thus, since readers use less space than writers, programs that only examine strings can save signficant space.

5.2.1 Readers and ReaderBodys

XString defines the following types for read-only strings:

```
XString.Reader: TYPE = LONG POINTER TO XString.ReaderBody;

XString.ReaderBody: TYPE = PRIVATE MACHINE DEPENDENT RECORD[
   context(0): XString.Context,
   limit(1): CARDINAL,
   offset(2): CARDINAL,
   bytes(3): XString.ReadOnlyBytes];

XString.Context: TYPE = MACHINE DEPENDENT RECORD [
   suffixSize(0:0..6): [1..2], --bit positions 0-6 in word 0
   homogeneous(0:7..7): BOOLEAN,
   prefix(0:8..15): Byte];

XString.ReadOnlyBytes: TYPE =
   LONG POINTER TO READONLY XString.ByteSequence;

XString.ByteSequence: TYPE = RECORD [
   PACKED SEQUENCE COMPUTED CARDINAL OF XString.Byte];

XString.Byte: TYPE = Environment.Byte;
```

The basic structure is the sequence of bytes pointed to by **bytes**. **limit** is the offset from the pointer to the byte after the last byte in the byte sequence (the "length" of the string); and **offset** is the offset from the pointer to the first byte (the "beginning" of the string). These fields are illustrated in Figure 5.1.

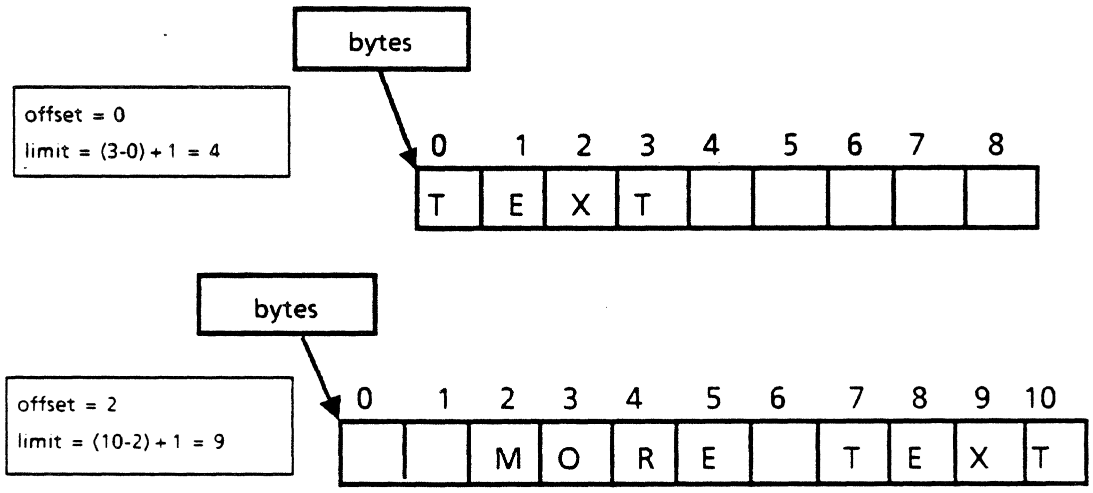
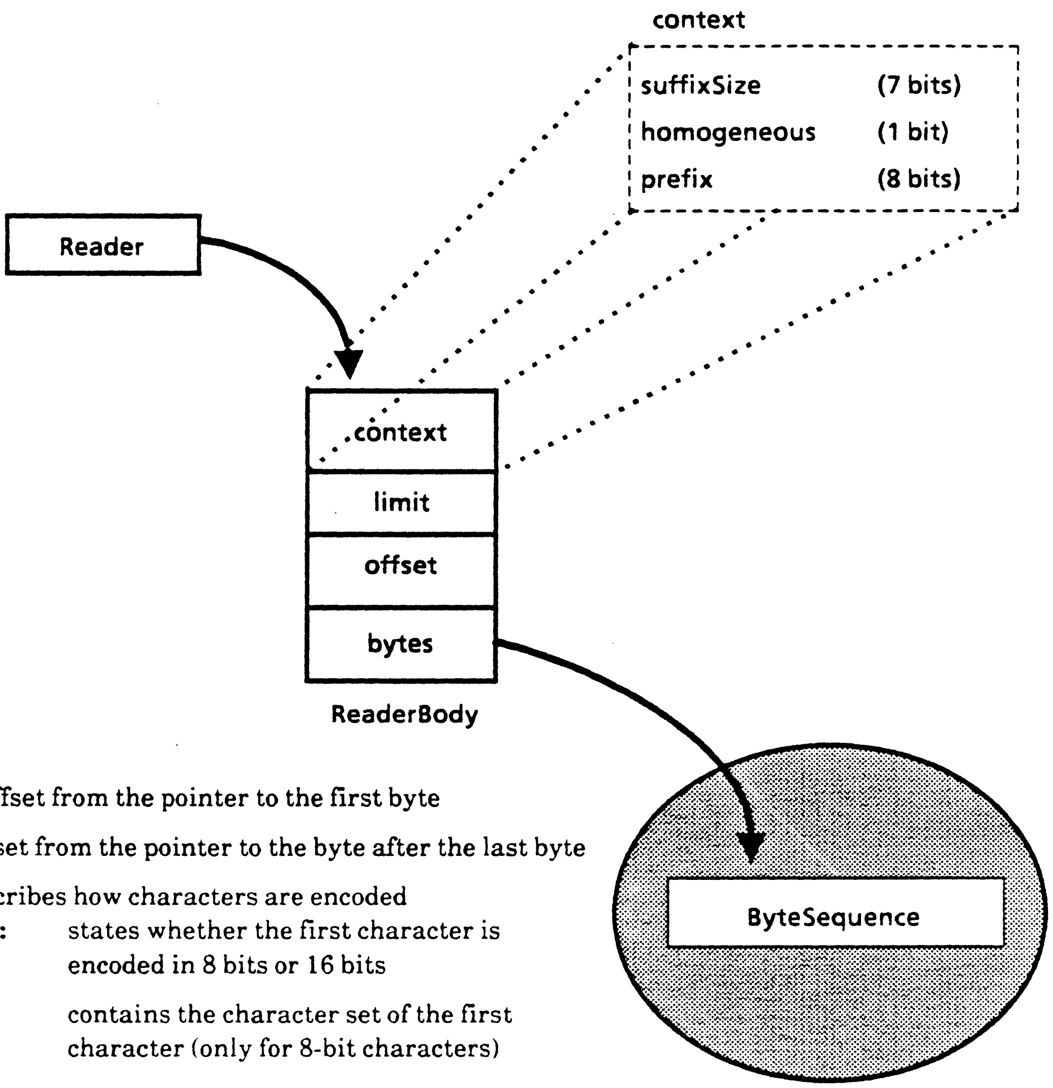


Figure 5.1. XString.ReaderBody

A **Context** contains information about how characters are encoded in the byte sequence. The **suffixSize** field describes whether the first byte is encoded as a 8-bit character or a 16-bit character. The **homogeneous** field is an accelerator specifying whether the byte sequence contains any character shifts. Setting it to TRUE may make some operations faster, but it's important to set it to TRUE only when it really is true. It is always safe to set it to FALSE. The **prefix** field specifies the character set of the first character. Subsequent characters in the string use the same prefix unless an encoding transition is encountered. (The **prefix** field is used only for 8-bit characters, since the 16-bit representation includes a character set.)

Figure 5.2 illustrates these data structures.



offset: the offset from the pointer to the first byte

limit: the offset from the pointer to the byte after the last byte

context: describes how characters are encoded

suffixSize:	states whether the first character is encoded in 8 bits or 16 bits
prefix:	contains the character set of the first character (only for 8-bit characters)
homogeneous:	TRUE if no character shifts in sequence

Figure 5.2 Reader and ReaderBody

5.2.1.1 Accessing the contents of a reader

Because of the character encodings, you shouldn't access the contents of a reader just by indexing. Instead, you should always use procedures from the **XString** interface:

```
XString.First: PROCEDURE [r: XString.Reader] RETURNS [c: Character];

XString.NthCharacter: PROCEDURE [r: XString.Reader, n: CARDINAL]
   RETURNS [c: Character];

XString.Lop: PROCEDURE [r: XString.Reader] RETURNS [c: Character];
```

First and **NthChar** *return* the specified character; **Lop** *removes* the first character and returns it. **First** and **Lop** are more efficient than **NthCharacter**; you should use them when appropriate. **XString** also provides procedures to determine other information about a reader, such as the number of logical characters that it contains; consult the **XString** chapter of the *ViewPoint Programmer's Manual* for details.

5.2.1.2 Creating readers

There are several ways to create readers. One way is to start with a writer; once the contents are fixed, you can use **XString.ReaderFromWriter** to convert from a writer to a reader. You can also use **XString.FromSTRING** or **XString.FromNSString** to convert a Mesa string or an NSString into a reader:

```
XString.ReaderFromWriter: PROCEDURE [w: XString.Writer]
   RETURNS [XString.Reader] = INLINE ... ;

XString.FromSTRING: PROCEDURE [s: LONG STRING,
   homogeneous: BOOLEAN ← FALSE]
   RETURNS [XString.ReaderBody];

XString.FromNSString: PROCEDURE [s: NSString.String,
   homogeneous: BOOLEAN ← FALSE]
   RETURNS [XString.ReaderBody];
```

You can also use procedures from the **XFormat** interface to format various types (such as a stream of characters or a sequence of strings) into **Readers**, or vice versa. The primary data structure of the **XFormat** interface is the **Handle**:

```
XFormat.Handle: TYPE = LONG POINTER TO XFormat.Object;

XFormat.Object: TYPE = RECORD [
   proc: XFormat.FormatProc,
   context: XString.Context ← XString.VanillaContext,
   data: XFormat.ClientData ← NIL];

XFormat.FormatProc: TYPE = PROCEDURE [r:XString.Reader, h:
XFormat.Handle];

XFormat.ClientData: TYPE = LONG POINTER;
```

There are two major classes of operations in **XFormat**. One class has a built-in format procedure, and the other does not. The four data structures for which there are default output procedures are: XString.Writer, Stream.Handle, TTY.Handle and NSString.String. Here is an example that uses both kinds of procedures to put the contents of a string directly into a stream:

```
fileStream: Stream.Handle ← foo;
rb: XString.ReaderBody ← XString.FromSTRING ["Lysol"L];
obj: XFormat.Object ← XFormat.StreamObject[sH:fileStream];
XFormat.Reader[h:@obj, r:rb];
```

XFormat.StreamObject has a built-in format procedure; it always directs its output to a Stream.Handle. Thus, the call to SreamObject constructs and returns an object whose FormatProc is StreamProc (the built-in default) and whose data is sH (the stream). Reader, on the other hand, is an example of the other kind of procedure. When you call this procedure, you need to provide a handle to an object with a format procedure. Thus, Reader calls StreamProc, with r as a parameter; StreamProc puts the bytes of the reader to the stream handle. Here is a second example of how to use XFormat:

```
writerBody: XString.WriterBody ←
   XString.NewWriterBody[maxLength:250, z:z];
xfo: XFormat.Object ← XFormat.WriterObject[
   w: @writerBody];
XFormat.String[h:@xfo, s:"My name is "L];
 --Concatenate strings into writer
XFormat.String[h:@xfo, s: namePassedInAsAParameter];
XFormat.String[h:@xfo, s:" and my age is "L];
XFormat.Decimal[h:@xfo, n: agePassedInAsAParameter];
XFormat.Char[h:@xfo, char: '..ORD];
[] ← SimpleTextDisplay.StringIntoWindow [
        string: XString.ReaderFromWriter[@writerBody], ...];
```

In this example, we first create a new writer (discussed later in the chapter), and then we call WriterObject to create an object initialized with the format procedure WriterProc and data writerBody. Next, we use the String procedure to concatenate a series of strings into the writer. Finally, we convert the finished writer into a reader, and then display the reader.

5.2.1.3 Readers vs. ReaderBodys

When you use readers, you must decide whether to use the actual ReaderBody or just the Reader. Obviously, since readers are just pointers, they require less space than ReaderBodys. However, you should use the ReaderBody itself when keeping track of who owns the storage is a problem. Thus, you should generally put a ReaderBody, not just a reader, in your data structure.

For procedures, the guideline is to take a Reader and return a ReaderBody. The idea is that passing readers as parameters reduces the number of words of parameters, while returning ReaderBodys allows the client to manage the storage for the ReaderBody.

Another guideline is that clients should be able to pass pointers to local ReaderBodys. That is, clients should be able to allocate ReaderBodies from the local frame, rather than from permanent storage. For example, consider the following fictional procedure that renames a file:

```
RenameFile: PROCEDURE [oldName:XString.Reader] = {
   rb: XString.ReaderBody ← SomeInterface.GetNewName[] ;
   file ← SomeInterface.LookupByName[oldName];
   SomeInterface.Rename[file: file, newName: @rb]};
```

The procedure **RenameFile** takes a reader, which it passes to **LookupByName**. In this case, we are just passing pointers around, so using readers is the right thing to do. **GetNewName**, on the other hand, returns a **ReaderBody**. If it returned a **Reader**, there would be a problem with where the storage for the **ReaderBody** was kept. Either it would have to be global, or it would have to be deallocated from a known place after **RenameFile** was done with it. Returning the **ReaderBody** itself makes it clear that **RenameFile** owns that storage and can deallocate it when appropriate. The **newName** parameter to the **Rename** operation is a pointer to a local **ReaderBody**. **Rename** should copy the **ReaderBody** (and the bytes) if it intends to save the characters.

5.2.2 Writers and WriterBodys

```
XString.Writer: TYPE = LONG POINTER TO XString.WriterBody;

XString.WriterBody: TYPE = PRIVATE MACHINE DEPENDENT RECORD [
   context(0): Context,
   limit(1): CARDINAL,
   offset(2): CARDINAL,
   bytes(3): Bytes,
   maxLimit(5): CARDINAL,
   endContext(6): Context,
   zone(7): UNCOUNTED ZONE];

Bytes: TYPE = LONG POINTER TO ByteSequence;
```

A **WriterBody** contains the same information as a **ReaderBody**, plus three additional fields. **maxLimit** describes the limits of the allocation unit, **endContext** is the context that describes how the last character is encoded (this is an accelerator for operations that append characters), and **zone** is the zone that contains the allocation unit.

Including a zone in the **WriterBody** enables operations that add characters to the writer to allocate a larger byte sequence, copy the old bytes, and update the byte pointer in the **WriterBody** without invalidating the writer variable that the caller owns.

You can allocate a writer with XString.**NewWriterBody**:

```
NewWriterBody: PROCEDURE [maxLength: CARDINAL,
   z: UNCOUNTED ZONE]
   RETURNS [XString.WriterBody];
```

NewWriterBody allocates a byte sequence that has room for **maxLength** bytes using **z** and returns an empty **WriterBody** that contains the bytes. You can expand a **WriterBody** with a call to XString.**ExpandWriter**:

```
ExpandWriter: PROCEDURE [w: XString.Writer, extra: CARDINAL];
```

ExpandWriter assures that at least **extra** bytes are available in the writer's bytes. There are several procedures for writing and editing writers; check the *Viewpoint Programmer's Manual* to find out what is available.

5.3 XMessage

The idea behind the **XMessage** interface is that all messages that the user sees (generally speaking, all the readers in a program) should be grouped together. Eventually, when an application is finished, you can use Message Tools to make your messages independent of your application. That way, you can change the messages without recompiling the application, which makes it easy to convert applications for multilingual use. We discuss how to make your messages independent of compilation in Chapter 13, Message Tools; for now, we present the messages mechanism used during development.

There are three pieces to the messages mechanism: a definitions module, an implementation module that provides the raw material for the messages, and program modules that use the messages provided by the implementation.

The definitions module defines the messages for the application, and defines a procedure for clients to call when the they need to access the messages. Programs access messages via an **XMessage.Handle**, which represents a collection of messages. A **handle** is normally associated with a particular application. The definitions file provides a procedure for programs to call when they need to access the handle. Thus, this procedure provides an easy way for message suppliers and message users to communicate.

The second piece is the module that provides the raw material for the messages. This module is used to supply the message text while running the application and supply the raw data to the message translators. The third piece is the module or modules that uses the messages. The example that follows shows each of these three pieces.

```
-- MsgDefs.mesa
DIRECTORY
   XMessage USING [Handle, MsgKey];

MsgDefs: DEFINITIONS = {
GetMessageHandle: PROC RETURNS [h: XMessage.Handle];
MessageKey: TYPE = {hiMom, elephant, missingFile, invalidInput};
khiMom: XMessage.MsgKey = MessageKey.hiMom.ORD;
kelephant: XMessage.MsgKey = MessageKey.elephant.ORD;
kmissingFile: XMessage.MsgKey = MessageKey.missingFile.ORD;
kinvalidInput: XMessage.MsgKey = MessageKey.invalidInput.ORD;

-- MsgImpl.mesa
DIRECTORY
   MsgDefs,
   XMessage,
   XString;

MsgImpl: PROGRAM IMPORTS XMessage, XString EXPORTS MsgDefs = {
```

```
OPEN XS: XString;

h: XMessage.Handle ← NIL;          -- The messages handle

GetMessageHandle: PUBLIC PROC RETURNS [h: XMessage.Handle] = {
 RETURN [h]};                -- Returns the message handle

Init: PROC = {          -- Creates, allocates, and registers messages
 msgArray: ARRAY MsgDefs.MessageKey OF XMessage.MsgEntry ← [
  hiMom: [
     msgKey: MsgDefs.khiMom,              -- Key used by msg customer
     msg: XS.FromSTRING ["Hi Mom"L],      -- Actual message
     id: 1],                 -- Internal key
  elephant: [
     msgKey: MsgDefs.kelephant,
     msg: XS.FromSTRING ["Elephants should be chartreuse."L],
     id: 2],
  missingFile: [
     msgKey: MsgDefs.kmissingFile,
     msg: XS.FromSTRING ["Error...file not found"L],
     type: errorMsg,              -- Hint as to how message will be used
     id: 3],
  invalidInput: [
     msgKey: MsgDefs.kinvalidInput,
     msg: XS.FromSTRING ["Invalid input."L],
     id: 4]];

     messages: XMessage.Messages ← DESCRIPTOR [LOOPHOLE [
        msgArray, ARRAY[
           0..MsgDefs.MessageKey.LAST.ORD] OF XMessage.MsgEntry]];

  h ← XMessage.AllocateMessages [
        applicationName: "TestApplication"L,
        maxMessages: MsgDefs.MessageKey.LAST.ORD + 1,
        clientData: NIL,
        proc: DeleteMessages];

  XMessage.RegisterMessages [
     h: h,
     messages: messages,
     stringBodiesAreReal: FALSE]};
  DeleteMessages: PROC [clientData: XMessage.ClientData] =
     {};
  --Mainline code
  Init [];}...

Typical message usage
mh: XMessage.Handle ← MsgDefs.GetMessageHandle[];
missingFile: XString.ReaderBody ← XMessage.Get [
  mh, MsgDefs.kmissingFile];
   .
   .
   .
Attention.Post [@missingFile];}.   --Discussed in the next section
```

The definitions module declares a procedure that can be used to get the message handle, as well as keys (which are just CARDINALS) to identify the messages. The implementation module declares the handle and exports **GetMessageHandle**. The **Init** procedure then initializes the elements in the message array, which are of type XMessage.**MsgEntry**:

```
MsgEntry: TYPE = RECORD [
   msgKey: Msgkey,
   msg: XString.ReaderBody,      --The actual message
   owner: LONG STRING ← NIL,     --Who owns the ReaderBody
   severity: MsgSeverity ← good,
   translationNote: LONG STRING ← NIL,
   translatable: BOOLEAN ← TRUE,
   type: MsgType ← userMsg,
   id: MsgID], ·
```

The **type** is used to give a hint as to how the message will be used. Some examples of possible types are **userMsg**, for messages that will appear in the Attention window, **pSheetItem** for messages that will appear in a property sheet, and **errorMsg**, for error messages. See the *ViewPoint Programmer's Manual* for a complete list of the possible types.

The **id** provides a unique identifier for a particular message. This id is what ties a message to previous occurrences of that message and helps translators to determine when a new message is added or an old one deleted. You should never change the id during the lifetime of a message. The **translationNote** parameter allows you to include extra information for the translator in case a message is potentially ambiguous.

Once the array has been set up, **Init** creates a descriptor for the **msgArray** and calls **XMessage.AllocateMessages** to define a range of messages to be associated with this application. This procedure returns a message handle, which must be used to access the messages for that application. Next, **Init** calls **XMessage.RegisterMessages** to associate the messages with the handle. Thus, **Init** provides the correspondence between the message keys declared in **MessageDefs**, and the actual text in the readers.

Client modules call **GetMessageHandle** to get the handle to the correct messages, and then call **Get** to retrieve the appropriate message.

5.4 Attention

The **Attention** interface implements a single window into which messages are displayed. The Attention window also has a menu to which you can add system-wide commands; this menu is discussed in section 7.1.

There are three types of messages: simple messages, sticky messages and confirmed messages. Simple messages have no special semantics. Sticky messages are redisplayed when a non-sticky message is cleared. **Attention** keeps track of one sticky message. Confirmed messages ask for confirmation by the user. **Attention** allows messages to be logically appended. There are three posting operations: **Post**, **PostSticky**, and **PostAndConfirm**.

```
Post: PROCEDURE [s: XString.Reader, clear: BOOLEAN ← TRUE];

PostSticky: PROCEDURE [s: XString.Reader,
   clear: BOOLEAN ← TRUE];
```

```
PostAndConfirm: PROCEDURE [
   s: XString.Reader,
   clear: BOOLEAN ← TRUE,
   confirmChoices: ConfirmChoices ← [NIL, NIL],
   timeout: Process.Ticks ← dontTimeout]
   RETURNS [confirmed, timedOut: BOOLEAN];
```

The **Post** procedures display the message **s** in the **Attention** window. If **clear** is TRUE, it clears the Attention window before displaying **s**, otherwise it displays it after whatever text is currently showing. **PostAndConfirm** acts like **Post** in displaying the message **s** but waits for confirmation by the user. See the *ViewPoint Programmer's Manual* for details on how to use **PostSticky** and **PostAndConfirm**. There are also the inverse operations:

```
Clear: PROCEDURE;

ClearSticky: PROCEDURE;
```

Clear clears the Attention window of any simple message. If a simple message is being displayed and there is a current sticky message, the sticky message will now be displayed. **Clear** has no effect if a sticky message is being displayed. **ClearSticky** clears any current sticky message. If a sticky message is being displayed, the window is cleared. **ClearSticky** has no effect if there is no sticky message.

Constructing messages in the single global Attention window does not work well if multiple processes try to display messages simultaneously. Thus, you should follow this guideline: only call procedures in the **Attention** interface from the notifier process. Following this rule guarantees that only well-formed messages will be displayed.

6. TIP

This chapter provides a brief overview of how user actions are translated into program actions. When the user presses a key or moves the mouse, that action must be recognized, directed to the correct window, and then acted upon. This multiplexing of user input is the job of the Terminal Interface Package (TIP), and two processes called the *Stimulus* and the *Notifier*.

The Stimulus is a high-priority process that just watches for user actions and enqueues them. The Notifier then dequeues each event and directs it to a window. All user actions are directed either to the window with the input focus or to the window with the cursor. Most user actions are sent to the input focus; only actions such as mouse clicks are sent to the window with the cursor.

Once it has determined the correct window for a user action, the Notifer checks for that action in the window's *TIP tables*. A TIP table is essentially a giant SELECT statement. The left side of the table contains various user actions, and the right side of the table has a list of results. When an action is located in the left side of a TIP table, the corresponding result on the right side is passed to a special procedure called a *NotifyProc*. The NotifyProc is then responsible for executing whatever program actions are to be associated with the user action.

The Notifier process is used to avoid multi-process interference. Some operations, such as setting the selection, must be guaranteed no asynchronous interference, and thus are restricted to happening only in the Notifier process. The Notifier process is also the one most closely tied to the user. If an operation will take an extended time to complete, it should be forked from the Notifier to run in a separate process so that the Notifier is free to respond to the user's actions.

6.1 TIP tables

TIP tables provide a flexible method of translating user actions into program actions. A given window always has at least one associated TIP table, and may have a chain of tables. The Notifier checks a given user action against each table in the chain until it finds a match or until it runs out of tables to check. If there are no more tables, the action is discarded. This sequence is illustrated in Figure 6.1.

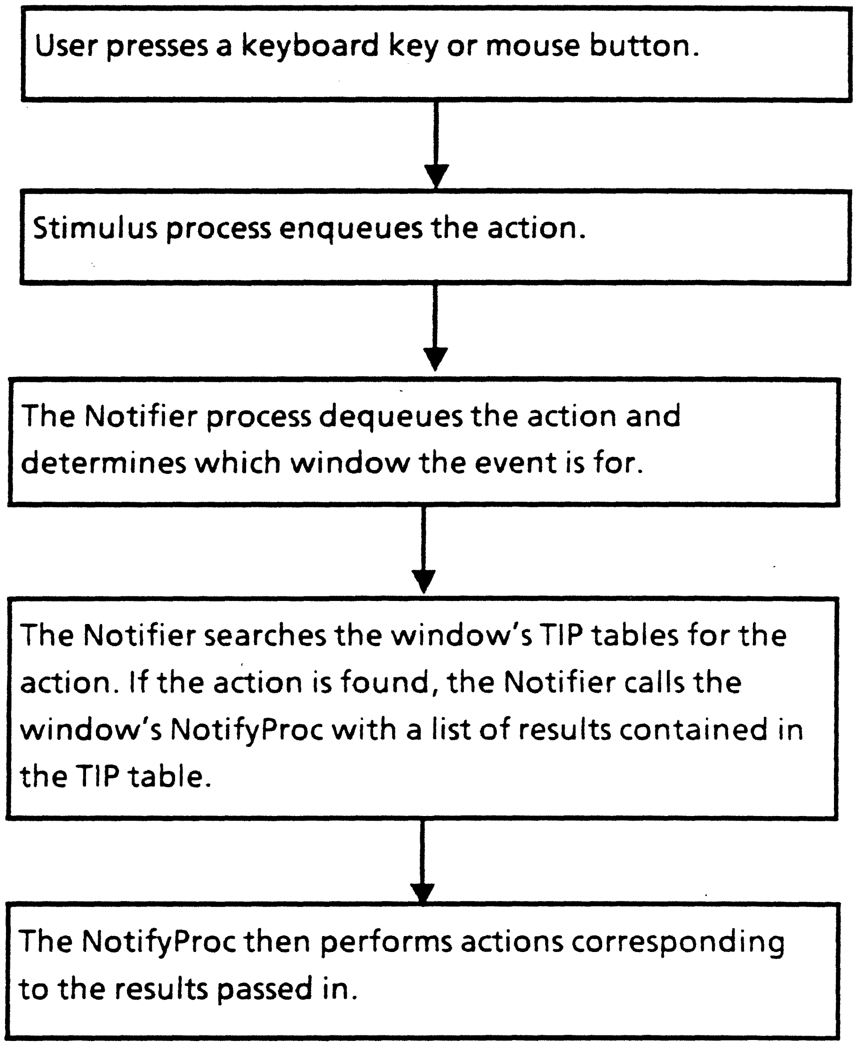

User presses a keyboard key or mouse button.
Stimulus process enqueues the action.
The Notifier process dequeues the action and determines which window the event is for.
The Notifier searches the window's TIP tables for the action. If the action is found, the Notifier calls the window's NotifyProc with a list of results contained in the TIP table.
The NotifyProc then performs actions corresponding to the results passed in.

Figure 6.1 How user input is handled

In its simplest form, a TIP table is a user-editable file. These TIP tables are stored in the system catalog, and have the extension .TIP. For every TIP table, there is a program somewhere that translates the user-readable TIP table into a program-readable TIP table, with a call to **TIP.CreateTable**. When the program is run, the information in the .TIP file is parsed into a runtime data structure. In addition, a compiled version of the TIP file (.TIPC) is created. Each subsequent time the program is run, it will use the .TIPC file to create the runtime data structure. This makes building the runtime data structure much faster, since there is no need to parse the TIP table.

Here is an example of a text version of a TIP table:

```
SELECT TRIGGER FROM
   Point Down => 
      SELECT TRIGGER FROM
         Point Up BEFORE 200 AND Point Down BEFORE 200 =>
            SELECT ENABLE FROM
```

```
LeftShift Down = > COORDS, ShiftedDoubleClick
ENDCASE        = > COORDS, NormalDoubleClick;

Adjust Down BEFORE 300 = > PointAndAdjust;
ENDCASE = > COORDS, SimpleClick;
 ...
```

A *trigger* action is an action that has just been dequeued from the user action queue; this is the action that caused the Notifier to check the TIP table. An *enabled* action is an action that is also true at the time, but did not necessarily just become true. Thus,this TIP table matches the two actions **Point Down** and **Adjust Down**. When the left mouse button goes down, remains there no longer than 200 milliseconds, and goes down again before another 200 milliseconds has elapsed, the state of the left shift key is checked. If the key is down, the result **ShiftedDoubleClick** is passed; otherwise, the result **NormalDoubleClick** is passed. The convention for writing TIP tables is to have information results precede the action result. Thus, in the example above, COORDS is information, and **ShiftedDoubleClick** or **NormalDoubleClick** is the actual action.

6.2 NotifyProcs

When the Notifer process recognizes a user action in the left side of a TIP table, it passes the associated results to a **NotifyProc**. **NotifyProcs** are ususally associated with the window, but they can also be associated with the TIP table itself. The job of a **NotifyProc** is to interpret the results and take appropriate program action. A **NotifyProc** is of type **TIP.NotifyProc**:

```
TIP.NotifyProc: TYPE = PROCEDURE [
   window: Window.Handle, results: TIP.Results];
```

Results is a pointer to a linked list of **ResultsObjects**. Each **ResultsObject** contains a pointer to the next **ResultsObject** and a body, as illustrated in Figure 6.2.

The **body** is a variant record that may be an atom or one of a number of standard results. Standard results represent information that is commonly needed, such as the time or the current mouse coordinates. An **atom** is a unique string It can be a string shared by many TIP tables, such as "point up," or it can be a special-purpose string defining a program-specific result.

```
ResultObject: TYPE = RECORD [
   next: Results,
   body: SELECT type: * FROM
      atom = > [a: ATOM],
      bufferedChar = > NULL,
      coords = > [place: Window.Place],
      int = > [i: LONG INTEGER],
      key = > [key: KeyName, downUp: DownUp],
      nop = > [],
      string = > [rb: XString.ReaderBody],
      time = > [time: System.Pulses],
      ENDCASE];
```

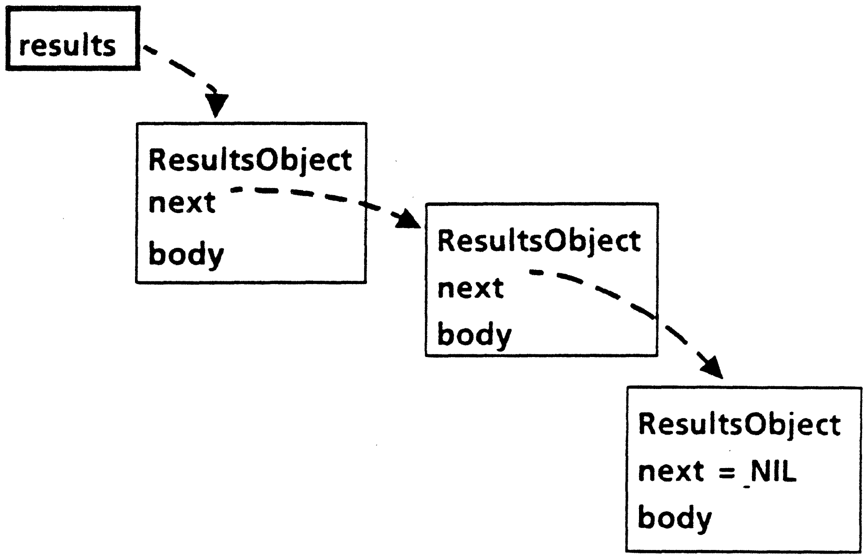


Figure 6.2: The structure of a results list

Thus, a **NotifyProc** for the above TIP table would look something like this:

```
TIPMe: TIP.NotifyProc = {
place: Window.Place;
FOR input: TIP.Results ← results, input.next UNTIL input = NIL
DO
   WITH z: input SELECT FROM
      coords = > place ← z.place;
      atom = > SELECT z.a FROM
         SimpleClick = > Simple[place];
         NormalDoubleClick = > NormalDouble[place];
         ShiftedDoubleClick = > ShiftedDouble[place];
         PointAndAdjust = > Chord[];
         ENDCASE;
      ENDCASE;
   ENDLOOP};
```

This is a contrived example, so this NotifyProc doesn't really do anything interesting. You should concentrate on the syntax.

Typically, when you receive an information result, such as COORDS, you store the value of the coordinates into another variable and act on them when a later result comes in. In this case, we store the coordinates into the variable **place**, and then pass **place** to our procedures **Simple**, **NormalDouble**, and **ShiftedDouble**. Naturally, you don't have to call another procedure from the **NotifyProc**; you can do whatever it is that you have to do straight from the **NotifyProc** if that is more convenient. Note that the **NotifyProc** is called once for every successful match in the TIP table. The loops in the **NotifyProc** are only there because the results list may have more than one element (e.g.,COORDS, **NormalDoubleClick**).

Note also that you must "create" all of the atoms that you wish to recognize in your NotifyProc. For each atom, you must make a call to the procedure **Atom.MakeAtom**. This is typically done in a separate procedure, but can be done within the **NotifyProc** if you so desire. **MakeAtom** returns the atom corresponding to the character string that you pass in. You have to call this for standard atoms as well as for atoms that you declare yourself; if **MakeAtom** can't find an existing atom corresponding to the string that you pass in, it creates a new one. (See the Atom chapter of the *ViewPoint Programmer's Manual* for details.) For example:

```
InitAtoms: PROCEDURE = {
   SimpleClick ← Atom.MakeAtom["SimpleClick"L];
   NormalDoubleClick ←
   Atom.MakeAtom["NormalDoubleClick"L];
   ShiftedDoubleClick ←
   Atom.MakeAtom["ShiftedDoubleClick"L];
   PointAndAdjust ← Atom.MakeAtom["PointAndAdjust"L]};
```

There are a number of procedures in the *ViewPoint Programmer's Manual* for setting up the structure of TIP tables. Check this manual to find out how to write your own TIP table and link it into the existing chain, and how to manipulate the relationships between windows, tables, and **NotifyProcs**. The chapters that you should look at are TIP, TIPStar, and Atom.

7. Creating a Simple Application

The ViewPoint architecture is an open-ended design, allowing Xerox applications, such as the document editor, to reside as equal citizens next to user-supplied applications. This chapter discusses how to create a simple application and integrate it into the existing environment.

7.1 The Attention menu

A Viewpoint application can be structured to run either from a command in the attention menu or from an icon. Using icons is discussed in Chapter 9, Icon Applications; for now, we discuss how to use the attention menu.

To make a tool run from the attention menu, you create a *menu item* and then call **Attention.AddMenuItem**:

```
MenuData.CreateItem: PROCEDURE [
    zone: UNCOUNTED ZONE,
    name: XString.Reader,
    proc: MenuProc,
    itemData: LONG UNSPECIFIED ← 0]
    RETURNS [Itemhandle];

Attention.AddMenuItem: PROCEDURE [item:
MenuData.ItemHandle];
```

The call to **CreateItem** returns a handle to a menu item, which you can then pass to **AddMenuItem**. (An **Itemhandle** is a pointer to a private object; you can't see what the object looks like.) In the call to **CreateItem**, **zone** is the name of the zone you want the storage to come from, **name** is the name of the command that you want to put in the menu, and **proc** is the procedure that will be called when the user invokes the command. **itemData** is for your own use. The **MenuData** implementation passes **itemData** to **proc**; if there is any information that you want to make available in **proc**, you can pass it in via **itemData**. Here is an example of how to register a command with the attention menu:

```
--This procedure gets called from the mainline code
 Init: PROC = {
       sampleTool: XString.ReaderBody ←
XString.FromSTRING["Sample Tool"L];
       Attention.AddMenuItem [
          MenuData.CreateItem [
          zone: sysZ,
          name: @sampleTool,
          proc: MenuProc] ];
```

7.2 StarWindowShells

In ViewPoint the abstract idea of a window is implemented by a *StarWindowShell*. A StarWindowShell is a basic window with a header that can have a title, commands, pop-up menus, and scrollbars (horizontal or vertical), as illustrated in Figure 8.1

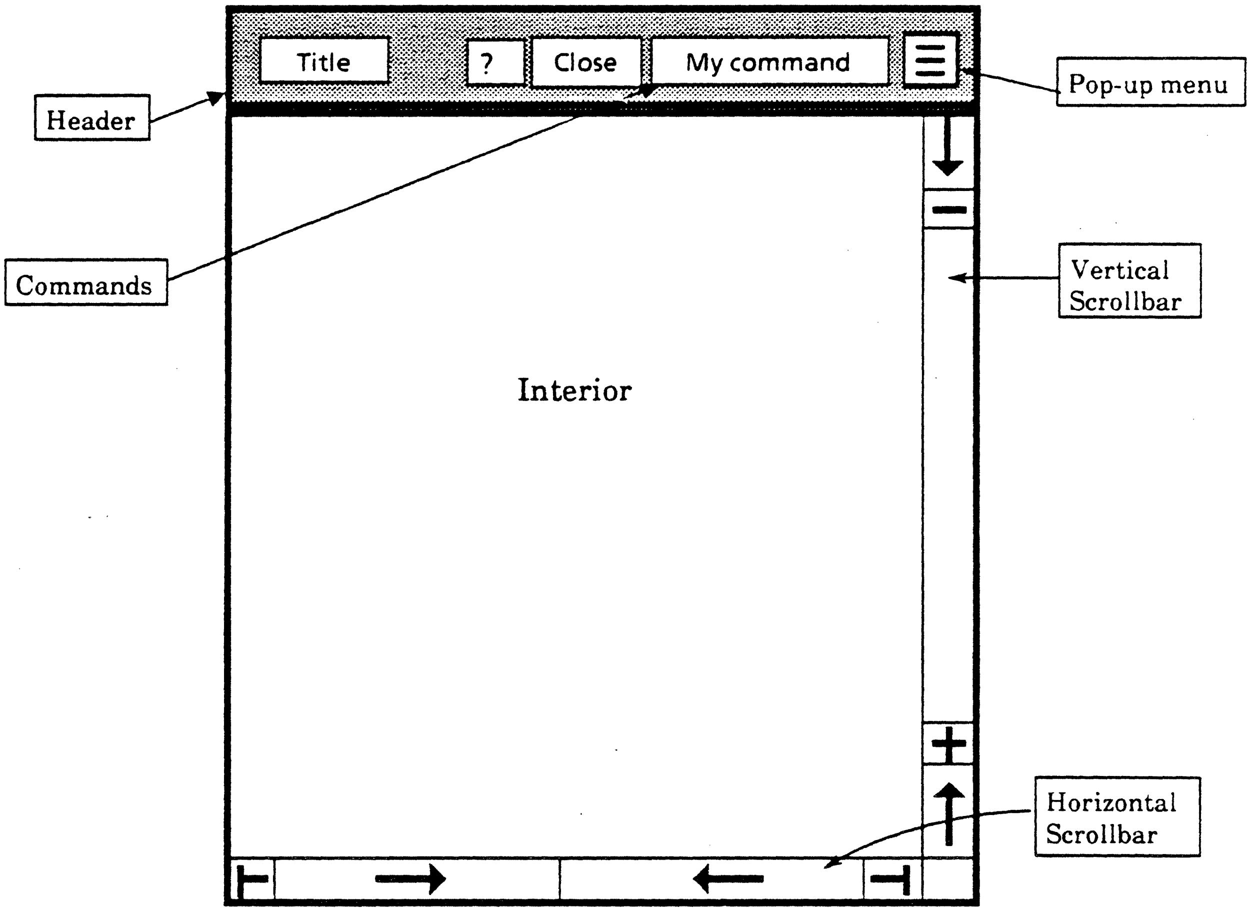


Figure 8.1 A StarWindowShell

ViewPoint windows are organized in a tree structure, with the desktop window at the root of the tree. When you write an application, the StarWindowShell for that application is generally a child of the desktop window. A StarWindowShell is in turn the parent of an *interior window* that is exactly the size of the available window space in the shell (i.e., the StarWindowShell minus its borders, header, and scrollbars). The interior window may in turn have children: children of interior windows are called *body windows*. Body windows are what define the functionality of a window shell.

7.2.1 Creating the shell

When the user invokes your command from the attention menu, your specified procedure will be called. This procedure should call StarWindowShell.Create:

```
Create: PROCEDURE [
   transitionProc: TransitionProc ← NIL, --See below
   name:XString.Reader ← NIL, --The name of the tool
   namePicture:XString.Character ← XChar.null, -
   host: Handle ← NIL,
   type: ShellType ← regular,
   sleeps: BOOLEAN ← FALSE,--see below
   considerShowingCoverSheet: BOOLEAN ← TRUE,
   currentlyShowingCoverSheet: BOOLEAN ← FALSE,
   pushersAreReadonly: BOOLEAN ← FALSE,
   readonly: BOOLEAN ← FALSE,
   scrollData: ScrollData ← vanillaScrollData,
   garbageCollectBodiesProc: PROCEDURE [Handle] ← NIL,
   isCloseLegalProc: IsCloseKegalProc ← NIL,   --See below
   bodyGravity: Window.Gravity ← nw,
   zone: UNCOUNTED ZONE ← NIL ]
   RETURNS [Handle];
```

Handle: TYPE = RECORD [Window.Handle];

Note that all of these parameters are defaulted, which means that they are all optional. However, most calls to **Create** include at least the first two (**name** and **transitionProc**), however. We discuss **transitionProc** and **isCloseLegal** below; for information on the other parameters, consult the *ViewPoint Programmer's Manual*.

7.2.2 Transition procedures

A **transitionProc** for a window shell is a procedure that will be called whenever the state of the shell is about to change. A StarWindowShell is always in one of three states: **awake**, **sleeping**, or **dead**. **awake** indicates that the shell is currently displayed. **sleeping** indicates that the shell still exists, but is not being displayed and therefore resources associated with the display state should be freed. dead indicates that the shell is just about to be destroyed and therefore all resources associated with the shell should be freed. If you have any storage associated with your shell, you should use a **transitionProc** to allocate and free that storage. (The **sleeps** parameter to **Create** indicates whether your application has any resources dedicated only to displaying information.) Here is a simple example of a transition procedure:

```
SimpleTransitionProc: StarWindowShell.TransitionProc =
   BEGIN
      SELECT state FROM
         awake = > IF data = NIL THEN AllocateData[sws];
         sleeping, dead = > FreeData[data];
      ENDCASE;
   END;
```

State is a parameter to the transition procedure, indicating the new state of the shell.

```
State: TYPE = {awake(0), sleeping, dead, last(7)};

TransitionProc: TYPE = PROCEDURE [sws: Handle, state: State];
```

7.2.3 IsCloseLegalProc

An **isCloseLegalProc** allows you to veto an attempt to close your window. The **isCloseLegalProc** that you supply will be called when the either a user or a client program attempts to close the StarWindowShell. If it's okay to close the window, it should return TRUE; otherwise, it should return FALSE. (The **isCloseLegalProc** is also a convenient way to get control when the window is being closed.) Here is a simple example:

```
SimpleIsCloseLegalProc: StarWindowShell.IsCloseLegalProc =
BEGIN
  IF YouDon'tCareIfTheWindowIsClosed THEN RETURN [TRUE];
  RETURN [FALSE];
END;
```

An **isCloseLegalProc** is of type **isCloseLegalProc**:

```
IsCloseLegalProc: TYPE = PROCEDURE [sws: Handle, closeAll:
BOOLEAN]
  RETURNS [BOOLEAN];
```

closeAll indicates whether the current command is a Close! or a CloseAll!.

7.2.4 Body windows

After you create a StarWindowShell, you can create an arbitrary number of body windows within the shell. Each body window will be a child of the StarWindowShell's interior window. The body windows define the functionality of the window; their arrangement depends on what you want your application to do. Here are some common arrangements of body windows:

- One very long body window. This makes scrolling easy; you simply slide the body window within the window shell. This is how the StarWindowShell default scrolling works. You can make a long window by specifying the dimensions of your body window during creation;the StarWindowShell will then take care of all scrolling operations.

- One body window with **BodyWindowJustFits** = TRUE. With this kind of body window, the size of the body window changes any time the size of the StarWindowShell changes. This type is difficult to implement, since you have to write the procedures for the display adjustment.

- Several body windows, each of which holds a segment of information. This is similar to a document that has been paginated. Here too scrolling is done by the StarWindowShell; you need to provide new pages as body windows are scrolled off the shell.

To create a body window, you call StarWindowShell.CreateBody:

```
CreateBody: PROCEDURE [
   sws: Handle,      --the StarWindowShell
   repaintProc: PROCEDURE [Window.Handle] ← NIL,
   bodyNotifyProc: TIP.NotifyProc ← NIL,
   box: Window.Box ← [[0,0],[0,29999]] ]
RETURNS [Window.Handle];
```

```
Window.Box: TYPE = RECORD [place: Place, dims: Dims];

Window.Place: TYPE = UserTerminal.Coordinate; --[x,y: INTEGER]

Window.Dims: TYPE = RECORD [w,h: INTEGER];
```

CreateBody creates a body window that is a child of the interior window of **sws**. **repaintProc** is the display proc that is called by the Window implementation whenever part or all of the body window needs to be displayed (more on this in the next section). **bodyNotifyProc** is a **TIP.NotifyProc** that is attached to the window.

box indicates the size and location of the body window within the shell's interior window. If **box.dims.w** and/or **box.dims.h** is zero, the body window will take on the **dims.w** and/or **dims.h** of the shell's interior window.

Note that body windows can themselves have child windows, and so on. There are a number of useful procedures (such as **GetBody**) in the StarWindowShell interface that allow you to take a look the window structure; see the *ViewPointProgrammer's Manual* for details.

7.2.5 Displaying information in a window

The simplest way to display text to a body window is to call SimpleTextDisplay.StringIntoWindow. (We discuss more complex ways of displaying information in a window in Chapter 8, Form Windows and Property Sheets). With **StringIntoWindow**, you can specify a string and a place in the window; the string will always be displayed in the system font. For example:

```
Redisplay: Window.DisplayProc = {
   wBody: XString.WriterBody ← XString.NewWriterBody [
      maxLength: 250, z: sysZ];
   xfo: XFormat.Object ← XFormat.WriterObject [w: @wBody];

   XFormat.String [h: @xfo, s: "This is a sample string
      displayed in a body window of a StarWindowShell
      using SimpleTextDisplay.StringIntoWindow."];

   [] ← SimpleTextDisplay.StringIntoWindow [
      string: XString.ReaderFromWriter [@writerbody],
      window: window,     --The body window
      place: [10,10],--Upper- left corner is [0,0]
      lineWidth: 300,    --Arbitrary (in pixels)
      maxNumberOfLines: 10, --Arbitrary
      flags: Display.replaceFlags]];;  --Clear old data
```

The standard way to paint information in a window is by letting the window implementation do most of the work. The basic idea is that you update your data structures, invalidate the area of the window that needs repainting, then call a validate routine to perform the repainting. The window implementation will call your **Window.DisplayProc** procedure to do the actual repainting. For example:

```
RepaintMenuProc: MenuData.MenuProc = {
   body: Window.Handle = StarWindowShell.GetBody[[window]];
   Window.InvalidateBox[body, [[0, 0], [30000, 30000] ]];
   Window.Validate[body]; };
```

The call to **Window.Validate** results in a call to the **Redisplay** procedure above, which in turn displays the desired information. Since this is a simple example, there are no internal data structures to update; **Redisplay** always displays the same thing.

7.2.6 Commands and menus

Every StarWindowShell can have commands and pop-up menus that the user can invoke. (Commands are actually individual menu items; the name of the item appears with a rounded corner box around it.) When you specify the commands for a window, you associate a procedure (of type **MenuData.MenuProc**) to go with each command. When the user invokes a command, the corresponding procedure is called.

```
z: UNCOUNTED ZONE ← StarWindowShell.GetZone [shell];
-- Gets the zone attached to the SWS

items: ARRAY [0..3) OF MenuData.ItemHandle ← [
   MenuData.CreateItem[zone: z, name: @another,
      proc: MenuProc],
   MenuData.CreateItem [zone: z, name: @repaint, proc:
      RepaintMenuProc],
   MenuData.CreateItem [zone: z, name: @post, proc: Post]];
.
.
myMenu: MenuData.MenuHandle = MenuData.CreateMenu [
   zone: z, -- Generally use zone attached to SWS
   title: NIL,
   array: DESCRIPTOR [items]];

StarWindowShell.SetRegularCommands[
   sws: shell, commands: myMenu];
.
.
Post: MenuData.MenuProc = {
   msg: XString.ReaderBody ← XString.FromSTRING [
      "This is a sample attention window message."L];
   Attention.Post [@msg];};
.
.
-- Destroy the Post command
DestroyItem[z: z, item: items[2]];
```

To add a command to the header of a StarWindowShell, you first call **MenuData.CreateItem** to associate a procedure with each command, then you store each menu item into an array. Next, you call **CreateMenu** to put the array into a menu. Once you have a **MenuData.MenuHandle**, you can call either **StarWindowShell.SetRegularCommands** or **StarWindowShell.Add-PopUpMenu**, depending on whether you want the item to appear as a command or in a menu. In the example above, we called **SetRegularCommands**. The procedure declarations are very similar:

```
SetRegularCommands: PROCEDURE [
   sws: Handle, commands: MenuData.MenuHandle] ;

AddPopupMenu: PROCEDURE [
   sws: Handle, menu: MenuData.MenuHandle] ;
```

AddPopupMenu adds **menu** to the available pop-up menus in **sws**. The title of **menu** is displayed in the StarWindowShell header with the small pop-up menu symbol (≡) just to the left of it, surrounded by a rounded corner box. When you have more commands than can fit in the window shell, the implementation automatically overflows the rightmost commands into an overflow pop-up menu.

7.2.7 Displaying windows on the screen

Create generates a StarWindowShell but does not display it on the screen. A shell is displayed on the screen with a call to **StarWindowShell.Push**, which inserts the new window into the existing tree structure.

```
Push: PROCEDURE [
   newShell: Handle, topOfStack: Handle ← NIL,
   poppedProc: PoppedProc ← NIL ];
```

Push displays **newShell** by inserting it into the visible window tree. If **topOfStack** is NIL, **newShell** is placed directly on the desktop. If **topOfStack** is not NIL, then **newShell** is "pushed on top of" **topOfStack** and **topOfStack** is removed from the display. If **poppedProc** is not NIL, it will be called when **newShell** is **Popped**. The **poppedProc** *must* either sleep the shell or destroy the shell, usually by calling **SleepOrDestroy**. If **poppedProc** is NIL, **newShell** will be destroyed when it is **Popped**.

Note that you do not always have to make the call to **Push** directly. For example, when the user selects an icon and presses OPEN, the application creates a StarWindowShell and returns it. The desktop implementation then displays the StarWindowShell by doing a **Push**.

You can remove a StarWindowShell from the screen by calling **StarWindowShell.Pop**. You will almost never call this procedure yourself, however; it is usually called by **StarWindowShell** as the result of an operation such as Close!.

7.3 Context

When possible, you should structure your application so that the user can have more than one copy of it running at any given time. This means that there may be many open windows associated with a particular program, giving rise to the problem of preserving state information for each window. For example, if you are editing three documents simultaneously on your desktop, the document editor must know which window you are typing in, which portion of the screen to update, and how to update it. To solve this problem, the notion of a *context* was introduced. A context is a data object associated with a window; thus, global state information is stored with the window rather than in the program's global frame (which is shared by all instances of the application.) Figure 7.2 illustrates this idea; notice that each window has the same context type but distinct data.

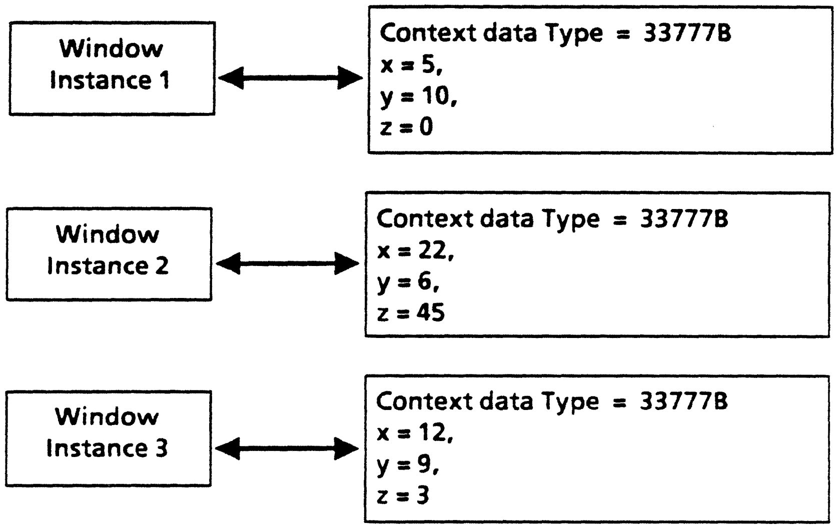


Figure 7.2 Context

In your mainline code, get a context type for the body window on which you intend to put the context. This type is unique for each client of the **Context** interface; you use it to identify your context in later calls to the **Context** interface.

```
context: Context.Type ← Context.UniqueType[]; --Global
```

Next, declare a data structure that represents your tool's global state variables:

```
DataObject: TYPE = RECORD [     --Global
   lastMouseButton: PointOrAdjust,
   place: Window.Place ← [0,0] ];
```

At initialization, allocate the context with a call to **Context.Create** :

```
Context.Create [
   type: context,
   data: sysZ.NEW[DataObject ← [neither] ],
   proc: DestroyContext,
   window: body];
```

In this example, we allocated the context from the **systemZone**. In this case, a **DestroyContext** procedure is unnecessary, since the default assumes the data was allocated from the **systemZone** and frees it from there. However, even though it is redundant, here is an example of a **DestroyContext** procedure:

```
DestroyContext: PROC [data: Data, window: Window.Handle]
= {
  sysZ.FREE [@data]};
```

When you need to look at or change the information in your context, call Context.Find or Context.Acquire. **Acquire** is just like **Find** except that it monitors the data so that only one process can have the context at a time. For example:

```
GetContext: PROC [body: Window.Handle] RETURNS [data:
Data] = {
  data ← Context.Find[context, body];
  IF data = NIL THEN ERROR; -- Just in case.
  RETURN [data]};
```

The only case where this is slightly tricky is when you want to retrieve context data from within a **MenuProc**. In this case, use the StarWindowShell handle, which was passed in to the **MenuProc**, to get a handle to the body window whose context you want. If you have only one body window, you can use the eldest child of the StarWindowShell by calling StarWindowShell.GetBody; to get the body's context you simply call Context.Find.

The following is an example of how to get the context from the MenuProc:

```
RepaintMenuProc: MenuData.MenuProc = {
    body: Window.Handle =
StarWindowShell.GetBody[[window]];
    data: Data ← GetContext[body]; --Find context
    Window.InvalidateBox[body, [[0, 0], [30000, 30000] ]];
    Window.Validate[body];
```

7.4 Complete example

Table 7.1 is a brief outline for writing a simple application. The program example that follows shows how to implement the outline.

```
-<< This is a sample tool that can be used as a template for ViewPoint
programs. Note that no special files or file types are needed to create such a
tool. This tool adds a command to the attention window menu. When the
user invokes this command, the MenuProc creates a StarWindowShell with
a single body window in it. Several menu items are placed in the header of
the StarWindowShell.>>

DIRECTORY
  Atom USING [ATOM, MakeAtom, null],
  Attention USING [AddMenuItem, Post],
  Context USING [Create, Data, Find, Type, UniqueType],
  Display USING [replaceFlags],
  Heap USING [systemZone],
  MenuData USING [CreateItem, CreateMenu, ItemHandle, MenuHandle,
    MenuProc],
  SimpleTextDisplay USING [StringIntoWindow],
  StarWindowShell USING [Create, CreateBody, GetBody, GetZone, Handle,
    Push, SetRegularCommands],
  TIP USING [NotifyProc, Results],
  Window USING [Dims, Handle, InvalidateBox, Object, Place, Validate],
  XFormat USING [Char, Decimal, Handle, Object, String, WriterObject],
```

Outline for creating a simple application:

1.	Create a StarWindowShell
2.	Create a body window inside the shell.
3.	Create menu items and a menu.
4.	Add the menu to the StarWindowShell header.
5.	Create a context for the body window.
6.	Push the StarWindowShell onto the visible window tree.

Table 7.1 Creating a Simple Application

```
   XString USING [FromSTRING, NewWriterBody, ReaderBody,
      ReaderFromWriter, WriterBody];

SampleVPTool: PROGRAM
   IMPORTS Atom, Attention, Context, Heap, MenuData, SimpleTextDisplay,
StarWindowShell, Window, XFormat, XString = BEGIN

 -- TYPEs
 Data: TYPE = LONG POINTER TO DataObject;
 DataObject: TYPE = RECORD [
  lastMouseButton: PointOrAdjust,
  place: Window.Place ← [0,0] ];

 PointOrAdjust: TYPE = {point, adjust, neither};

-- Constants
   bodyWindowDims: Window.Dims = [1000, 1000];
   sysZ: UNCOUNTED ZONE = Heap.systemZone;

-- Data
 context: Context.Type ← Context.UniqueType[];
 pointDown, adjustDown

-- Procedures
DestroyContext: PROC [data: Data, window: Window.Handle] = {
   -- Note that since Data was allocated out of the
   systemZone, this procedure is unnecessary, but it is
   included here as an example of a Context.DestroyProcType.
   The default assumes the data was allocated out of the
   systemZone and frees it from there.
      sysZ.FREE [@data];
      };

GetContext: PROC [body: Window.Handle] RETURNS [data: Data] = {
      data ← Context.Find[context, body];
      IF data = NIL THEN ERROR; --Just in case.
      RETURN [data];
      };

 Init: PROC = {
      sampleTool: XString.ReaderBody ←
         XString.FromSTRING["Sample Tool"L];
```

```
    Attention.AddMenuItem [
      MenuData.CreateItem [
      zone: sysZ,
      name: @sampleTool,
      proc: MenuProc] ];
    };

  InitAtoms: PROC = {
    pointDown ← Atom.MakeAtom["PointDown"L];
    adjustDown ← Atom.MakeAtom["AdjustDown"L];
    };

  MenuProc: MenuData.MenuProc = {
    another: XString.ReaderBody ← XString.FromSTRING["Another"L];
    repaint: XString.ReaderBody ← XString.FromSTRING["Repaint"L];
    post: XString.ReaderBody ← XString.FromSTRING["Post Message"L];
    sampleTool: XString.ReaderBody ←
      XString.FromSTRING["Sample Tool"L];

    -- Create the StarWindowShell.
    shell: StarWindowShell.Handle = StarWindowShell.Create [
      name: @sampleTool];

    -- Create a body window inside the StarWindowShell.
    body: Window.Handle = StarWindowShell.CreateBody [
      sws: shell,
      box: [ [0,0], bodyWindowDims ],
      repaintProc: Redisplay,
      bodyNotifyProc: NotifyProc ];

    -- Create some menu items and a menu.
    z: UNCOUNTED ZONE ← StarWindowShell.GetZone [shell];
    items: ARRAY [0..3) OF MenuData.ItemHandle ← [
      MenuData.CreateItem [zone: z, name: @another, proc: MenuProc],
      MenuData.CreateItem [zone: z, name: @repaint,
        proc: RepaintMenuProc],
      MenuData.CreateItem [zone: z, name: @post, proc: Post]
      ];
    myMenu: MenuData.MenuHandle = MenuData.CreateMenu [
      zone: z,
      title: NIL,
      array: DESCRIPTOR [items]];

    -- Add the menu to the StarWindowShell header.
    StarWindowShell.SetRegularCommands [sws: shell, commands:
myMenu];

    -- Create a context for the body window.
    Context.Create [
      type: context,
      data: sysZ.NEW[DataObject ← [neither] ],
      proc: DestroyContext,
      window: body];

    -- Put the StarWindowShell on the screen.
    StarWindowShell.Push [shell];
    };

  NotifyProc: TIP.NotifyProc = {
    data: Data ← GetContext [window];
    FOR input: TIP.Results ← results, input.next UNTIL input = NIL DO
      WITH z: input SELECT FROM
        coords = > data.place ← z.place;
        atom = > SELECT z.a FROM
```

```
                pointDown = > data.lastMouseButton ← point;
                adjustDown = > data.lastMouseButton ← adjust;
                ENDCASE;
              ENDCASE;
            ENDLOOP;
        Redisplay [window]};

--This procedure is called when the user invokes the Post command.
    Post: MenuData.MenuProc =
        msg: XString.ReaderBody ←
            XString.FromSTRING ["This is a sample attention window
message."L];
        Attention.Post [@msg];
        };


    Redisplay: Window.DisplayProc = {
        data: Data ← GetContext [window];
        writerBody: XString.WriterBody ← XString.NewWriterBody [
            maxLength: 250, z: sysZ];
        xfo: XFormat.Object ← XFormat.WriterObject [w: @writerBody];

        XFormat.String [h: @xfo, s: "This is a sample string displayed in a body
window of a StarWindowShell using
SimpleTextDisplay.StringIntoWindow."];
  XFormat.String [h: @xfo, s: SELECT data.lastMouseButton FROM
    point = > "Point"L,
    adjust = > "Adjust"L,
    ENDCASE = > "Neither"L];
  XFormat.String [h: @xfo, s: " and the mouse was at window relative
location: [x: "L];
  XFormat.Decimal [h: @xfo, n: data.place.x];
  XFormat.String [h: @xfo, s: ", y: "L];
  XFormat.Decimal [h: @xfo, n: data.place.y];
  XFormat.Char [h: @xfo, char: '].ORD ];
        [] ← SimpleTextDisplay.StringIntoWindow [
            string: XString.ReaderFromWriter [@writerbody],
            window: window, --The body window
            place: [10,10], --Upper- left corner is [0,0]
            lineWidth: 300,   --Arbitrary (in pixels)
            maxNumberOfLines: 10, --Arbitrary
            flags: Display.replaceFlags]];; --Clear old data
    };

    RepaintMenuProc: MenuData.MenuProc = {
        body: Window.Handle = StarWindowShell.GetBody[[window]];
        Window.InvalidateBox[body, [[0, 0], [30000, 30000] ]];
        Window.Validate[body]; };

    -- Mainline code

    Init[];
    InitAtoms[];

    END...
```

8. FORM WINDOWS AND PROPERTY SHEETS

Form windows and property sheets are specialized windows that provide an intuitive and consistent user interface for invoking commands and setting parameters. This chapter discusses how to write an application that uses form windows.

8.1 Form windows

If your application has commands that require parameters, you should put those commands and parameters in a *form window*. Form windows are made up of *form items*, such as commands, booleans, and strings; each form item has a specific user interface. The form window is based on the abstraction of a form, such as a personnel form or an income tax form, that has specific blanks to be filled in by the person using it. A form window contains keywords, such as the name of a command or parameter, and space for the user to fill in values for those parameters. The user fills in the appropriate parameters and then invokes a command. Form windows thus standardize parameter collection and reduce the restrictions on the order in which parameters are provided. Figure 8.1 illustrates a form window, showing the possible form items.

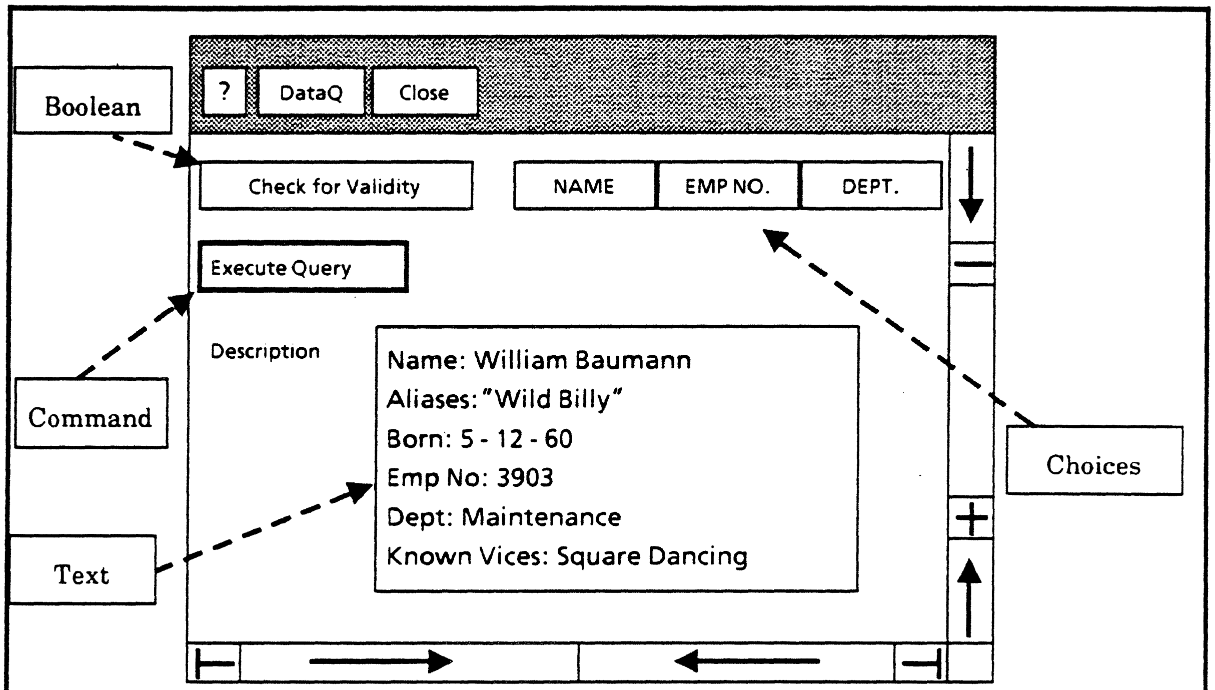
8.1.1 Creating a form window

You create a form window by calling FormWindow.Create:

```
Create: PROCEDURE[
   window: Window.Handle,
   makeItems: MakeItemsProc,
   layoutProc: LayoutProc ← NIL,
   windowChangeProc: GlobalChangeProc ← NIL,
   zone: UNCOUNTED ZONE ← NIL,
   clientData: LONG POINTER ← NIL ];
```

Create takes an ordinary window and makes it a form window. Typically, the window that you pass to this procedure will be one that you created by calling StarWindowShell.CreateBody.

The three most imortant parameters are the call-back procedures **makeItems**, **layoutProc**, and **windowChangeProc**. **makeItems** is responsible for creating the items that you want in your form window; **layoutProc** specifies the desired position of the items in the window, and **windowChangeProc** is called whenever the user changes the value of an an item in the window. We discuss each of these three call-back procedures in detail in the following sections.



Name: William Baumann
Aliases: "Wild Billy"
Born: 5 - 12 - 60
Emp No: 3903
Dept: Maintenance
Known Vices: Square Dancing

boolean item	has two states: on and off (or TRUE and FALSE). When the value is TRUE, the item is highlighted.
choice item	has an enumerated list of choices, only one of which can be selected. A choice item's value is of type FormWindow.ChoiceIndex.
text item	a user-editable string; its value is of type XString.ReaderBody.
decimal item	a text item that has a value of type XLReal.Number.
integer item	a text item that has a value of type LONG INTEGER.
command item	has an associated procedure. When the user invokes the command, the procedure is called.
tagonly item	a string that the user can neither select nor edit.
window item	a window that is a child of the form window and can contain anything you like. A window item's value is a Window.Handle.

Figure 8.1. Form window

zone is the zone out of which storage for the items will be allocated. If you don't supply a zone, **FormWindow** will use its own private zone.

clientData is passed to **makeItems**, **layoutProc**, and **windowChangeProc**. This parameter is for your own use; if there is any additional information that you want to pass to your **makeItems**, **layoutProc**, or **windowchangeProc**, you can do it via **clientData**.

8.1.2 Creating form items

The first step is to write a **MakeItems** procedure to pass to Create. This procedure should be of type FormWindow.MakeItemsProc:

```
MakeItemsProc: TYPE = PROCEDURE [
   window: Window.Handle,
   clientData: LONG POINTER];
```

This procedure is responsible for creating the various items that you want to have displayed in your form window. **FormWindow** provides a procedure for making each type of item: **MakeBooleanItem**, **MakeChoiceItem**, **MakeCommandItem**, **MakeDecimalItem**, **MakeIntegerItem**, **MakeMultipleChoiceItem**, **MakeTagOnlyItem**, **MakeTextItem**, **MakeWindowItem**. Thus, all you need to do in this procedure is call the appropriate procedure for each item that you want in your form window.

Here is a part of the **MakeItemsProc** for the application illustrated in Figure 8.1:

```
MakeFormItems: FormWindow.MakeItemsProc = {
  boolLabel: XString.ReaderBody ←
       XString.FromSTRING["Check for Validity"L];
  commandLabel: XString.ReaderBody ←
       XString.FromSTRING["Execute Query"L];
  choice1Label:  ... ("NAME")
  choice2Label:  ... ("EMP NO.")
  ...
  FormWindow.MakeBooleanItem [
     window: window, myKey: 1, label: boolLabel];
  FormWindow.MakeCommandItem [
     window: window,
     myKey: 2,
     commandProc: CommProc,
     commandName: commandLabel;
  FormWindow.MakeTextItem [
     window: window, myKey: 3, ...];
  FormWindow.MakeChoiceItem [window: window, myKey: 4,
  ...];
  ...};
```

8.1.3 MakeCommandItem

As an example of the make item procedures, we discuss **MakeCommandItem**. If you want information on the other procedures, consult the FormWindow chapter of the *ViewPoint Programmer's Manual*.

```
  MakeCommandItem: PROCEDURE [
     window: Window.Handle,
     myKey: ItemKey,
     tag: XString.Reader ← NIL,
     suffix: XString.Reader ← NIL,
     visibility: Visibility ← visible,
     boxed: BOOLEAN ← TRUE,
     readOnly: BOOLEAN ← FALSE,
commandProc: CommandProc,
commandName: XString.Reader];
```

This procedure creates a command item. The first seven parameters are common to all the procedures that create form items.

window is the form window that the item is in. (This should be the same as the window passed to your **MakeItemsProc**.)

myKey is a key that you define for the item. The item key uniquely identifies the item and should be used to make calls on other **FormWindow** procedures. The key must be *unique within this form window*.

tag is the text to be displayed before (to the left of) the item on the same line. The default is for no tags.

suffix is the text to be displayed after (to the right of) the item on the same line.

visibility indicates whether the item should be displayed on the screen. If an item is displayed in the form window, it is **visible**. If an item is not currently displayed, it is either **invisible** or **invisibleGhost**. If it is invisible, it does not take up any space on the screen, i.e. any items below it move up to take its screen space. If an item is **invisibleGhost**, the space that it would occupy were it visible is white on the screen. You can change an item's visibility by calling FormWindow.**SetVisibility**.

boxed indicates whether the item should have a box drawn around it.

readOnly = TRUE indicates whether the user can change the value of the item. If an item is **readOnly**, you can still change the value by calling appropriate procedures in the **FormWindow** interface.

Only the last two parameters are specific to a command item. The **commandName** is the name that will appear in the form window. When the user clicks over the **commandName**, **commandProc** is called. The **commandProc** is of type FormWindow.**CommandProc**:

```
CommandProc: TYPE = PROCEDURE [
   window: Window.Handle,
   item:ItemKey];
```

8.1.4 Layout

Once you have written a **MakeFormItems** procedure to create the items in your form window, you need to write a **LayoutProc** to specify how those items are to be displayed. The layout procedure must be of type FormWindow.**LayoutProc**:

```
LayoutProc:TYPE = PROCEDURE [
   window: Window.Handle,
   clientData: LONG POINTER];
```

If an item is not explicitly laid out, it will not appear in the form window at all. If you don't want to write your own layout procedure, you can use FormWindow.DefaultLayout, which places each item on a separate line. If you prefer to write your own layout procedure, you can use either *flexible layout* or *fixed layout*.

Flexible layout allows text, decimal, integer, and window items to grow and shrink (and other items to move around accordingly) as the user or program changes values. Fixed layout, on the other hand, does not allow any movement; you specify where the items are to go, and they remain there until you explicitly move them. Flexible layout is the preferred method.

8.1.4.1 Flexible layout

A form window with flexible layout consists of horizontal lines with zero or more items on each line. Each line may be a different height, but should be at least FormWindow.defaultLineHeight to avoid overlap. You can control vertical spacing by using appropriate heights for the lines. Similarly, you can control horizontal spacing by using appropriate margins between items. Items may be lined up horizontally with TabStops; see the *ViewPoint Programmer's Manual* for details.

The first step to creating a layout is to create a line by calling either FormWindow.AppendLine or FormWindow.InsertLine. Once you have a line, you put items on that line by calling FormWindow.AppendItem or FormWindow.InsertItem. The Append routines add items after the previously created line or item; the Insert routines add items between previously created items or lines.

```
AppendLine: PROCEDURE [
   window: Window.Handle,
   height: CARDINAL ← defaultLineHeight]          --In pixels
   RETURNS [line: Line];

InsertLine: PROCEDURE [
   window: Window.Handle,
   before: Line,
   height: CARDINAL ← defaultLineHeight]
   RETURNS [line: Line];

AppendItem: PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   line: Line,
   preMargin: CARDINAL ← 0,
   tabStop: CARDINAL ← nextTabStop,
   repaint: BOOLEAN ← TRUE];

InsertItem: PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   line: Line,
   beforeItem: ItemKey,
   preMargin: CARDINAL ← 0,
   tabStop: CARDINAL ← nextTabStop,
   repaint: BOOLEAN ← TRUE];
```

Here is an example of a layout procedure using the flexible method:

```
LayoutFormItems: FormWindow.LayoutProc = {
   line: FormWindow.Line ← FormWindow.AppendLine
      [window: window,
      -- height ← defaultLineHeight --];
   FormWindow.AppendItem [
      window: window, item: 1, line: line];
   FormWindow.AppendItem
      [window: window, item: 4, line: line];
   line ← FormWindow.AppendLine [
      window: window];
   FormWindow.AppendItem [
      window: window, item: 2, line: line];
   line ← FormWindow.AppendLine [window: window];
   FormWindow.AppendItem [
      window: window, item: 3, line: line];
   };
```

8.1.4.2 Fixed layout

With fixed layout, you call **FormWindow.SetItemBox** to specify the exact position of each item:

```
SetItemBox: PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   box: Window.Box];
```

With this method, all items stay where you put them unless you make another call to **SetItemBox**. Thus, text, decimal, integer, and window items will not grow or shrink. **SetItemBox** is incompatible with flexible layout: either all layout must be flexible, or all layout must be fixed. Here is an example of laying out a window using the fixed method:

```
LayoutFormItems: FormWindow.LayoutProc = {
   FormWindow.SetItemBox [
      window: window, item: 1, box: [[10,20],[60,20]];
   FormWindow.SetItemBox [w
      indow: window, item: 2, box: [[10,50],[45,20]];
   FormWindow.SetItemBox [
      window: window, item: 3, box: [[20,80],[150,120]];
   FormWindow.SetItemBox [w
      indow: window, item: 4, box: [[70,20],[60,80]];
   };
```

8.1.5 Recognizing changes in the form window

When the user changes something in the form window, you typically need to recognize that change and act upon it. There are three ways that you can monitor changes in a form window: with a global change procedure, with a local change procedure, or with a changed boolean. A global change procedure is a procedure that is called whenever a user or a program changes the value of an item in the form window. A **GlobalChangeProc** is called whenever *anything* in the form window changes. You associate a **GlobalChangeProc** with a window by passing it in as a call-back procedure in the call to **FormWindow.Create**.

You can also associate local change procedures with particular kinds of items, such as booleans and choice items. You can associate a local change procedure with an item when you make the item. (Note: if a window has both a global change procedure and a local change procedure, the local one will be called first.)

The third way to keep track is with the "changed boolean." Every item that has a value that the user can change (all except tag-only, command, and window items) has a changed boolean associated with it. When an item is created, this boolean is set to FALSE. When the user changes the value of the item, **FormWindow** automatically sets the boolean to TRUE. Once you look at a boolean and act accordingly, you are responsible for setting its value back to FALSE.

8.1.6 Getting and setting values

Every item that has a value that the user can change (all except tagonly and command items) also has procedures for the client to get and set the value. For example, the procedures for boolean items are called **GetBooleanItemValue** and **SetBooleanItemValue**. See the *ViewPoint Programmer's Manual* for details.

8.1.7 Destroying a form window

Destroy: PROCEDURE [window: Window.Handle];

Destroy destroys all **FormWindow** data associated with **window**, turning it back into an ordinary window. This procedure does not destroy the window itself; it destroys the form items within the window. You can also use either **FormWindow.DestroyItem** or **FormWindow.DestroyItems** to destroy individual items without destroying all of the items in the window.

8.2 Property sheets

In addition to functioning as forms to be filled in, form windows also function as the basis for property sheets. Figure 8.2 shows a generic property sheet.

Property sheets are specialized forms that show properties of an object. *Object* is a very general term and can refer to almost anything; printer icons, documents, paragraphs within documents, mail baskets, and the Directory icon are all examples of objects that have properties. *Properties* for a paragraph in a document include margins, whether to justify, and line spacing, for example. Properties for a printer icon include number of copies to print and paper size. User-supplied applications have properties, too; naturally they vary according to the semantics of the objects defined by the application.

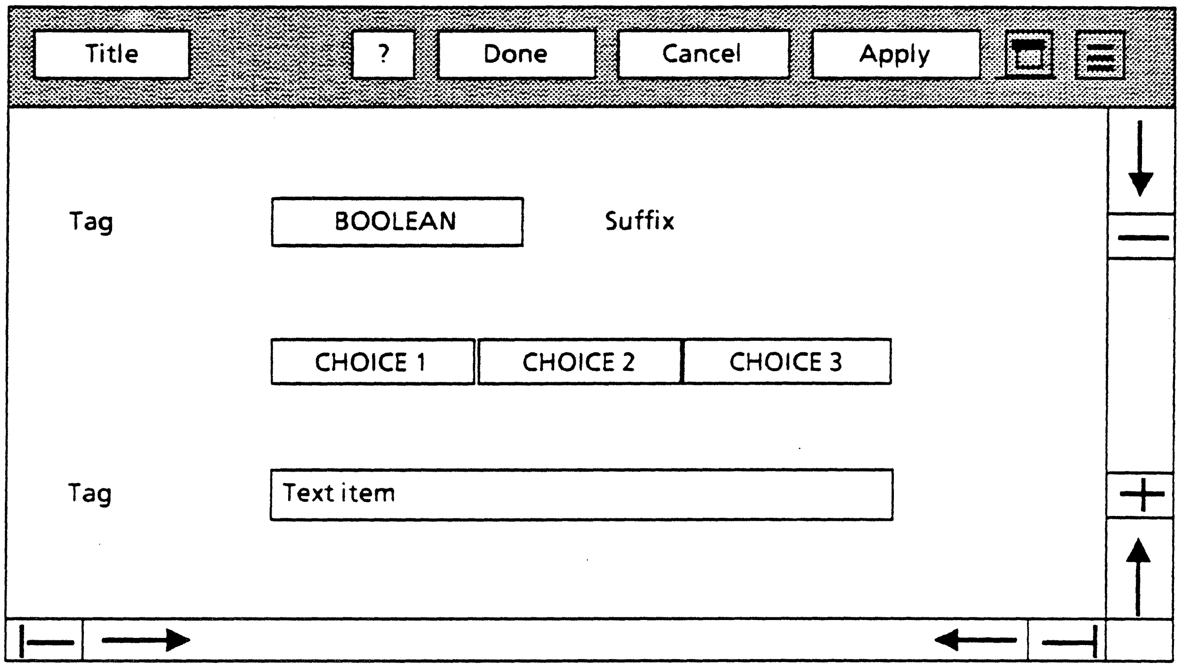


Figure 8.2. Property sheet

8.2.1 Creating a property sheet

A property sheet is a StarWindowShell with a formwindow. Therefore, creating a property sheet requires that you first create a form window by making form items, laying them out, and writing procedures for items that require some action. The major difference between creating a form window and a property sheet is that you do not have to call FormWindow.Create to change the body window into a form window; instead, call PropertySheet.Create to get a property sheet. Another significant difference is that you will need a procedure that updates the object's properties when the user modifies items in the property sheet. ApplyAnyChanges below is an example of such a procedure:

```
MakePropertySheet: PROC [...] = {
   mh: XMessage.Handle = Defs.GetMessageHandle[];
   title: XString.ReaderBody ← XMessage.Get [mh, Defs.xxx];
   pSheetShell: StarWindowShell.Handle ← PropertySheet.Create[
        formWindowItems: MakeFWItems,
        menuItemProc:MyMenuItemProc,
        menuItems: [done:TRUE, cancel:TRUE],
        size:size,
        title: @title,
        formWindowItemsLayout: DoLayout]];
   };

MakeFWItems: FormWindow.MakeItemsProc = {
   ...
   };

MyMenuItemProc: PropertySheet.MenuItemProc = {
   SELECT menuItem FROM
   done => RETURN [ok:ApplyAnyChanges[formWindow].ok];
   cancel => RETURN [ok:TRUE];
   ENDCASE;
RETURN [ok:FALSE];
};

   DoLayout: FormWindow.LayoutProc = {
     ...
     };

   ApplyAnyChanges: PROC [fw:Window.Handle] RETURNS
   [ok:BOOL] = {
     IF NOT FormWindow.HasAnyBeenChanged[fw] THEN RETURN
     ok:TRUE];
     Otherwise, check each item in the form; if it's been
     changed, make the appropriate update
     . . .
     };
```

8.2.2 Linked property sheets

One property sheet can contain several other property sheets. These sheets are considered to be linked, and each has its own form window. Only one form window is displayed at a time; the displayed form window is indicated by a choice item in a link window. The **Text Property Sheet** available with the document editor is an example of a linked property sheet. In the link window you see choices for Character, Paragraph, and Tab Setting property sheets; selecting one of them causes the appropriate sheet to be displayed. Figure 7.3 shows a generic linked property sheet with three possible property sheets: PSHEET1, PSHEET2, and PSHEET3.

To create a linked property sheet, first create form windows for all choices. Next, declare a **ChoiceChangeProc** that swaps form windows each time the user selects a new property sheet. Last, in your **MakeItemsProc**, make a choice item for the property sheet choices, initializing the window to display the desired initial property sheet and passing in your **ChoiceChangeProc** for swapping the form windows. For example:

```
   MakePropertySheet: PROC [...] = {
   ...
   pSheetShell: StarWindowShell.Handle ←
     PropertySheet.CreateLinked [
       formWindowItems: MakeSheet1,
       menuItemProc: MakeMenuItems1,
       size: ...,
       linkWindowItems: MakeLinkWindowItems,
       linkWindowItemsLayout: NIL];          --Use default
   layout of link window
     };

   MakeLinkWindowItems: FormWindow.MakeItemsProc = {
   ...
   --Create an array of FormWindow.ChoiceItem
   FormWindow.MakeChoiceItem [
     window:window, myKey:Items.radix.ORD,
     tag:@rb, values:DESCRIPTOR[radixChoices],
     initChoice:1,
```

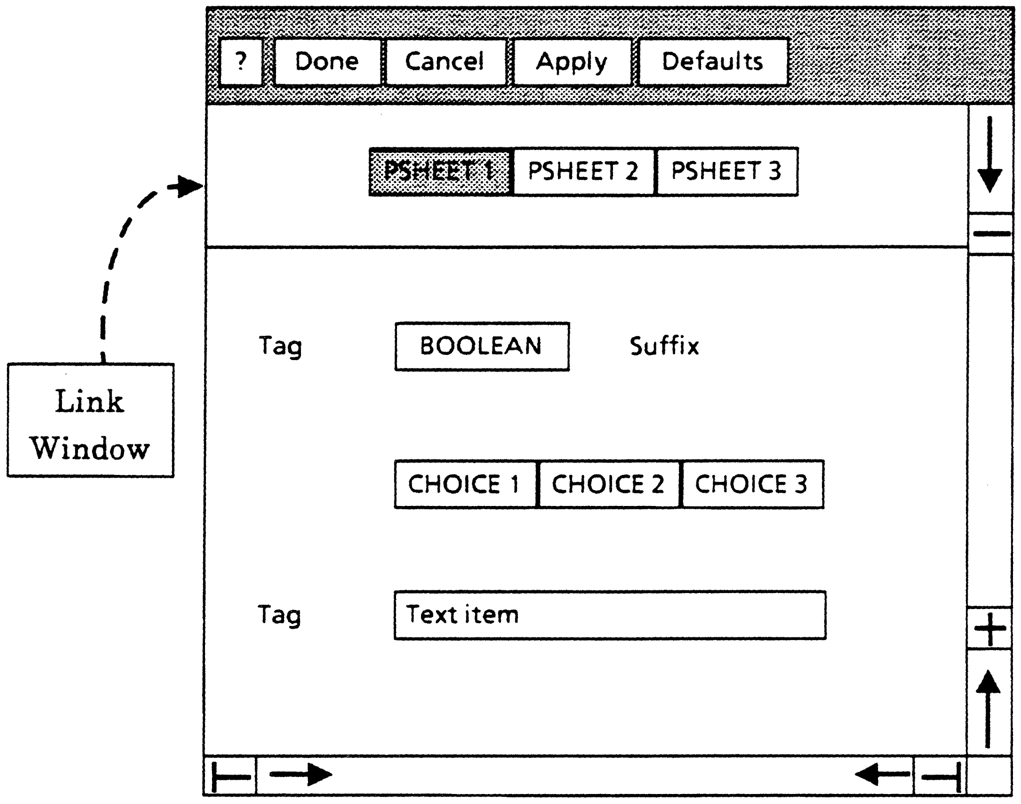


Figure 8.3 Linked property sheet

```
   changeProc: ChangeFormWindow];
};

ChangeFormWindow: FormWindow.ChoiceChangeProc = {
SELECT newValue FROM
   1 => PropertySheet.SwapFormWindows [
           shell:pSheetShell,
           newFormWindowItems: MakeSheet1];
   2 => PropertySheet.SwapFormWindows [
           shell:pSheetShell, newFormWindowItems:
           MakeSheet2];
   3 => PropertySheet.SwapFormWindows [
           shell:pSheetShell, newFormWindowItems:
           MakeSheet3];
ENDCASE;
```

9. ICON APPLICATIONS

As mentioned earlier, you can write applications to run either from a command in the Attention menu or from an icon. We discussed the Attention window method in Chapter 7; this chapter covers how to write an icon application.

9.1 What is an icon?

Icons are pictorial representations of applications. You can operate on them in various ways, with each operation carrying an application-specific significance. Generally, you can SELECT, OPEN, COPY and MOVE icons. The OPEN operation is generally implemented similarly across applications and is usually associated with opening and displaying the application's window. Copying and moving on the other hand, tend to have more application-specific semantics. For example, if you copy or move a document to a folder icon, it will be added to the list of documents in that folder. However, moving or copying documents to a printer icon is quite different; it causes an associated Interpress master to be generated and transferred to a local printer.

9.2 File types and icon applications

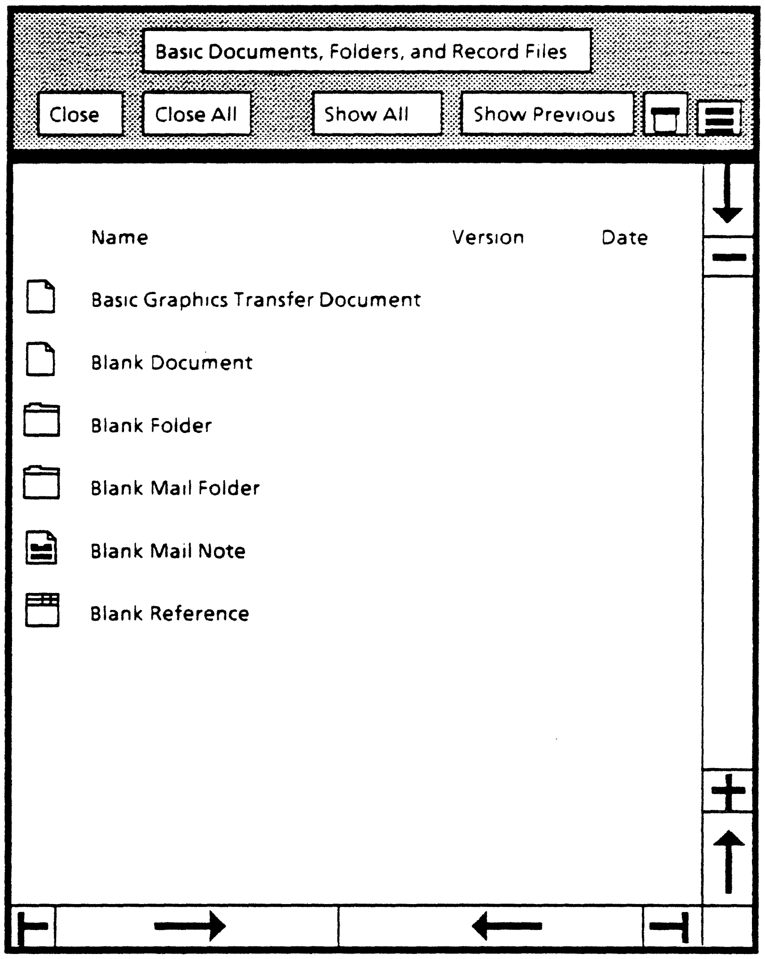
Applications represented by icons are software packages that implement the manipulation of one type of file. All icons corresponding to a particular application have the same file type and are distinct from icon file types of other applications. Icon applications "register" with the desktop (see below) and in so doing, they state the type of file on which they operate.

Since file types must be unique across applications, it is important to establish a central distribution mechanism to keep track of previously allocated file types. During development, you can pick an arbitrary file type with the single constraint that it be distinct from other applications on the workstation. However, when you want to distribute your application for public use, it cannot use an arbitrary file type but instead must use one guaranteed to be different from all other public applications. Only a central distribution mechanism can provide such a guarantee.

File types and other filing capabilities are provided by the **NSFile** interface. See the *Services Programmer's Guide* for more information on **NSFile**.

9.3 Getting icons on the desktop: the Prototype folder

When you run an application, its icon does not automatically appear on the desktop. Instead, you must open a specialized system folder, known as the Prototype folder, select the desired icon in it, and copy it to the desktop. Figure 9.1 illustrates the Prototype folder when opened.



Name | Version | Date

Basic Graphics Transfer Document
Blank Document
Blank Folder
Blank Mail Folder
Blank Mail Note
Blank Reference

Figure 9.1 Prototype folder

9.3.1 Opening the Prototype folder: the user's perspective

There are two ways for a user to access the Prototype folder. The first method is automatically included with the ViewPoint boot file, but is initially confusing, since the prototype folder is called Basic Documents, Folders, and Record Files. To access it, open the Directory icon, then the Workstation folder, and finally open the Basic Documents, Folders, and Record Files folder. Inside the Basic Documents folder you will probably find such icons as Blank Folder, Blank Document, and any additional icons associated with system and user-supplied applications.

The second method is a shortcut, and requires that the program **SystemFolder** be running. To run **SystemFolder**, start it either from CommandCentral or by dropping it on the ViewPoint Loader icon. (See Chapter 3 for details on how to run an application.) **SystemFolder** registers three commands in the Attention window menu, one of which is Prototype Folder. Simply select the command to open the Prototype folder.

9.3.2 Putting icons in the Prototype folder: the programmer's perspective

Your icon application must include code that places its icon in the Prototype folder so that the user can select and copy it to the desktop. Icons in the Prototype folder, referred to as prototype files, are uniquely identified by file **type**, **subtype**, and **version**. **subtype** distinguishes objects of the same **type**; **version** helps determine if the prototype is current. Use the ViewPoint interface **Prototype** for creating prototype files. Its main procedures are **Find** and **Create**:

```
Find: PROCEDURE [type: NSFile.Type,
   version: Version,
   subtype: Subtype ← 0,
   session: NSFile.Session ← NSFile.nullSession]
   RETURNS [reference: NSFile.Reference];

Create: PROCEDURE [
   name: XString.Reader,
   type: NSFile.Type,
   version: Version,
   subtype: Subtype ← 0,
   size: LONG CARDINAL ← 0,
   isDirectory: BOOLEAN ← FALSE,
   session: NSFile.Session ← NSFile.nullSession]
   RETURNS [prototype: NSFile.Handle];
```

Find returns a reference for the file with the specified type, version, and subtype. If the file doesn't exist, **Find** returns **NSFile.nullReference**. **Create** creates a file in the Prototype catalog with the specified name, type, version, subtype, size (in bytes), and **isDirectory** attribute. The following code fragment shows typical usage of **Prototype.Find** and **Prototype.Create**:

```
sampleType: NSFile.Type = ...;
version: CARDINAL = ...;

--This procedure is called from the mainline code
FindOrCreateIconFile: PROCEDURE [name: XString.ReaderBody]
= {
   IF Prototype.Find[type:sampleType, version:version] =
   NSFile.nullReference
      THEN NSFile.Close[Prototype.Create[
         name:@name, type:sampleType, version:version]];
```

In this example, **Find** is called; if that call returns **NSFile.nullReference**, then **Create** is called. **Create** returns an **NSFile.Handle** (opens the file), so **NSFile.Close** is called to release the handle.

9.4 Registering an application with the desktop

As mentioned above, icon applications must register themselves with the desktop. Registration informs ViewPoint of the application's existence, states the file type the application will manipulate, and defines operations that can be performed on it. You register an icon by calling Containee.SetImplementation and passing to it the application's file type and a number of procedures. These procedures specify operations such as how to paint the application's icon, what to do when the user OPENs the icon, what to do when the user selects the icon,.

```
Containee.SetImplementation: PROCEDURE[
   NSFile.Type, Implementation] RETURNS[Implementation];

Implementation: TYPE = RECORD [
   implementors: LONG POINTER ← NIL,
   smallPicture: XString.Character ← XChar.null,
   smallRefPicture: XString.Character ← XChar.null,
   pictureProc: PictureProc ← NIL,
   convertProc: Selection.ConvertProc ← NIL,
   genericProc: GenericProc ← NIL];
```

The first parameter of SetImplementation is the file type the application will operate on. The second parameter is a record containing state data and procedures of type Containee.PictureProc for displaying the icon picture, and Containee.GenericProc for performing various operations on the icon, such as OPEN. Don't worry about the other fields in the Implementation record for now.

9.4.1 Painting icons

If you want your icon to have a picture, you need to write a Containee.PictureProc and pass it to Containee.SetImplementation. There are two methods for defining and using the actual iconic bitmap. The first solution involves determining the bitmap values and hardcoding them into the implementation as shown in the following example:

```
InitBigPicture: PROC = {
   myIconPic ← Space.ScratchMap[1].pointer;
   myIconPic ↑ .← [177777B, 177777B, 000063B, ...] --
   Hardcoded bitmap
   };

PaintIconName: PROC [...] = {
   ...
   [] ← SimpleTextDisplay.StringIntoWindow [...];
   };

PictureProc: Containee.PictureProc = {
   textBox: Window.Box ← [[x:7, y:10],[w:55, h:36]];
   name: XString.ReaderBody;
   ticket: Containee.Ticket;
   IF new = garbage THEN RETURN;
   box.dims ← [64,64];
   [name, ticket] ← Containee.GetCachedName[data];
   SELECT old FROM
```

```
      garbage,ghost = > {Display.Bitmap[
        window:window, address:myIconPic,...];
        PaintIconName[window, box, textBox, @name]};
    ENDCASE;
    SELECT new FROM
      highlighted = > Display.Invert[window, box];
      ghost = > {Display.White[window, box];
            PaintIconName[window, box, textBox,
    @name]};
    ENDCASE;
    Containee.ReturnTicket[ticket];
    };
```

The second method is to use icon files. An icon file associates an icon bitmap with a file type. When an application is loaded, the icon file in the application folder is opened to read the bitmap and display the icons. You can create new icon files or edit existing ones with the icon editor, which is discussed in Chapter 12.

If you are going to use an icon file, rather than a hardcoded bitmap, you can either use an icon file from the standard directory, or you can package an icon as part of your application. We discuss packaging an application in Chapter 11. If you choose to use icon files, you should omit the reference to the **pictureProc** in your **SetImplementation** procedure.

9.4.2 Generic procs

The second kind of procedure that you need to include in your **Impelmentation** record that you pass to **SetImplementation** is a **GenericProc**. A **GenericProc** is where most of the real implementation for an application resides. A **GenericProc** is of type **Containee.GenericProc**:

```
GenericProc: TYPE = PROCEDURE [
   atom: Atom.ATOM,
   data: DataHandle,
   changeProc: ChangeProc ← NIL,
   changeProcData: LONG POINTER ← NIL]
   RETURNS [LONG UNSPECIFIED];
```

The **atom** specifies which operation to perform. For example, when the user selects an icon and presses the OPEN key, the application's **GenericProc** is called with an **atom** of "Open." Here is an example of a **GenericProc**; for a complete discussion of **GenericProcs** and a list of possible atoms, see the **Containee** chapter of the *ViewPoint Programmer's Guide*.

```
GenericProc: Containee.GenericProc = {
   SELECT atom FROM
      canYouTakeSelection = >  RETURN[
            IF CanITake[] THEN @true ELSE @false];
      open = >   RETURN[
            MakeShell[data, changeProc, changeProcData]];
      props = > RETURN[
            Defs.MakePropertySheet[
            data, changeProc, changeProcData]];
      takeSelection, takeSelectionCopy = > RETURN[
            IF Take[data, changeProc, changeProcData] THEN
   @true ELSE @false;
ENDCASE => RETURN[
        oldImpl.genericProc[
           atom, data, changeProc, changeProcData]];
};
```

9.5 Examples

A general program format for registering an application with the desktop is:

1. Initialize the atoms you want reconized from the **GenericProc** (see Chapter 6 for more information on TIP and atoms).
2. Write the **GenericProc** that determines what operations are performed on the file associated with the application.
3. Write picture procs for iconic representation of the file on the desktop.
4. Register the application with the desktop, passing the **GenericProc** and picture proc.

The following program fragment illustrates registering an application with the desktop:

```
open, props, canYouTakeSelection,
takeSelection, takeSelectionCopy: Atom.ATOM ← Atom.null;

InitAtoms: PROC = {
   open ← Atom.MakeAtom["Open"L];
   props ← Atom.MakeAtom["Props"L];
   takeSelection ← Atom.MakeAtom["TakeSelection"L];
   takeSelectionCopy← Atom.MakeAtom["TakeSelectionCopy"L];
   };

GenericProc: Containee.GenericProc = {
   SELECT atom FROM
      open = > RETURN[MakeShell[data, changeProc, changeProcData]];
      props = > RETURN[
         Defs.MakePropertySheet[data, changeProc, changeProcData]];
      takeSelection, takeSelectionCopy = > RETURN[
         IF Take[data, changeProc, changeProcData] THEN @true
         ELSE @false;
   ENDCASE = > RETURN[
         oldImpl.genericProc[atom, data, changeProc, changeProcData]];
   };

PictureProc: Containee.PictureProc = {
   textBox: Window.Box ← [[x:7, y:10],[w:55, h:36]];
   name: XString.ReaderBody;
   ticket: Containee.Ticket;
   IF new = garbage THEN RETURN;
   box.dims ← [64,64];
   [name, ticket] ←Containee.GetCachedName[data];
   SELECT old FROM
      garbage, ghost = > {Display.Bitmap[...];
         PaintIconName[window, box, textBox, @name]};
      ENDCASE;
   SELECT new FROM
```

```
        highlighted => Display.Invert[window, box];
        ghost => {Display.White[window, box];
                  PaintIconName[window, box, textBox, @name]};
        ENDCASE;
    Containee.ReturnTicket[ticket];
    };

bits: ARRAY[0..13) OF WORD ← [...];
smallIconPicture: XString.Character ←
    SimpleTextFont.AddClientDefinedCharacter[
        width: 13, height: 13, bitsPerLine: 16, bits: @bits];

SmallPictureProc: Containee.SmallPictureProc = {
    RETURN[smallIconPicture];
    };

SetImplementation: PROC = {
    oldImpl ← newImpl ← Containee.GetImplementation[myFileType];
    newImpl.convertProc ← Containee.DefaultFileConvertProc;
    newImpl.genericProc ← GenericProc;
    newImpl.pictureProc ← PictureProc;
    newImpl.smallPictureProc ← SmallPictureProc;
    [] ← Containee.SetImplementation[myFileType, newImpl];
    };
```

10. SELECTION

The **Selection** interface defines the abstraction of the user's current selection. **Selection** provides procedures that allow programs to do operations such as setting, saving, and clearing the selection.

There are two kinds of programs that use the facilities of the **Selection** interface. Most programs wish merely to obtain the *value* of the current selection in some particular format; such programs are called *requestors*. If your program is just a requestor, you don't need to understand all the details of the **Selection** interface.

The other class of programs consists of those who wish to own or set the current selection; these are called *managers*. If you are going to write such an application, you do need to understand all the details of the interface.

The goal of the **Selection** interface is that the requestor need never know, and should never care, what module is managing the selection. All that matters is whether the selection can be rendered in a suitable form. For example, suppose the user presses COPY and selects a printer icon as the destination. The printer implementation needn't know what is printable and what isn't. It simply queries the selection to determine whether it can be rendered as an Interpress master, and if so it obtains it and sends it. Otherwise, it queries whether the selection can be enumerated as a sequence of Interpress masters (as would be true of a folder, for instance). If this also fails, the object is rejected.

10.1 Some basic guidelines

The selection belongs to the user, and he should be free to change it at any time. To synchronize this correctly, changes to the selection occur only in the Notfier. (See Chapter 6 for more information on the Notifier.)

If you are writing an application that needs to read the selection, you must deal with the fact that the user can change the selection at any time that the notifier process is running. The obvious solution to this problem is that the selection should only be read from within the Notifier. This guarantees that the user cannot alter the selection while the application is reading it. Thus the first rule for dealing with the selection is:

The selection may only be read or changed in the Notifier.

Once an application returns to the Notifier, any knowledge it retains about the selection is no longer guaranteed to be valid. Similarly, if an application running in the Notifier passes some information about the selection to another process, that information may similarly be invalidated at any time. In these circumstances, the application must copy the selection's value, using Selection.Copy, Selection.Move, or Selection.CopyMove, to assure that its data remains valid. Thus the second rule for dealing with the selection is

Copy the selection's value before returning to the system or before passing it to another process.

10.2 Requestors

The fundamental operation performed by a selection requestor is to obtain the value of the current selection by calling Selection.Convert:

```
Convert: PROCEDURE [
   target:Target,
   zone: UNCOUNTED ZONE ← NIL]
   RETURNS [value: Value];

Target: TYPE = MACHINE DEPENDENT{
   window(0), shell, subwindow, string, length, position,
   integer, interpressMaster, file, fileType, token, help,
   interscriptScript, interscriptFragment, serializedFile,
   name, firstFree, last(1777B)};

Value: TYPE = RECORD [value: LONG POINTER, . . .];
```

The **target** is the TYPE of data to which the selection should be converted. The **value** is a RECORD containing a pointer to the converted selection. For example, Selection.**Convert** [**target: string**] returns an XString.**Reader**. Note that not all selections can be converted to all **Targets**; in fact most selections can be converted to only a small number of **Targets**. For example, if the selection is a text string, it can be converted to Target **string** and perhaps to **integer**, but probably not to **file** or **fileType**.

Converting to some **Targets** is not so much requesting the value of the selection as requesting some general information about the selection or its environment. For example, Selection.**Convert** [**target: window**] is a request for the window that the selection is in, Selection.**Convert** [**target: help**] is a request for user help information about the selection, etc.

You should remember that **Convert** returns a *read-only value* and you must free any storage associated with that value when you are finished. Here is an example of using Selection.**Convert**:

```
streamHandle: Stream.Handle ← GetStreamToSomeFile[];
xfo: XFormat.Object;
xfo ← XFormat.StreamObject[streamHandle];
-- Convert returns NIL if selection can't be converted
savedString: Selection.Value ← Selection.Convert[string];
IF saveString = NIL THEN RETURN;
XFormat.Reader[@xfo, LOOPHOLE[savedString]];
Stream.Delete[streamHandle];
Selection.Free[@savedString];
```

10.2.1 Can you convert the selection?

Since all selections do not convert to all target types, you may want to ask whether a given selection will convert. To ask such a question, you should call Selection.CanYouConvert. CanYouConvert returns a BOOLEAN specifying whether the value will convert to the particular target type.

```
CanYouConvert: PROCEDURE [
   target: Target, enumeration: BOOLEAN ← FALSE]
   RETURNS [yes: BOOLEAN] = INLINE {
      RETURN[HowHard[target, enumeration] # impossible]};
```

Here is an example of calling CanYouConvert. This type of procedure is generally called from the canYouTakeSelection arm of a Containee.GenericProc:

```
CanITake: PROCEDURE RETURNS[yes: BOOLEAN] =
BEGIN
   -- Take anything that is a string, token, or integer
   RETURN[
      Selection.CanYouConvert[
         target: string, enumeration: FALSE] OR
      Selection.CanYouConvert[
         target:integer,enumeration:FALSE] OR
      Selection.CanYouConvert[
         target: token, enumeration: FALSE]];
END;
```

10.2.2 Enumerating selections

A selection is often a collection of items (several files in a folder) or a single large item that can be split up (e.g., a long string that can be broken up). A requestor can ask that each item or part of such a selection be converted to some target by calling Selection.Enumerate.

```
Absorb: PROCEDURE[data: Containee.DataHandle,
   changeProc: Containee.ChangeProc ← NIL,
   changeProcData: LONG POINTER ← NIL]
   RETURNS[absorbed: BOOLEAN ← FALSE] =
   BEGIN

   AbsorbString: Selection.EnumerationProc =
      XFormat.Reader[@xfo, LOOPHOLE[element.value]];
      Selection.Free[@element];
   END;

   xfo: XFormat.Object
   fileStream: NSFileStream.Handle ← GetStream [data];
   Stream.SetPosition[
      fileStream, NSFileStream.GetLength[fileStream]];
   xfo: XFormat.StreamObject [fileStream];
   SELECT TRUE FROM
      Selection.CanYouConvert[target: file, enumeration: TRUE] = >
         [] ← Selection.Enumerate[AbsorbString, string, NIL];
      Selection.CanYouConvert[target: file, enumeration: FALSE] = >
         [] ← AbsorbString[Selection.Convert[string], NIL];
   ENDCASE;
   NSFileStream.SetLength [
      [fileStream], Stream.GetPosition[fileStream]];
```

```
    Stream.Delete[fileStream];
    -- Ensure that the file will be read again when the icon is opened
    IF changeProc # NIL THEN
      changeProc[changeProcData, data, [interpreted: [name:
TRUE]]];
END;
```

10.2.3 Resource allocation and deallocation

It is a strict rule that the Values produced by Selection.Convert and Selection.Enumerate describe objects *owned by the selection manager*. The requestor may examine the data referenced by the value field, but must not alter it. Furthermore, the requestor must free the Value (using Selection.Free) once he no longer needs it.

If you want to keep the value or pass the value to another process, you must call Selection.Copy, Selection.Move, or Selection.CopyMove. After the Move or Copy, you own any storage associated with the Value. You can free this storage by calling Selection.Free.

For example, suppose the selection manager uses a Mesa STRING as the internal selection representation. Then Convert[string] simply builds the string pointer into an XString.Reader using XString.FromSTRING. If the requestor wants to save the string for very long, he should call Copy, and the manager will allocate a copy of the original string using the zone passed to Convert. An alternative, somewhat simpler, is for the requestor to call XString.CopyReader or XString.CopyToNewReaderBody or XString.CopyToNewWriterBody to copy the bytes, and then call Selection.Free to dispose of the original Reader.

10.3 Managers

The fundamental operation performed by a selection manager is to become the current manager by calling Selection.Set:

```
Set: PROCEDURE [
  pointer: ManagerData, conversion: ConvertProc, actOn:
ActOnProc];

ManagerData: TYPE = LONG POINTER;

ConvertProc: TYPE = PROCEDURE [
  data: ManagerData,
  target: Target,
  zone: UNCOUNTED ZONE,
  info: ConversionInfo ← [convert[]] ]
  RETURNS [value: Value];

ConversionInfo: TYPE = RECORD [SELECT type: * FROM
  convert = > NULL,
  enumeration = > [proc: PROCEDURE [Value]
    RETURNS [stop: BOOLEAN]],
  query = > [
    query: LONG DESCRIPTOR FOR ARRAY OF QueryElement],
  ENDCASE];
```

```
ActOnProc: TYPE = PROCEDURE [data: ManagerData,
  action: Action]
  RETURNS [cleared: BOOLEAN ← FALSE];Selection.HowHard.
```
ConversionInfo is a variant record passed to the **ConvertProc** that indicates which operation to perform: **convert, enumeration,** or **query.**

The **ActOnProc** is called to perform various **Actions** on the selection, such as **mark, unmark,** and **clear.**

The **ManagerData** passed to **Set** is passed back to the **ConvertProc** and the **ActOnProc**. Typically, the **ManagerData** identifies exactly what portion of the manager's domain is currently selected. For example, if the current selection is some text in a document, the actual manager is the document application, which has some **ManagerData** that indicates exactly which characters are currently selected.

When a manager calls **Selection.Set**, the previous manager is told to **ActOn [clear]**, and **Selection** forgets about the previous manager. Hence, there is only one selection at a time. However, **Selection** also supports the notion of a "saved" selection. A client can become the current manager by calling **Selection.SaveAndSet**, which does a **Set** but also saves the previous selection. Later, the manager that did the **SaveAndSet** can do a **Selection.Restore**, which restores the previous selection.

For more details on managing the selection, consult the **Selection** chapter of the *ViewPoint Programmer's Manual*.

11. PACKAGING AN APPLICATION

During the last stages of development, you generally focus your attention on testing and debugging the code modules of your application. You load and run your modules from CommandCentral in XDE, and use standard icons provided by ViewPoint. If your application posts messages, they are probably contained in a message bcd.

However, once your code is running to your satisfaction, it is time to package it as a finished application. You may want to design an icon specifically for your application; if so, you will generate an icon file. If your application posts messages and is potentially multilingual, you should create a separate message file from your message bcd with the **MessageTool**. Having a separate message file makes it easy for a translator to convert all of the application's messages to another language.

The motivation behind packaging an application is to make the details of finding the application and all of its dependent files transparent to the user. Instead of being represented as a collection of files, a packaged application is a single object. Additionally, packaging makes multinationalization easier, since it permits language dependent aspects, such as message and keyboard files, to be extracted, translated, and reinserted without touching the application code.

To facilitate the packaging of an application, ViewPoint provides an object called an *application folder*. An application folder groups the components of an application together into one object.

This chapter defines the components of an application, describes the steps for building an application folder, and gives examples of the code you need to add to your modules before you put them in an application folder. For more information on the topics in this chapter, consult the *ViewPoint Programmer's Manual*.

11.1 Building an application folder

There are five steps to building an application folder:

- Identify the application's components.
- Build the data files for the application (such as messge files, icon files, TIP files).
- Write an application description file.
- Change or add code to access the data files in the application folder.
- Integrate the components into an application folder.

11.1.1 Identify the components of an application

An *application* consists of its object files and data files. The object files contain the executable application code. There must be at least one object file.

An application may also have the following data files:

- A message file if the application posts messages. This makes multilingual conversion easier.
- An icon file if the application's icon is not in the standard icons file. For example, you might want your application to be represented by a picture of a duck. Since a duck icon is not part of the standard ViewPoint package, you would have to create your own duck icon file and include it in the application folder.
- One or more TIP files if the application requires its own TIP files. An application requires its own TIP files if it redefines the meaning of the keyboard in any way; for example, having the MOVE key map to the COPY operation.
- A keyboard file if the application uses a keyboard that is not in the standard keyboard file.
- Other private data files, such as translation tables. If your application translates ASCII to EBCDIC, you might wish to include a translation table for this purpose.

11.1.2 Build the data files

There are four common types of data files: message, icon, keyboard, and TIPC. Detailed documentation for creating message, icon, and keyboard files is contained in the Tools section of this manual. Use the following table as a quick reference:

Data File	Chapter: Title
Message file	13: Message Tools
Icon file	12: Icon Editor
Keyboard file	15: Keyboard Tool & 14:Bitmap Edit Tool
TIPC file	6:TIP

11.1.3 Write an Application Description File

In addition to the application components, an application folder contains an *Application Description File (ADF).* There is only one ADF in each application folder. The ADF contains the names of the data files in the application folder, somewhat like a table of contents. The application uses the ADF to determine the names of the data files in its application folder so it can open and read them. An ADF is required so that data file names are not hardcoded into the application; if they were, multilingual applications would be more difficult.

An ADF also indicates the loading priority for an application. The priority is important only for an application that depends on another application. In such a situation, the application that must start first has a lower priority number than the dependent application.

An ADF's file type must be that of an option file, and it is read using the **OptionFile** interface. See the OptionFile chapter of the *ViewPoint Programmer's Manual* for details of option files. The syntax for an ADF follows option file syntax, and is given at the end of this chapter. Here is an example of an ADF:

```
[SampleApplication]
bcd: Sample.bcd    -- Object file
MessageFile: Sample.messages
IconFile: Sample.icons
KeyboardFile:Sample.keyboards
TIPFile: Sample.TIP      -- Really a TIPC file
Priority: 0
```

Only the section name and bcd entry are mandatory.

The easiest way to create an ADF is to copythe user profile from the directory to the desktop, delete the text in it, and enter the ADF text. To do this, your Workstation Profile must have

```
[System]
Developer: TRUE
```

(See Chapter 3 for details on the Workstation Profile).

The application folder has an *external name*, which appears on the folder. The external name can be changed, making multilingual conversion easier and more complete. The application also has an unchangeable *internal name*, supplied by the implementor. The internal name is used by the application code to reference components of the application folder, and is not modified by multilingual conversion or by any other process.

The section name that appears in an ADF is the application's internal name. In the example, the internal name is **SampleApplication**. For this example, the name of the file would typically be **SampleApplication.adf**.

11.1.4 Change or add code to access data files in the application folder

Once you have created your data files and written an ADF for them, you must change your code to access the data files. There are three steps to accessing a data file in an application folder:

- Getting a reference to the ADF

- Getting the name of or reference to the data file of interest, and
- Using the data file.

The following ADF will be used to illustrate the three steps:

```
[SampleBWSApplication]
bcd: SampleBWSApplication.bcd
MessageFile: SampleBWSApplication.messages
TIPFile: SampleBWSApplication.TIP
```

The **SampleBWSApplication** folder contains the above ADF, the bcd, one message file, and one TIPC file. The complete code for accessing the data files appears at the end of this chapter.

11.1.4.1 Getting a reference to the ADF

As you develop your application code, you should include the capability of running either from the system folder or from an application folder. If you do this, during development you can run your application from Command Central without having to worry about your data files. Later, you can incorporate your code and data files into an application folder without having to change as much of the code. When an application runs in the system folder, it uses the Workstation Profile as its ADF and its data files reside in the system folder. Here is an example of getting a reference to the ADF, whether it is a real ADF or the Workstation Profile:

```
--Get the internal name of the application
internalName: XString.ReaderBody ←XString.FromSTRING
["SampleBWSApplication"L];


--Get a reference to the application folder
 folder ←ApplicationFolder.FromName [@internalName]

-- Check to see in there's an application folder or if it's the
system folder
IF folder = NSFile.nullReference THEN {

  -- No application folder, so use the system folder and the
WorkstationProfile
   folderHandle ←Catalog.Open
[BWSFileTypes.systemFileCatalog];

-- Get a reference to the ADF
   adf ←OptionFile.GetWorkstationProfile []}
  ELSE {
   -- There was an application folder, so use the folder and the
ADF inside it
   folderHandle ←NSFile.OpenByReference [folder];

-- Get a reference to the ADF
   adf ←ApplicationFolder.FindDescriptionFile [folderHandle]};
```

11.1.4.2 Getting a reference to the data file

Once you have a reference to the ADF, you need to get a reference to the data file of interest. This example illustrates getting a reference to the message file. Note that the complete file for this example is included at the end of this chapter.

```
msgFile: NSFile.Reference ← NSFile.Reference;
internalName: XString.ReaderBody ←
   XString.FromSTRING["SampleBWSApplication"L];
--Get the name of the entry in the ADF
messageFile: XString.ReaderBody ← XString.FromString["MessageFile"L];
.
.
.
FindMessageFileFromName: PROCEDURE [value: XString.Reader] = {
   nssName: NSString.String ←
      XString.NSStringFromReader [r: value, z: localZone];
   msgFileHandle: NSFile.Handle ← NSFile.nullHandle;
-- Look for the message file
   msgFileHandle ← NSFile.Find [directory: folderHandle,
      scope: [filter: [matches[attribute: [name[nssName]]]]] !
      NSFile.Error = > {msgFileHandle ← NSFile.nullHandle; CONTINUE}];
   IF msgFileHandle = NSFile.nullHandle THEN ERROR; --No message file
--Since there is a message file, get a reference to it
   msgFile ← NSFile.GetReference [msgFileHandle];
--Free the file when you're done using it
   NSFile.Close [msgFileHandle];
   NSString.FreeString [z: localZone, s: nssName];};
.
.
.
   OptionFile.GetStringValue [section: @internalName,
      entry: @messageFile,
      callBack: FindMessageFileFromName,
      file: adf];
```

11.1.5 Integrate the components into an application folder

Once you have constructed all of the components of your application, you must run the application folder tool to combine them into an application folder.

- Copy all of the components, including the ADF, into a folder.
- Run the application folder tool, **Applize.bcd**. This will put two items in the Attention window menu:

  Folder → Application

  Application →Folder

  The first item takes the folder and turns it into an application folder. It does this by changing the file type of the folder and stamping the create date with the current date and time. It also sets the version to OS 6.0.

  The second item turns an application folder back into a regular folder. It changes the file type back to "folder" and sets the version to NIL.

Thus, to turn a folder into an application folder, just.select the folder, and then invoke **Folder → Application**. Figure 11.1 illustrates the steps of buildng an application folder.

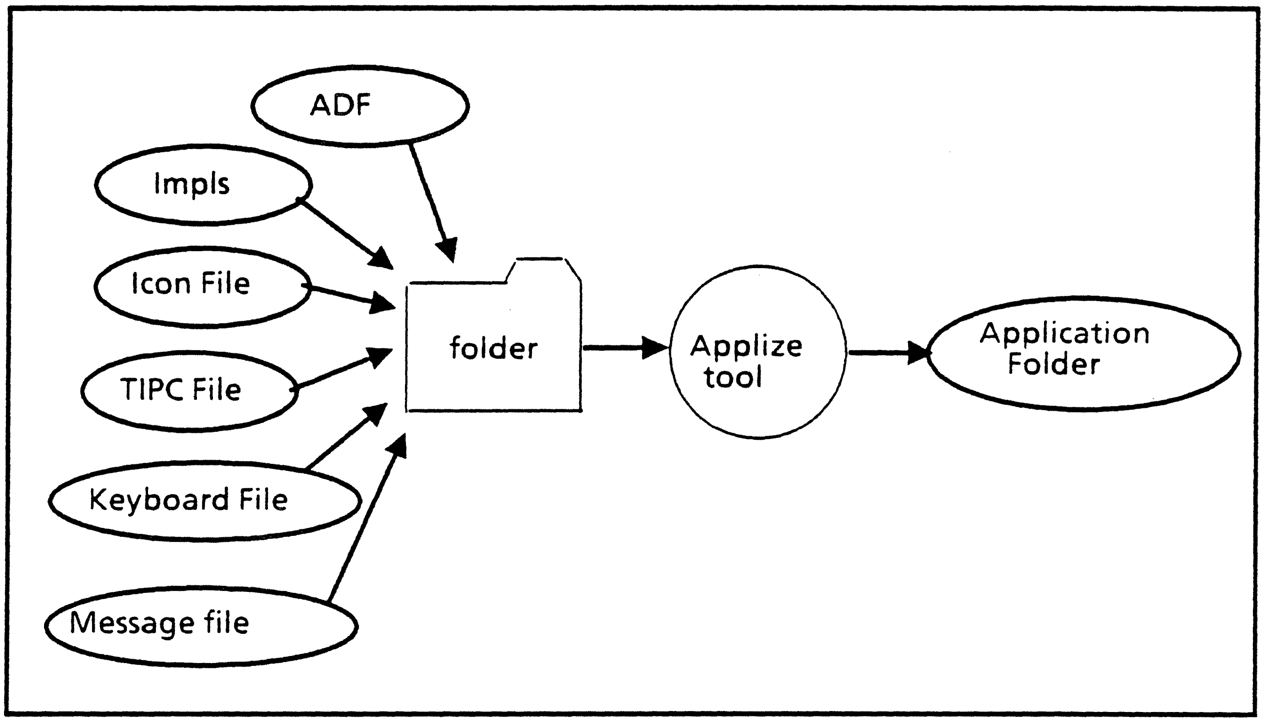


Figure 11.1 Building an application folder

11.2 Code Samples

11.2.1 Message Files

This is the message file impl from the sample application.This example shows how to write an application so it can run either in an application folder or as a bcd in the System folder. It also illustrates the method for supporting wildcards in the ADF.

```
-- File: SampleMsgFileImpl.mesa
-- Created by editing SampleBWSApplicationMessagesImpl

-- Copyright (C) 1985 by Xerox Corporation. All rights reserved.

DIRECTORY
  ApplicationFolder USING [FindDescriptionFile, FromName],
  BWSFileTypes USING [systemFileCatalog],
  Catalog USING [Open],
  Heap USING [systemZone],
  NSFile USING [Close, Error, Find, GetReference, Handle, nullHandle,
    nullReference, OpenByReference, Reference],
  NSString USING [FreeString, String],
  OptionFile USING [GetStringValue, GetWorkstationProfile],
  SampleBWSApplicationOps,
  XMessage USING [ClientData, FreeMsgDomainsStorage, Handle,
    MessagesFromReference, MsgDomains],
  XString USING [FromSTRING, NSStringFromReader, Reader, ReaderBody];
```

```
SampleMsgFileImpl: PROGRAM
   IMPORTS ApplicationFolder, Catalog, Heap, NSFile, NSString, OptionFile,
      XMessage, XString
   EXPORTS SampleBWSApplicationOps = {

   -- Data
   h: XMessage.Handle ← NIL;
   localZone: UNCOUNTED ZONE ← Heap.systemZone;

   -- Procedures
   DeleteMessages: PROCEDURE [clientData: XMessage.ClientData] = {};

   GetMessageHandle: PUBLIC PROCEDURE RETURNS [XMessage.Handle] =
      {RETURN[h]};

   InitMessages: PROCEDURE = {
      internalName: XString.ReaderBody ←
         XString.FromSTRING ["SampleBWSApplication"L];
      msgDomains: XMessage.MsgDomains ← NIL;
      msgDomains ← XMessage.MessagesFromReference [
         file: GetMessageFileRef
            [ApplicationFolder.FromName[@internalName]],
         clientData: NIL,
         proc: DeleteMessages ];
      h ← msgDomains[0].handle;
      XMessage.FreeMsgDomainsStorage [msgDomains];
      };

   GetMessageFileRef: PROCEDURE [folder: NSFile.Reference]
      RETURNS [msgFile: NSFile.Reference ← NSFile.nullReference] = {
      folderHandle: NSFile.Handle ← NSFile.nullHandle;
      adf: NSFile.Reference ← NSFile.nullReference;
      internalName: XString.ReaderBody ←
         XString.FromSTRING ["SampleBWSApplication"L];
      messageFile: XString.ReaderBody ←
         XString.FromSTRING ["MessageFile"L];

      FindMessageFileFromName: PROCEDURE [value: XString.Reader] = {
         nssName: NSString.String ← XString.NSStringFromReader [
            r: value, z: localZone];
         msgFileHandle: NSFile.Handle ← NSFile.nullHandle;
         -- We do NSFile.Find here in case the name has an asterisk in it
         msgFileHandle ← NSFile.Find [directory: folderHandle,
            scope: [filter: [matches[attribute: [name[nssName]]]]] !
      NSFile.Error = > {msgFileHandle ← NSFile.nullHandle; CONTINUE}];
         IF msgFileHandle = NSFile.nullHandle THEN ERROR; -- No message file
         msgFile ← NSFile.GetReference [msgFileHandle];
         NSFile.Close [msgFileHandle];
         NSString.FreeString [z: localZone, s: nssName];
         };

      IF folder = NSFile.nullReference THEN {
-- No application folder, so use the system catalog and the Workstation
Profile
         folderHandle ← Catalog.Open [BWSFileTypes.systemFileCatalog];
         adf ← OptionFile.GetWorkstationProfile []}
      ELSE {
-- There was an application folder, so use the folder and the adf inside it.
         folderHandle ← NSFile.OpenByReference [folder];
         adf ← ApplicationFolder.FindDescriptionFile [folderHandle]};
      OptionFile.GetStringValue [section: @internalName,
         entry: @messageFile,
         callBack: FindMessageFileFromName,
        file: adf];
      NSFile.Close [folderHandle];
      };

    -- Mainline code

    InitMessages[];

    }..
```

11.2.2 Private icons file

This example shows how to register an application that uses a private icons file. Note that this is no different from an application that uses the standard icons file. All the application must do is register its type. ViewPoint will locate the icon file when the user loads the application and associate the application and its icon by type.

```
sampleIconFileType: NSFile.Type = 100100; -- arbitrary

SetImplementation: PROCEDURE = {
   mh: XMessage.Handle = SampleBWSApplicationOps.GetMessageHandle[];
   oldImpl ← newImpl ←Containee.GetImplementation[sampleIconFileType];
   newImpl.convertProc ← Containee.DefaultFileConvertProc;
   newImpl.genericProc ← GenericProc;
   newImpl.name ← XMessage.Get [
   mh, SampleBWSApplicationOps.kApplicationName];
   [] ← Containee.SetImplementation [sampleIconFileType, newImpl];
   };
```

Note that for this to work you need to omit any reference to either pictureProc or smallPictureProc. For more information, see Chapter 9 of this manual.

11.1.3 Private TIP file

This example shows how to find a TIP file in an application folder and create a TIP table by calling TIP.CreateTable. Applications should include only TIPC files in their application folders. Name them <name>.TIP in the TIPFile entry of the ADF, and give <folder name>/<name>.TIP to TIP.CreateTable. If you want to load applications that use their own TIP files, you must boot ViewPoint with the 'O switch.

```
sampleTIPTable: TIP.Table ←NIL;
InitTIPTable: PROCEDURE = {
   separator: XChar.Character = LOOPHOLE[
      NSFileName.nameVersionPairSeparator];
   pathName: XString.WriterBody ←XString.NewWriterBody[40, zone];
   AppendTIPFileName [@pathName];
   sampleTIPTable ← TIP.CreateTable [
      file: XString.ReaderFromWriter [@pathName]];
   XString.FreeWriterBytes [@pathName];
   };

AppendTIPFileName: PROCEDURE [writer: XString.Writer] = {
   separator: XChar.Character = LOOPHOLE
      [NSFileName.nameVersionPairSeparator];
   internalName: XString.ReaderBody ←XString.FromSTRING
      ["SampleBWSApplication"L];
tipFile: XString.ReaderBody ← XString.FromSTRING ["TIPFile"L];
folderHandle: NSFile.Handle;
folderRef: NSFile.Reference ← ApplicationFolder.FromName
   [@internalName];

AppendName: PROCEDURE [value: XString.Reader] = {
   XString.AppendReader [to: writer, from: value];
   };

IF folderRef = NSFile.nullReference THEN {
   XString.AppendSTRING [writer, "SampleBWSApplication.TIP"L];
   RETURN};
-- ELSE --
folderHandle ← NSFile.OpenByReference [folderRef];
AppendFolderName [folderHandle, writer];
XString.AppendChar [to: writer, c: separator];
OptionFile.GetStringValue [section: @internalName, entry: @tipFile,
   callBack: AppendName,
   file: ApplicationFolder.FindDescriptionFile [folderHandle]];
NSFile.Close [folderHandle];
};
AppendFolderName: PROCEDURE [
   applFolder: NSFile.Handle, writer: XString.Writer] = {
   attrs: NSFile.AttributesRecord;
   rb: XString.ReaderBody;
   NSFile.GetAttributes[applFolder, [interpreted: [name : TRUE]], @attrs];
   rb ← XString.FromNSString [attrs.name];
   XString.AppendReader [writer, @rb];
   NSFile.ClearAttributes[@attrs];
   };
```

11.3 ADF Syntax

An ADF consists of the application's internal name, the names of the data files, the loading priority of the application, and any other entries an application requires. The application's object files are listed in starting order. All other entries may occur in any order.

An application that must start before another application is assigned a lower priority number. The loader starts applications in increasing priority number order. Thus you can think of the priority number as indicating the number of dependencies for an application. If your application has no dependencies, use a priority of zero.

The **Requires** entry in the ADF lists the internal names of applications that must be loaded and started for this application to run.

An ADF is named <application internal name>.adf. Its contents follow this syntax:

<ADF> :: = [<application internal name>]<keyword series> | NIL

<keyword series> :: = <keyword series><component>
|<component>

```
<component> ::= <object file> | <message file> | <icon file> |
                <keyboard file>|<TIPC File>|<priority>|
                <requires>

<object file> ::= bcd: <any legal NSFile name>.bcd

<message file> ::= <Entry Identifier>: <any legal NSFile name>

<icon file> ::= <Entry Identifier>: <any legal NSFile name>.icons

<keyboard file> ::= <Entry Identifier>: <any legal NSFile name>

<TIPC file> ::= <Entry Identifier>: <any legal NSFile name>

<priority> ::= Priority: <Integer>

<requires> ::= Requires: <Required application internal Name 1>,. ,
             <Required application internal Name n>
```

Note that the **<Entry Identifier>**s may be any identifier of your choice, but should indicate the type of entry. For example, the entry name for translation tables could be **TransTable**. The standard identifiers are **MessageFile**, **IconFile**, **KeyboardFile**, and **TIPFile**.

Section III. Programming Tools

This section documents four tools that were designed specifically to aid programmers in developing ViewPoint applications. These tools are:

- Chapter 12: Icon Editor
- Chapter 13: Message Tools
- Chapter 14: Bitmap Edit Tool
- Chapter 15: Keyboard Tool

You should note that these tools have limited functionality, as this is their initial release. Read the restrictions for each tool carefully, and use the tool with regard to the restrictions.

12. ICON EDITOR

The icon editor is a tool that is used to modify or delete an existing icon picture or to create a new icon picture. You will want to use this tool if you want to change an icon for a standard tool, or if you want to design an icon for a new application. The Icon Editor helps you create the new icon; once you have the icon, you can include it in an application folder (see Chapter 11), or add it to the file of standard icons.

12.1 Getting started

To use this tool, run the files **BWSIconEditor.bcd** and Standard.icons from CommandCentral with the /-e switch. Your Run line should look something like this:

```
un: BWSIconEditor.bcd/-e Standard.icons/-e
```

Note that you can also load them directly from ViewPoint using the Application Loader, as described in Chapter 3.

To get started, you need to perform the following steps:

1. Open your System folder and copy **Standard.icons** to your desktop. By copying this file, you ensure that you don't accidentally overwrite any of the existing icons. Standard.icons contains a list of the icons currently available, and the file types with which those icons are associated.

2. Use the PROPS key to rename the icon to be **New.icons**, and change the file type to be 6010. (The new name that you choose is arbitrary; in fact, you don't even have to rename it. However, the new file type must be 6010.)

3. Move or copy **BWSIconEditor.bcd** to the Loader.

4. Open the **New.icons** file, and you will get a list of icons and associated file types. Choose an icon that you want to modify, and open it. If you want to modify one of the standard icons, you should open that icon; if you want to create a new one, you can select any of them to modify.

   When you open an icon, you will get a list of sizes, such as 8 X 8 or 65 X 65. These sizes correspond to the various possible forms of that icon, such as tiny, cursor, and reference. Select the size that you want to modify and open it to start editing. As an illustration of the possible icon forms, consider a document. When you select a standard document icon and invoke an operation such as MOVE, the icon changes to a tiny copy of the document icon. This is the cursor form of that icon. Similarly, if you make an icon a reference icon, the form of the icon will change. (See the *ViewPoint Series Reference Library* for more information on icons.)

5. When you have finished editing the file, you need to save the bitmap before you close everything, or you will lose your new icon.

12.2 Editing an icon

While you are editing an icon file, you have the following operations available:

Left mouse button	Make a white box black
Right mouse button	Make a black box white
Magnification	Change the size of the icon. Provides a popup menu that allows you to choose the power of the magnification.
Shift	Shift the current bitmap pixel by pixel. Supplies a menu that allows you to specify the direction of the shift.
Save	Save the current bitmap in a file. You should use this command when you have finished.
Reset	Restore a bitmap to its original condition (before any edits)
Clear	Clear bitmap completely

You can also use the PROPS key to change the dimensions of the text box (where the icon name is displayed.) To do this, select an icon from the list of icons in the .icons file, and press PROPS.

13. MESSAGE TOOLS

This chapter describes the tools that are available for creating, modifying, and translating *message files*. Using message files for an application enables you to change or translate the messages without recompiling the application. There are three ViewPoint tools that support this process: the Message Master File Creation tool, the Message Master Editor, and the Message Runtime File Creation tool. These tools build files that facilitate the task of translating text messages and placing the translated messages in the running application.

The Message Master File Creation tool builds a Message Master file from the message bcd files. A Message Master file is a file that contains the original text, a translation of that text, and additional information used by the translator. The File Creation tool produces an icon for each Message Master File. The application programmer (or a translator) can then open the icon to edit the messages it represents.

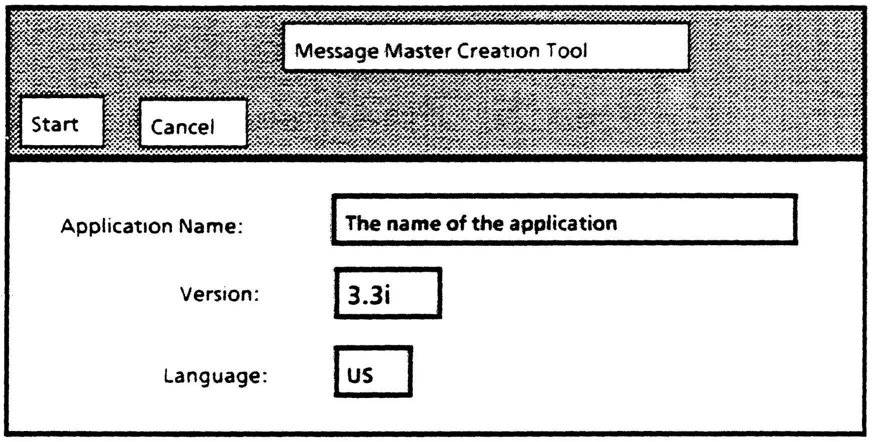
The Message Master Editor is used both by application programmers and by translators. It allows them to merge current and previous versions of a Message Master File, translate, edit, and print the file, check the validity of its translated text, compare two message files, and search for messages that have been translated.

The Message Runtime File Creation tool builds a Message Runtime file from a Message Master file. The runtime file contains only the textual information that is required by a runnning application. It cannot be edited.

13.1 Message Master File Creation tool

The first step is to copy the file containing your message information to the Message Master File Creation tool. To use this tool, you need to run the file **MasterFileCreate.bcd**; this will create an icon identified by the words "Msg Master Maker." (The file with the message information is the compiled version of the message implementation module; it can be either a single file or a folder containing several files. See Chapter 5, Strings and Messages.)

When you copy a file to the Message Master icon, an options window appears that allows you to specify the application name, language abbreviation, and version. The default application name is the name of the Message bcd file or folder. The default language is US. You must specify a version number, however; there is no default. Figure 13.1 illustrates this option sheet.



Application Name: The name of the application

Version: 3.3i

Language: US

Figure 13.1 Message Master Creation Tool

To start the actual creation of the Message Master file, select Start in the tool window header. If you do not enter a version number, you will get an error message; the message master creation process will not continue until you enter a version number and select Start again. During file creation, the window disappears from the screen. Until the window disappears, you can abort the operation by selecting Cancel in the tool window header.

This tool produces an icon and places it either on the desktop or in the folder containing the source files. The name of the Message Master file that is created is:

<Application Name>.<version>.<language>.master

For example: **Cusp.3.3i.US.master** is the Message Master file for version 3.3i of the application, Cusp. Its language is US. For Message Master files created using a folder, the number of bcd files included in the Message Master file is indicated by the number of dots posted between the user messages "Creating Message Master File" and "Placing on Desktop" or "Placing in Folder" displayed in the Attention window. This allows you to gauge the progress of the creation application; each dot indicates one file.

This file contains an untranslated version of all the messages. The next step is typically to edit or translate the messages, using the Message Master Editor, as described in the next section.

The Message Master Creation tool also produces an errorlog and places it on the desktop. The possible errors are:

- Duplicate IDs: This error indicates that a set of **MessageImpl bcds** has messages in identical domains with identical IDs. This error means that the source will have to be changed and recompiled. (See Chapter 5 for a discussion of IDs.)

- Unbound Procedure: This error indicates that a **MessageImpl bcd** contains references to other bcds that are not bound in. To fix this, you must remove the offending unbound reference or bind the files with the appropriate implementation.

- No domains: This error indicates that the **MessageImpl bcd** contains no calls to **xMessage.AllocateMessages** or **xMessage.RegisterMessages**.
- RegisterMessages/AllocateMessages Error: This error indicates that the bcd has either called **xMessage.AllocateMessages** and **xMessage.RegisterMessages** in the wrong order or has only referenced one of them.

13.2 Message Master Editor

The file generated by the Message Master Creation tool will appear on the desktop as illustrated in Figure 13.3:

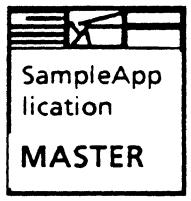


Figure 13.3 Master File icon

Once you have the message master icon, you can edit its contents with the Message Master Editor. To use this tool, load the program **MessageFileTool.bcd**. The editor allows you to search, edit, translate or print the text of the messages. To edit a message master, select the icon and press OPEN. This will bring up the editor window, as illustrated in Figure 13.4.

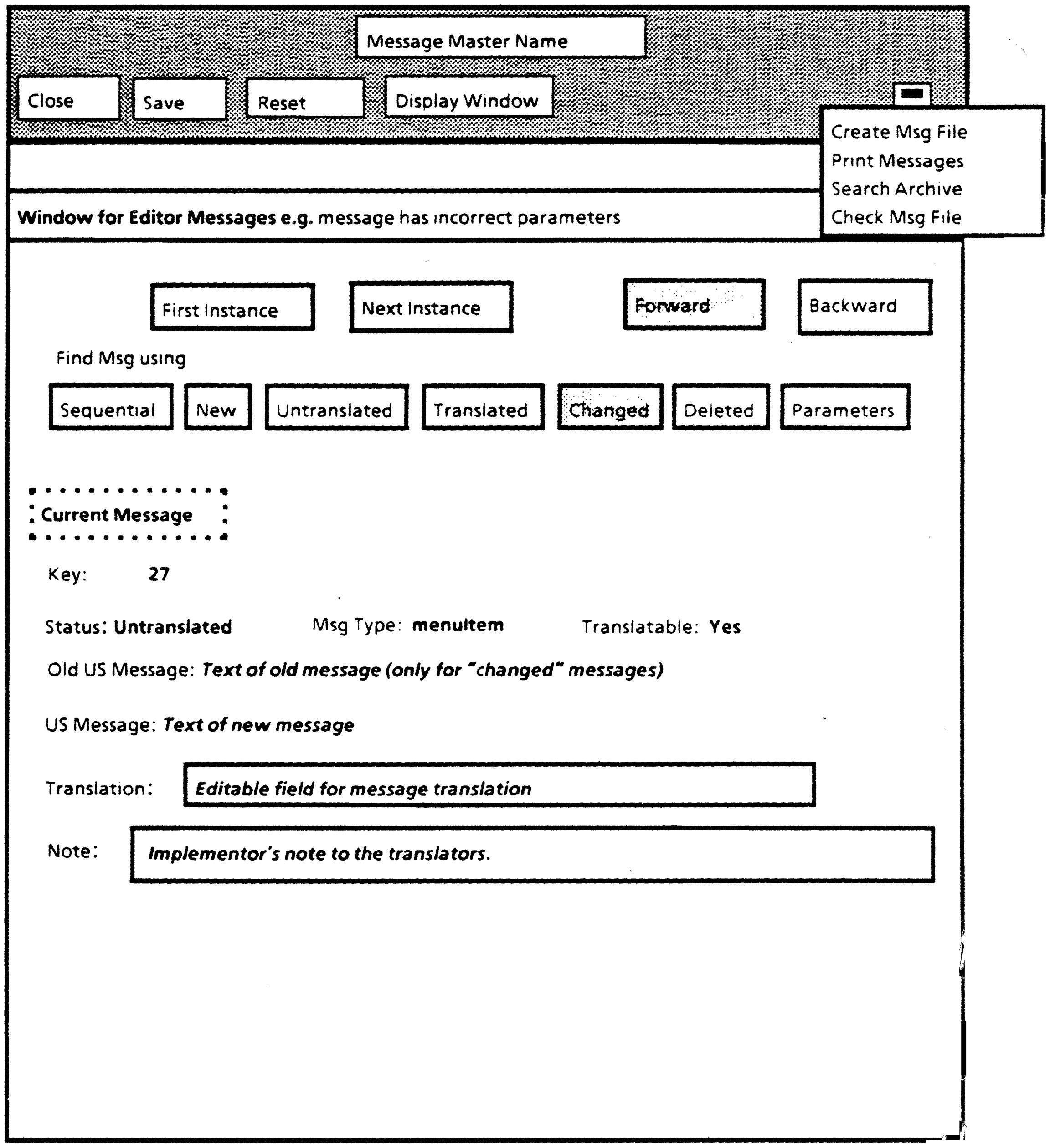
The top message window of the editor window is used for error messages concerning the message text. For example, if a translated message contains mismatched leading or trailing spaces or parameters, an error message appears when you attempt to initiate a further search. The search will not be performed until you correct the translated text or indicate that you want to ignore the errors.

13.2.1 Searching Message Master files

To search the message file to find a message that satisfies a particular criterion, use the commands **FirstInstance** and **NextInstance**:

First Instance - searches the Message Master file from the beginning to find and display the first message that satisfies the given search criteria.

Next Instance - finds the next instance (going either forward or backward from the last successful search) of the specified message type. You can also execute this command by pressing the NEXT key.



Window for Editor Messages e.g. message has incorrect parameters

Key: **27**

Status: **Untranslated** Msg Type: **menuItem** Translatable: **Yes**

Old US Message: ***Text of old message (only for "changed" messages)***

US Message: ***Text of new message***

Translation: ***Editable field for message translation***

Note: ***Implementor's note to the translators.***

Figure 13.4 Message Master Editor window

The Find Msg Using field allows you to specify the type of message searched for. The search proceeds either forward or backward, depending on the value selected in the form window. The choices are:

Sequential	finds the first or next message in the file.
New	finds the next message with the status "new."
Untranslated	finds the next message with the status "untranslated."

Translated	finds the next message with the status "translated."
Changed	finds the next message with the status "changed." If an entry of this type is found, an additional field appears that contains the previous version of the original text. (This field is not displayed for other types of messages.) If you have not specified the previous original text in the file properties, an error message appears and no text is displayed.
Deleted	finds the next message with the status "deleted."
Parameters	displays a set of options that specify the search criteria. This is discussed more fully in the next section.

If no message of the type specified can be found, "No message Found" appears in the Message window and the currently displayed message remains in the tool window.

13.2.2 Search parameters

You can also search by parameter, as illustrated in Figure 13.5. A search is successful only if all specified criteria are met. If you specify Message Text, you can search for the original text (by specifying US) or for a translation of the original text (by specifying Trans).

Simple searches by Key, ID, Status field, or Msg Type are reasonably quick. However, specifying the Key or ID fields in conjunction with other fields is slow and unproductive, as these fields are specific to the message.

When searching for strings (either translated or original text), setting the Status field or the Msg Type field accelerates the search. String searches are based on a search for the substring entered in the Message Text field.

13.2.3 Editing

When a search is successful, a message appears with the following information:

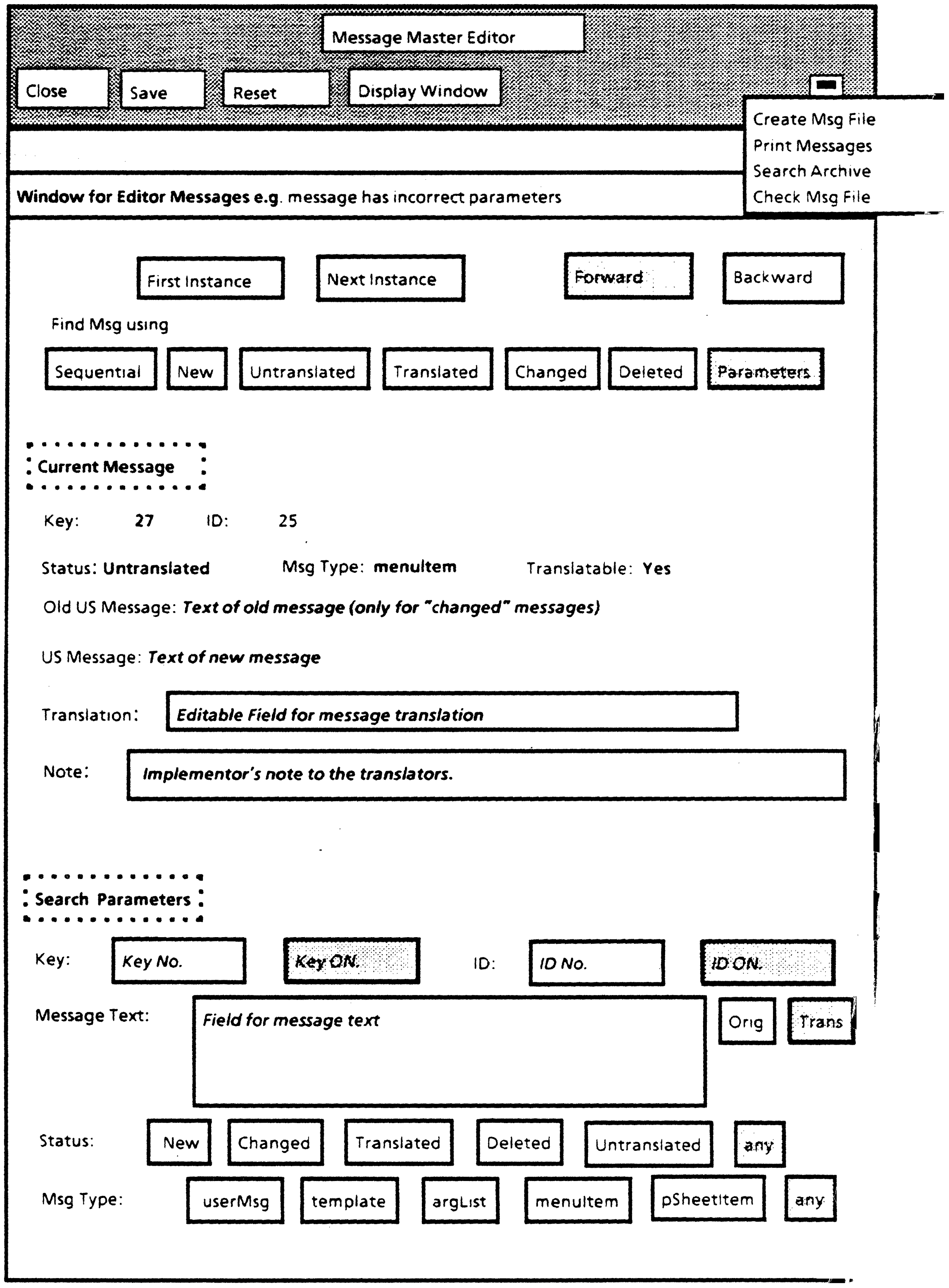
- Message key number

  The developer-assigned number of the message.

- Message ID number

  The unique message ID. It is developer-assigned and can never be reassigned.

- Message status (e.g., new, changed, translated)



Window for Editor Messages e.g. message has incorrect parameters

Find Msg using

Key: **27** ID: 25

Status: **Untranslated** Msg Type: **menuItem** Translatable: **Yes**

Old US Message: ***Text of old message (only for "changed" messages)***

US Message: ***Text of new message***

Translation: ***Editable Field for message translation***

Note: ***Implementor's note to the translators.***

Key: *Key No.* ID: *ID No.*

Message Text: *Field for message text*

Status:

Msg Type:

Figure 13.5 Searching by parameter

See section 13.2.2, Search Parameters, above, for a complete list of parameters. Message status is assigned by the Message Translation tools.

- Msg type (e.g., userMsg, template)

  Message type defines the use of the message (e.g., whether it is an argument list or a template for a composed message). The possible message types are:

  userMsg - an Attention Window message .

  template -contains fields to be filled in later by an item of type **argList.**

  argList - a list of arguments for use with a template.

  menuItem - an item used in a form window item (e.g., an item tag) or in a user menu selection.

  pSheetItem - a property sheet message.

  errorMsg - a highlighted error message that appears within the application.

  infoMsg - an informative message displayed to the user.

  promptItem - a user prompt.

  windowMenuCommand - a menu command in a window header.

  others - all other types.

  Note that these meanings are only a suggestion; you are free to allocate the types as you see best.

- Whether the message is translatable

- Previous original text (for changed messages only)

  When the text for a message has changed from a previous version, both versions of the original text appear on the screen. This field presents the previous text.

- Present original text

  When the text for a message has changed from a previous version, both versions of the original text appear on the screen. This field presents the present original text.

- Translation field (for translatable mesages only)

  The translator uses this field to enter his translation.

- Iimplementor's note

  This field allows the developer to enter any notes that he thinks might be useful to the translator. The translator may add additional notes. These additional notes are saved when two message master files are merged if the complete text from the original note is retained within the new note.

Translators can only edit the contents of the translation field and implementors can only edit note fields. If the message is non-translatable, the translation field cannot be edited.

Important: *Translators particularly should note that opening a new, untranslated message file does not give the file translated status. However, if either of the editable fields within any message is altered (even if it is subsequently returned to its previous state), the message file is considered translated, thus prohibiting any merging operation.*

13.2.4 Closing, saving, and resetting

Once a developer or a translator has edited a file, there are three ways to save the resulting file:

1 Use the Save command in the main window header. This function saves all changes made since the file was opened or since the last save (user or automatic).

2 Use the Automatic Save function in the Message File property sheet. See the property sheet section for details.

3 Use the Close command. This function closes the edit window and saves any changes made to the file since it was opened or since the last save (user or automatic).

If you make some edits that you want to get rid of, you can use the Reset command. If no Save operation has taken place, Reset returns the file its state when first opened. If you have changed the file, you must confirm the command (by reselecting the Reset command.) You can only reset back to the last user or automatic save performed.

13.2.5 Merging message files

When you need to change a message file that has been translated, make sure that the changes are reflected in the translated version as well. To approach this problem, start by editing the file in the original language. Then copy the translated version of the earlier version (without the changes) into the untranslated version with the changes. To copy the text, select the translated file, press COPY, and select the new message file as the destination of the copy.

When you copy the translated version into the untranslated version, the two files are merged and the status of each message is set to one of the following states:

- **new** - This a new message that was not present in the earlier version. This message will have to be translated.

- **changed** - The message was present in the earlier version, but either the text itself or the translator's note has been changed.

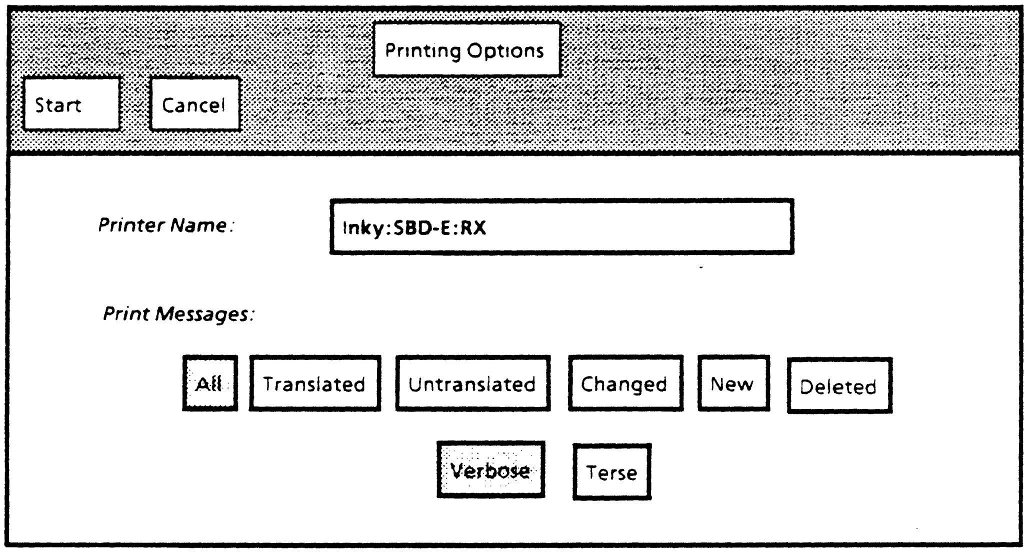
- **notTranslated** - The message is exactly the same in both languages, so there is no need to translate it. An example of such a message might be a roman numeral, a direct quotation, or the name of a city.

- **translated** - The translation field has been changed.
- **deleted** - The message was included in the early version, but not in the new version.

Note that for this to work, the new version must not include

13.2.6 Printing message files

To obtain hardcopies of information contained within the Message Master, use the Print Messages pop-up menu command. This command displays a list of print options, as illustrated in Figure 13.6.



Inky:SBD-E:RX

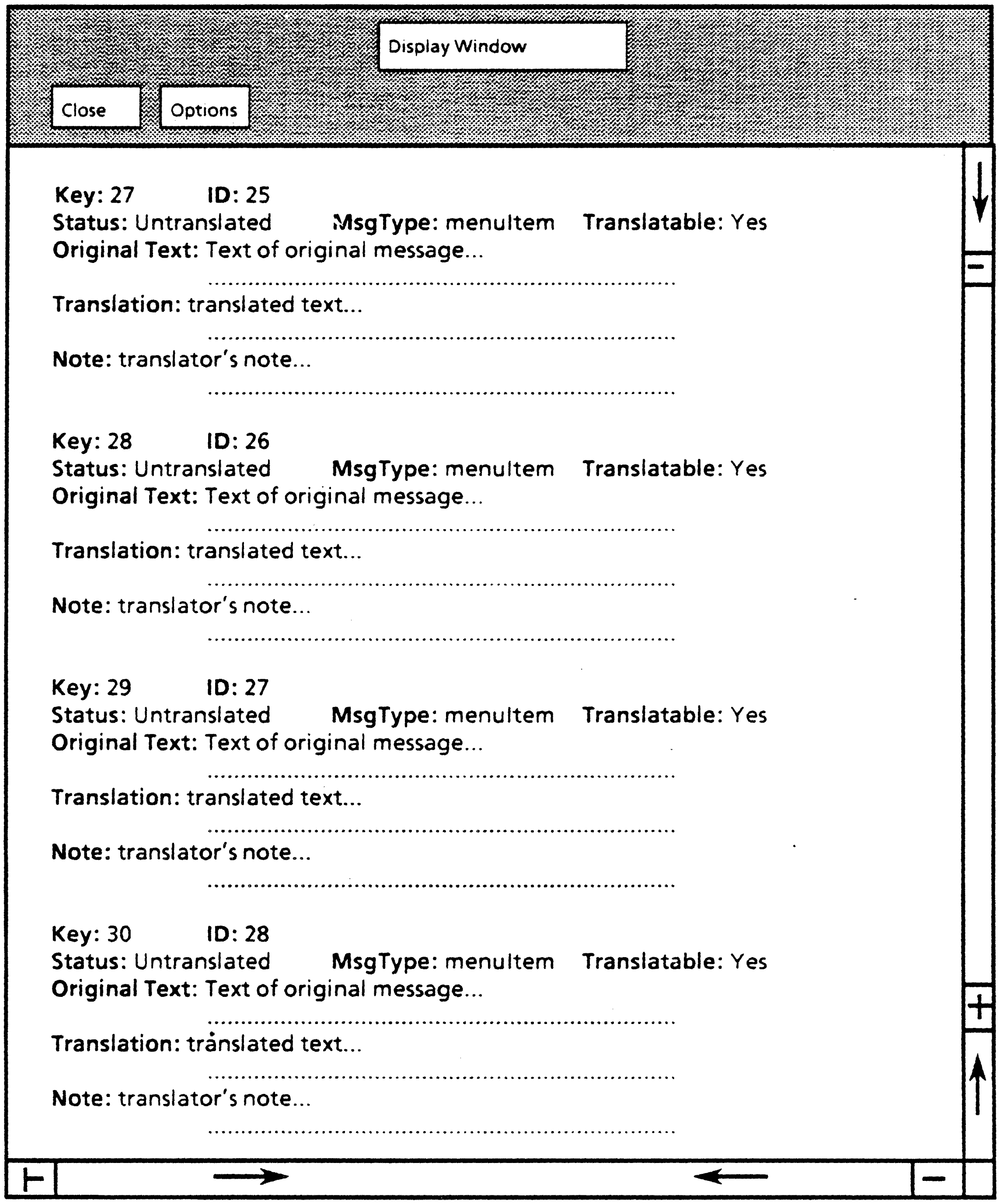
Figure 13.6 Print options

Selecting the Start command produces an Interpress-format document and sends it to the printer specified in the Printer Name field. You can print all messages or a category of messsages. Specify any one of the following categories: translated, untranslated, changed, new, or deleted messages.

In additiona, you can specify verbose or terse. Verbose produces a document containing all available information (i.e., original text, translated text and translation information). Terse produces a document containing only the original text, message key number, and message ID.

13.2.7 Displaying message entries

This feature allows you to view several messages simultaneously in one window, as illustrated in Figure 13.7. To do so, select the Display Window menu command in the main window header. A window for displaying the text required and an option sheet for selecting messages appears, as illustrated in Figure 13.8.



Key: 27 **ID:** 25
Status: Untranslated **MsgType:** menuItem **Translatable:** Yes
Original Text: Text of original message...
Translation: translated text...
Note: translator's note...

Key: 28 **ID:** 26
Status: Untranslated **MsgType:** menuItem **Translatable:** Yes
Original Text: Text of original message...
Translation: translated text...
Note: translator's note...

Key: 29 **ID:** 27
Status: Untranslated **MsgType:** menuItem **Translatable:** Yes
Original Text: Text of original message...
Translation: translated text...
Note: translator's note...

Key: 30 **ID:** 28
Status: Untranslated **MsgType:** menuItem **Translatable:** Yes
Original Text: Text of original message...
Translation: translated text...
Note: translator's note...

Figure 13.7 Viewing several messages simultaneously

The display window has two commands:

- Close - closes the window.

- Options - presents a window in which you specify the search options. The same search options are available as in the main window.

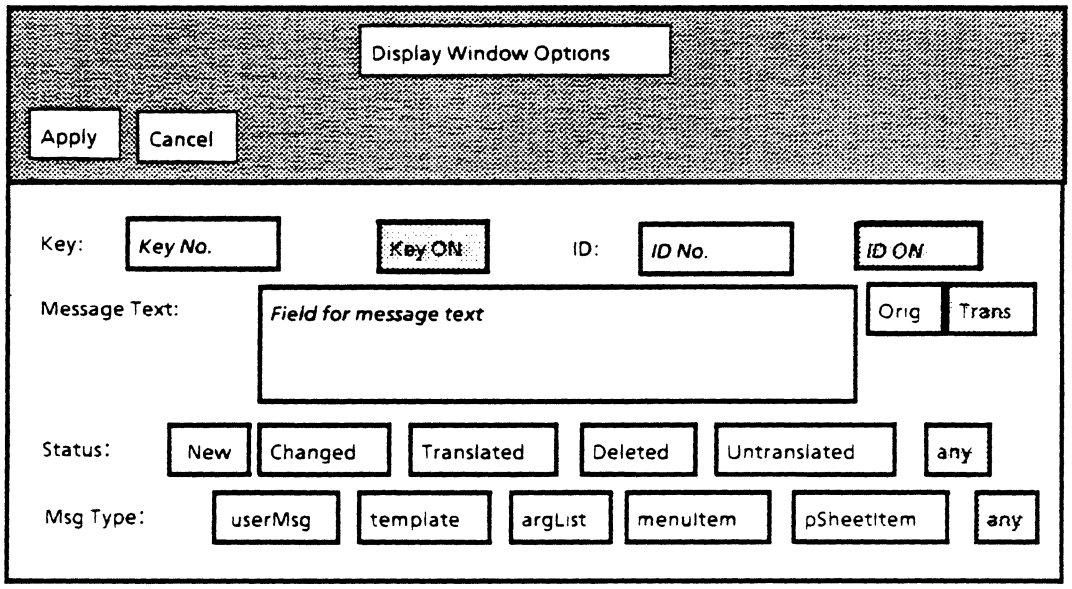


Figure 13.8 Display window options

13.3 Message File property sheet

To display the properties of a file, select the corresponding Message File icon and press the PROPS key. The property sheet that appears is shown in Figure 13.2.

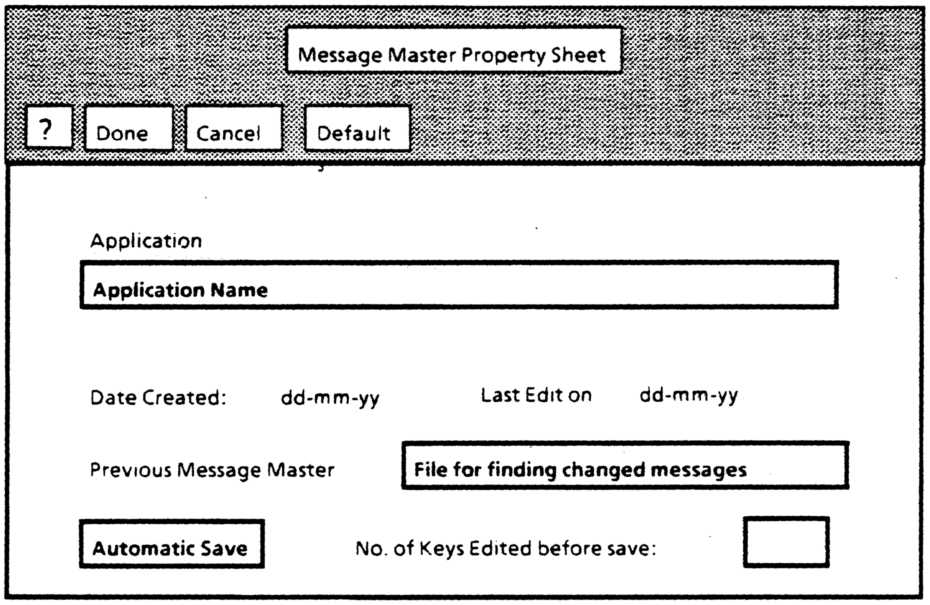
You cannot change the application name, creation date, or last edit date. (You can use the Message Master Editor tool Runtime Message file to specify the format of the creation date and last edit date). You can, however, change the remaining fields of the property sheet, as described below:

Application Name - the name of the Message Master file.

Previous Message File - This is the file you use to find the previous original text of changed messages. The default is the name of the last Message Master file that has been merged with this file. If no name appears in this field, an error message appears and no previous text is displayed.

Automatic Save - When set, edits are saved automatically. Use the "No. of Keys Edited before save" field to specify the save frequency. This function affects the execution of the main editor window RESET function. When you specify the save interval, keep in mind that Automatic Save entails considerable overhead since it writes the complete file to disk. Keep this in mind when specifying the save interval.

No. of Keys Edited before save - This field defines the Automatic Save frequency. It is defined as type CARDINAL. An edit is defined as editing one message and requesting another.



Application

Application Name

Date Created: dd-mm-yy Last Edit on dd-mm-yy

Previous Message Master File for finding changed messages

Automatic Save No. of Keys Edited before save:

Figure 13.2 Message Master property sheet

When the value in this field is zero, Automatic Save is ignored. The default value for this field is zero.

13.4 Runtime File Creation tool

The final step is to create a Runtime Message file. A Runtime Message file is essentially a compiled version; it can be loaded with an application, but it can't be edited or viewed. To create a Runtime Message file, you need to run the program **RuntimeFileCreate.bcd**. Once you have done so, select the Create Message File command from the pop-up menu. Invoking this command creates the window illustrated in Figure 13.9.

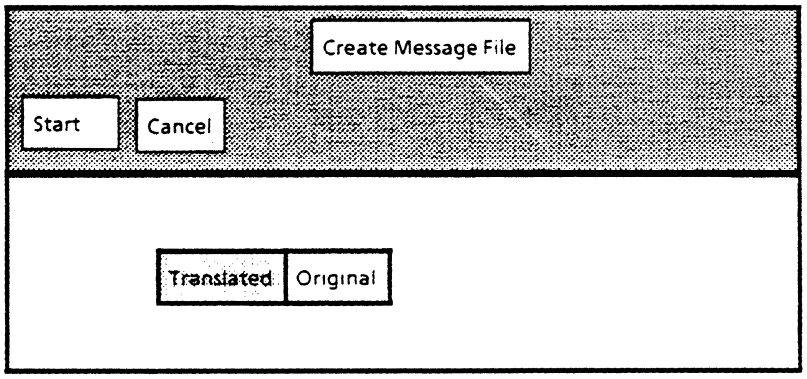


Figure 13.9 Using the Create Message File command

This window allows you to specify the type of text (original or translated) included in the Runtime Message file. If you specify Original, the command produces a Runtime Message file that contains only the original text. If you specify Translated, the command produces a Runtime Message file that contains only the translated text.

The Runtime Message file is produced when you select Start. The resulting file appears as an icon on the current desktop. As illustrated in Figure 13.10, this icon is similar, but not identical, to the Message Master file icon. You cannot perform any operations on the runtime file.

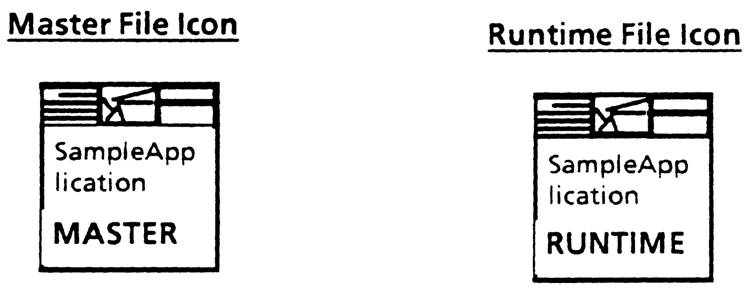


Figure 13.10 The resulting icon

To create a Runtime Message file without using the Message File Editor, use the Runtime Message Creation tool. The icon for this tool is identified by the words "Runtime Msg Maker". Use the COPY key to copy a Message Master file containing the message information to this icon. The tool then produces an original language Runtime Message file with the same name as the Message Master file followed by the extension .messages.

When the operation is complete, the tool places a Runtime Message file in the container (either a folder or the desktop) containing the Message Master file. If the Runtime Message file has been placed in a folder, you must redisplay the file.

14. BITMAP EDIT TOOL

The Bitmap Edit tool is an XDE tool that allows you to edit any bitmap, including keyboard bitmaps. It is used in conjunction with the Keyboard tool to create new keyboards. (See the Keyboard Tool chapter for more information.) It also provides limited freehand drawing capability.

The Bitmap Edit tool provides two modes, Draw and Edit. Draw mode is used for freehand drawing with a variety of brushes. Edit mode is used to read in a bitmap, move, copy, delete, or magnify portions of the bitmap, and store the edited bitmap.

14.1 Getting started

Running **BitmapEditTool.bcd** produces the window shown in Figure 14.1:

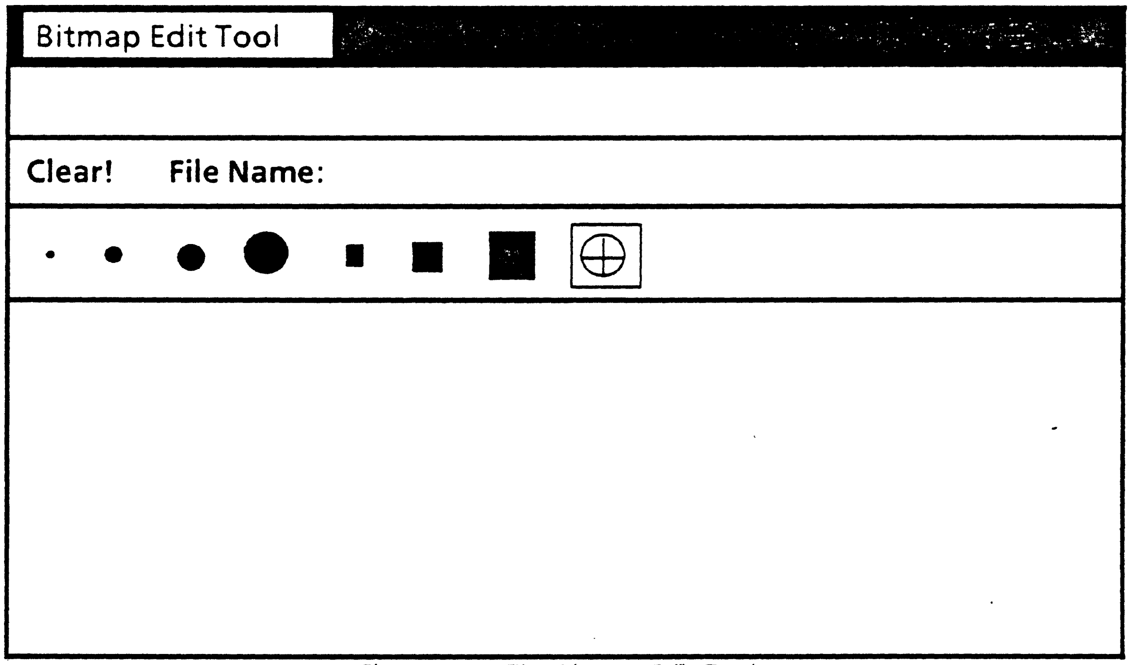


Figure 14.1 The Bitmap Edit Tool

The uppermost subwindow is a message subwindow for feedback and error messages. The second subwindow is a command subwindow, with two items, Clear and File Name, in it. Clear erases the bitmap subwindow. You can store the current bitmap to File Name or read in a bitmap to edit from File Name.

The third subwindow is the brush subwindow, which contains several sizes of rectangular and circular brushes that are available in Draw mode. If you select the crosshairs symbol, you will enter Edit mode.

The lowermost window is the bitmap subwindow, where the current bitmap will be displayed.

14.2 Draw mode

To use the Bitmap Edit ool in Draw mode, select a brush and move the cursor into the bitmap subwindow. To draw, move the mouse while holding down the left button; to erase, hold down the right button. Selecting Clear erases the entire bitmap subwindow.

14.3 Edit mode

Edit mode is used to select and manipulate portions of the current bitmap. When you are in Edit mode, you can

- Select a rectangular portion of any size from the bitmap
- Move, copy, delete or magnify any portion of the bitmap
- Write portions of the bitmap to press or bitmap files
- Read in press or bitmap files to edit or examine

14.3.1 Selection

To enter Edit mode, select the crosshairs brush. Holding down the left mouse button brings up a pair of crosshairs that can be moved across the bitmap to the section that you want to manipulate. Releasing the left button sets one corner of a rectangle. Holding down the right mouse button gives you a rectangular highlighted area that you can stretch to include any portion of the bitmap; releasing the right button sets the rectangular selection.

The granularity of the selection is controlled by the Grid menu, which is invoked by chording (pressing both mouse buttons simultaneously) in the command or message subwindow. The Grid menu offers the following choices, as shown in Figure 14.2:

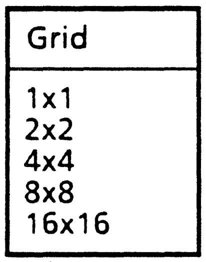


Figure 14.2
Grid menu

With Grid set to 1x1, you can use the cursor to point at an individual pixel. With Grid set to 16x16, you can only point to each 16x16 pixel block. When editing keyboards, it is most useful to have Grid set to 1x1.

14.3.2 Move, copy, and erase

You can manipulate portions of the bitmap with the pop-up Edit menu or with the function keys on the left side of the keyboard.

You activate the Edit menu, illustrated in Figure 14.3, by chording in the command or message subwindow.

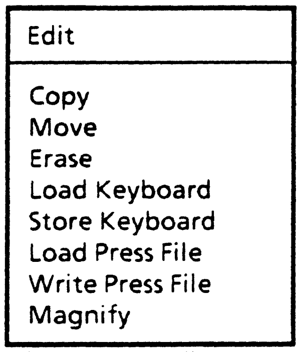


Figure 14.3 Edit menu

If you want to use the keyboard keys instead of the Edit menu , the key mapping is shown in Figure 14.4:

Function	Key
Copy	COPY
Move	MOVE
Erase	DELETE
Load Kbd	AGAIN
Store Kbd	FIND
Load Press	SAME
Write Press	OPEN
Magnify	PROPS

Figure 14.4 Key mapping

Select the portion of the bitmap that you want to manipulate, then select the desired operation from the menu or press the appropriate key. The Bitmap Edit tool will give you feedback and instructions in the message subwindow.

14.3.3 Magnify

You can also select and magnify a portion of the bitmap. This is the easiest way to edit a keyboard bitmap, since the pictures of the keys are usually quite detailed. Invoke the Magnify window from the Edit menu or by pressing the FIND key. Figure 14.5 shows a picture of the Magnify window, with the letter "B" magnified in it:

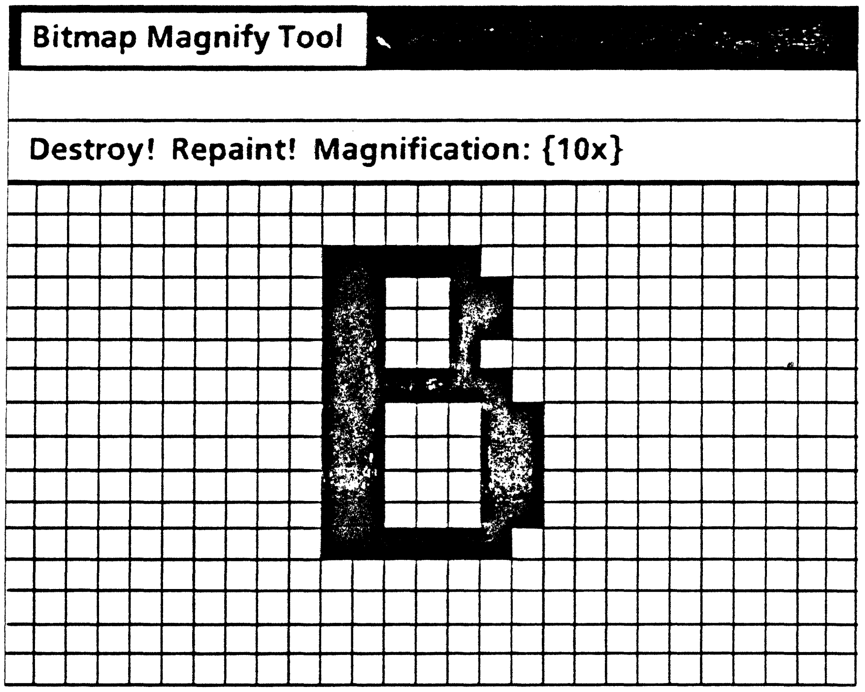


Figure 14.5 The Magnify window

To draw, turn the grid squares on (black) with the left mouse button and off (white) with the right mouse button. By selecting the desired value from the Magnification enumerated field, you can set the magnification of the window to any even value between 4 and 16. As you draw in the Magnify window, the scaled-down version of the drawing appears in the bitmap subwindow. .

The Repaint command repaint sthe Magnify window to reflect any changes you have made by drawing in the bitmap window. For example, if you magnify a letter, return to Draw mode, and draw a circle around the letter in the bitmap subwindow, you won't see the circle appear in the Magnify window until you select Repaint.

The Destroy command destroys the Magnify window. Any changes you have made will remain reflected on the non-magnified bitmap in the bitmap subwindow of the Edit tool. You can open up to four magnify windows at once; thus, you can edit several portions of the bitmap at the same time.

14.3.4 Loading and storing bitmaps and press files

You can store your edited bitmap or read in a bitmap to edit or examine with the Load Keyboard and Store Keyboard commands, or with the Load Press File and Write Press File commands. Load Keyboard and Store Keyboard are somewhat limited in that they will only load or store a bitmap that is exactly the size of a standard keyboard bitmap (505x145 pixels.) For example, if you are building the middle row of keys, and you want to store only that portion of the bitmap, you select the middle row. However, when you invoke Store Keyboard, the selection will be automatically extended from the upper-left corner downward and to the right, until it is the size of a standard keyboard bitmap.

If, however, you select Write Press File, you can store any portion of the bitmap, no matter how small. There is no functional difference between the Load Keyboard and Load Press File commands.

14.4 Restrictions

The tool has limited space to store bitmaps. Reading a press file or bitmap file that is too large will crash the tool.

The mapping of edit functions to function keys is only in effect if the cursor is in the bitmap subwindow.

Pop-up menus can only be accessed from the message and command subwindows.

15. KEYBOARD TOOL

ViewPoint is equipped with a standard keyboard file that contains definitions for many keyboards, including European and Arabic languages, math and logic symbols, and terminal emulators. However, in case you are writing an aoplication that requires a keyboard that is not part of the standard release, ViewPoint also provides facilities for defining your own keyboard. A new keyboard is usually defined by editing an existing keyboard that has similar characteristics; the new keyboard is limited by the available font set.

The Keyboard tool is an XDE tool for building and maintaining the black keys of ViewPoint keyboards data files. (Function keys are changed via TIP tables; see the TIP chapter for more information.) The Keyboard tool allows you to examine the keyboards in a keyboard file, change existing entries, and add new entries.

15.1 Getting started

A keyboard file is actually a collection of definitions for different keyboards. A keyboard definition is composed of three items: a bitmap, an interpretation table, and a geometry table. The bitmap is the picture of the keyboard that would appear on the screen. The interpretation table defines what happens when you press a particular key or key combination. The geometry table defines the area of the keycap on the bitmap that is highlighted when you use the mouse to select the keycap. Each of these tables has an index in the keyboard file, and these indices are used by the Keyboard tool to refer to the components of the file.

The KeyboardTool is activated by running **KeyboardTool.bcd** in XDE. It has six subwindows. The uppermost subwindow is a message subwindow for displaying feedback and error messages. The lowermost window is a log window for listings of keys and keyboards. The middle four windows will be discussed in detail in the following sections.

15.2 Keyboard file subwindow

The file subwindow contains facilities for manipulating the keyboards in the keyboard file. In the keyboard file subwindow, you can:

- Open a keyboard file
- List the keyboard definitions in the open keyboard file
- Add a newly created keyboard to the keyboard file
- Close the keyboard file

Here is an example with a keyboard file that has already been opened:

```
Open!  Close!  ReadOnly  List!  Keyboard File: Keyboards
Set!  Installed Keyboards: USEnglish French Russian German
          American UKEnglish Hiragana

Default Keyboard: {}
Phsyical Keyboard: {}          Default Language: {}
Date Format: {}                Time Format: {}
Decimal Separator: {}          Thousands Separator: {}
Default Units: {}              Paper Sizes: {}
Sort Order: {}                 Default Print Wheel: {}
```

The Open command opens the file whose name is specified in the string **Keyboard File**. If the boolean **ReadOnly** is set, you can examine the file but not edit it. Otherwise, the file is read/write, and the Close command will write all changes to the file and close it.

When you select Open, the list of installed keyboards will be filled in with the contents of the keyboard file. This is the list of keyboards that will appear on the soft keys when the KEYBOARD key is pressed. In the example above, the keyboard file is named "Keyboards", and the keyboard definitions contained in that file are listed on the Installed Keyboards line. You can change this list by editing the line and using the Set command. If you try to add the name of a keyboard whose definition does not appear in the keyboard file, you will get an error message.

The List! command provides a brief summary of each keyboard definition in the bottom subwindow, listing the indices of the bitmap, interpretation, and geometry tables. Here is a sample list:

```
Keyboard: USEnglish
 Bitmap index: 29
 Interpretation index: 4
 Geometry index: 21
Keyboard: MathSymbols
 Bitmap index: 1
 Interpretation index: 1
 Geometry index: 2
```

The rest of the parameters shown are obsolete. Because of changes in recent software releases, these parameters are now set by the Workstation Profile.

15.3 Keyboard and bitmap subwindows

The bitmap subwindow contains the bitmap associated with the keyboard specified by the Keyboard parameter. This bitmap is the picture of the keyboard that you see in ViewPoint when you press KEYBOARD and select the keyboard you want to view.

The keyboard subwindow contains the facilities for manipulating the bitmap, interpretation, and geometry tables. You can:

- Select a keyboard definition for examination or editing
- Remove a keyboard definition from the keyboard file
- Insert into the open definition a bitmap, geometry table, or interpretation table from another keyboard definition
- Write the bitmap to a file for editing
- Write the geometry table to a file for editing
- Load a bitmap or geometry table into the definition
- List the functions and parameters of the black keys.

The keyboard subwindow has the following format:

```
Keyboard: {}          Remove!
Bitmap Index =         Replace!
KbdInt Index =         Replace!
Geom Table Index =     Replace!
Source Keyboard: {}        Same File  Keyboard File:
Load Bitmap!  Store Bitmap!  Bitmap File:
Load Geom!    Store Geom!    Geom File:
List!   AllKeys
```

When you select a keyboard from the Keyboard enumerated menu, the indices and the bitmap subwindow will be filled in according to the definition of the keyboard you selected.

The Remove command removes a keyboard from the file. Note, however, that Remove only removes the index entry from the file; it leaves the components (the bitmap, interpretation table, and geometry table) in the keyboard file. This will be corrected in a later version of the Keyboard tool.

The Store Bitmap command stores the current bitmap, pictured in the bitmap subwindow, to the file specified by Bitmap File. You can then load this bitmap file into the Bitmap Edit tool for editing. (See the Bitmap Edit Tool chapter for more information.) The Load Bitmap command takes the specified bitmap and writes it back into the current keyboard. This is used for replacing the current bitmap with a bitmap that you have edited.

Store Geom is used to write the current geometry file to Geom File so that you can bring it up in a file window and examine or alter it. Load Geom loads the file specified by Geom File into the geometry table of the current keyboard. A sample geometry file looks like this:

```
[[75, 75], 27, 14], k3, One
[[75, 88], 27, 14], k3, None
[[75, 75], 27, 14], k3, One
[[75, 88], 27, 14], k3, None
```

Each keycap is divided into an upper and lower section, and there is a one-line entry for each section.On each line, the first set of coordinates is the location of the upper-left corner of the keycap section. The second set of coordinates is the width and height of the keycap section. The next parameter is the keystation name (as defined in the *Pilot Programmer's Manual*), followed by the number of shifts allowed with the key. For example, keystation k3 produces the character "a" when pressed, and the character "A" when one SHIFT key is pressed along with the key; therefore it has one significant shift. (Shifts are noted only on the line for the upper section of the keycap.)

By editing the geometry file, you can change the area that is highlighted when you select a keycap on the bitmap. This feature is useful when defining new keyboards with odd-sized keycaps. For example, store the geometry table to a file, bring it up in a file window, double the width of a keycap, load the edited geometry file, and select the altered keycap with the mouse in the bitmap subwindow. You will notice that the selected (highlighted) area is now twice as wide as the keycap. For clarity, the measurements of a keycap in the geometry table should correlate to the area of that keycap on the bitmap.

The List command produces a listing of the keys in the bottom subwindow. Here is the list for the letter "A" on the USEnglish keyboard:

```
k3 REPEAT LOCKABLE OneShift
   UNSHIFTED: Character CHSET: 0 CODE: 141B
   SHIFTED: Character CHSET: 0 CODE: 101B
```

You can replace various components of the current keyboard with the components of other keyboards. Source Keyboard, Same File, and Keyboard File are used to interchange the components of keyboards. For instance, you might want to have two different keyboard definitions with identical bitmaps, or even have two keyboards point to the same bitmap. To do this, you would use the Replace command on the Bitmap Index line. This will replace the bitmap for the current keyboard with the bitmap for the source keyboard. If you wanted to replace the interpretation table, you would use the Replace command on the KbdInt Index line. For replacing the geometry table, use the Replace command on the Geom Table Index line. Geometry tables can easily be shared by many different keyboards, since the physical area occupied by each keycap often remains the same, although the characters on the key faces and the Ascii codes that they generate are different.

Source Keyboard is the source of the bitmap, geometry table, or interpretation table that you want to insert into the current keyboard. If the source keyboard is in the same keyboard file as the current keyboard, the boolean **Same File** should be on. If not, you should type the name of the source keyboard file. For the following examples, assume that USEnglish is the source keyboard, and French is the current keyboard that we wish to alter, as illustrated in Figure 15.1.

If Same File is on, you have an additional parameter choice, Copy, for each keyboard component. If you turn on Copy, an identical copy of the source keyboard's component (for example, a geometry table) will be created and the current keyboard's pointer set to the new geometry table. If Copy is off, the current geometry table pointer will to point to the source keyboard's geometry table, instead of creating a new copy of the geometry table. Figure 15.2 illustrates the action of the Copy command:

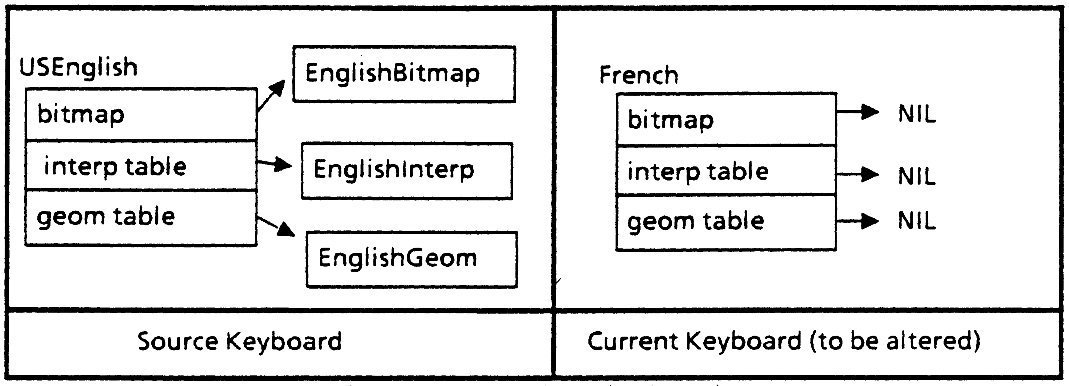


Figure 15.1 Altering a keyboard

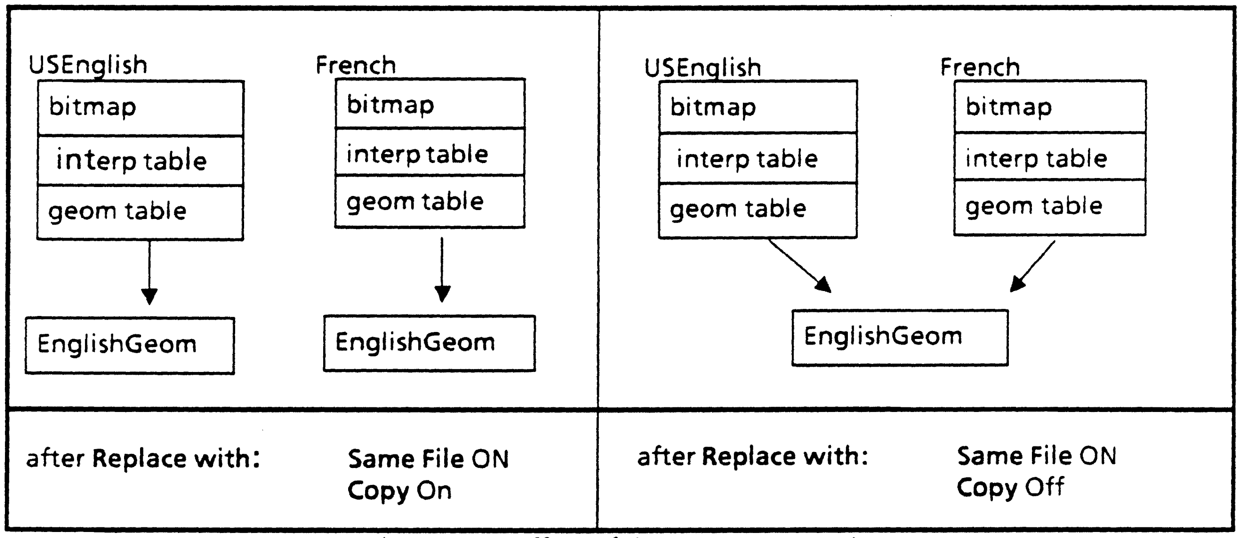


Figure 15.2 Effect of the Copy command

If Same File is off, no Copy parameters will appear, but there may be a parameter called All References. This appears if a component is referenced by more than one keyboard. Let's assume that there is a geometry table that is shared by several keyboards. If you try to replace that geometry table and All References is on, the geometry table will simply be replaced, so that all pointers now point to the new geometry table. If, however, All References is off, a copy of the new geometry table will be made and only the current keyboard will be set to point to the new copy. Figure 15.3 illustrates the effect of All References.

15.4 Key subwindow

When you select a key in the bitmap subwindow, information about the key will appear below in the key subwindow. It is the same information that you get from List, described above. You use the key subwindow to:

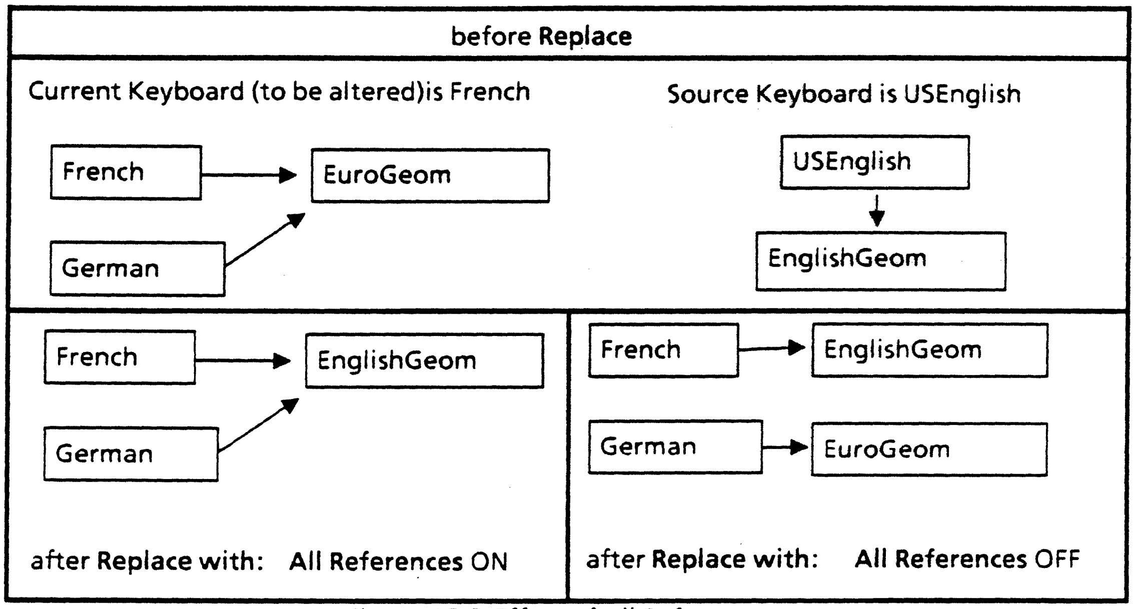

before Replace
Current Keyboard (to be altered)is French
Source Keyboard is USEnglish
French
EuroGeom
German
USEnglish
EnglishGeom
French
EnglishGeom
German
after Replace with: All References ON
French
EnglishGeom
German
EuroGeom
after Replace with: All References OFF

Figure 15.3 Effect of **All References**

- Select a key and examine its attributes
- Change the attributes of the key

Remember, you can only examine and change the interpretation of black keys, which are the only ones shown on the bitmap. The interpretation of function keys is controlled by TIP tables. Here is an example of the key subwindow:

KeyStation: {k3} **Set! Reset!**
Ignore **Repeat Type:** {OneShift}
Unshifted: {Character} **ChSet** = 0B **Code** = 141B
Shifted: {Character} **ChSet** = 0B **Code** = 101B
Lockable

KeyStation is the key station name of the key as defined by the KeyboardWindow interface in the *ViewPoint Programmer's Manual*. You can also select the key station from the enumerated list of KeyStation, instead of selecting the key from the bitmap.

The Set command causes any changes you make to take effect; Reset sets the parameters back to the way they are in the keyboard file.

Ignore will be set if the key is ignorable, Lockable if the Lock key can be used to lock the key, and Repeat if the key is repeatable. Lockable only appears if the key can be shifted. If the key can be shifted, its Type will be **OneShift** or **TwoShift**, and the **Shifted** and **Unshifted** interpretations must be defined. **TwoShift** is used on keyboards with many characters, such as the Japanese keyboard, where a single key may have three separate meanings, depending on whether the SHIFT key has been pressed once, twice, or not at all. At the time of this release, it is recommended that you always define interpretations as Character, since the other choices are functions that are no longer defined by the keyboard, but are instead defined by TIP tables. (See the TIP chapter for more information.)

The Type option **SendUpStroke** refers to whether the character is sent when the key is pressed down or let up--normal operation is to use the down stroke, but some emulators may need to use the up stroke. A key can also be defined as Type **ShiftKey**.

ChSet refers to the Xerox Character Set, and Code shows theASCII codes that are generated by the key.

15.5 How to edit a keyboard

Now you'll see a complete example of how to alter a keyboard file for your own uses. Let's say you want to interchange the "T" and the "S" key on the English keyboard. (This is a contrived example, but it will illustrate the use of the Keyboard tool.) T ake the following steps:

1. Change the appearance of the bitmap.

2. Assign the edited bitmap to a new keyboard file.

3. Change the interpretation of the keys so that pressing the new "S" will really print an "S" on the screen, and pressing the new "T" will print a "T".

15.5.1 Getting started

Run **KeyboardTool.bcd.** You must have on your search path the standard keyboard file, which is currently called "Keyboards". Open Keyboards, and select USEnglish from the Keyboard enumerated menu. This will paint the USEnglish keyboard bitmap in the bitmap subwindow, and fill in the other parameters for USEnglish.

First, you need to write the bitmap to a file so that you can edit it. In the Keyboard subwindow, type *TempBitmap* on the Bitmap File: line, and select Store Bitmap. This saves the USEnglish bitmap in the file **TempBitmap**.

Type *TempGeom* on the Geom File: line, and use the Store Geom command to save the USEnglish geometry tables. They will enable you to select keys and change their interpretations.

Select the "T" key. Note that the octal codes in the key subwindow are 164B and 124B. The codes for the "S" key are 163B and 123B.

15.5.2 Editing the bitmap

In the Bitmap Edit tool, enter Edit mode by selecting the crosshairs brush. Type *TempBitmap* on the File name: line and select Load Keyboard from the Edit menu, which you get by chording in the command subwindow. In the bitmap window, press the left mouse button to see where the bitmap will be placed in the window; move the mouse until you are satisfied with the bitmap's location. Then release the left button. The

USEnglish keyboard will now be painted in the Bitmap Edit tool.

Select the "T" key by pressing the left mouse button while in the upper-left corner of the "T" key and holding down the right button to adjust the selection to include the whole key. Select Move from the Edit menu (or use the MOVE key on the keyboard) to move the "T" to a vacant part of the window, outside of the keyboard bitmap. Similarly, move the "S" to the old "T" key, and move the "T" to the "S" key. (If you wanted to alter the appearance of any of the characters on the keyboard at this time, you could select the character, select Magnify from the Edit menu, and change the appearance of the character by turning the grid bits on and off. See the Bitmap Edit Tool chapter for more information.) Select the entire keyboard, and save the changes you have made to **TempBitmap** by selecting Store Keyboard from the Edit menu. Now you are finished editing the bitmap.

15.5.3 Creating a new keyboard file

There are no facilities for naming your own keyboard. You must use an existing keyboard definition and write your definition on top of it. In the Keyboard tool, choose a keyboard name from the enumerated Keyboard list. Choose a keyboard which you aren't likely to need for anything else. For this example, we'll use the Korean keyboard. First, select the Korean keyboard. Unless you have Korean installed on your machine, a blank keyboard will probably be painted in the bitmap window. Load your edited file TempBitmap with the Load Bitmap command. The "T" and "S" keys should appear interchanged.

15.5.4 Changing the interpretation of the keys

Your new keyboard, Korean, has no geometry file or interpretation file attached to it yet. The geometry file enables you to select the keycap labeled "T" and have the interpretation for the "T" key, which is provided by the interpretation table, appear in the key subwindow. It is much easier to edit an existing file than to create a new one, so we'll use the USEnglish tables.

Set Source Keyboard to USEnglish and turn on the Copy booleans on the KbdInt and Geom Table lines. This will copy the USEnglish tables into the Korean keyboard definition, rather than just having the Korean definition point to the USEnglish definition.

Invoke Replace on the KbdInt line and then on the Geom Table line.

Select the "T" key. The codes displayed are 163B and 123B, which are really the codes for "S". Change the codes to 164B and 124B and invoke Set!. Select "S" and set its codes to 163B and 123B.

You now have a keyboard file called Korean, which is really the USEnglish keyboard with the "T" and "S" keys interchanged.

15.5.5 Using a keyboard file

Creating a new keyboard definition is only part of the process you must complete to be able to use your new keyboard. The Keyboard tool only provides facilities for changing the pictorial representation and the interpretation of the black keys. In addition, the interpretation of the black keys is limited to characters; the Keyboard tool cannot be used to map functions to keys. For more information about assigning functions to keys, read the TIP chapter of this manual and the TIP chapter of the *ViewPoint Programmer's Manual*.

To integrate your new keyboard file into a program, you will need to become familiar with the following interfaces in the *ViewPoint Programmer's Manual*:

- BlackKeys
- KeyboardKey
- KeyboardWindow

15.6 Restrictions

File subwindow: The Keyboard tool will crash if you type an invalid file name in **File:** and select Open. Also, all of the default parameters are obsolete.

Keyboard subwindow: The parameter AllKeys does not currently have any effect.

Key subwindow: Use Character as the Type for all keys. There are other types available from the menu, but they date back to an earlier release, before functions were implemented by TIP tables.

A APPENDIX

A.1 Boot switches

The following list represents boot switches that affect the booting sequence and are consulted at boot time. They are recognized either by the ViewPoint boot file or by Pilot, the XDE operating system. Generally, you decide on a default switch combination and set it from the utility program called Othello to avoid having to set them every time you boot.

In addition to the switches presented below, see also Chapter 4 for a list of CommandCentral client switches. They too take affect at boot time, but are implemented by CommandCentral.

'd Disable the debugger substitute; for example, run with a debugger (CoPilot) volume.

'l Empty the Prototypes folder. This is useful for upgrading from one version to another. Boot time is increased with the 'l switch, and it should only need to be used once.

'N Do not run .autorun applications. Invisible applications are still run. If Developer is TRUE in the System section of the Workstation Profile, then nothing is run.

'O Use only **.TIPC** files and do not look for **.TIP** files.

'P Copy, load, and start CommandCentral files in parallel

A.1.1 More details on the 'P switch

When ViewPoint is booted with the 'P switch, the files handed to it by CommandCentral are processed in parallel: that is, one process copies the files to the System folder, one process loads them, and another process starts them up. This speeds system startup.

This overlapped loading and starting are fine if a program's imports are satisfied only by programs in the boot file, or by programs started previously; this should be the usual case.

If instead a program's imports are satisfied by a program that is later in the load sequence, the importer could get confused by the partially loaded following program. This should not be much of a problem in practice because applications should only import interfaces from ViewPoint and common software packages (which therefore, should be loaded first).

Note: Do not use the 'P switch when installing data files needed by ViewPoint itself, or when installing .autoRun files. (See Chapter 4 for a discussion of .autoRun).

The /p CommandCentral Run line switch is used to handle loading programs whose imports will be satisfied by configs loaded subsequently (not the normal case). /p means "make Parallel loading activity Pause while this program is being started." See Chapter 4 for details.

'T Use software **TextBlt**.

'u Use a volume named User as the data volume. If this switch is not given, the system volume will be used for the data volume.

'y Allocate Pilot backing store from available space on neighboring volume named Scavenger, System, or Star. This allows you to save 2000 pages of disk space on the User volume when running ViewPoint on a single User volume. You can boot with the '{ switch (see below) when booting with 'y.

'J	VP Standalone and Remote operations should be run at full duplex.
's	Enables local thermal printer.
'U	Tells VP Standalone and Remote that the workstation is a standalone or a remote workstation.
'V	Indicates to VP Standalone and Remote that the workstation administration option sheet should appear after the idle procedure is interrupted. If this switch is not used (the normal case), the Logon option sheet appears.
'W	Enables foreground/background operation for the impact printer.

A.1.2 Pilot switches

The remaining switches are recognized by Pilot. See the *Pilot Programmer's Manual* for more information.

&	Hang with a maintenance panel code instead of going to the debugger.
0	Go to the debugger as early as possible in Pilot initialization.
1	Go to the debugger as soon as all code is map-logged.
2	Go to the debugger just before calling **PilotClient.Run**.
3	Simulate 192K memory size for a Dandelion with no display.
4	Initialize scratch memory pages to zero.
5	Go to the Ethernet for the debugger.
6	Turn owner checking on for the system zones.
7	Disable map logging.
8	Create a Pilot interrupt key watcher.
9	Simulate 256K memory size for a Dandelion with display.
.	Suppress clock failure errors.

:	Go to the debugger as early as possible in the initialization of the File manager.
;	Go to the debugger as early as possible in the initialization of the VM manager.
<	Pretend that there is no Ethernet 1 attached to the system element.
=	Do not initialize the Communication package at system startup.
>	Pretend that there is no Ethernet attached to the system element.
{	Set the VM backing file size to 750 pages.
|	Set the VM backing file size to 1400 pages.
}	Set the VM backing file size to 2000 pages.
↑	Turn on checking, for the system zones.
?	Make loadstate resident (for debugging on UtilityPilot - based clients).
[	Create a tiny heap,with tiny increment values.
%	Create a medium-size heap with medium-size increment values (default).
]	Create a large heap with large increment values.

/350	Helps to save MDS space. ViewPoint is usually booted with this switch. However, be advised that using this switch will affect the use of the XDE Packager, and integral swap units will no longer be guaranteed (contrary to the documentation in the Packager chapter of the *XDE User's Guide*.) Also, you cannot use this switch if you boot from CommandCentral.
/360	Display error code, global frame, and pc on boot loader errors.
/370	Bypass the debugger substitute by going to the real debugger.
/371	Tile code with one page swap units.
/372	Give display memory to Pilot for client use.
/373	Give display memory to Pilot for client use if there is no bitmap display.
/374	Allows special clients to set parameters of system zones.
/375	Disable map logging.
/376	Delete boot loader so that the memory that it uses can be recycled.

Booting your workstation with boot switches to give it more data space backing storage cache may significantly improve performance.

Many ViewPoint applications get good performance by caching data in scratch virtual memory. However, more scratch virtual memory may be used up than is expected. Also, more of Pilot's VM backing file cache may be used up than might be expected.

B APPENDIX

B.1 Scavenging

This section describes how to scavenge the **NSFiling** volume that holds the desktop and system files for ViewPoint. Scavenging is required when the system crashes at an inopportune moment and the file system is not consistent.

The scavenger for **NSFiling** cannot scavenge the data volume if it is the same as the volume the boot file is running on. In this case, attempting to scavenge eventually arrives in the debugger with a message that says "Can't scavenge system volume".This makes scavenging more difficult since users often use the same volume both for the boot file and for data.

If the data volume needs scavenging, ViewPoint will break to CoPilot with the message, "Proceed to scavenge data volume".

If the boot file volume is the same as the data volume, proceeding will automatically boot another volume named Scavenger, System, or Star. This allows you to put a Scavenger boot file (for example, **BWSScavengerDLion.boot)** on the other volume. The Scavenger boot file is much smaller than the full ViewPoint boot file. When the Scavenger is finished, it will return to CoPilot with the message "Done scavenging". The ViewPoint boot file can then be successfully booted.

If the boot file volume is different from the data volume, proceeding will scavenge the volume, and everything will be in order.

If there is no debugger volume and the data volume needs scavenging, maintenance panel code 7501 will be displayed. Pressing the F and C keys simultaneously will change the MP code to 7502, then releasing the keys will change the MP code to 7500, indicating that the scavenge is being done. As above, if there is a boot file on another volume named Scavenger, System, or Star, that volume is booted. After the Scavenger boot file finishes on a machine with no debugger, the physical volume is booted.

C APPENDIX

C.1 ViewPoint Scavenger Maintenance Panel Codes

From time to time, codes will appear on the maintenance panel of your hard disk during routine operations. They are listed here, together with any actions you should take when they appear.

7500	Scavenging in progress. No action required.
7501	User volume requires scavenging. Press the F and C keys to proceed.
7502	Appears after you press F and C from 7501, should go to 7500 when you release F and C;
7504	Volume needs initializing. Press and release the I and V keys.
7508	Volume needs converting Press and release the F and C keys to proceed.
7511	There is no scavenger boot file on the Scavenger, System, or Star volume.

C.1.1 Normal MP Codes

7920	Starting ViewPoint.
7940	Starting invisible application folders.
7960	Starting visible application folders with autorun = TRUE.
8000	Done booting.

C.1.2 Debugger Substitute MP Codes

7540	AddressFault
7541	Breakpoint
7542	Bug
7543	CallDebugger
7544	CleanMapLog
7545	Diskerror
7546	Interrupt
7547	Rreturn
7548	ReturnAborted
7549	UncaughtSignal
7550	VisitDebugger
7551	WriteProtectFault
7552	Other