

VIEWPOINT PROGRAMMING COURSE

XEROX

**Xerox Corporation
Information Systems Division
475 Oakmead Parkway
Sunnyvale, California 94086**

Copyright © 1986, Xerox Corporation. All rights reserved.
Printed in U.S. A.

TABLE OF CONTENTS

1. Introduction

1.1 If you won't write ViewPoint applications	1-1
1.2 If you will write ViewPoint applications	1-1
1.3 Your investment	1-2
1.4 What you get out of it	1-2
1.5 Course structure	1-2
1.6 Reference documentation	1-3
1.7 Setting up your machine	1-3
1.8 Getting help and reporting problems	1-5
1.9 Future editions	1-5

2. User interface

2.1 The desktop and icons	2-1
2.2 Windows	2-2
2.3 Pop-up menus	2-3
2.4 Attention Window	2-4
2.5 Form windows and property sheets	2-4
2.6 The Directory icon	2-6
2.7 The Prototype Folder	2-6
2.8 Other user interface features	2-7

3. Strings and messages

3.1 XChar	3-1
3.2 XString	3-2
3.2.1 Readers and ReaderBodys	3-2
3.2.2 Writers and WriterBodys	3-5
3.3 XFormat	3-6
3.4 XMessage	3-10
3.4.1 The definitions module	3-11
3.4.2 The implementation module	3-12
3.4.3 The client module	3-14
3.5 Attention	3-14
3.6 Summary	3-15
3.7 Exercise: Concordance Tool	3-15

4. Creating a simple application

4.1 Adding a command to the Attention menu	4-1
4.2 Creating a StarWindowShell	4-2
4.2.1 Transition procedures	4-4
4.2.2 IsCloseLegalProc	4-4
4.3 Body windows	4-5
4.4 Commands	4-6
4.4.1 Putting commands in the header	4-7
4.4.2 Putting commands in a popup menu	4-9
4.5 Displaying windows on the screen	4-10
4.6 Summary	4-10
4.7 Exercise: DMT I	4-13

5. Context

5.1 Context types	5-1
5.2 Creating the context	5-2
5.3 Using the context	5-3
5.4 Summary	5-4
5.5 Exercise: DMT II	5-7

6. Displaying information on the screen

6.1 Overview	6-1
6.2 Invalidating and validating	6-1
6.3 Writing a display procedure	6-3
6.3.1 SimpleTextDisplay	6-4
6.3.2 Display	6-5
6.4 LimitProcs and AdjustProcs	6-10
6.5 Summary	6-11
6.6 Exercise: Text Window	6-12

7. Form windows

7.1 Overview	7-1
7.1 Creating form windows	7-2
7.2.1 MakeItemsProcs	7-3
7.2.2 LayoutProcs	7-6
7.2.3 ChangeProcs	7-9
7.2.4 Example	7-10
7.3 Getting and setting values	7-13
7.4 Destroying a form window	7-13
7.5 Summary	7-13
7.6 Exercise: Time Clock I	7-13

8. Property sheets

8.1 Creating a property sheet	8-1
8.1.1 MenuItems	8-2
8.1.2 MenuItemsProc	8-2
8.1.3 Example	8-3
8.2 Linked property sheets	8-4
8.3 Summary	8-7
8.4 Exercise: Time Clock II	8-8

9. TIP

9.1 Overview	9-1
9.2 TIP tables	9-2
9.2.1 TIP table syntax	9-3
9.2.2 Results	9-4
9.3 The NotifyProc	9-6
9.4 Incorporating the new TIP table	9-7
9.4.1 Creating a compiled TIP table	9-7
9.4.2 Associating tables and NotifyProcs	9-9
9.4.3 Input focus	9-10
9.4.4 Example	9-11
9.5 Periodic notifiers	9-13
9.6 User aborts	9-15
9.7 Summary	9-15
9.8 Exercise: Tank	9-16

10. NSFiling attributes

10.1 Content and attributes	10-1
10.2 Interpreted and uninterpreted attributes	10-1
10.3 Specifying attributes	10-2
10.3.1 Specifying interpreted attributes	10-3
10.3.2 Specifying extended attributes	10-4
10.4 Getting interpreted attributes	10-5
10.4.1 Selections	10-6
10.4.2 Attributes	10-7
10.4.3 Sessions	10-7
10.4.4 Storage management	10-8
10.4.5 Example of GetAttributes and SetAttributes	10-8
10.5 Getting extended attributes	10-10
10.6 Summary	10-11
10.7 Exercise: Music Man I	10-12

11.NSFiling operations

11.1 Naming files	11-1
11.2 Opening local files	11-2
11.2.1 NSFile.Open	11-2
11.2.2 NSFile.OpenByReference	11-3
11.2.3 NSFile.OpenByName	11-4
11.2.4 Catalog.Open	11-4
11.3 Opening remote files	11-5
11.4 Closing files	11-7
11.5 Creating files	11-7
11.5.1 NSFile.Create	11-7
11.5.2 Catalog.Create	11-8
11.6 Deleting files	11-9
11.7 Finding files	11-9
11.7.1 Filters	11-10
11.7.2 NSFile.Find	11-11
11.8 Listing files	11-11
11.9 Errors	11-13
11.10 Summary	11-14
11.11 Exercise: Music Man II	11-15

12.Streams

12.1 Overview	12-1
12.2 Creating a stream	12-2
12.3 Stream I/O	12-3
12.3.1 Example of I/O	12-3
12.3.2 Block I/O	12-4
12.3.3 Random access	12-6
12.3.4 Miscellaneous operations	12-7
12.4 Deleting streams	12-7
12.5 Summary	12-8
12.6 Exercise: Stream Tool	12-8

13. NSSegment

13.1 Definition of terms	13-1
13.2 Virtual memory overview	13-1
13.3 Mapping	13-2
13.3.1 Setting up the mapping	13-2
13.3.2 Accessing the file	13-4
13.3.3 Updating the file	13-5
13.3.4 Complete mapping example	13-6
13.4 CopyIn and CopyOut	13-8
13.5 Summary	13-9
13.6 Exercise: Black Book I	13-17

14. Selection requestors

14.1 Converting the selection	14-1
14.2 Resource management	14-3
14.3 Can you convert the selection?	14-5
14.4 Enumerating selections	14-6
14.5 Summary	14-7
14.5 Exercise: Checkers	14-8

15. Icon applications

15.1 Overview	15-1
15.2 Registering with the desktop	15-1
15.2.1 PictureProcs	15-3
15.2.2 SmallPictureProcs	15-5
15.2.3 GenericProcs	15-6
15.3 The Prototype interface	15-11
15.4 Summary	15-12
15.5 Exercise: Tic-Tac-Toe	15-14

16. Application folders

16.1 Building an application folder	16-1
16.1.1 Application description files	16-2
16.2 Modifying the code	16-3
16.2.1 Message files	16-4
16.2.2 Private TIP file	16-7
16.2.3 Private icons file	16-9
16.3 Create the application folder	16-9
16.4 Summary	16-10
16.5 Exercise: Black Book II	16-11

17. DocInterchange

17.1	Creating documents	17-1	
	17.1.1	Adding information to a document	17-2
	17.1.2	Finalizing a document	17-7
17.2	Properties	17-8	
	17.2.1	Anchored frame properties	17-9
	17.2.2	Font properties	17-9
	17.2.3	Page properties	17-9
	17.2.4	Field properties	17-10
	17.2.5	Utilities for getting and setting properties	17-10
17.3	Example: creating a document	17-11	
17.4	Enumerating documents	17-12	
17.5	Summary	17-14	
17.6	Example: copying a file	17-14	
17.7	Exercise: Form Letter application	17-17	

18. Graphics

18.1	Creating graphics	18-1	
	18.1.1	Start routines	18-1
	18.1.2	Add routines	18-5
	18.1.3	Finish routines	18-11
18.2	Reading graphics	18-12	
18.3	Summary	18-14	

A. Running an application

A.1	The programming cycle	A-1
A.2	The WorkstationProfile	A-1
A.3	The SystemFolder	A-2
A.4	The Application Loader	A-2
A.5	.autorun files	A-3

B. Icon editor

B.1	Getting started	B-1
B.2	Editing an icon	B-2

C. Message tools

C.1 Message Master File Creation tool	C-1
C.2 Message file property sheet	C-3
C.3 Message Master Editor	C-3
C.3.1 Searching Message Master files	C-4
C.3.2 Search parameters	C-5
C.3.3 Closing, saving, and resetting	C-7
C.3.4 Printing message files	C-7
C.4 Runtime File Creation tool	C-8

1-1	The Course directory structure	1-4
2-1	Icons on a desktop	2-1
2-2	A basic window	2-2
2-3	Attention window	2-4
2-4	Sample form window	2-5
2-5	Property sheet for the Wastebasket icon	2-5
2-6	Prototype folder	2-6
3-1	<code>xstring.ReaderBody</code>	3-2
3-2	Reader and <code>ReaderBody</code>	3-3
3-3	The <code>XFormat</code> abstraction	3-7
3-4	Using a writer as output sink	3-8
3-5	An <code>xFormat.Object</code> with a stream as output sink	3-8
3-6	Diagram of <code>XFormat</code> example	3-10
3-7	The Concordance Tool	3-16
4-1	A <code>StarWindowShell</code>	4-3
4-2	Commands	4-7
4-3	The DMT Tool	4-13
5-1	Contexts	5-1
6-1	The Checkers display	6-6
6-2	The Text Window Tool	6-12
7-1	Form Window	7-1
7-2	The <code>TimeClock</code> application	7-14
8-1	Property sheet	8-1
8-2	Linked property sheet	8-4
8-3	The <code>Time Clock</code> property sheet	8-8

9-1	Path of user input	9-1
9-2	The Placeholder tables	9-2
9-3	The normal TIP tables	9-3
9-4	A possible results list	9-5
9-5	Pushing NewTableA onto mouseActions	9-9
9-6	Pushing NewTableB onto mouseActions	9-9
9-7	The Tank application	9-15
10-1	An NSFile	10-1
10-2	ARRAY of NSFile.Attributes	10-3
10-3	ARRAY of NSFile.Attributes	10-5
10-4	An NSFile.Selections record	10-6
10-5	selections and myAttributes	10-9
10-6	After a call to GetAttributes	10-9
10-7	attributeArray	10-9
10-8	Attributes record	10-11
10-9	The Music Man application	10-12
12-1	A stream to a file	12-1
12-2	After a GetBlock operation	12-5
12-3	Using stream block operations	12-6
13-1	Mapping a file to virtual memory	13-3
13-2	An NSsegment.Origin	13-3
13-3	Mapping a file then LOOPHOLING	13-5
13-4	Unmapping	13-6
13-5	The CopyOut operation	13-8
13-6	The File Tool	13-10
13-7	Little Black Book	13-17
14-1	The Checkers tool	14-8
15-1	Icons	15-3
15-2	Document header with tiny icons	15-5
15-3	The Tic-Tac-Toe application	15-14
16-1	ADF syntax	16-3
16-2	Building an application folder	16-10

17-1	Appending to a document	17-3
17-2	Adding text to a header	17-3
17-3	Data file	17-17
17-4	Template document	17-17
17-5	Form letter icon	17-18
18-1	Graphics frames	18-3
18-2	Arrowheads	18-6
18-3	Lines	18-6
18-4	Ellipse	18-7
18-5	Defining curves	18-8
18-6	Curves	18-9
18-7	Text frame	18-11
C-1	Message Master Creation Tool	C-1
C-2	Master File icon	C-2
C-3	Message Master property sheet	C-3
C-4	Message Master Editor window	C-4
C-5	Searching by parameter	C-6
C-6	Print options	C-7
C-7	Using the Create Message File command	C-8
C-8	The resulting icon	C-8

As the authors of this course, we have made some assumptions about your background and why you will be taking this course. As the readers, you undoubtedly would like to know what you are getting into and what you will get out of it. This chapter describes our assumptions and should help you set your expectations. It also describes the course structure and reference documentation, how to get started, how to get help, how to report problems, and how to have your suggestions incorporated into future editions of this manual.

1.1 If you won't write ViewPoint applications

If you do not intend to write ViewPoint applications we assume the following about you:

- You have Mesa programming experience but little if any knowledge of ViewPoint application development.
- You want an understanding of the principal components of a typical ViewPoint application and how they interact.

We assume that you know how to read Mesa code.

This manual is intended for programmers. If you are an experienced Mesa programmer reading this manual to get the flavor of ViewPoint application programming, all well and good. If you are not an experienced Mesa programmer and do not intend to do any programming exercises, be warned: the material is dense and you are not properly dressed for the terrain.

1.2 If you will write ViewPoint applications

If you want to develop ViewPoint applications we assume the following about you:

- Your knowledge of ViewPoint application programming is limited and you want to expand it to the point that you can write significant ViewPoint applications.
- You are familiar with the Xerox Development Environment (XDE).
- You can program in Mesa. We assume that, at a minimum, you have worked through at least the first ten chapters of the Mesa Course (see Section 1.6). Nothing in this course depends directly on the Mesa Course, however, so you are free to acquire expertise in Mesa in other ways.

1.3 Your investment

You can read the text of this manual in two days.

We estimate that it will take between three to eight weeks of full time study to complete all readings and associated exercises.

1.4 What you get out of it

Whether or not you intend to program in ViewPoint, you will acquire an appreciation of the architecture of a typical ViewPoint application.

If you complete the course and its associated exercises, you will be able to write significant ViewPoint applications, and you will know where to look for information on more advanced topics.

1.5 Course structure

The course has 18 chapters and 3 appendices. This is the first chapter. The second chapter discusses the user interface. The remaining chapters discuss how to create a new application, starting with the basics and gradually increasing in complexity. Each chapter concentrates on a particular topic, providing a description and examples to illustrate the topic. You should work through the chapters in order.

Each chapter also has an associated programming exercise. Some of the exercises build on one another, although we have tried to keep them largely independent. We expect that those of you who intend to develop ViewPoint applications will do all of the exercises. The exercises vary in complexity and difficulty; the easiest can be done in a matter of hours, the hardest may take as long as a week. We provide solutions to all of the exercises.

The appendices contain information that you need to know, but that is not directly part of this course.

Appendix A, Programming in ViewPoint, describes the logistics of writing, testing, and debugging a ViewPoint application. If you don't already know how to do this, you should read Appendix A immediately after you read this introduction.

Appendix B, Icon Editor, and Appendix C, Message Tools, describe how to use some ViewPoint applications that you may need to use in the final chapters. The course text directs you to these appendices at the appropriate point. There are no exercises associated with the appendices.

1.6 Reference documentation

The goal of this course is to familiarize you with the basic structure of a ViewPoint application and get you started; it is not a reference manual, and it does not replace the existing reference manuals. Here is a list of other documentation that you may need to use before, during, or after the course.

DF Software Reference Manual

This describes how to use the df software to simplify storing and retrieving large numbers of files. You may need to consult this manual to find out how to use the df files to retrieve the course software. (See the next section.)

Mesa Course (version 12.0, July, 1985)

This is a programming course that covers the Mesa language and programming for XDE.

Mesa Language Manual (version 3.0, November 1984)

This is the reference for the Mesa programming language.

Services Programmer's Manual

This manual contains individual reference manuals for Services interfaces. In particular, you will need the *Filing Programmer's Manual*, (November, 1984), which documents the ViewPoint filing system.

ViewPoint Programmer's Guide (September, 1985)

This is a complete reference for ViewPoint programming. It describes all procedures in the public ViewPoint interfaces. This course is basically a condensed version of the information in this manual. This is the **primary reference for the course**.

ViewPoint Series Reference Library

This is the set of user manuals for ViewPoint applications software. It contains information on documents, folders, and the like.

XDE Tutorials (September, 1985)

This is a printed version of the on-line tutorials.

XDE User Guide (version 3.0, November 1984)

This is the user manual for the development environment. It discusses the XDE user interface, and the tools that run in XDE.

1.7 Setting up your machine

This section describes how you should set up your machine. In most cases, you should show this section to an experienced user and let him or her set up your machine for you.

We assume that you have a Dandelion or Daybreak running the XDE 4.0 (or later), and ViewPoint 1.0 (or later). Your volume configuration isn't critical, but you should have at least a CoPilot volume and a ViewPoint (aka User) volume.

Your machine should have the SystemFolder application, and a WorkstationProfile with the following two entries:

[Application Loader]
Developer: TRUE

[System]
Developer: TRUE

You should also check your user.cm file, which lives on your CoPilot volume. The only section that will affect this course is the [Executive] section. In particular, you need to check your ClientVolume, CompilerSwitches, BinderSwitches, and ClientSwitches entries. Here is one possible [Executive] section:

[Executive]
CompilerSwitches: eub-j
BinderSwitches: ec
ClientVolume: User
ClientSwitches: Ody\365

The files that you will need for the course are stored on [Bob:OSBU North:Xerox]<ViewPointProgrammingCourse>. Figure 1.1 illustrates the structure within this drawer.

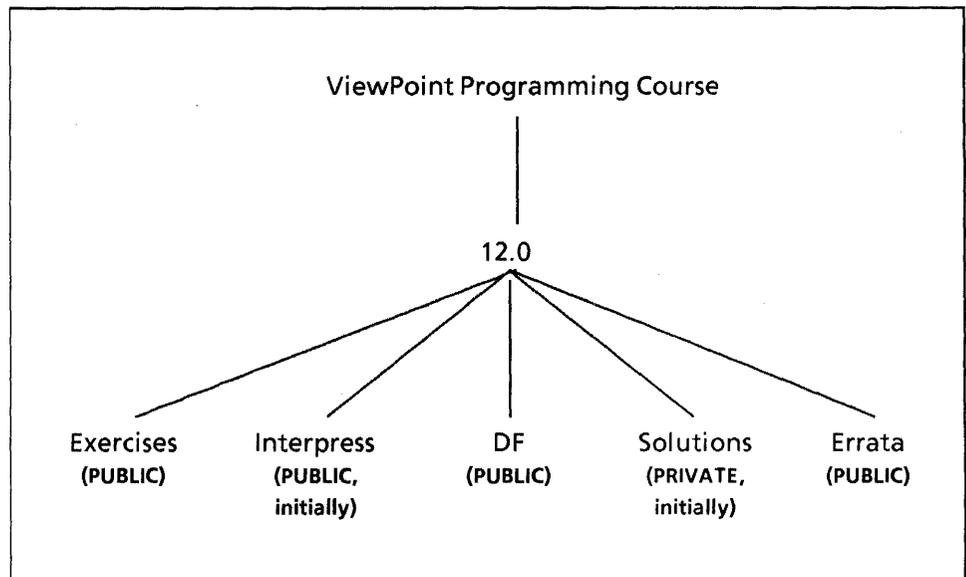


Figure 1.1: The course directory structure

The **Exercises** folder contains the files that you will need to do the exercises at the end of each chapter. This folder contains a folder for each chapter in the course.

The **Interpress** folder contains interpress masters for each chapter of the course. You can print a copy of the course from these if you must, but we recommend that you do not overload your printer. Double sided, printed, bound copies of this course are available from:

Patience Nason
XDE Technical Services
Xerox Corporation
475 Oakmead Parkway
Sunnyvale, CA., 94086

There is a charge for the manual.

The **DF** folder contains df files for each chapter in the course. You can use these files to retrieve everything you need to do the exercises for a given chapter. To find out how to use these, consult the *DF Software Reference Manual*.

The **Solutions** folder contains our solutions to the programming exercises.

The **Errata** folder contains a description of the mistakes that we have found since the last printing.

1.8 Getting help and reporting problems

This is the first edition of this course, and we expect that you will find some problems and have some suggestions for improvements. If you do, we would appreciate it if you would report them. Internal users can submit ARs against the course (System Documentation:OSBU South, subsystem Programming Course); external users can send mail messages to the distribution list XDESupport.osbunorth@Xerox.COM.

1.9 Future editions

If you have suggestions about topics that you would like to see included in future editions, we would like to hear from you. If you have written ViewPoint applications that are elegant, relevant, and reasonably compact and would like to see them immortalized in subsequent editions, please submit them to us for review. You can contact the authors through the distribution list XDE-Training:osbu north:Xerox if you are an internal user or XDE-Training.osbunorth@Xerox.COM otherwise.

Notes:

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

ViewPoint also provides programming interfaces to support these user interface characteristics. By using these interfaces when you write new applications, you ensure that your applications integrate well with existing software. This chapter describes the basic components of the user interface; the rest of this course describes how 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 represents an object in that office. A typical desktop might include icons that represent various documents, folders, mail baskets, a printer, and so on.

The user accesses an object through its icon, generally by selecting the icon and pressing the OPEN key. The MOVE, COPY, and DELETE keys also apply to icons. Figure 2.1 illustrates a desktop with several different icons and an open document.

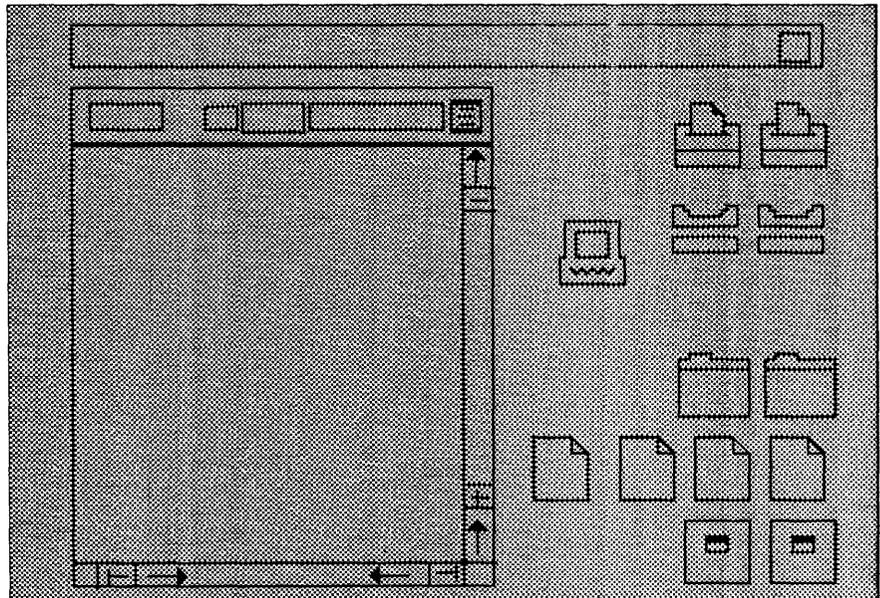


Figure 2.1: Icons on a desktop

The user can move the icons to different positions on the surface, but two icons cannot occupy the same square at the same time.

The use of icons is simple and intuitive. Therefore, you should use icons to represent 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.

2.2 Windows

A *window* is a rectangular region of the display screen in which an application can display information to the user. Figure 2.2 illustrates the various parts of a basic window.

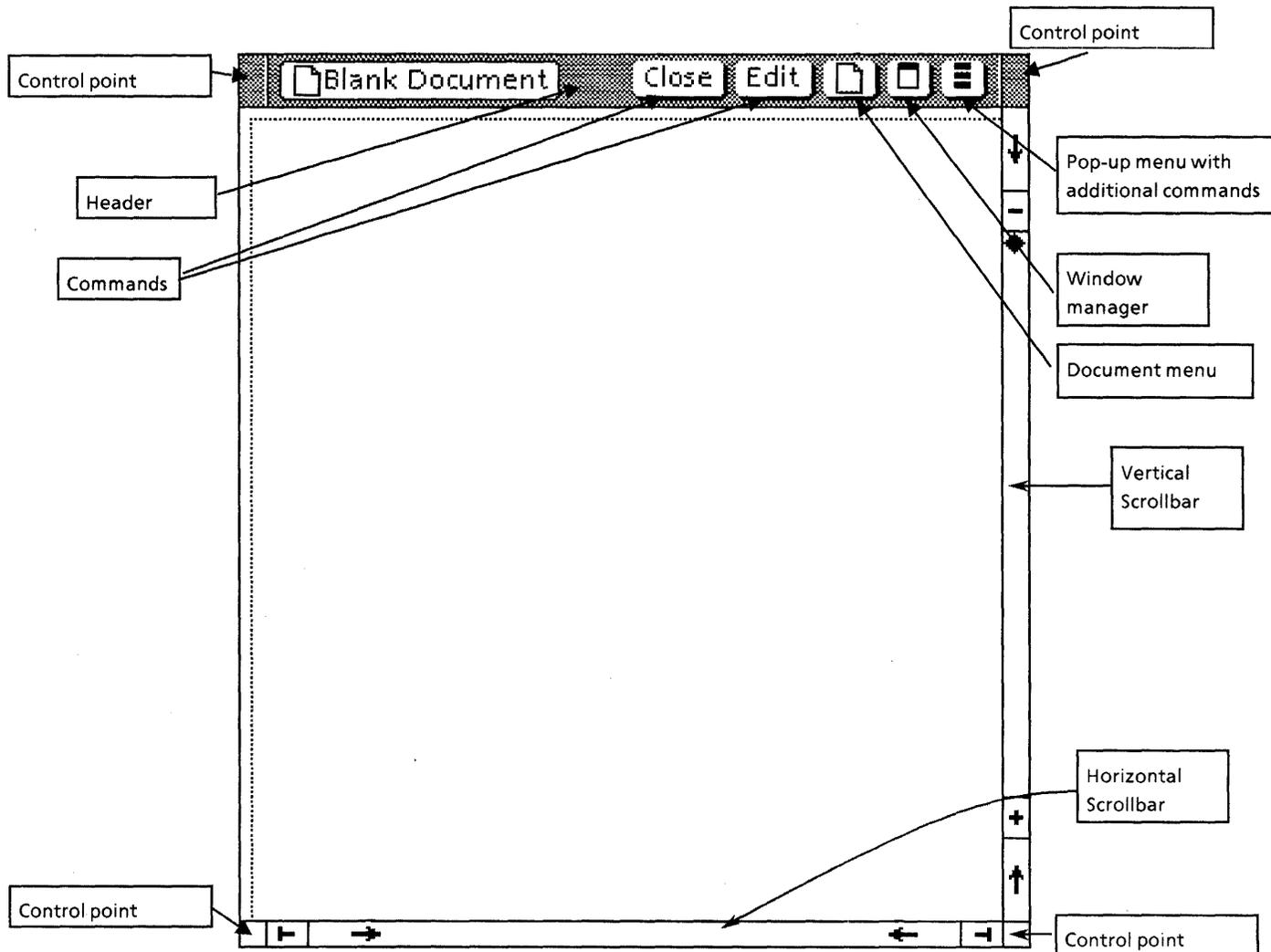


Figure 2.2: A basic window

Windows can be in one of two modes: *overlapping* or *tiled*. In overlapping mode, windows can appear on top of each other and 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 changes the size of 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 reopen it, the system remembers the window's initial position and redisplay 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. (The next section discusses the Attention Menu.)

If you want to change the defaults for these parameters, you can edit the User Profile. (For more information on the User Profile, consult the ViewPoint user documentation.) 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 (\equiv). Each application generally has a pop-up menu; the

author of the application chooses which commands go directly in the window header and which go in the pop-up menu. However, if the window is too small for all the specified commands to fit in the header, the rightmost header commands will automatically overflow into the pop-up menu instead of appearing in the header. 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.

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; you can access the menu by mousing *anywhere* in the Attention window, not necessarily over the menu symbol itself. 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.)

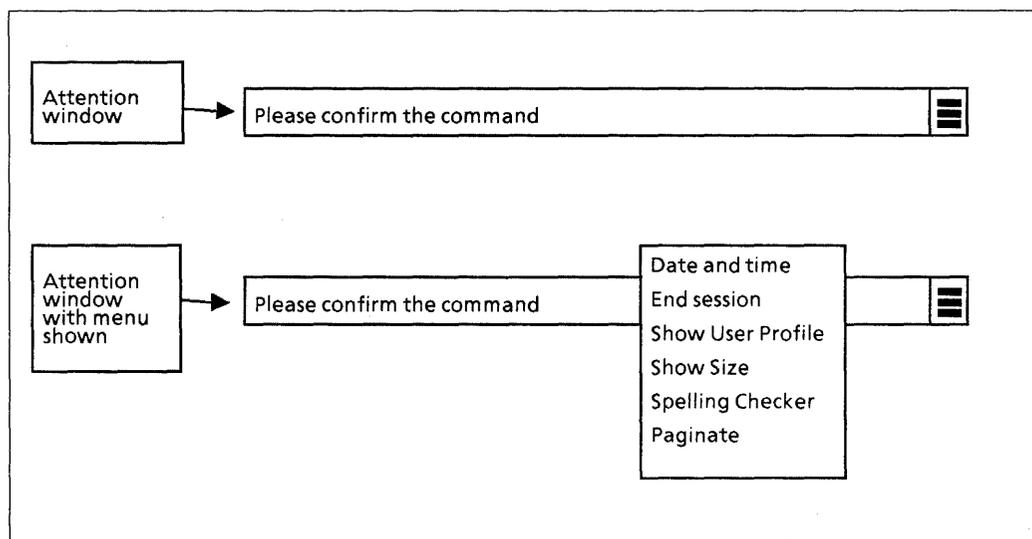


Figure 2.3: Attention window

You can access a standard set of commands available from the Attention menu, and applications can add 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.

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.

The figure shows a window titled "CALENDAR" with a menu bar containing "Close", "Next", "Previous", "Another", "Destroy", and "Hardcopy". The main area contains several form elements:

- Scan:** A row of four buttons: "Time", "Place", "Title", and "None".
- For:** A text input field.
- Time:** A text input field containing "9 Aug 85 19:00".
- Place:** A text input field.
- Title:** A text input field.
- Duration:** A row of four buttons: "30", "Minutes", "Hours", and "Days".
- Details:** A text input field.
- ReminderType:** A row of four buttons: "Mail", "Window", "Both", and "None".

Callouts on the right side of the window:

- "Text items (to be filled in by user)" points to the "For", "Time", "Place", and "Title" fields.
- "Choice items" points to the "Duration" and "ReminderType" rows.

Figure 2.4: Sample Form window

A property sheet is a form window in which the items control the *properties* of an object. To see the properties of an object, select the object and then press the `PROPS` key. Different objects have different properties; for example, the properties of a paragraph include left and right margins, justification, and line height. The header of a property sheet usually contains some subset of the standard commands: ? (Help), Done, Cancel, Apply, and Defaults. Figure 2.5 illustrates a property sheet for the Wastebasket icon.

The figure shows a window titled "WASTEBASKET PROPERTIES" with a menu bar containing "Done" and "Cancel". The main area contains the following elements:

- Purge deleted items:** A row of two buttons: "IMMEDIATELY" (highlighted) and "NEVER".
- Number of contained items:** The text "Number of contained items: 24".
- Total size:** The text "Total size: 1027 Disk Pages".

A vertical scrollbar is located on the right side of the window.

Figure 2.5: Property sheet for the Wastebasket icon

2.6 The Directory icon

Every ViewPoint desktop has a Directory icon, which 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 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.

2.7 The Prototype folder

When you run an application that has an icon, the icon does not automatically appear on the desktop. Instead, you must open a special system folder, known as the *Prototype Folder*, selected the desired icon, and copy it to the desktop. Figure 2.6 illustrates the Prototype folder.

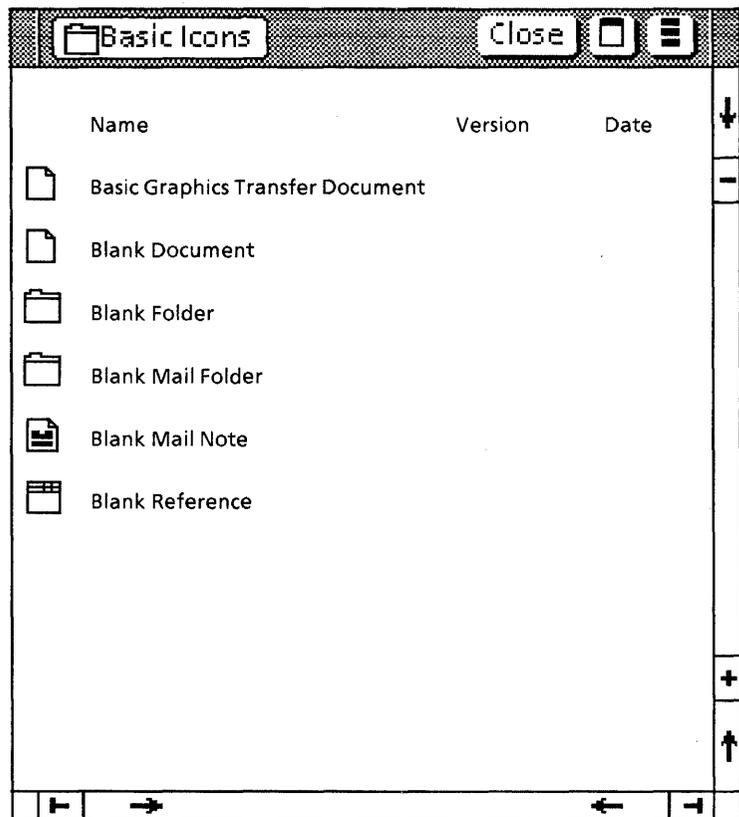


Figure 2.6 Prototype folder

There are two possible ways to access the Prototype folder. The first method is to open the Directory Icon and then the Workstation folder. Inside the Workstation folder is a folder called Basic Icons; this is the Prototype folder. Inside this folder, you will find such icons associated with the various applications

running on your machine; you can copy any or all of these icons to your desktop.

The second way to access the Prototype folder is a shortcut, but it requires that you be running the program **SystemFolder.bcd**. (The easiest way to run this program is to drop it on the Loader directly from a file drawer; see Appendix A, *Programming In ViewPoint*, for more information on running a program.)

SystemFolder registers three commands in the Attention Menu, one of which is Prototype Folder. Selecting this command will open the Prototype folder on your screen. (The other commands are System Folder, and Set System Folder Filter. See Appendix A, *Programming In ViewPoint*, for more information on these commands.)

2.8 Other user interface features

This chapter discussed the major user interface features that you might want to incorporate into a new application. For a complete discussion of the ViewPoint user interface from the user's point of view, however, you should refer to the *ViewPoint Series Reference Library* and the *ViewPoint Series Training Guides*.

Notes:

This chapter discusses some basic aspects of ViewPoint programming, such as string representation, manipulation, and management.

The specific interfaces covered are **XChar**, which defines the structure of a character; **XString**, which defines the structure of a string and various operations on strings; **XFormat**, which supports conversion among various data types; **XMessage**, which helps separate messages to the user from the rest of the code for an application; and **Attention**, which supports posting messages to the user.

The material in this chapter is difficult, but it is necessary background to the rest of the course.

3.1 XChar

Most computer systems 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. To allow true multinationality, character codes need to be much bigger, which has obvious attendant disadvantages.

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 65,512. This 65,512 range is partitioned into 256 blocks (*character sets*) of 256 character codes each. (Actually, there can be at most 255 x 255 characters; the last block is reserved.)

A character is thus composed of two 8-bit quantities: a character set and a character code. The character set is optional when all characters in a given string are part of the same character set. When there is no character set, the character code conforms to ASCII and ISO. This approach provides both versatility 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];
```

3.2 XString

The **XString** interface provides data structures and operations for strings encoded with the Character Code Standard. **XString** declares two kinds of strings: one for read-only access ("reader") and one for writing ("writer"). Readers occupy less space than writers. Thus, programs that create strings once and do not later need to modify them can save significant space.

3.2.1 Readers and ReaderBodys

XString defines the following types for readers:

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): **XString.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 referenced by **bytes**. **limit** is the offset from the pointer to the byte after the last byte in the byte sequence; and **offset** is the offset from the pointer to the first byte (the "beginning" of the string). Figure 3.1 shows two **ReaderBodys**, one that starts at the beginning of the byte sequence (**offset** = 0), and one that starts in the middle of the byte sequence (**offset** ≠ 0.)

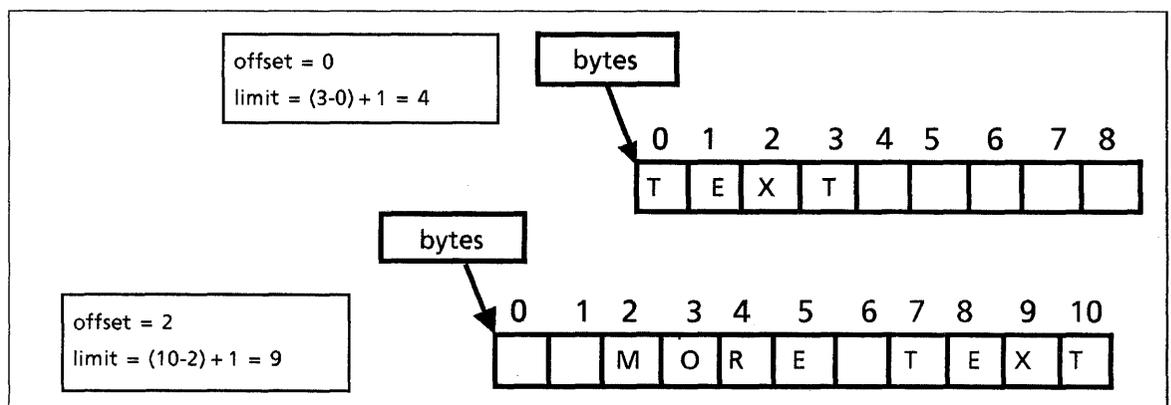


Figure 3.1: **XString.ReaderBody**

A **Context** contains information about the character encoding within 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 set shifts. The **prefix** field specifies the character set of the first character. Subsequent characters in the string use the same prefix unless there is an encoding transition. (The prefix field is used only for 8-bit characters, since the 16-bit representation includes a character set.)

Figure 3.2 illustrates these data structures.

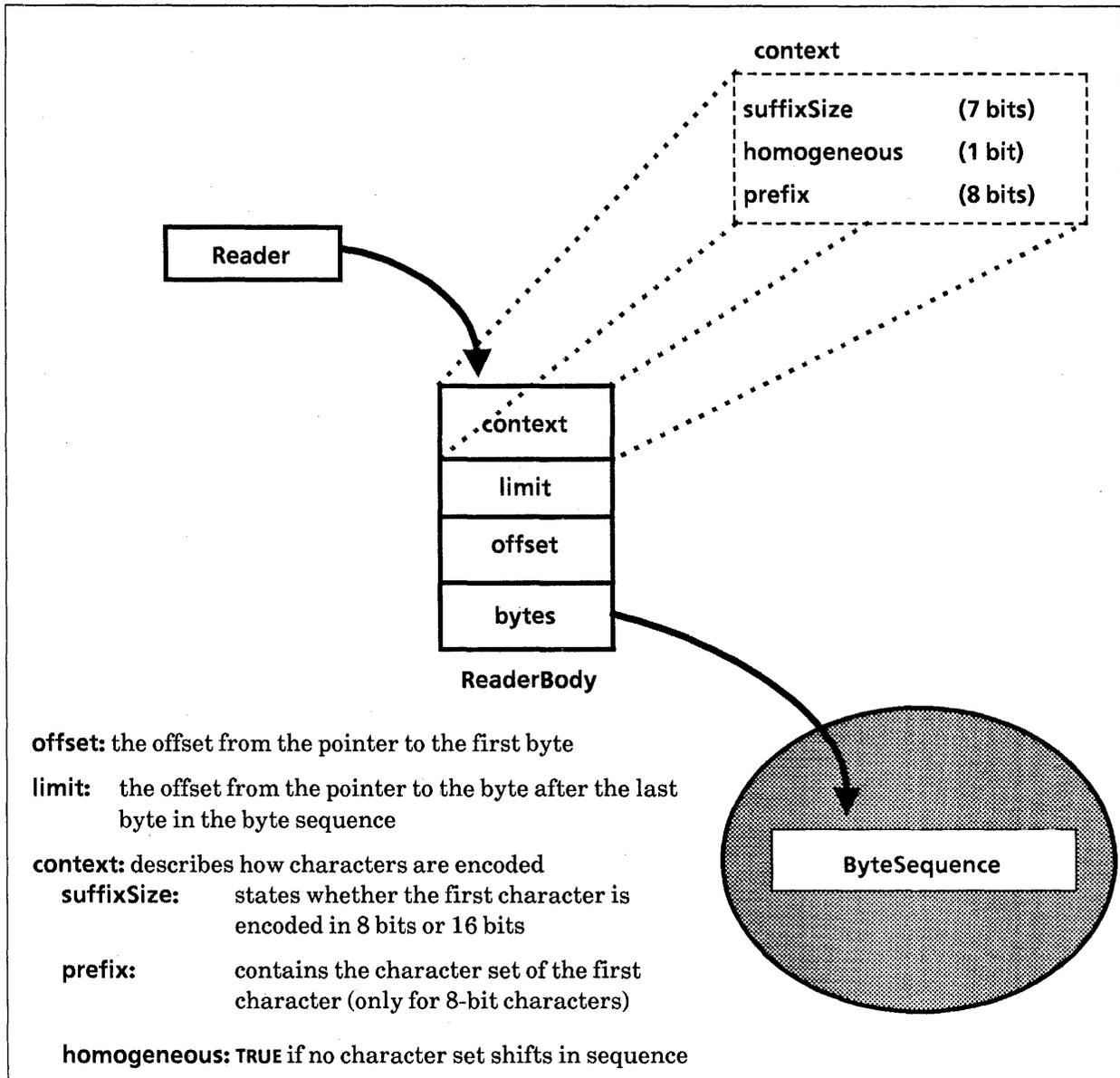


Figure 3.2: Reader and ReaderBody

Examples of character encodings and character set shifts are beyond the scope of this course; if you are interested, consult the **XChar** chapter of the *ViewPoint Programmer's Manual*, the *Xerox Character Code Standard*, or the *Xerox NetworkSystems Architecture General Information Manual*.

3.2.1.1 Accessing the contents of a reader

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

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

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

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

3.2.1.2 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];
```

3.2.1.3 Readers vs. ReaderBodys

When writing procedures and data structures, you must decide when 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** 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**. This is an example of using a reader as a parameter. **GetNewName**, on the other hand, returns a **ReaderBody**. If it returned a **Reader**, there would be a problem with the storage for the **ReaderBody**. 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.

3.2.2 Writers and WriterBodys

A **Writer** is much like a **Reader**, except that it has additional fields to permit editing:

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

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

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

The first four fields are the same as in a reader; the last three fields contain information to support editing. **maxLimit** describes the limits of the allocation unit; **endContext** is the context that describes the encoding of the last character (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 caller's writer variable.

3.2.2.1 Allocating writers

There are several ways to initially create a writer. To allocate a brand new writer, call **xstring.NewWriterBody**:

XString.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 also create writers from existing strings or NSStrings:

XString.WriterBodyFromNSString: PROCEDURE [
s: NSString.String,
homogeneous: BOOLEAN ← FALSE]
RETURNS [XString.WriterBody];

XString.WriterBodyFromSTRING: PROCEDURE [
s: LONG_STRING,
homogeneous: BOOLEAN ← FALSE]
RETURNS [XString.WriterBody];

3.2.2.2 Expanding writers

You can expand a **WriterBody** with **XString.ExpandWriter**:

XString.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.

3.2.2.3 Editing writers

There are a number of procedures that you can use to add information to a writer or to edit a writer. For example, there is a procedure to add a reader to a writer (**XString.AppendReader**), to add a character to a writer (**XString.AppendChar**), to append a mesa string to a writer (**XString.AppendSTRING**), and so on. See the *ViewPoint Programmer's Manual* for the declarations of these procedures.

Another common way to add contents to a writer is with the **XFormat** interface, as described in the next section.

3.3 XFormat

The **XFormat** interface provides facilities for formatting data types into other data types. For example, you could use **XFormat** to convert a series of characters into an **XString**, or a series of cardinals into an **NSString**. Instead of providing a different routine for every possible conversion that you might want to do, however, the **XFormat** interface provides a standardized way of converting one type to another.

The basic idea is to convert the input data to an intermediate format, and then convert from that intermediate format into a specified output format. (The intermediate format is just a

Reader, but you don't need to know that to use the XFormat facilities.) Figure 3.3 illustrates this idea.

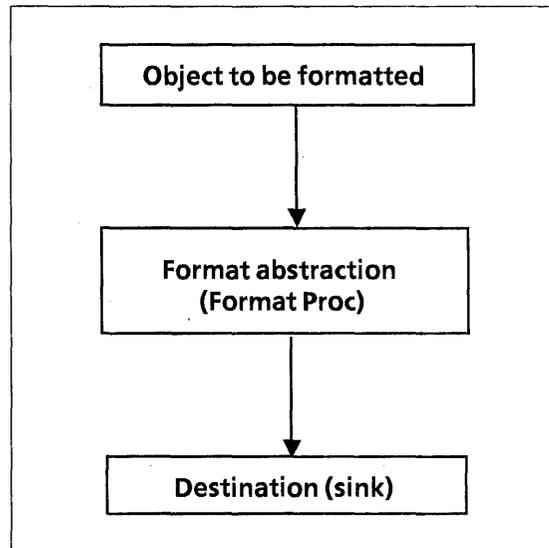


Figure 3.3: The XFormat abstraction

The major 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;
```

A handle is a pointer to an **Object**; the principle field of an **Object** is the format procedure, **proc**. The format procedure is responsible for converting from the intermediate format (reader) to the output format. It takes a reader and a handle as parameters, and it should pass its reader parameter to its *output sink*. Obviously, there must be a different format procedure for every type of output; a format procedure that produces streams is not the same as a format procedure that produces writers.

The **context** field of an **Object** contains context information on the last character sent to the format procedure; the format procedure is responsible for updating this information.

To use the XFormat facilities, the first step is to create an **Object** that has a format procedure that implements the output sink you are interested in. For example, you might want to use a writer or a stream as your output sink. You can then pass in various data types that you want to add to the writer, as illustrated in Figure 3.4. Note that you can either pass several items individually to several different writers, or you can concatenate them into one writer.

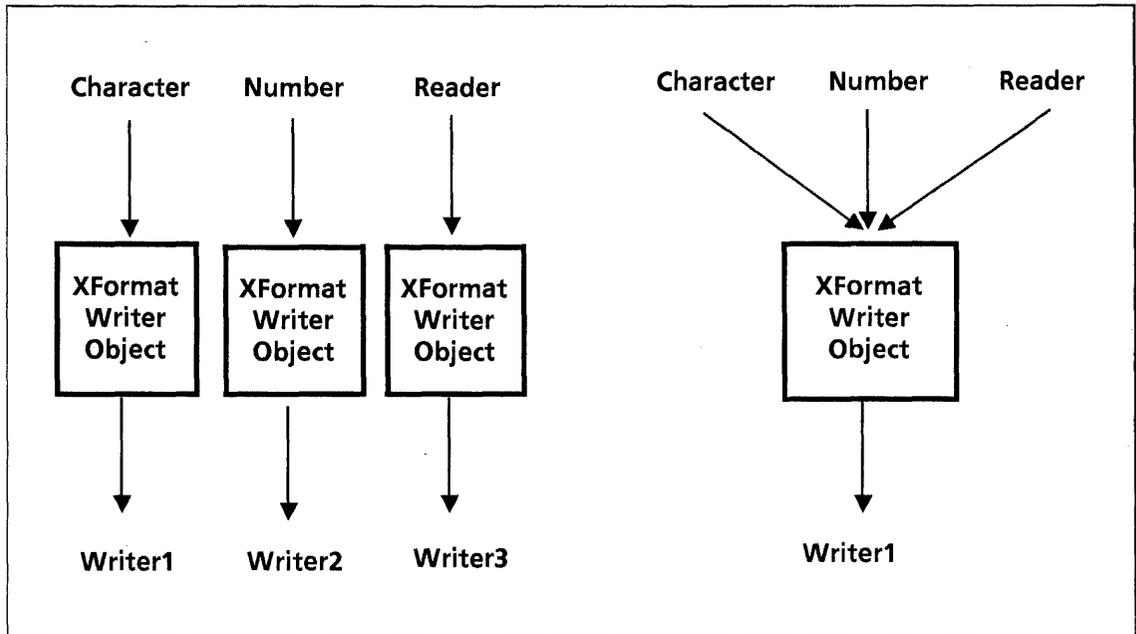


Figure 3.4: Using a writer as output sink

The `XFormat` interface provides format procedures for four common data types: `xstring.Writer`, `stream.Handle`, `TTY.Handle`, and `nsstring.String`. Thus, if you want to use any of these four data types as your output sink, then you don't have to write the procedure that converts from a reader to your specified output format. If you want to use an output sink other than these four, then you do need to write the procedure yourself.

For example, suppose that you want to use a stream as your output sink. (For now, don't worry about it if you don't know what a stream is; we discuss streams at length in Chapter 12, *Streams*. For now, you just need to get a basic idea of how `XFormat` works.) You can make a call to the procedure `xFormat.StreamObject`, which will create an `Object` with the correct format procedure, and with the stream as its data parameter.

The format procedure itself is called `xFormat.StreamProc`:

```
xFormat.StreamProc: xFormat.FormatProc;
```

```
xFormat.StreamObject: PROCEDURE [sH: Stream.Handle]
    RETURNS [xFormat.Object];
```

Figure 3.5 illustrates the format object that `StreamObject` returns.

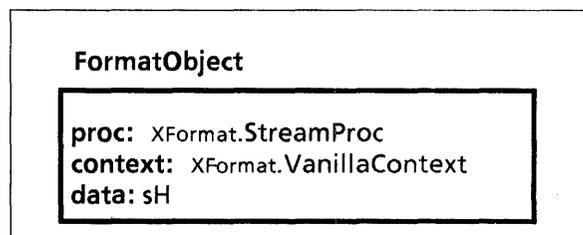


Figure 3.5: An `xFormat.Object` with a stream as output sink

Once you have initialized the format object, you can then pass in the data that you want to format. **XFormat** provides a number of procedures that you can call to pass various data types to a format object. For example:

```
XFormat.Char: PROCEDURE [h: XFormat.Handle ← NIL,  
char: XString.Character];
```

```
XFormat.Decimal: PROEDURE [h: XFormat.Handle ← NIL,  
n: LONG INTEGER];
```

```
XFormat.Reader: PROEDURE [h: XFormat.Handle ← NIL,  
r: XString.Reader];
```

These procedures all take two parameters, a piece of data of the specified type, and a **Handle**. They take the piece of data, format it into a reader, and then pass it to the format procedure in the format object, which formats it into the desired output format.

Here is an example that concatenates several different data types into a writer:

```
--Statement1: create a new writer  
writerBody: XString.WriterBody ←  
XString.NewWriterBody[maxLength:250, z: sysZ];  
--Statement2: Create an object with writer format object  
xfo: XFormat.Object ← XFormat.WriterObject[  
w: @writerBody];  
--Statement3: Concatenate data types into writer  
XFormat.String[h:@xfo, s:"My name is "L];  
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];
```

This example first creates a new writer, and then calls **WriterObject** to create an object initialized with the format procedure **WriterProc** and data **@writerBody**. This sets up the writer as the output sink, as illustrated in the innermost box of Figure 3.6.

The next step is to call **String**, **Decimal**, and **Char** to put the various data types to the writer. Each of these procedures takes a piece of data, puts it in the intermediate format, and then calls **h.proc**, passing in the data. Thus, **String** calls **h.proc**, (which is the writer format procedure **WriterProc**), passing in the string "My name is,", and **WriterProc** puts the bytes of the string to the writer. Note that the argument to **Char** is just a period; the statement above takes the ordinal value of the period character.

This code will create a writer whose contents are something like this: "My name is Lucille and my age is 11."

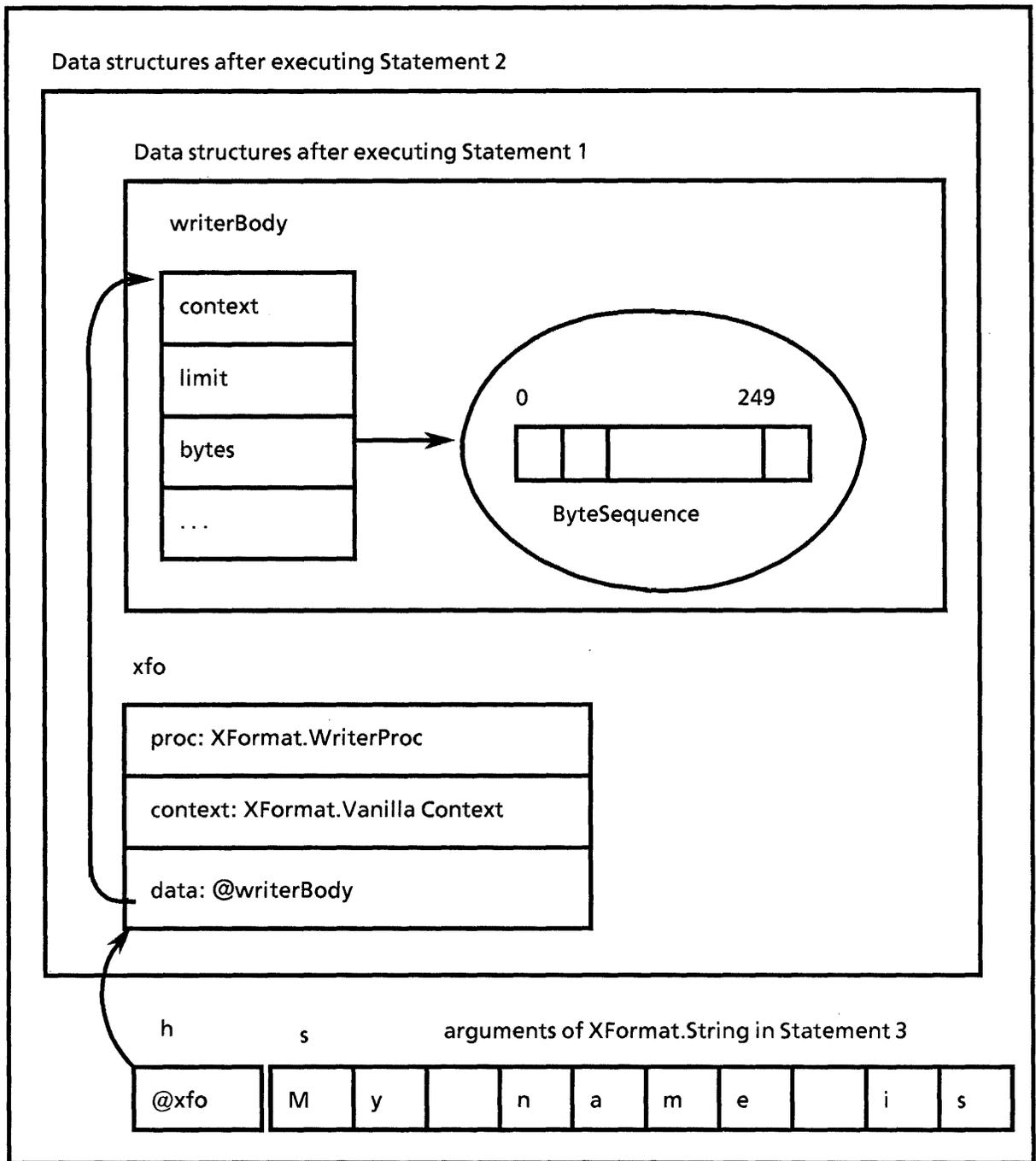


Figure 3.6: Diagram of XFormat example

For the purposes of this course, you do not need to see the actual code for a format procedure. If you want to write your own format procedure for a data type other than the four that XFormat supports, see the XFormat chapter of the *ViewPoint Programmer's Manual*.

3.4 XMessage

The idea behind the XMessage interface is to group all messages that the user sees (generally speaking, all the readers

in a program) into a single module. Eventually, when you are through with your application, you can use *Message Tools* to remove the messages from the code altogether. The advantage of this approach is that you can change the messages without changing the code; this is particularly important when dealing with multinational applications.

During development, however, while things are still changing, you should use the method described here. Chapter 16, *Application Folders* and Appendix C, *Message Tools* describe how to handle messages for a completed application.

The messages mechanism used during development uses the standard three-module structure: a definitions module, an implementation module, and one or more client modules.

3.4.1 The definitions module

The definitions module defines the messages for the application and a procedure for clients to call when they need to access the messages. (A message definitions module does not have to be distinct from other interfaces; you can just add the messages definitions to another interface, if you like.)

First, you need to define a type **Key** that includes a name for each of the messages. Thus, for example:

```
Key: TYPE = {hiMom, elephant, noFile, badInput};
```

You also need a procedure that clients can call to access the messages. For example:

```
Get: PROC [key: Key] RETURNS [XString.ReaderBody];
```

The job of this procedure is to return the actual message corresponding to the specified key.

3.4.2 The implementation module

The second piece is the implementation module that supplies the actual messages. In the implementation module, you need to define the actual text for each message, and implement the **Get** procedure.

The first step is to call **XMessage.AllocateMessages**, which defines the domain of messages for your application:

```
XMessage.AllocateMessages: PROCEDURE [  
  applicationName: LONG STRING,  
  maxMsgIndex: CARDINAL,  
  clientData: XMessage.ClientData,  
  proc: XMessage.DestroyMsgsProc]  
  RETURNS [h: XMessage.Handle];
```

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

```
XMessage.Object: TYPE;
```

applicationName is the name of the application, **maxMsgIndex** is the number of messages, **clientData** is for your own use. **DeleteMessages** is a call back procedure to deallocate any storage associated with the message handle.

AllocateMessages returns a message handle for the application; here is an example of calling this procedure:

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

To implement the messages, you need to create an object of type **XMessage.Messages**:

```
XMessage.Messages: TYPE =
  LONG DESCRIPTOR FOR ARRAY OF XMessage.MsgEntry;
```

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

```
XMessage.MsgKey: TYPE = CARDINAL;
```

msgKey is the ordinal value of the key defined in the interface (for example, **khiMom**), and **msg** is the actual text for the message. The other fields are for the purposes of translation. The only one that you must supply is **id**, which is a unique identifier. This **id** is for the use of the translators and should not change; note that the **id** and the **msgKey** for a message don't have to be the same.

For example, here is a fragment that illustrates how to implement the **hiMom** key:

```
msgArray: ARRAY Defs.Key OF XMessage.MsgEntry ← [
  hiMom: [
    msgKey: Defs.Key.hiMom.ORD,
    msg: XString.FromString["Hi, Mom!"],
    id: 1].
  ...];
```

The final step is to call **XMessage.RegisterMessages** to actually initialize your messages:

```
XMessage.RegisterMessages: PROCEDURE [
  h: XMessage.Handle,
  messages: XMessage.Messages,
  stringBodiesAreReal: BOOLEAN];
```

This procedure takes a message handle and the messages for the application, and initializes the messages. If **stringBodiesAreReal** is **FALSE**, then **RegisterMessages** will copy

the bytes for the string; if it is TRUE, then RegisterMessages assumes that the bytes will remain valid.

Once you have made this call, clients can access your messages by calling Get; to implement Get, just call XMessage.Get:

```
XMessage.Get: PROCEDURE [
  h: XMessage.Handle, msgKey: XMessage.MsgKey]
  RETURNS [msg: XString.ReaderBody];
```

For example:

```
Get: PUBLIC PROCEDURE [key: Defs.Key] RETURNS [XS.ReaderBody] =
  RETURN h.Get[key.ORD];
```

Here is a complete implementation module:

```
DIRECTORY Defs, XMessage, XString;
```

```
MsgImpl: PROGRAM IMPORTS XMessage, XString EXPORTS Defs = {
  OPEN XS: XString;
  h: XMessage.Handle ← NIL;    -- The messages handle
```

```
  Get: PUBLIC PROC [key: Defs.Key] RETURNS [XS.ReaderBody] =
    RETURN h.Get[key.ORD];
```

```
  Init: PROC = { -- Creates, allocates, and registers messages
```

```
    msgArray: ARRAY Defs.Key OF XMessage.MsgEntry ←
      [hiMom: [
        msgKey: Defs.Key.hiMom.ORD,
        msg: XS.FromString["Hi, Mom!"],
        id: 1],
      elephant: [
        msgKey: Defs.Key.elephant.ORD,
        msg: XString.FromSTRING ["Elephants are pink."],
        id: 2],
      noFile: [
        msgKey: Defs.Key.noFile.ORD,
        msg: XString.FromSTRING ["Error...file not found"],
        id: 3],
      badInput: [
        msgKey: Defs.Key.badInput.ORD,
        msg: XString.FromSTRING ["Invalid input."],
        id: 4]];
```

```
    messages: XMessage.Messages ← DESCRIPTOR [LOOPHOLE [
      msgArray, ARRAY[0..Defs.Key.LAST.ORD] OF
      XMessage.MsgEntry]];
```

```
  h ← XMessage.AllocateMessages [
    applicationName: "TestApplication",
    maxMessages: Defs.Key.LAST.ORD + 1,
    clientData: NIL,
    proc: NIL];
```

```
  XMessage.RegisterMessages [
    h: h,
    messages: messages,
    stringBodiesAreReal: FALSE];
```

```
--Mainline code
  Init [];}...
```

3.4.3 The client module

From the client side, things are much simpler: you just call `Defs.Get` to retrieve a particular message. For example:

Typical message usage

```
noFile: XString.ReaderBody ← Defs.Get [Defs.Key.noFile];
```

```
.
```

```
Attention.Post [@noFile];. --Discussed in the next section
```

One final note on `XMessage`: the method we present here is slightly different (and simpler) than that suggested in the *ViewPoint Programmer's Manual*. Unfortunately, many of our examples either don't use messages or use the other method. Do as we say, not as we do.

3.5 Attention

The `Attention` interface implements a single window for displaying messages. The `Attention Window` also has an associated menu; Chapter 4, *Simple Application*, describes this menu.

There are three types of messages that you can put in the `Attention Window`: *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 the user to confirm something.

There are three posting operations: `Post`, `PostSticky`, and `PostAndConfirm`.

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

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

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

The `Post` procedures display the message `s` in the `Attention` window. If `clear` is `TRUE`, the procedure 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 the user to confirm. See the *ViewPoint Programmer's Manual* for details on how to use `PostSticky` and `PostAndConfirm`. There are also the inverse operations:

Attention.Clear: PROCEDURE;

Attention.ClearSticky: PROCEDURE;

Clear clears the Attention window of any simple message. If there is a current sticky message, **Attention** will display it after clearing the simple message. **Clear** has no effect if the current message is sticky. **ClearSticky** clears any current sticky message. **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, to avoid interference, you should follow this guideline: only call procedures in the **Attention** interface from the *Notifier* process. Chapter 9, *TIP*, discusses the Notifier process in detail; for now, you should just realize that there is a potential conflict.

3.6 Summary

The **XChar** interface provides the definition of a character. ViewPoint characters are encoded with the *Xerox Character Code Standard*, which provides increased generality at the price of slightly increased complexity.

The **XString** interface defines the data structures and operations for strings. **XString** defines two different kinds of strings, *readers* (read-only strings) and *writers* (editable strings.) **XString** provides a wide variety of procedures for manipulating both readers and writers.

The **XFormat** interface provides procedures to format various data types into readers, and vice versa.

The **XMessage** interface provides facilities for keeping messages that the user will see (readers) separate from the actual code for the application. To use the **XMessage** approach, you need three modules: a definitions module, a client module, and an implementation.

The **Attention** interface allows you to post messages to the global attention window. The chief procedures are **Post** and **Clear**.

3.7 Exercise

The exercise for this chapter is the Concordance Tool, which determines the number of times that a particular pattern occurs in a textual selection. To use this tool, you fill in a pattern and a selection, specify a context, and then invoke the Find All command. (The *context* is the number of words on each side of the pattern; note that the pattern cannot include wildcard characters.) The tool then provides a list of matches in the specified context, and a count of the total number of matches. Figure 3.7 illustrates this tool.

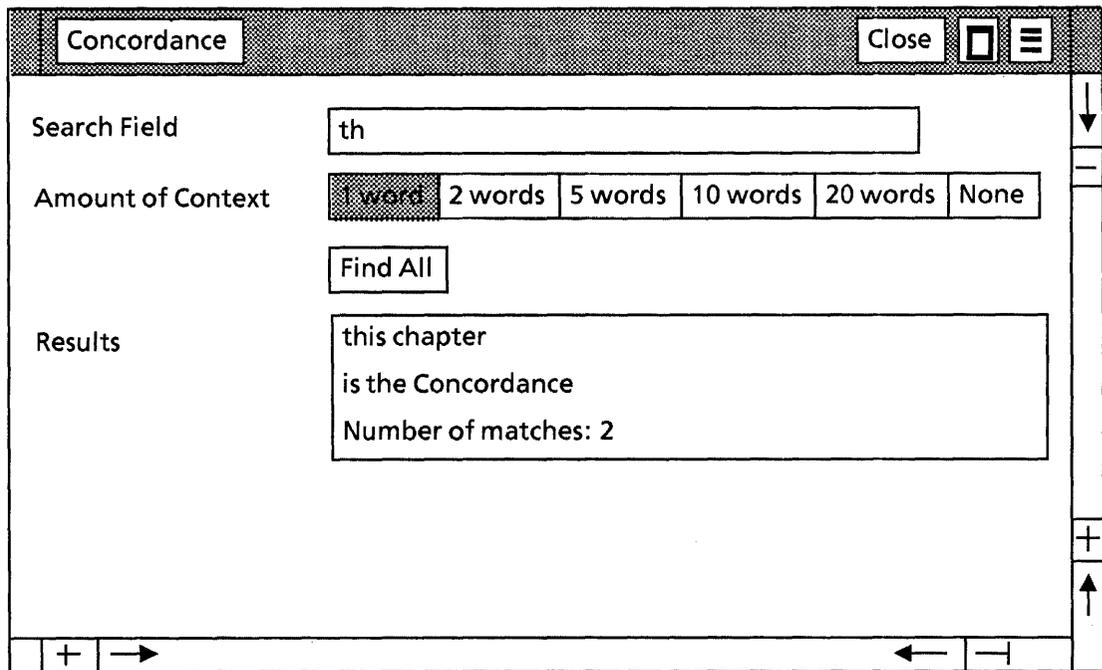


Figure 3.7 The ConcordanceTool

Your assignment is to complete the module `ConcordFormImplTemplate.mesa`. Specifically, you need to write the two procedures `Find`, and `ProcessString`. `Find` is the `FormWindow.CommandProc` that implements Find All: it first calls a procedure to get the current selection, and then calls `ProcessString`, which searches the text for pattern matches and extracts concordances for each match. `Find` then displays any matches and also frees any storage.

For a complete explanation of what you need to do, see the comments in `ConcordFormImplTemplate.mesa`. You will also need the following modules:

```
ConcordDefs.bcd
ConcordMsgImpl.bcd
ConcordImpl.bcd
ConcordSelectionImpl.bcd
Concord.config
```

4. CREATING A SIMPLE APPLICATION

This chapter discusses how to create a basic user interface for an application: how to add a command to the Attention Menu, how to create a window, and how to display the window on the screen. The information in this chapter is just a skeleton; the next four chapters discuss how to add additional functionality and features to your user interface.

4.1 Adding a command to the Attention menu

You can structure a Viewpoint application to run either from a command in the Attention Menu or from an icon. Chapter 14, *Icon Applications*, discusses how to write applications that use icons; this chapter describes how to write an application that runs from a command in the Attention Menu.

To have your application run from the Attention Menu, you need to add a command to the menu and supply a procedure to implement that command. To add a command to the Attention Menu, call `Attention.AddItem`:

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

```
MenuData.ItemHandle: TYPE = LONG POINTER TO MenuData.Item;
```

```
MenuData.Item: TYPE = MenuData.PrivateItem; --hidden
```

`AddMenuItem` requires a parameter of type `ItemHandle`, which represents a menu item. To get an `ItemHandle`, call `MenuData.CreateItem`:

```
MenuData.CreateItem: PROCEDURE [  
    zone: UNCOUNTED_ZONE,  
    name: XString.Reader,  
    proc: MenuData.MenuProc,  
    itemData: LONG_UNSPECIFIED ← 0]  
    RETURNS [MenuData.Itemhandle];
```

`CreateItem` builds an item record in `zone`. `name` is the string that will appear in the menu, and `proc` is a *call back procedure* that will be called when the user invokes the command from the Attention Menu.

A *call back procedure* is a procedure that is passed in as a parameter to another procedure, and is later called by that procedure. Thus, you write a `MenuProc` but don't call it; you just pass it to `CreateItem`, and the `MenuData` implementation will call it when the user invokes your command.

The `proc` parameter is of type `MenuData.MenuProc`:

```
MenuData.MenuProc: TYPE = PROCEDURE [  
    window: Window.Handle,  
    menu: MenuData.MenuHandle,  
    itemData: LONG UNSPECIFIED];
```

The parameters to this procedure are a *handle* to your application's window, a handle to the menu, and the `itemData` parameter that you passed to `CreateItem`. `window` identifies a particular window on the screen, and `menu` identifies the menu from which the command was invoked.

There is a `MenuProc` associated with every command in a menu, whether the command is in the menu of commands for an application, or in the Attention Menu. For some `MenuProcs`, you will need to use the `window`, `menu`, and `itemData` parameters. The examples in this chapter, however, do not use any of these parameters. (Section 4.4 discusses `MenuProcs` associated with commands in the header of an application.)

`itemData` is referred to as *client data*; it is entirely for your own use. If there is no extra information that you need to have available in `proc`, you can leave `itemData` defaulted in the call to `CreateItem`.

Here is an example of how to add a command to the Attention Menu:

--This procedure gets called from the mainline code

```
Init: PROC = {  
    command: XString.ReaderBody ← Defs.Get[  
        Defs.Key.commandName];  
  
    Attention.AddItem [  
        MenuData.CreateItem [  
            zone: Heap.systemZone,  
            name: @command,  
            proc: SampleMenuProc]] };
```

This procedure will add the command *Sample Tool* to the Attention Menu. When the user selects this command, `ViewPoint` will call the procedure `SampleMenuProc`. `SampleMenuProc` is then responsible for putting the application's window on the screen, as described in the next section.

4.2 Creating a StarWindowShell

When the user invokes your command from the Attention Menu, `ViewPoint` will call your `MenuProc`. Typically, the first thing you want to do is create a window on the screen.

`ViewPoint` implements windows with something called a *StarWindowShell*. A `StarWindowShell` is a basic window that can have a title, commands, pop-up menus, and scrollbars (horizontal or vertical), as illustrated in Figure 4.1

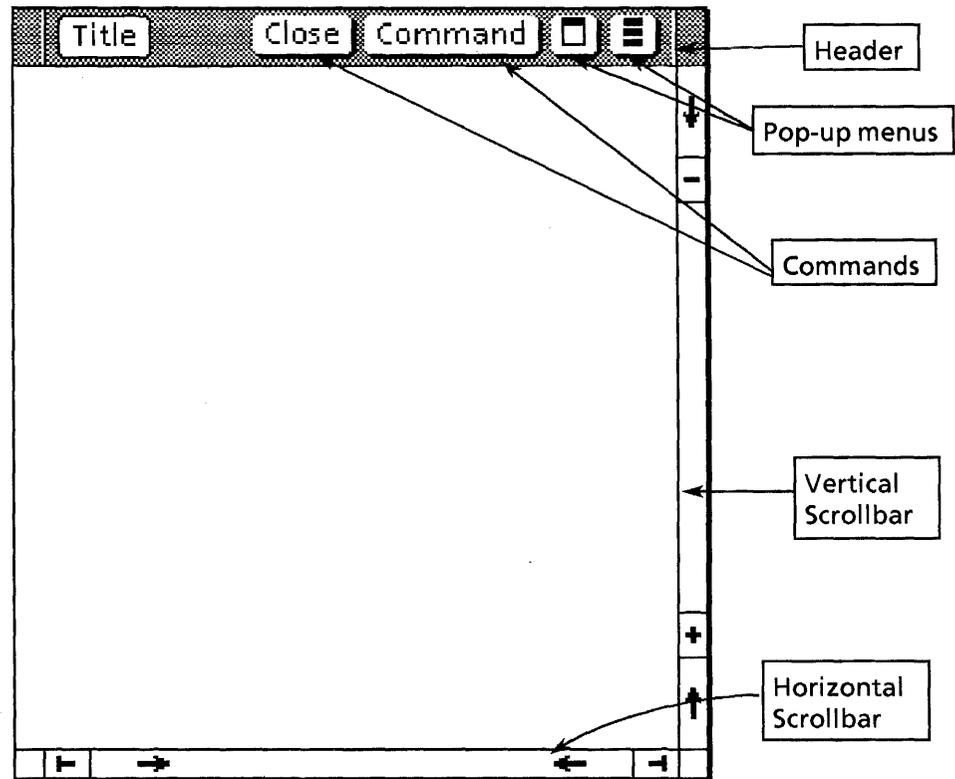


Figure 4.1 A StarWindowShell

To create a shell, call `StarWindowShell.Create`:

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

`StarWindowShell.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`.) We discuss `transitionProc` and `isCloseLegalProc` below; for information on the other parameters, consult the `StarWindowShell` chapter of the *ViewPoint Programmer's Manual*.

`Create` returns a `StarWindowShell.Handle`, which is essentially a `Window.Handle`. As mentioned earlier, a handle identifies your particular window; you will need to pass this handle to various procedures.

Thus, the first part of your `MenuProc` will look like this:

```
SampleMenuProc: MenuData.MenuProc = {
  toolName: XString.ReaderBody ← Defs.Get[
    Defs.Key.toolName];

  shell: StarWindowShell.Handle ← StarWindowShell.Create[
    name: @toolName];
  ...};
```

4.2.1 Transition procedures

A `transitionProc` for a shell is a procedure that `ViewPoint` will call when the state of the shell is about to change:

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

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

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 you should free resources associated with the display state. `dead` indicates that the shell is about to be destroyed and therefore you should free all resources associated with the shell. If you have any storage associated with your shell, you should use a `transitionProc` to allocate and free that storage. Here is a "generic" example of a transition procedure:

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

4.2.2 IsCloseLegalProc

An `isCloseLegalProc` allows you to veto an attempt to close your window. An `isCloseLegalProc` is of type `IsCloseLegalProc`:

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

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

ViewPoint will call the `isCloseLegalProc` that you supply when either a user or a client program attempts to close the `StarWindowShell`. If it's okay to close the window, you should return `TRUE`; otherwise, return `FALSE`. (The `isCloseLegalProc` is also a convenient way to get control when the window is being closed.) Here is a simple example that prints a message and ignores the close under certain circumstances:

```
SimpleIsCloseLegalProc: StarWindowShell.IsCloseLegalProc =
BEGIN
  IF --YouDon'tCareIfTheWindowIsClosed-- THEN RETURN [TRUE];
  ELSE { --print a message, and then abort the close request
    abortMsg: Xstring.Reader ← Defs.Get[Defs.Key.abortMsg];
    Attention.Post[@abortMsg];
    RETURN [FALSE]};
END;
```

4.3 Body windows

ViewPoint windows are organized in a tree structure, with the desktop window at the root of the tree. The `StarWindowShell` for an application is generally a child of the desktop window. A `StarWindowShell` is just a shell, however; before the window can do anything useful, you need to put *body windows* within the `StarWindowShell`.

Body windows are what define the functionality of a shell. For example, if you want to display in a window, you display that information to a body window, not to the `StarWindowShell`. You can create various different arrangements of body windows, depending on what you want your application to do.

The simplest arrangement is to have one very long body window that is much longer than the shell. This makes scrolling easy: you simply slide the body window within the window shell. This is how the `StarWindowShell` default scrolling works, so if you use this approach the `StarWindowShell` implementation will take care of all scrolling for you.

However, if you use this approach, the dimensions of the body window won't change when the user changes the size of the shell. Thus, the body window may be much bigger or much smaller than the shell.

An alternate way of handling body windows is to specify that the body window should change size whenever the size of the `StarWindowShell` changes. If you do this, you are responsible for keeping track of what is in the window, and you must perform all scrolling operations yourself.

This chapter discusses only how to create a single body window whose size never changes. If you are interested in creating body windows that change size when the shell does, or if you want to create multiple body windows within a given shell, see the `StarWindowShell` chapter of the *ViewPoint Programmer's Manual*.

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

```
StarWindowShell.CreateBody: PROCEDURE [
  sws: StarWindowShell.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 within the **sws**. The **Window** implementation will call **repaintProc** whenever it needs to redisplay part or all of the information in the body window. Chapter 6, *Displaying information on the screen*, discusses repaint procedures in detail.

bodyNotifyProc is a procedure that is responsible for determining how the window handles user input; Chapter 8, *TIP*, discusses this subject in detail.

box indicates the size and location of the body window within the shell. 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. If you are going to create one long body window, you should use **box** to specify the dimensions that you want the body window to have. For example, here is a code fragment from a **MenuProc**:

```
...  
--dimensions of the body window in pixels  
bodyWindowDims: Window.Dims = [1000, 1000];  
  
-- Create the StarWindowShell.  
shell: StarWindowShell.Handle = StarWindowShell.Create [  
    name: @sampleTool];  
  
-- Create one long body window inside the StarWindowShell  
body: Window.Handle = StarWindowShell.CreateBody [  
    sws: shell,  
    box: [ [0,0], bodyWindowDims ],  
    repaintProc: SomeRedisplayProc, --discussed in chapter 6  
    bodyNotifyProc: SomeNotifyProc];--discussed in chapter 8
```

4.4 Commands

StarWindowShells have commands in the header that the user can invoke. The commands can appear either directly in the header, or in a pop-up menu available from the header, as illustrated in Figure 4.2.

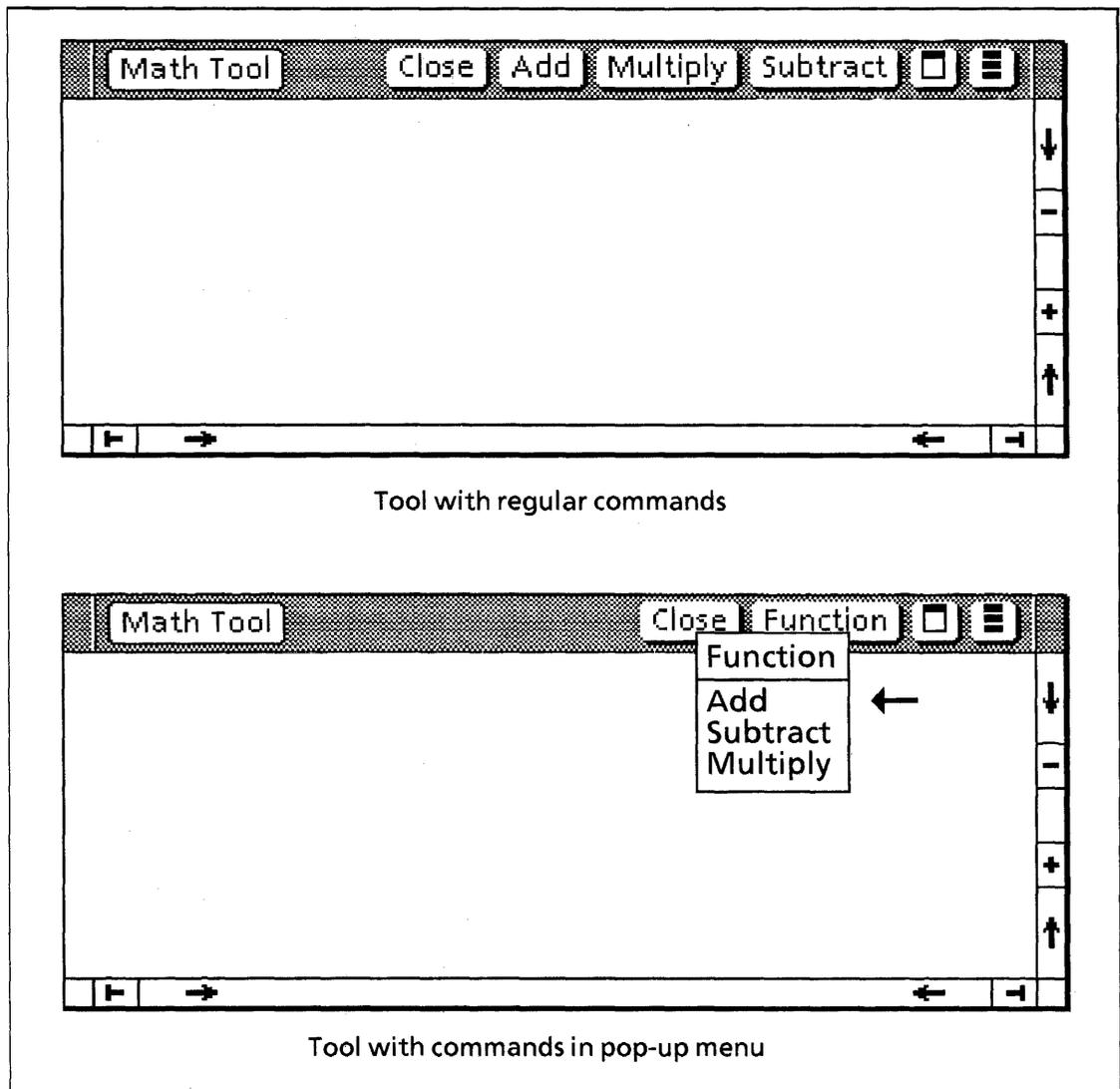


Figure 4.2 Commands

Commands for an application work just like commands in the Attention Menu: when you specify a command, you associate a procedure (of type `MenuData.MenuProc`) with the command. When the user invokes the command, the corresponding procedure is called.

`StarWindowShell` automatically puts the `Close` command in the header of every application, and you can also add additional commands. The following sections provide examples of how to put commands directly in the header and how to put them in a pop-up menu. (Note: if you specify commands in the header and they don't fit, they will automatically overflow into the pop-up menu.)

4.4.1 Putting commands directly in the header

To put commands in the header, you start by calling `MenuData.CreateItem` to create an individual `ItemHandle` for each command. (Remember, this is what we did in the first

section, when we created the `ItemHandle` for the command in the Attention Menu.). Once you have called `CreateItem` for each command that you want to have, you store them in an array, and then call `MenuData.CreateMenu`:

```
MenuData.CreateMenu: PROCEDURE [
  zone: UNCOUNTED_ZONE,
  title: MenuData.ItemHandle,
  array: MenuData.ArrayHandle,
  copyItemsIntoMenusZone: BOOLEAN ← FALSE]
  RETURNS [MenuData.MenuHandle];
```

```
MenuData.ArrayHandle: TYPE = LONG_DESCRIPTOR_FOR_ARRAY_OF
  MenuData.ItemHandle;
```

This procedure returns a handle to a menu; you can then put the "menu" of commands in the header. (Note: the idea that commands in the header are actually a menu may seem a bit strange, but in fact a command in the header is considered to be an individual menu item with a box around it.)

Finally, you call `StarWindowShell.SetRegularCommands` to display the commands in the header:

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

Here is an example of how to create the commands Add, Multiply, and Subtract, and put them directly in the header:

```
-- retrieve zone attached to the StarWindowShell
z: UNCOUNTED_ZONE ← StarWindowShell.GetZone [shell];
--declare ReaderBodys for the commands that you want to have
add: XString.ReaderBody ← XString.FromSTRING ["Add"];
subtract: XString.ReaderBody ← XString.FromSTRING["Subtract"]L];
multiply: XString.ReaderBody ← XString.FromSTRING["Multiply"]L];

--call CreateItem for each command, passing in the ReaderBodys
--and a call back procedure for each command. Store the
-- ItemHandles into an array.
items: ARRAY [0..3] OF MenuData.ItemHandle ← [
  MenuData.CreateItem [zone: z, name: @add, proc: Add],
  MenuData.CreateItem [zone: z, name: @subtract,
  proc: Subtract],
  MenuData.CreateItem [zone: z, name: @multiply,
  proc: Multiply]];

--Create a MenuHandle by calling CreateMenu, passing in
--a descriptor for the array of ItemHandles.
myMenu: MenuData.MenuHandle = MenuData.CreateMenu [
  zone: z, -- Generally use zone attached to SWS
  title: NIL,
  array: DESCRIPTOR [items]];

--specify that the commands should appear as individual
--items in the header.
StarWindowShell.SetRegularCommands[
  sws: shell, commands: myMenu];
```

The first step is to retrieve the heap attached to the shell. When you call `StarWindowShell.Create` to create a `StarWindowShell`, the implementation creates a private heap and uses it as storage

for all shell-related items, such as name strings. It is a good idea to use this zone for your own storage, since it will be deleted when you delete the StarWindowShell.

The next step is to call `MenuData.CreateItem` once for each command, and store the individual `ItemHandles` into an array.

Once you have an array of `ItemHandles`, you can call `CreateMenu`, which puts the array into a menu and returns a `MenuHandle`. The last step is to pass that `MenuHandle` to the procedure `StarWindowShell.SetRegularCommands`.

4.4.2 Putting commands in a popup menu

Putting the commands in a pop-up menu is much like putting them directly in the header, except that you need to include a title for the menu, and you need to call `AddPopupMenu` instead of `SetRegular Commands`:

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

Here is an example of how to put commands in a pop-up menu:

```
-- retrieve zone attached to the StarWindowShell
z: UNCOUNTED ZONE ← StarWindowShell.GetZone [shell];
--declare ReaderBodys for the commands and the title
add: XString.ReaderBody ← XString.FromSTRING ["Add"];
subtract: XString.ReaderBody ← XString.FromSTRING ["Subtract"];
multiply: XString.ReaderBody ← XString.FromSTRING ["Multiply"];
title: XString.ReaderBody ← XString.FromSTRING ["Function"];

--call CreateItem for each command, passing in the ReaderBodys
-- and a call back procedure for each command.Store the
ItemHandles into an array.
items: ARRAY [0..3] OF MenuData.ItemHandle ← [
  MenuData.CreateItem[zone: z, name: @add,
    proc: Add],
  MenuData.CreateItem [zone: z, name: @subtract,
    proc: Subtract],
  MenuData.CreateItem [zone: z, name: @multiply,
    proc: Multiply]];

--Create a menu item for the title (but don't store it in array.)
-- Notice that there is no proc; when the user invokes this
--command, MenuData will display the menu.
titleItem: MenuData.ItemHandle ← MenuData.CreateItem [
  zone: z, name: @title, proc: NIL];

--Create a MenuHandle by calling CreateMenu, passing in
--a descriptor for the array of ItemHandles.
--Notice that title is not NIL.
myMenu: MenuData.MenuHandle = MenuData.CreateMenu [
  zone: z, -- Generally use zone attached to SWS
  title: titleItem,
  array: DESCRIPTOR [items]];

--specify that commands should appear in menu.
StarWindowShell.AddPopupMenu[
  sws: shell, commands: myMenu];
```

4.5 Displaying windows on the screen

Once you have created a window shell, body window, and associated commands, you still need to display the shell on the screen. (Note that `Create` generates a `StarWindowShell` but does not display it on the screen.) To display a shell on the screen, call `StarWindowShell.Push`, which inserts the new window into the existing tree structure:

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

`Push` displays `newShell` by inserting it into the visible window tree. If `poppedProc` is `NIL`, popping `newShell` will destroy the shell. You can write your own `PoppedProc` if you want to do something other than destroy the shell; see the *ViewPoint Programmer's Manual* for details.

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!`.

4.6 Summary

To create a bare-bones user interface for a new application, you need to do the following:

1. In your initialization code, add a command to the Attention Menu with the following steps:
 - A. Call `MenuData.CreateItem` to create an `ItemHandle`. The most important parameters to this procedure are the name of the command and the call back procedure that `ViewPoint` will call when the command is invoked.
 - B. Call `Attention.AddMenuItem` to add your new command to the Attention Menu.
2. In your `MenuProc`, do the following:
 - A. Call `StarWindowShell.Create`, which creates a window shell. You can optionally associate a *TransitionProc* and an *IsCloseLegal* procedure with your shell.
 - B. Call `StarWindowShell.CreateBody` to create one or more body windows within the `StarWindowShell`. There are many different possible arrangements of body windows; this chapter discussed only the simplest case.
 - C. Add commands to the shell with the following steps:
 1. Call `MenuData.CreateItem` to get an `ItemHandle` for each command. Store the `ItemHandles` into an array.
 2. Call `MenuData.CreateMenu` to create a menu of commands. If you want the commands to appear in a

popup menu, you need to include a title in the call to `CreateMenu`; if you want the commands to appear directly in the header, you don't need a title.

3. Call either `StarWindowShell.SetRegularCommands` or `StarWindowShell.AddPopupMenu`, depending on whether you want your commands to appear directly in the header or in a popup menu.

- D. Display the window on the screen with a call to `StarWindowShell.Push`.

Here is a complete example:

<< This is a sample tool that you can use as a template for creating a new user interface. We will add to this program in the next few chapters.

This tool adds the command Sample Tool to the attention window menu. When the user invokes this command, the MenuProc creates a StarWindowShell with a single body window in it. The commands Post and Redisplay are placed in the header of the StarWindowShell.>>

DIRECTORY

```
Attention USING [AddMenuItem, Post],
Heap USING [systemZone],
MenuData USING [CreateItem, CreateMenu, ItemHandle,
MenuHandle, MenuProc],
StarWindowShell USING [Create, CreateBody, GetZone,
Handle, Push, SetRegularCommands],
Window USING [Dims, Handle],
XString USING [FromSTRING, ReaderBody, WriterBody];
```

SampleTool: PROGRAM

```
IMPORTS Attention, MenuData, StarWindowShell, XString =
BEGIN
```

```
-- Constant; used for size of body window
bodyWindowDims: Window.Dims = [1000, 1000];
```

```
--Procedures
```

<< This procedure is called from the mainline code. It registers the command Sample Tool in the Attention Menu. When the user invokes this command, MenuProc will be called.>>

```
Init: PROC = {
  command: XString.ReaderBody ←
    Defs.Get[Defs.Key.commandName];
  Attention.AddMenuItem [
    MenuData.CreateItem [
      zone: Heap.systemZone,
      name: @commandName,
      proc: MenuProc ] ];
};
```

<< Called when the user invokes the Sample Tool command. Create a StarWindowShell, a single body window within that shell, and some commands, and then display the shell on the screen. >>

```

MenuProc: MenuData.MenuProc = {
  redisplay: XString.ReaderBody ←
    XString.FromSTRING["Redisplay"L];
  post: XString.ReaderBody ← XString.FromSTRING["Post"L];
  sampleTool: XString.ReaderBody ←
    XString.FromSTRING["Sample Tool"L];

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

  -- Create one long body window inside the shell.
  body: Window.Handle = StarWindowShell.CreateBody [
    sws: shell,
    box: [ [0,0], bodyWindowDims ],
    repaintProc: SomeRedisplayProc, --chapter 6, Display
    bodyNotifyProc: SomeNotifyProc];--chapter 8, TIP

  --Get the zone attached to the window shell and use it
  --to create the menu items.
  z: UNCOUNTED_ZONE ← StarWindowShell.GetZone [shell];

  --Create an array of menu items
  items: ARRAY [0..2] OF MenuData.ItemHandle ← [
    MenuData.CreateItem [zone: z, name: @redisplay,
      proc: RedisplayMenuProc],
    MenuData.CreateItem [zone: z, name: @post, proc: Post]];

  --Create the menu. If you want to put commands directly in
  --the header, title can be NIL; if you want to put them in
  --a popup menu, you need to include a title.
  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];

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

--Procedures to implement the commands in the header.
Post: MenuData.MenuProc = {
  msg: XString.ReaderBody ←
    Defs.Get[Defs.Key.sampleMessage];
  Attention.Post [@msg];

RedisplayMenuProc: MenuData.MenuProc = {
  --chapter 6 discusses how to display on the screen};

-- Mainline code
Init[];
END...

```

4.4 Exercise

The exercise for this chapter is to write the user interface for a simple tool, called DMT. This tool takes a string and displays it in random places in the tool's window, much like the DMT program that runs in XDE. This tool registers the command DMT Tool in the Attention Menu. Figure 4.3 illustrates the window that will appear when the user invokes this command.

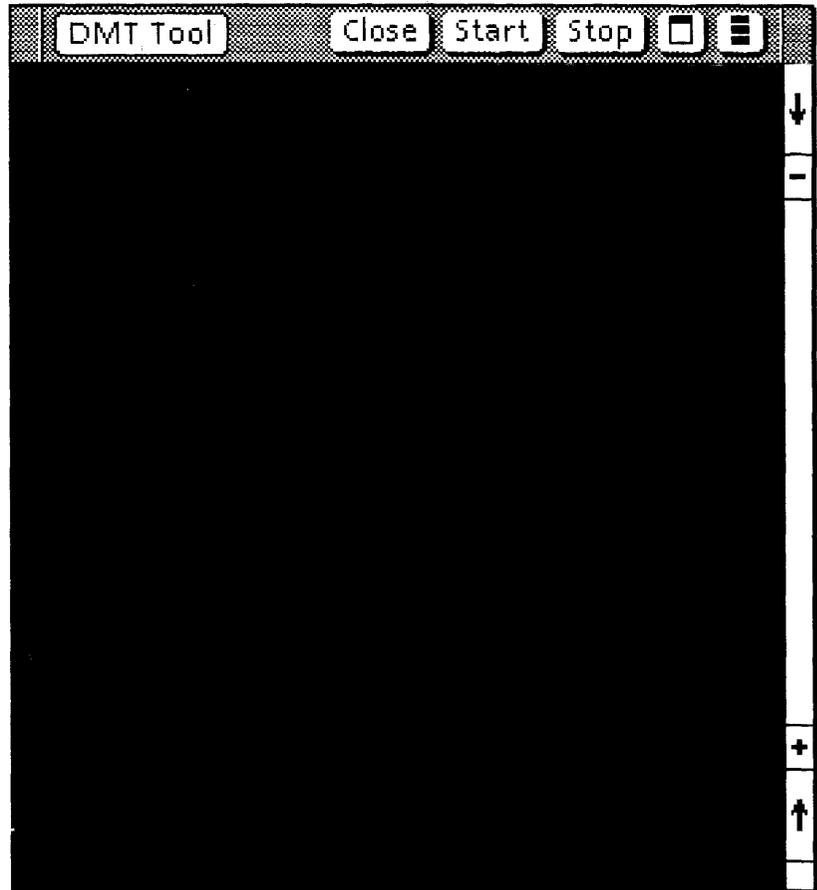


Figure 4.3 The DMT Tool

Invoking the Start command will take the current selection, if it is a string, and display it in the window at random locations. If the current selection cannot be recognized as a string, the default string will be My DMT. (Note that the selected string must be in a simple document, not a standard document.)

Invoking Stop will stop the DMT. You can only have one DMT at a given time. Invoking Close will destroy the window.

Your job is to write the following procedures:

Init This procedure registers the DMT Tool command in the AttentionMenu.

MenuProc This procedure is called when the user invokes DMT Tool command. This procedure

should create the shell and display it on the screen.

IsCloseLegalProc This is the **IsCloseLegalProc** parameter to **StarWindowShell.Create**.

Redisplay This is the **repaintProc** parameter to **StarWindowShell.CreateBody**.

Start This is the command procedure for the Start command.

Stop This is the command procedure for the Stop command.

We provide a template, called **DMTExercise.mesa**, that contains detailed instructions for each of these procedures. To run the program, you will also need the files **DMTDefs.bcd**, **DMTImpl.bcd**, and **DMTConfig.mesa**.

In general, you should structure applications so that the user can run more than one copy ("instance") simultaneously. This doesn't typically involve much restructuring, but it does create a problem with storage for global data.

All copies of an application share the same global frame, so you can't use the global frame to store data that will be different for each instance of the application. Examples of such window-relative data include the contents of the window, a handle to the StarWindowShell itself, the state of some options associated with the window, and so on.

ViewPoint solves this problem with the notion of a *context*. A context is a data object associated with a window instance; contexts allow you to store global state information with a *window* rather than in the program's global frame. This approach has the additional advantage that it helps minimize global frame size, which is an important consideration.

5.1 Context types

To use contexts, the first step is to get a *context type* for your application. Every client of the **Context** interface has a unique type; this is how the client identifies itself in later calls to the **Context** interface. Figure 5.1 illustrates this idea; notice that each window has the same context type but distinct data.

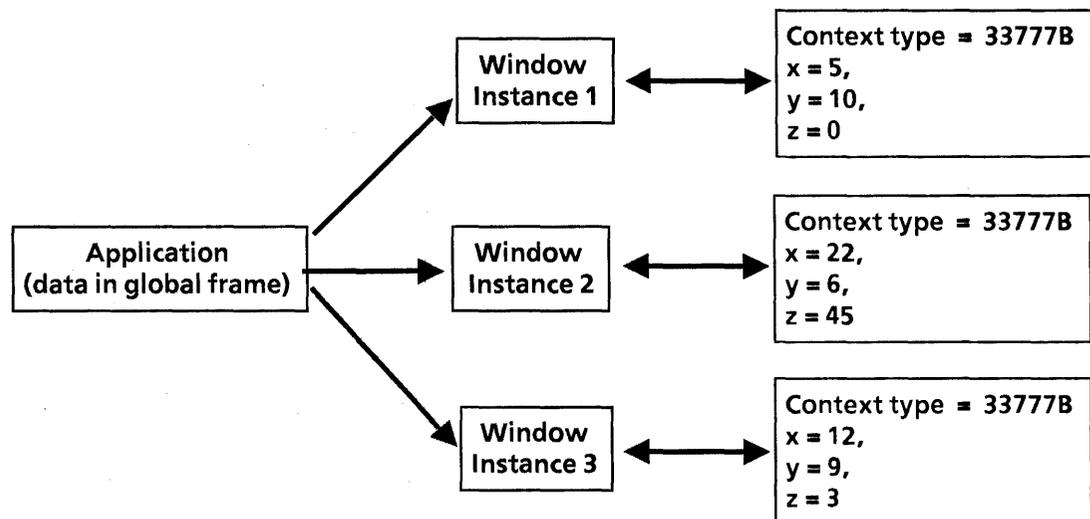


Figure 5.1: Contexts

To get a context, call `Context.UniqueType`:

```
--Note that all instances share one context type
context: Context.Type ← Context.UniqueType[];
```

This call returns a context type. You don't need to know anything about the actual number that the type represents; you just need to use it in future calls to the `Context` interface.

You only need to make this call once for each time the application is run, not once for each instance of the window, since every instance of a given application uses the same context type. Thus, you should store your context type in the shared global frame.

5.2 Creating the context

The next step is to allocate the data that you want to keep in your context. You need to allocate a context each time that you initialize an instance of the window with which you are going to associate the context; this will typically be in your `MenuProc`. You can associate the context either with the shell or with a particular body window.

To allocate a context, call `Context.Create`:

```
Context.Create: PROCEDURE = [
  type: Context.Type,
  data: Context.Data,
  proc: Context.DestroyProcType,
  window: Window.Handle];
```

```
Context.Data: TYPE = LONG POINTER TO UNSPECIFIED;
```

```
Context.DestroyProcType: TYPE = PROCEDURE [
  data: Context.Data,
  window: Window.Handle];
```

`type` is your unique context type, `data` is the data that you want to keep in the context, `proc` is a procedure that the system will call to free the context when it is about to destroy the window, and `window` is the body window with which you want to associate the context.

Notice that `Data` is a pointer to an unspecified data structure; this means that it can be a pointer to any data structure you declare. Obviously, there is no general structure for this data; every application will be different. You should declare a type that represents the data you want to keep in your context. For example, the data for a game might look like this:

```
--global type declaration
DataObject: TYPE = RECORD [
  currentUser: XString.ReaderBody ← XString.nullReaderBody,
  level: {beginner, intermediate, advanced} ← beginner,
  score: CARDINAL ← 0];
```

The actual call to `Create` would then look like this:

```
Context.Create [
  type: context, --your context type
  data: sysZ.NEW[DataObject]; --data to be stored in context
  proc: DestroyContext,
  window: body]; --body window for context
```

The call to **Create** allocates a **DataObject**, associates it with **type**, and stores it with **window**.

DestroyContext is a procedure that is responsible for deallocating the storage associated with the context. When it is about to destroy a window with an associated context (typically in response to a **Close** command), **ViewPoint** will call **Context.Destroy**, which will in turn call your **DestroyProc**. (If you like, you can call **Destroy** yourself, but this is rare. You never call your **DestroyContext** procedure directly.)

Your **DestroyContext** procedure should deallocate the context data. The **Context** interface provides two procedures of this type:

```
Context.NopDestroyProc: Context.DestroyProcType;
```

```
Context.SimpleDestroyProc: Context.DestroyProcType;
```

NopDestroyProc does nothing; **SimpleDestroyProc** just deallocates **data** from the system heap. Thus, if you allocate your context from the **systemZone**, you can use **SimpleDestroyProc** instead of writing your own **DestroyProc**. If you use a private heap to allocate your data, or if you have additional deallocation that you need to do, then you should write your own **DestroyProc**. Here is a simple example of a **DestroyProc**:

```
DestroyContext: Context.DestroyProcType = {
  z.FREE[@data]};
```

5.3 Using the context

Once you have stored the context, you can retrieve it at any point to inspect or change the information in the context. To retrieve your context, call **Context.Find** or **Context.Acquire**:

```
Context.Find: PROCEDURE [type: Context.Type,
  window: Window.Handle] RETURNS [Context.Data];
```

```
Context.Acquire: PROCEDURE [type: Context.Type,
  window: Window.Handle] RETURNS [Context.Data];
```

These procedures both retrieve the data associated with a particular context (**type** specifies which context to retrieve). The difference between the two is that **Acquire** monitors the data so that only one process can have the context at a time. You can use either one, depending on the needs of your application.

The only case where retrieving the context is slightly tricky is when you have the wrong window handle: that is, if the context is associated with the body window, and you have a handle to the shell, or vice versa.

If you have a body window, and would like the shell, you can call `StarWindowShell.ShellFromChild`; if you have the shell and want the body window, you can call `StarWindowShell.GetBody` to get a handle to the body window. (Note that this only works if there is only one body window.)

```
StarWindowShell.GetBody: PROCEDURE [  
    SWS: StarWindowShell.Handle] RETURNS [Window.Handle];
```

```
StarWindowShell.ShellFromChild: PROCEDURE [  
    child: Window.Handle] RETURNS [StarWindowShell.Handle];
```

Here is an example of retrieving the context:

```
--This procedure actually retrieves the context. It is called  
--from various other procedures  
GetContext: PROC [body: Window.Handle] RETURNS [data: Data]  
= {data ← Context.Find[context, body];  
   RETURN [data]};
```

```
--This procedure makes the game more difficult, if possible.  
--If the player is already an expert, call another procedure to  
--post an appropriate message.
```

```
MakeGameHarder: MenuData.MenuProc = {  
    body: Window.Handle = --get handle to body window  
        StarWindowShell.GetBody[[window]];  
    data: Data ← GetContext[body]; --Find context  
    SELECT data.level FROM  
        beginner = > data.level ← intermediate,  
        intermediate = > data.level ← expert,  
        expert = > MessageUser[expert];  
    ENDCASE};  
};
```

In this case, we have associated the context with the body window, and not the shell. Thus, inside the `MenuProc` we need to call `GetBody`, since we have a handle to the `StarWindowShell` (which was passed in to the `MenuProc`), but we don't have a handle to the body window itself.

5.4 Summary

To allow the user to have multiple copies of your application, you need to do the following:

1. Declare a `Context.Type` in your (shared) global frame, and initialize it with a call to `Context.UniqueType`.
2. Create the context with a call to `Context.Create`. You should make this call in your window creation code, after you create the body window with which you are going to associate the context. One of the parameters to `Create` is a call back procedure to destroy the context.
3. Access the context as often as necessary with `Context.Find` or `Context.Acquire`.
4. Include a command (such as `Another`) so that the user can conveniently create new instances of the application. (The `Close` command will destroy an instance.)

To illustrate the use of contexts, here is the example from the last chapter, with the addition of Context information.

```

DIRECTORY
  Attention USING [AddMenuItem, Post],
  Context USING [Create, Data, Find, Type, UniqueType],
  Heap USING [systemZone],
  MenuData USING [CreateItem, CreateMenu, ItemHandle,
    MenuHandle, MenuProc],
  StarWindowShell USING [Create, CreateBody, GetZone,
    Handle, Push, SetRegularCommands],
  Window USING [Dims, Handle],
  XString USING [FromSTRING, nullReaderBody, ReaderBody,
    WriterBody];

SampleVPTool: PROGRAM
  IMPORTS Attention, Context, MenuData, StarWindowShell,
    XString = BEGIN

  -- TYPES
  Data: TYPE = LONG POINTER TO DataObject;
  DataObject: TYPE = RECORD [
    currentUser: XString.ReaderBody ← XString.nullReaderBody,
    level: Level ← beginner,
    score: LONG CARDINAL ← 0];

  Level: TYPE = {beginner, intermediate, advanced}

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

  -- Shared global data
  context: Context.Type ← Context.UniqueType[];

  -- Procedures
  -- This procedure retrieves the context associated with a given
  -- body window and returns it to the calling procedure.
  GetContext: PROC [body: Window.Handle] RETURNS [data: Data]
  = {data ← Context.Find[context, body];
    IF data = NIL THEN ERROR; --Just in case.
    RETURN [data]; };

  -- Register the command Sample Tool in the Attention Menu.
  Init: PROC = {
    sampleTool: XString.ReaderBody ←
      XString.FromSTRING["Sample Tool"L];
    Attention.AddMenuItem [
      MenuData.CreateItem [
        zone: Heap.systemZone,
        name: @sampleTool,
        proc: MenuProc] ];
  };

```

<< *This procedure is called when the user invokes the Sample Tool command from the Attention Menu or the Another command from the header. It creates a StarWindowShell, creates a single body window within that shell, allocates a context to go with that body window, adds some commands to the shell, and then displays the shell on the screen.* >>

```

MenuProc: MenuData.MenuProc = {
  another: XString.ReaderBody ←
    XString.FromSTRING["Another"L];
  redisplay: XString.ReaderBody ←
    XString.FromSTRING["Redisplay"L];
  post: XString.ReaderBody ← XString.FromSTRING["Post"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: SomeRedisplayProc, --chapter 6, Display
    bodyNotifyProc: SomeNotifyProc ];--Chapter 8, TIP

  --Get the zone attached to the window shell and use it
  --to create the menu items.
  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: @destroy,
      proc: DestroyMenuProc],
    MenuData.CreateItem [zone: z, name: @post, proc: Post]
  ];

  --Create the menu.
  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 and "hang it" from the body window.
  Context.Create [
    type: context,
    data: sysZ.NEW[DataObject],
    proc: Context.SimpleDestroyProc,
    window: body];

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

```

<<Procedures to implement header commands. Note that the Another command just calls MenuProc again. Recall that window is a parameter to the MenuProc.>>

```
Post: MenuData.MenuProc = {
  msg: XString.ReaderBody;
  body: Window.Handle = StarWindowShell.GetBody[[window]];
  data: Data ← GetContext [body];
  IF XString.Empty[data.currentUser] THEN
    msg ← XString.FromSTRING ["No current user."L]
  ELSE msg ← ...;
  Attention.Post [@msg];
};
```

--see chapter 6, *Displaying information on the screen*

```
RedisplayMenuProc: MenuData.MenuProc = { };
```

-- Mainline code

```
Init[];
```

```
END...
```

5.5 Exercise

The exercise for this chapter is to take your solution to the exercise for the last chapter and modify it so that you store the global tool data in a context rather than in the global frame. (If you did not do the exercise for the last chapter, you should retrieve our solution from the file server and modify it for this exercise.)

Once again, there is a template, in this case called DMTExercise2.mesa, that describes exactly what you need to write. You will also need the following files:

```
DMTDefs.mesa
DMTImpl.mesa
DMTConfig2.mesa
```

The interface and the implementation file are identical to the ones in the last chapter; the configuration file is slightly different..

Notes:

6. DISPLAYING INFORMATION ON THE SCREEN

Most applications need to display some sort of information on the screen. For example, the document editor needs facilities to display text and user-defined graphics, and the loader needs to display a list of applications. This chapter describes how to use the **Window**, **Display** and **SimpleTextDisplay** interfaces to display text and graphics in a body window.

6.1 Overview

In general, every application that displays information on the screen has a *display procedure* that actually paints the display. When called, this procedure can display all or part of the information in the application's window. The display procedure typically creates the display based on an associated data structure that represents the current data.

A display procedure is a call back procedure: the system calls the appropriate display procedure when it needs to repaint a particular portion of the screen. The system decides when to paint a window by keeping an *invalid list*. This list represents portions of the screen that contain invalid information. For example, if the user uncovers a window that was previously covered by another window, the window implementation will call the display procedure for the window that is being uncovered. A client can also call the window implementation to specify that a portion of its display has become invalid.

The window implementation accumulates invalid areas until a client requests a screen *validation*. At that point, it will call the appropriate display procedures to repaint the screen.

When performance is critical, an application can also paint directly to the screen instead of going through the display procedure. Even if you choose to display directly to the screen, however, you still need to provide a display procedure. (Otherwise, when the user moves another window on top of yours and then moves it back, there will be no way for the window implementation to repaint the display.) This chapter has examples of both methods.

6.2 Invalidating and validating

Either a client or a user can invalidate a portion of the screen. The window package handles repainting as the result of user actions, but if your program changes something that invalidates your application's display, you are responsible for ensuring that the display is correctly updated.

To specify that a portion of the screen contains invalid information, you call `InvalidateBox`:

```
Window.InvalidateBox: PROCEDURE [
    window: Window.Handle,
    box: Window.Box,
    clarity: Window.Clarity ← isDirty];
```

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

```
Window.Place: TYPE = UserTerminal.Coordinate;
```

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

```
Window.Clarity: TYPE = {isClean, isDirty};
```

`box` describes an invalid region with its position in the window and its dimensions (width and height). `place` is relative to the window's upper-left corner, which is defined to be at [0,0]. The `place`, `width`, and `height` are all measured in pixels.

`clarity` specifies the current state of the window. `isClean` means that the region is already white and there is nothing to erase; `isDirty` means that there is some information displayed that the system should erase before it displays the new information.

Calling `Invalidate` does not actually initiate a repaint operation; it merely adds the specified area to the invalid list for that window. To initiate the repaint, you need to call either `Validate` or `ValidateTree`:

```
Window.Validate: PROCEDURE [window: Window.Handle];
```

```
Window.ValidateTree: PROCEDURE [
    window: Window.Handle ← Window.rootWindow];
```

Calling `Validate` will affect only your window; calling `ValidateTree` will update the tree of windows whose root is `window`. As a simple example of calling `Invalidate` and `Validate`, suppose that your application has a `Redisplay` command in the header. When the user invokes this command you should invalidate the entire window and then validate it:

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

Recall that within a `MenuProc` you have a handle to the `StarWindowShell` and not the body window. Thus, as in the example above, you must get a handle to the body window before you do the invalidation and validation.

The [30000, 30000] dimensions are arbitrary; they just need to be large enough to guarantee that you invalidate the entire window. It is never wrong to invalidate a box that is larger than you really need; the window implementation ensures that you cannot paint outside the boundary of your window.

6.3 Writing a display procedure

The display procedure is of type `Window.DisplayProc`:

```
Window.DisplayProc: TYPE = PROCEDURE [
    window: Window.Handle];
```

Within a display procedure, you have two choices for how to display information. You have access to a list of invalid regions for the window, and you can repaint just the invalid regions, or you can ignore the list and repaint the entire window.

If you want to enumerate the invalid regions and repaint each one individually, you call `Window.EnumerateInvalidBoxes`:

```
Window.EnumerateInvalidBoxes: PROCEDURE [
    window: Window.Handle,
    proc: PROCEDURE [Window.Handle, Window.Box]];
```

`EnumerateInvalidBoxes` will call `proc` once for each box on `window`'s invalid list, passing in the window handle and the invalid region. The `window` passed to `proc` is the same as the `window` passed to `EnumerateInvalidBoxes`; this should be the body window (the `window` parameter to the `DisplayProc`.) `proc` is responsible for redisplaying the correct information in the specified region.

You must call `EnumerateInvalidBoxes` from within a `DisplayProc`. Here is an example of using this method:

```
DisplayGraphicSW: Window.DisplayProc =
    BEGIN
        data: DataHandle = FindContext>window];

        RepaintGraphicSW: PROC [
            window: Window.Handle, box: Window.Box] =
            {RepaintBox[data.bitmap, window, box]};

        Window.EnumerateInvalidBoxes[
            window, RepaintGraphicSW];
    END;
```

When this display procedure is called, it makes two calls: one to `FindContext`, and the other to `EnumerateInvalidBoxes`. The call to `EnumerateInvalidBoxes` will in turn result in a call to `RepaintGraphicSW` for each of the invalid boxes in `window`; `RepaintGraphicSW` is then responsible for repainting the area described by `box`. For now, don't worry about how `RepaintBox` displays to the screen; the following sections discuss this.

The other way to write a `DisplayProc` is to ignore the invalid list and just repaint the entire window. This isn't as inefficient as it sounds, because `ViewPoint` keeps a *bad phosphor list* for each window. The bad phosphor list consists of the visible portions of the window's invalid areas; it thus represents the parts of the window that need to be repainted. If your `DisplayProc` ignores the invalid list and repaints the entire window, the painting will be *clipped* to this list. This means that the window implementation will only paint areas that are on the bad phosphor list, and will ignore requests to paint other areas.

Thus, even when you try to repaint the entire window, the window implementation will only repaint the areas that are both invalid and visible. Thus, you should usually just repaint your entire window within the `DisplayProc`; you only need to use the `EnumerateInvalidBoxes` method when there is no bad phosphor list (the window has never been validated) or when redisplaying the entire window will take too long.

Whichever method you use, your display procedure should not change the data structure; it should just paint the screen. To be safe, you should generally **MONITOR** your display data structure and make the display procedure an **ENTRY** procedure. (See section 50.3.1 of the *ViewPoint Programmer's Manual* for a complete discussion of this problem.)

The following sections discuss how to use the facilities of the `SimpleTextDisplay` and `Display` interfaces to actually paint information on the screen.

6.3.1 SimpleTextDisplay

The `SimpleTextDisplay` interface provides facilities for displaying text in a window. The primary procedure in this interface is `StringIntoWindow`:

```
SimpleTextDisplay.StringIntoWindow: PROCEDURE [  
  string: XString.Reader,  
  window: Window.Handle,  
  place: Window.Place,  
  lineWidth: CARDINAL ← CARDINAL.LAST,  
  maxNumberOfLines: CARDINAL ← 1,  
  lineToDeltaY: CARDINAL ← 0,  
  wordBreak: BOOLEAN ← TRUE,  
  flags: BitBlit.BitBlitFlags ← Display.paintFlags]  
  RETURNS [lines, lastLineWidth: CARDINAL];
```

This procedure displays `string` in `window`, starting at `place`. The other parameters describe formatting details; see the `SimpleTextDisplay` documentation for details. For example:

```
Redisplay: Window.DisplayProc = {  
  data: Data ← GetContext [window];  
  writerBody: XString.WriterBody ← XString.NewWriterBody [  
    maxLength: 250, z: sysZ];  
  xfo: XFormat.Object ← XFormat.WriterObject [  
    w: @writerBody];  
  
  --put text and value of data.count in string  
  XFormat.String [h: @xfo, s: "This is a string displayed in a  
    body window using StringIntoWindow. The current  
    value of data.count is "L];  
  XFormat.Decimal [h: @xfo, n: data.count];  
  
  --display the string  
  [] ← SimpleTextDisplay.StringIntoWindow [  
    string: XString.ReaderFromWriter [@writerbody],  
    window: window, --The body window  
    place: [10,10], --Upper-left corner is [0,0]  
    lineWidth: 300, --number of pixels wide  
    maxNumberOfLines: 10]; --Arbitrary-- };
```

This display procedure displays a constant string and the current value of `data.count`. `data.count` is just a count of something interesting; it doesn't matter exactly what is being counted. (Remember, `window` is a parameter to the `DisplayProc`; it specifies the body window that is to be painted.) Notice that the information displayed in the window depends on the value of the variable `count`; if your program changes the value of `count`, you need to do an `Invalidate` and then a `Validate` to update the display.

6.3.2 Display

The `Display` interface provides procedures to paint points, lines, bitmaps, repeating patterns, boxes, circles, arcs, ellipses, and so on. Because of the wide variety, we won't try to cover all of the routines in this interface. As examples, however, here are the declarations of two of the more common `Display` procedures:

```
Display.Black: PROCEDURE [
  window: Window.Handle,
  box: Window.Box];
```

```
Display.Bitmap: PROCEDURE [
  window: Window.Handle,
  box: Window.Box,
  address: Environment.BitAddress,
  bitmapBitWidth: CARDINAL,
  flags: BitBlt.BitBltFlags ← paintFlags];
```

```
Environment.BitAddress: TYPE = MACHINE DEPENDENT RECORD [
  word: LONG POINTER,
  reserved: [0..LAST[WORD]/Environment.bitsPerWord) ← 0,
  bit: [0..Environment.bitsPerWord)];
```

`Black` just paints the specified portion of the display black. There are similar procedures to paint a portion white or to invert it.

`Bitmap` displays the bitmap specified by `address` and `bitmapBitWidth` into `box` in `window`.

`bitmapBitWidth` specifies the width of the bitmap in pixels. This must be a multiple of 16.

`address` is the field that describes the bitmap to be painted. Within an address, `word` is a pointer to an array of bits (the actual bitmap.) `reserved` and `bit` are primarily for the purposes of lower-level routines; for the purposes of this course, they will both always be 0.

`flags` specifies how `Bitmap` should interact with the pixels that are already displayed in the specified area. The default, `paintFlags`, specifies that black source pixels should cause black display pixels, and white source pixels have no effect. There are various other alternative ways to interact with existing pixels, such as XORing; check the `Display` chapter of the *ViewPoint Programmer's Manual* for details.

6.3.2.1 Example: Checkerboard

Here is a `DisplayProc` that draws the checkerboard in Figure 6.1.

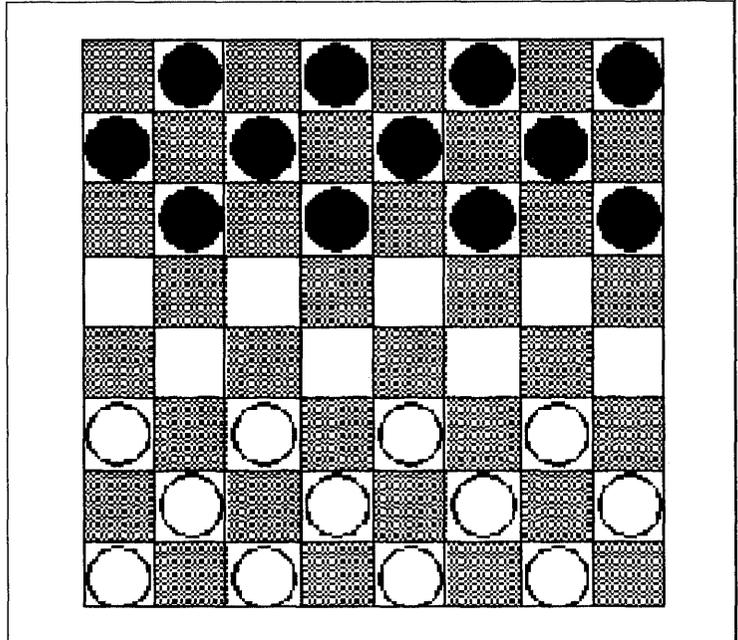


Figure 6.1: The checkers display

```

DisplayProc: window.DisplayProc = {
  boardWidth: INTEGER ← (8*Defs.squareDims.w) + 8;
  checkerWidth: INTEGER = Defs.squareDims.w;
  checkerHeight: INTEGER = Defs.squareDims.h;
  --the offset is the difference between the upper left hand
  --corner of the window and the upper left hand corner of
  --the checkerboard
  offsetWidth: INTEGER ← Defs.Offset.w;
  offsetHeight: INTEGER ← Defs.Offset.h;
  mydata: Defs.Data ← GetContext>window;

  --draw edges of checkerboard. Use boxes of width 1 pixel
  Display.Black[ --upper left to upper right
    window: window,
    box: [place: [offsetWidth, offsetHeight],
          dims: [boardWidth, 1]];
  Display.Black[ --lower left to lower right
    window: window,
    box: [place: [offsetWidth,
                  (offsetHeight + boardWidth)],
          dims: [boardWidth, 1]];
  Display.Black[ --lower left to upper left
    window: window,
    box: [place: [offsetWidth, offsetHeight],
          dims: [1, boardWidth]];
  Display.Black[ --lower right to upper right
    window: window,
    box: [place: [(offsetWidth + boardWidth),
                  offsetHeight],
          dims: [1, boardWidth]];

```

```

-- Use Display.Line instead of Display.Black for variety
--draw 7 horizontal lines to make 8 boxes.
FOR i: CARDINAL IN [1..8) DO
  Display.Line[
    window: window,
    start: [x: offsetWidth + 0,
            y: (offsetHeight + i*(checkerHeight + 1))],
    stop: [x: (offsetWidth + boardWidth),
           y: (offsetHeight + i*(checkerHeight + 1))]];

--draw 7 vertical lines
Display.Line[
  window: window,
  start: [x: (offsetWidth + i*(checkerWidth + 1)),
          y: offsetHeight + 0],
  stop: [x: (offsetWidth + i*(checkerWidth + 1)),
         y: (offsetHeight + boardWidth)];
ENDLOOP;

--draw gray boxes
FOR i: CARDINAL IN [0..8) DO
  FOR j: CARDINAL IN [0..8) DO
    IF (i MOD 2 = j MOD 2) THEN Display.Gray[
      window: window,
      box: [place: [
              x: (offsetWidth + 1 + i*(checkerWidth + 1)),
              y: (offsetHeight + 1 + j*(checkerHeight + 1))],
            dims: [checkerWidth, checkerHeight]]];
    ENDLOOP;
  ENDLOOP;

-- put the checkers on the board
FOR m: CARDINAL IN [0..8) DO
  FOR n: CARDINAL IN [0..8) DO
    bitmap: Defs.BitmapItems;
    SELECT mydata[m][n].piece FROM
      black = > bitmap ← black;
      blackKing = > bitmap ← blackKing;
      white = > bitmap ← white;
      whiteKing = > bitmap ← whiteKing;
    ENDCASE = > LOOP; --no piece here
    Display.Bitmap[
      window: window,
      box: [place: mydata[m][n].place,
            dims: Defs.squareDims],
      address: Defs.GetChecker[bitmap],
      bitmapBitWidth: checkerWidth,
      flags: Display.replaceFlags];
    IF mydata[m][n].marked THEN Display.Invert[
      window: window,
      box: [place: mydata[m][n].place,
            dims: Defs.squareDims]];
    ENDLOOP;
  ENDLOOP; };

```

Once an actual game is in progress, areas of the screen (checkerboard squares) will gradually become invalid as the user moves the checkers around. Thus, the code to move and delete checkers requires invalidations and validations. For example, the code to delete a piece might look like this:

```

DeletePiece: PROC[
  square: Defs.Square, window: Window.handle] = {
  IF square = NIL THEN RETURN;
  square.piece ← none;
  Window.InvalidateBox[
    window, [square.place, Defs.squareDims]];
  Window.Validate>window];
};

```

This code assumes that the data type `Defs.Square` contains fields for the tool window, the location of the square in the matrix, and the current contents of the square.

6.3.2.2 Example: FlySwatter

Here is an example that uses `Display.White` and `Display.Bitmap`. This example is part of a "Fly Swatter" game that displays flies in the body window of an application, turns the cursor into a flyswatter, and lets the user swat flies on the screen.

```

--Here is part of the interface FlyDefs
Data: TYPE = LONG POINTER TO DataObject;

```

```

DataObject: TYPE = RECORD [
  iterations: CARDINAL ← 0, -- number of times fly has appeared
  numberOfHits: CARDINAL ← 0, -- successful hits
  stopGame: BOOLEAN ← TRUE]; -- game in progress or not

```

```

--messages stuff
Key: TYPE = {startGame, ...};

```

```

Get: PROCEDURE [key:Key] RETURNS [XString.ReaderBody];

```

```

--the implementation
FlySwatterImpl: PROGRAM ... =
BEGIN
  sysZ: UNCOUNTED_ZONE = Heap.systemZone;
  --the width of a "fly"
  FlyBitMapWidth: CARDINAL = 32;
  --bitmap is 32x 32, so it takes 64 16-bit words to represent a fly.
  WordsInPicture: CARDINAL = 64;
  IconPictureBits: TYPE = ARRAY [0..WordsInPicture) OF WORD;

```

```

--procedure to start game
PlayGame: PUBLIC PROCEDURE [
  body: window.Handle, data: FlyDefs.Data] =
  BEGIN
  ticks: Process.Ticks = Process.MsecToTicks[1000];
  startGame: XString.ReaderBody ←
    Defs.Get[Defs.Key.startGame];

  << Create a temporary area of storage for the bitmap, to
  keep it out of the program's frame. This is the standard way
  to allocate space for a bitmap. >>
  bitBuffer: LONG POINTER TO IconPictureBits ←
    space.ScratchMap[1].pointer;

  -- make the cursor turn into a flyswatter
  newCursorBitMap: UserTerminal.CursorArray ← [177776B, ...];
  swatter: Cursor.Object ← [[last,7,7], newCursorBitMap];
  Cursor.Store[@swatter];

  --start out with whole body window white
  Display.White[body,[[0,0],FlyDefs.bodyWindowDims]];

  --initialize bitBuffer to contain bitmap picture of fly
  bitBuffer ↑ ← [000007B, 174000B. . .];

  -- reset all the data variables
  data.iterations ← 0;
  data.numberOfHits ← 0;
  data.stopGame ← FALSE;

  --post message telling user that game is starting
  Attention.Post[@startGame];

  --choose a random location for the fly; store in myBox
  FOR i: CARDINAL IN [0..FlyDefs.iterationsPerGame)
  WHILE NOT data.stopGame DO
    xPos: INTEGER ← ..random location
    yPos: INTEGER ← ..random location
    myBox: Window.Box ←
      [[xPos, yPos], [FlyBitMapWidth, FlyBitMapWidth]];

    --display the fly in appropriate place
    Display.Bitmap[window: body,
      box: myBox,
      bitmapBitWidth: FlyBitMapWidth,
      address: [bitBuffer, 0, 0]];

    --increment count of number of flies
    data.iterations ← data.iterations + 1;

    -- wait until its time to do next fly
    Process.Pause[ticks:ticks];
  ENDLOOP;

  --Game is over, so clear the display, reset the pointer, and
  --print out results of game.
  Display.White[body,[[0,0],FlyDefs.bodyWindowDims]];
  data.stopGame ← TRUE;
  Cursor.Set[textPointer];
  PrintStats[body,data];
  bitBuffer ← space.Unmap[bitBuffer];

  END;

```

The first step is to create a bitmap of a flyswatter, and store that as the current cursor representation. (**Note:** Appendix B, *Icon Editor*, discusses how to create bitmaps.) The second step is to create a bitmap picture of a fly, and display it periodically on the screen. This example displays a fly once a second; a real game would make this time variable. We did not include the code that decides where to put the fly, since it is incidental to the example.

When the game is over, make the entire body window white, and then call `PrintStats` to print the results of the game.

You should notice that the procedure `PlayGame` is not a display procedure. When an application wants to display something on the screen, the standard method is to update your data structures, call `Invalidate` and then call `Validate`. This will result in a call to the application's display procedure, which will update the display.

However, when painting performance is critical, you can paint directly to the screen without going through your display procedure, as illustrated in this example. This example puts a fly on the screen once every second, so soing regular invalidations and validations would be inefficient.

In fact, there is no display procedure for this example. Even if you generally paint directly to the display, you generally still need a display procedure to recreate the state of the screen when the user moves windows around. However, in this case, there is no "current state" of the display to be lost if you cover the window and then uncover it, so there is no need for a display procedure.

6.4 LimitProcs and AdjustProcs

Depending on the kind of information that you display, there are times when you want to put some restrictions on the size or location of your shell. For example, you might want to ensure that the shell always retains a certain minimum size, or that it is always on the screen somewhere. You can exercise this kind of control with *LimitProcs* and *AdjustProcs*.

A `LimitProc` gives you control over the size and placement of a shell. A `LimitProc` is of type `StarWindowShell.LimitProc`:

```
StarWindowShell.LimitProc: TYPE = PROCEDURE [  
    SWS: StarWindowShell.Handle,  
    box: Window.Box]  
    RETURNS [Window.Box];
```

The window system will call the `LimitProc` for a window whenever the size or location of the shell is about to change. You then have veto or modification rights over the size and location of the shell. `box` is the requested size of the shell; the return value is the size that you want the shell to be. A `LimitProc` is thus just a way to potentially change `box` before the actual size change takes place. You set and obtain `LimitProcs` with the following procedures:

```
StarWindowShell.GetLimitProc: PROCEDURE [
  SWS: StarWindowShell.Handle]
  RETURNS [StarWindowShell.LimitProc];
```

```
StarWindowShell.SetLimitProc: PROCEDURE [
  SWS: StarWindowShell.Handle,
  PROC: StarWindowShell.LimitProc]
  RETURNS [old: StarWindowShell.LimitProc];
```

You should call `SetLimitProc` from your `MakeShell` procedure. There is a default `LimitProc`, `StarWindowShell.StandardLimitProc`, that keeps shells on the screen and keeps them from getting too small. You only need to write your own limit procedure if you have some special needs.

If your body window display depends on the size of the surrounding shell, then you need to write an `AdjustProc`. An `AdjustProc` is of type `StarWindowShell.AdjustProc`:

```
StarWindowShell.AdjustProc: TYPE = PROCEDURE [
  SWS: StarWindowShell.Handle,
  BOX: Window.Box,
  WHEN: StarWindowShell.When];
```

```
StarWindowShell.When: TYPE = {before, after};
```

```
StarWindowShell.GetAdjustProc: PROCEDURE [
  SWS: StarWindowShell.Handle]
  RETURNS [StarWindowShell.AdjustProc];
```

```
StarWindowShell.SetAdjustProc: PROCEDURE [
  SWS: StarWindowShell.Handle,
  PROC: StarWindowShell.AdjustProc]
  RETURNS [old: StarWindowShell.AdjustProc];
```

An `AdjustProc` will be called both before and after a shell changes size. `box` is the interior size of the `StarWindowShell`. `when` indicates whether the current call is before or after the size changes. Typically, the `after` case is more interesting, since you may have to adjust the display to fit the new box size.

For more information on `LimitProcs` and `AdjustProcs`, see the `StarWindowShell` documentation.

6.5 Summary

Using `SimpleTextDisplay` and `Display`, an application can display arbitrary text and graphics in a body window. Most body windows that display information must provide a *display procedure* that can recreate all or part of that display on demand. The window implementation will call this procedure whenever it needs to redisplay the application's window. The only exceptions to this rule are applications like the Fly Swatter, which have no state to recreate.

If an application wants to change its display, it can either just display the new information directly to the screen, or it can update its data structures to represent the correct information, call `window.InvalidatBox` to specify that a portion of the screen is invalid, and then call `window.Validate` to force the window system to update the display. The latter method is preferred. If

you want to display information directly to the screen without going through your `DisplayProc`, you should read the caveats in section 50.3 of the *ViewPoint Programmer's Guide*.

If the information that you display depends on the size or location of the shell, then you need to write a `StarWindowShell.LimitProc` and/or a `StarWindowShell.AdjustProc`. The `LimitProc` allows you to exercise control over the location of your shell; the `AdjustProc` allows you to adjust the contents display according to the size and location of the shell.

6.6 Exercise

Text Window provides a non-editable window for displaying text. You display text by selecting an icon (simple document) on the desktop and invoking the Load command. If no file is currently selected, Load will clear the window.

Text Window also has a Wrap command that toggles the state of the *wrapping*. When the wrapping is on, the line width will be equal to the width of the window (the text will fit the body window.) When wrapping is off, the line width will be constant, and you will have to make the window bigger if you want to see more of the text. Figure 6.2 illustrates the Text Window tool with a file loaded in it and wrapping turned on.

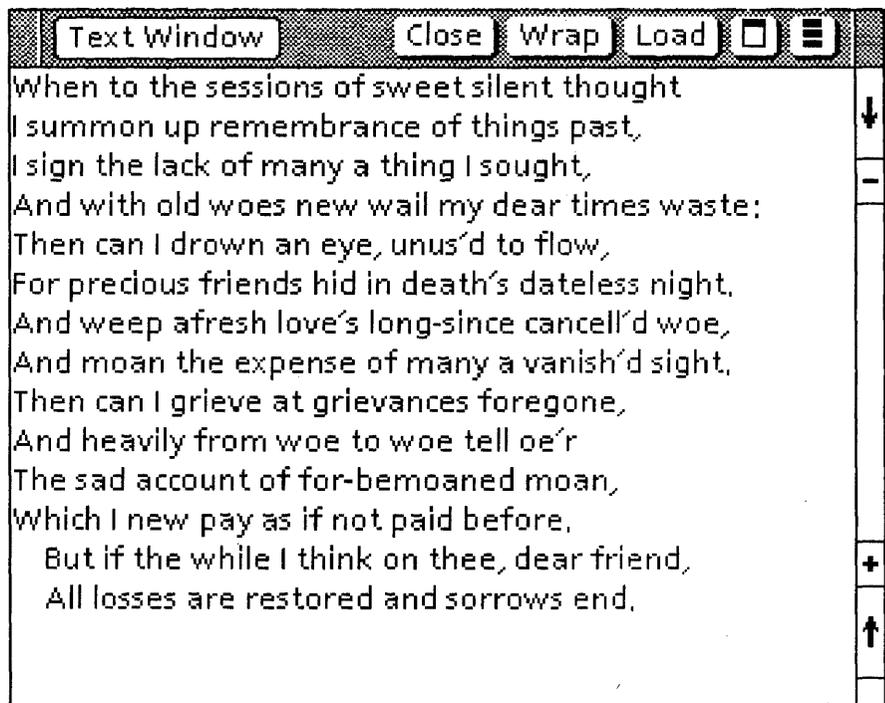


Figure 6.2 The Text Window tool

You are to write the user interface for the Text Window and implement the Load and Wrap commands. The procedures that you will implement are in the template `TextExercise.mesa`. Other files that you will need are `TextImpl.mesa`, `TextDefs.mesa`, and `Text.Config`.

Once you have created a body window for an application, you need to define its user interface. The last chapter discussed how to display arbitrary text and graphics in a body window. This chapter discusses how to convert a body window into a *form window*.

A form window is a body window whose purpose is to display the parameters and commands associated with an application. Thus, you might want to use a form window if your application requires specific user input, commands, and feedback.

Form windows are also the basis for *property sheets*, which provide a standardized way to examine and change the properties of an object. This chapter discusses the facilities of the `FormWindow` interface; the next chapter discusses the `PropertySheet` interface.

7.1 Overview

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 for the user to fill in. Figure 7.1 illustrates a typical form window.

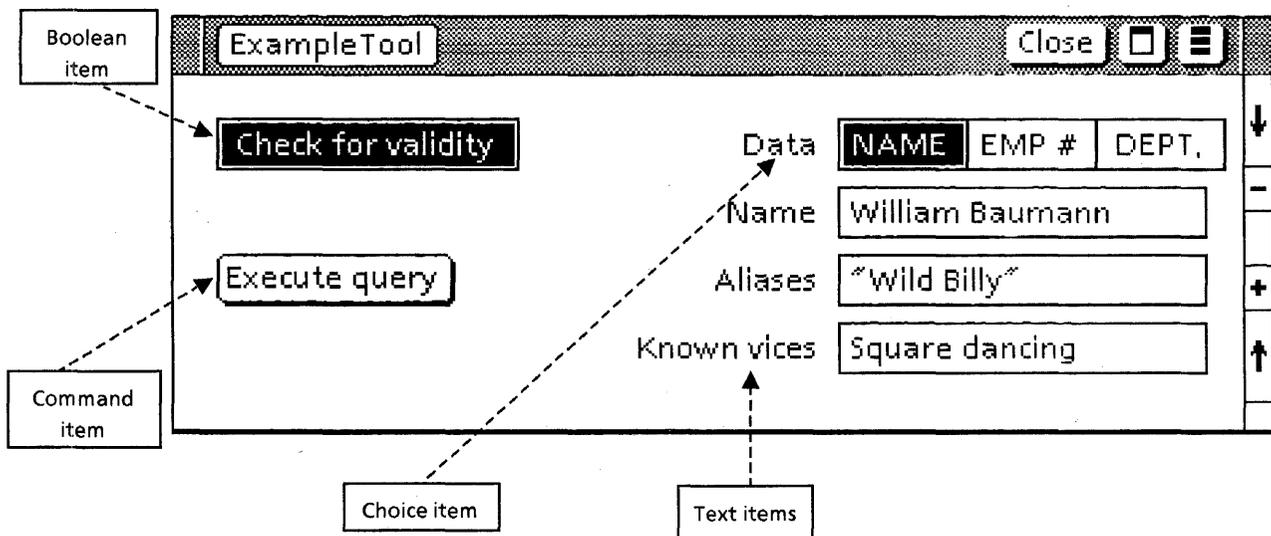


Figure 7.1: Form window

A form window contains *form items*, which generally consist of a keyword, such as the name of a command or parameter, and space for the user to fill in values for parameters. The user fills

in the appropriate parameters and then invokes a command. Here is a list of form item types:

- boolean item** A boolean item has two states: on and off (or **TRUE** and **FALSE**). When the value is **TRUE**, the item is highlighted.
- choice item** A choice item has an enumerated list of choices, from which the user can select one value at a time. A choice item's value is of type **FormWindow.ChoiceIndex**.
- text item** A text item is a user-editable string; its value is of type **XString.ReaderBody**.
- decimal item** A decimal item is a text item that has a value of type **XLReal.Number**.
- integer item** An integer item is a text item that has a value of type **LONG INTEGER**.
- command item** A command item is an item that has an associated procedure. The system calls this procedure when the user invokes the command.
- tagonly item** A tagonly item is a string that the user can neither select nor edit.
- window item** A window item is 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**.

7.2 Creating form windows

To create a form window, call **FormWindow.Create**:

```
FormWindow.Create: PROCEDURE[  
  window: Window.Handle,  
  makeltems: FormWindow.MakeltemsProc,  
  layoutProc: FormWindow.LayoutProc ← NIL,  
  windowChangeProc: FormWindow.GlobalChangeProc ← NIL,  
  zone: UNCOUNTED_ZONE ← NIL,  
  clientData: LONG_POINTER ← NIL];
```

Create takes an ordinary window and makes it a form window. Typically, the **window** parameter will be a body window that you created by calling **StarWindowShell.CreateBody**.

makeltems, **layoutProc**, and **windowChangeProc** are call-back procedures. **makeltems** creates the items that you want in your form window; **layoutProc** specifies the desired position of the items in the window, and **windowChangeProc** allows you to react when the value of something in the form window changes. The following sections discuss each of these parameters in detail. There is also an example in section 7.2.4 that contains a full example of a call to **Create**.

zone is the zone from which **FormWindow** will allocate storage for the items in the form window. 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**.

7.2.1 MakeItemsProcs

The **MakeItems** parameter to **Create** is of type **FormWindow.MakeItemsProc**:

```
FormWindow.MakeItemsProc: TYPE = PROCEDURE [
    window: Window.Handle, --form window
    clientData: LONG POINTER]; --pointer passed to Create
```

This procedure allocates the various items that you want to display in your form window. The **FormWindow** interface provides a procedure for creating each type of item: **MakeBooleanItem**, **MakeChoiceItem**, **MakeCommandItem**, and so on. In your **MakeItems** procedure, you should just call the appropriate procedure for each item that you want in your form window.

Obviously, since each type of form item has different characteristics, the procedures to make the various kinds of form items are slightly different. The following sections describe **MakeCommandItem**, **MakeTextItem**, and **MakeChoiceItem**; consult the *ViewPoint Programmer's Manual* for the declarations of any of the other **MakeXXX** procedures.

The example in section 7.2.4 contains an example of a **MakeItemsProc**.

7.2.1.1 MakeCommandItem

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

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

window is the form window. (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. You will need to use this key to make calls on other **FormWindow** procedures. You can use any

scheme you want for defining keys, but *each item in a given form window must have a unique key.* (An `ItemKey` is just a `CARDINAL`.)

`tag` is text that will appear before (to the left of) the item on the same line; `suffix` is text that will appear after (to the right of) the item on the same line. Typically, there is no tag or suffix associated with a command item.

`visibility` indicates whether the item should be displayed on the screen. The default is `visible`. For information on invisible items, see the *ViewPoint Programmer's Manual*.)

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

`readOnly` indicates whether the *user* can change the value of the item. (If an item is `readOnly`, the client can still change the value by calling appropriate procedures in the `FormWindow` interface.)

The last two parameters are specific to a command item. The `commandName` is the name that will appear in the form window; invoking this command will result in a call to `commandProc`. A `commandProc` is of type `FormWindow.CommandProc`:

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

7.2.1.2 MakeTextItem

```
FormWindow.MakeTextItem: PROCEDURE [  
    window: Window.Handle,  
    myKey: FormWindow.ItemKey,  
    tag: XString.Reader ← NIL,  
    suffix: XString.Reader ← NIL,  
    visibility: FormWindow.Visibility ← visible,  
    boxed: BOOLEAN ← TRUE,  
    readOnly: BOOLEAN ← FALSE,  
    width: CARDINAL,  
    initString: XString.Reader ← NIL,  
    wrapUnderTag: BOOLEAN ← FALSE,  
    passwordFeedback: BOOLEAN ← FALSE,  
    hintsProc: FormWindow.TextHintsProc ← NIL,  
    nextOutOfProc: FormWindow.NextOutOfProc ← NIL,  
    SPECIALKeyboard: BlackKeys.Keyboard ← NIL];
```

The first seven parameters are identical to those in `MakeCommandItem`.

`width` is the desired width of the item, in screen dots.

`initString` is the initial string (default value) for the text item.

`wrapUnderTag` is not yet implemented.

`passwordFeedback` indicates that the text should be displayed in an unreadable form (asterisks) rather than as normal

characters. The correct value of the string is maintained internally.

`hintsProc` will be called to make a popup hint menu for the user. `nextOutOfProc` will be called when the user presses the next key while the input focus is in this text item. This gives the client an opportunity to create more text items. `SPECIALKeyboard` allows clients to make a special keyboard available to the user when he is typing into a text field.

These last three parameters are beyond the scope of this course. For more information, see the `FormWindow` documentation in the *ViewPoint Programmer's Manual*.

7.2.1.3 MakeChoiceItem

```
FormWindow.MakeChoiceItem: PROCEDURE [
  window: Window.Handle,
  myKey: FormWindow.ItemKey,
  tag: XString.Reader ← NIL,
  suffix: XString.Reader ← NIL,
  visibility: FormWindow.Visibility ← visible,
  boxed: BOOLEAN ← TRUE,
  readOnly: BOOLEAN ← FALSE,
  values: FormWindow.ChoiceItems,
  initChoice: FormWindow.ChoiceIndex,
  fullyDisplayed: BOOLEAN ← TRUE,
  verticallyDisplayed: BOOLEAN ← FALSE,
  hintsProc: FormWindow.ChoiceHintsProc ← NIL,
  changeProc: FormWindow.ChoiceChangeProc ← NIL,
  outlineOrHighlight: FormWindow.OutlineOrHighlight ←
    highlight];
```

This procedure creates a *choice item*. A choice item is an enumerated list of choices, from which the user selects the current value of the item. When the user clicks on a choice, that choice becomes the current choice.

`values` is the list of all possible choices. It is of type `ChoiceItems`:

```
FormWindow.ChoiceItems: TYPE = LONG DESCRIPTOR FOR ARRAY
  FormWindow.ChoiceIndex OF FormWindow.ChoiceItem;
```

```
FormWindow.ChoiceIndex: TYPE = CARDINAL [0..37777B];
```

```
FormWindow.ChoiceItem: TYPE = RECORD [
  var: SELECT type: FormWindow.ChoiceItemType FROM
  string = > [
    choiceNumber: FormWindow.ChoiceIndex,
    string: XString.ReaderBody],
  bitmap = > [
    choiceNumber: FormWindow.ChoiceIndex,
    bitmap: FormWindow.Bitmap],
  wrapIndicator = > NULL];
```

```
FormWindow.Bitmap: TYPE = RECORD [
  height, width: CARDINAL,
  bitsPerLine: CARDINAL,
  bits: Environment.BitAddress];
```

Thus, each choice consists of either a string or a bitmap, and an associated `ChoiceIndex`. (Chapter 6, *Displaying information on*

the screen, discusses bitmaps in more detail.) You define the item indices; the values don't matter, except that each item within a given choice must have a unique index.

For string choices, the **FormWindow** implementation will copy the storage for the string; for bitmap choices, it will not. Thus, if you use a bitmap, you must ensure that the bitmap pointer is valid for the life of the form window.

wrapIndicator is not really a choice; it allows you to specify that the choices should go onto a new line. **wrapIndicator** is thus exclusively for formatting.

initChoice specifies which value should be the initial choice.

fullyDisplayed indicates whether to display all choices or just the current one. If **fullyDisplayed = FALSE**, the rest of the choices are available from a popup menu.

verticallyDisplayed indicates whether to display the choices vertically or horizontally. This is only important when **fullyDisplayed** is **TRUE**.

hintsProc allows you to create a popup hint menu. The default creates a hint menu with all possible choices. See the **FormWindow** documentation for details.

changeProc will be called whenever the choice changes. Section 7.2.3 discusses change procedures in more detail.

outlineOrHighlight specifies whether to highlight or underline the selected item .

FormWindow.OutlineOrHighlight: TYPE = {outline, highlight};

7.2.2 LayoutProcs

The second procedure parameter to **Create** is a **LayoutProc**, which defines the layout of the form items on the screen. The layout procedure is of type **FormWindow.LayoutProc**:

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

Unless you explicitly lay out an item, 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 of items in the form window. Among other things, this kind of layout simplifies multinational conversion. 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. The following sections discuss both methods in detail.

7.2.2.1 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 with `TabStops`. The example in Section 7.2.4 uses `TabStops`; for more detail, you will have to consult the `FormWindow` documentation.

To create a flexible layout, start by calling either `FormWindow.AppendLine` or `FormWindow.InsertLine` to create a line. 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:

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

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

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

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

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

```

LayoutFormItems: FormWindow.LayoutProc = {
  --create first line
  line: FormWindow.Line ← FormWindow.AppendLine
    [window: window,
     -- height ← defaultLineHeight --];
  --add items whose keys are 1 and 4 to first line
  FormWindow.AppendItem [
    window: window, item: 1, line: line];
  FormWindow.AppendItem
    [window: window, item: 4, line: line];
  --Create second line
  line ← FormWindow.AppendLine [
    window: window];
  --add item whose key is 2 to second line
  FormWindow.AppendItem [
    window: window, item: 2, line: line];
  --create third line
  line ← FormWindow.AppendLine [window: window];
  --add item whose key is 3 to third line
  FormWindow.AppendItem [
    window: window, item: 3, line: line];
};

```

7.2.2.2 Fixed layout

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

```

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

```

```

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

```

```

Window.Place: TYPE = UserTerminal.Coordinate;

```

```

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

```

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. This method 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]];
};

```

7.2.3 ChangeProcs

The third procedure parameter to **Create** is a *change procedure*. 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.

The change procedure that you pass to **Create** is a global change procedure that **ViewPoint** will call whenever a user or a program changes the value of an item in the form window.

```
FormWindow.GlobalChangeProc: TYPE = PROCEDURE [
    window: Window.Handle,
    item: FormWindow.ItemKey,
    calledBecauseOf: FormWindow.ChangeReason,
    clientData: LONG POINTER];
```

```
FormWindow.ChangeReason: TYPE = {user, client, restore};
```

Whenever the value of any item in the window changes, **ViewPoint** will call your **GlobalChangeProc** with the key of the item that was changed. If more than one item was changed, item will be **nullItemKey** and you will have to use the *changed boolean* (discussed below) to decide which items have changed.

calledBecauseOf specifies the kind of change. (Restore means that the client called **FormWindow.Restore** to return the form window to a former state.)

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 an associated *changed boolean*. The initial value of this boolean is always **FALSE**. When the value of an item changes, the **FormWindow** implementation sets the boolean to **TRUE**. You can thus check the boolean for a given item to find out if that item has changed. You are responsible for setting the boolean back to **FALSE**.

The **FormWindow** interface provides procedures that you can call to see if any items in the window have changed (**HasAnyBeenChanged**), to see if a specific item has been changed (**HasBeenChanged**), to reset the boolean for an individual item (**ResetChanged**), and to reset all booleans (**ResetAllChanged**). See the **FormWindow** chapter of the *ViewPoint Programmer's Manual* for details.

7.2.4 Example

Here is an example that creates the application illustrated in Figure 7.1.

DIRECTORY.

```

...
Example: PROGRAM IMPORTS... = {

Items: TYPE = {checkforvalidity, data, executeQuery, name,
aliases, born, knownVices};

formWindowDims: window.Dims ← [450, 250];
shellDims: Window.Dims = [550, 750];-- size of tool
name: XString.ReaderBody ← Xstring.FromSTRING["Example"L];

tabStopInterval: CARDINAL = 50;
context: Context.Type ← Context.UniqueType[];

-- Procedures

--register command in Attention Menu
Init: PROCEDURE = {
  Attention.AddMenuItem [
    MenuData.CreateItem [
      zone: Heap.systemZone,
      name: @name,
      proc: MakeShell ] ];
};

<<create StarWindowShell with one body window. Make
body window a form window, and then display shell on
screen.>>
MakeShell: MenuData.MenuProc = {
  shell: StarWindowShell.Handle = StarWindowShell.Create [
    name: @name];
  formWindow: Window.Handle ←
    StarWindowShell.CreateBody [
      sws: shell,
      box: [[0,0], formWindowDims] ];
  FormWindow.Create [
    window: formWindow,
    makeltemsProc: Makeltems,
    layoutProc: DoLayout];
  StarWindowShell.SetPreferredDims [shell, shellDims];
  StarWindowShell.Push [shell];
};

--create the items in the form window
Makeltems: FormWindow.MakeltemsProc = {
  fwz: UNCOUNTED_ZONE = FormWindow.GetZone[window];

  --create Check for validity boolean
  BEGIN
  rb: XString.ReaderBody ←
    XString.FromSTRING["Check for validity"L];
  FormWindow.MakeBooleanItem [
    window: window,
    myKey: Items.checkforvalidity.ORD,
    initBoolean: TRUE,
    label: [string[rb]] ];
  END;

```

```

--create Data choice item
BEGIN
choice0: XString.ReaderBody ←
  XString.FromSTRING["NAME"L];
choice1: XString.ReaderBody ←
  XString.FromSTRING["EMP NO."L];
choice2: XString.ReaderBody ←
  XString.FromSTRING["DEPT."L];
tag: XString.ReaderBody ← XString.FromSTRING["Data"L];
choices: ARRAY [0..3) OF FormWindow.ChoiceItem ← [
  [string[0, choice0] ],
  [string[1, choice1] ],
  [string[2, choice2] ] ];
FormWindow.MakeChoiceItem [
  window: window,
  myKey: Items.data.ORD,
  tag: @tag,
  values: DESCRIPTOR[choices],
  initChoice: 0 ];
END;

--Create command item
BEGIN
rb: XString.ReaderBody ← XString.FromSTRING[
  "Execute Query"L];
FormWindow.MakeCommandItem [
  window: window,
  myKey: Items.executeQuery.ORD,
  commandProc: ExecuteQuery,
  commandName: @rb];
END;

--Create first text item. Omit code for other text items,
--since they are all nearly identical.
BEGIN
initString: XString.ReaderBody ←
  XString.FromSTRING["William Baumann"L];
tag: XString.ReaderBody ← XString.FromSTRING["Name:"L];
FormWindow.MakeTextItem [
  window: window,
  myKey: Items.name.ORD,
  tag: @tag,
  width: 40,
  initString: @initString ];
END;
...};

```

```

--layout the form window with flexible layout
DoLayout: FormWindow.LayoutProc = {
  lineLeading: CARDINAL = 6; --space between lines
  topMargin: CARDINAL = 16; --space before first line
  line: FormWindow.Line;

  --tabs fixed at 50 spaces apart. Line up data, name,
  --aliases, and known vices vertically
  tabChoice: fixed FormWindow.TabStops =
    [fixed[ tabStopInterval]];
  FormWindow.SetTabStops[
    window: window, tabStops: tabChoice];
  -- Line 1: check for validity boolean and data choice
  line ← FormWindow.AppendLine [
    window: window,
    spaceAboveLine: topMargin];
  FormWindow.AppendItem [
    window: window,
    item: Items.checkforvalidity.ORD,
    line: line,
    tabStop: 16 / tabStopInterval,
    preMargin: 16 MOD tabStopInterval];
  FormWindow.AppendItem [
    window: window,
    item: Items.data.ORD,
    line: line,
    tabStop: 212 / tabStopInterval,
    preMargin: 212 MOD tabStopInterval];
  -- Line 2: blank line
  line ← FormWindow.AppendLine [
    window: window,
    spaceAboveLine: lineLeading];
  -- Line 3: execute query command and name text item
  line ← FormWindow.AppendLine [
    window: window,
    spaceAboveLine: lineLeading];
  FormWindow.AppendItem [
    window: window,
    item: Items.executeQuery.ORD,
    line: line,
    tabStop: 16 / tabStopInterval,
    preMargin: 16 MOD tabStopInterval];
  FormWindow.AppendItem [
    window: window,
    item: Items.name.ORD,
    line: line,
    tabStop: 201 / tabStopInterval,
    preMargin: 201 MOD tabStopInterval];
  -- Line 4: aliases ...
  --Line 5: known vices...
};

-- Mainline code
Init[];

}...

```

7.3 Getting and setting values

Every item that has a value that the user can change (all except tag-only and command items) also has procedures for the client to get and set the value. For example, the procedures for boolean items are called `FormWindow.GetBooleanItemValue` and `FormWindow.SetBooleanItemValue`:

```
FormWindow.GetBooleanItemValue: PROCEDURE [
  window: Window.Handle,
  item: FormWindow.ItemKey]
  RETURNS [value: BOOLEAN];
```

```
FormWindow.SetBooleanItemValue: PROCEDURE [
  window: Window.Handle,
  item: FormWindow.ItemKey,
  newValue: BOOLEAN,
  repaint: BOOLEAN ← TRUE];
```

For example, to get the value of the Check for validity boolean, you could make the following call:

```
valid: BOOLEAN ← FormWindow.GetBooleanItemValue[
  window,
  items.checkForValidity.ORD];
```

The procedures to get and set the values of other types are nearly identical, except for the type of the value. See the *ViewPoint Programmer's Manual* if you want more details.

7.4 Destroying a form window

```
FormWindow.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.

7.5 Summary

Form windows provide a standardized user interface for collecting parameters.

To create a form window, perform the following steps:

- Call `FormWindow.Create`, passing in the following parameters:
 - An existing body window
 - A `MakeItemsProc`, in which you need to create each item that you want to have in your form window.
 - An optional `LayoutProc`, which specifies where the items are to appear in the formwindow. Your layout

procedure can use either *fixed layout* or *flexible layout*. If you don't supply a layout procedure, the default is to put each item on a separate line.

- An optional *global change procedure*, which will be called whenever the user changes the value of anything in the form window. This allows you to recognize the change and act on it.

For more information on the other procedures in the `FormWindow` interface, see the `FormWindow` chapter of the *ViewPoint Programmer's Guide*.

7.6 Exercise

The exercise for this chapter is the Time Clock application, which keeps track of how much time you spend on various tasks. Figure 7.2 illustrates this tool.

Task	Billing	Job #
Editing	<input checked="" type="checkbox"/>	1
Reading/answering mail	<input type="checkbox"/>	2

Figure 7.2: The TimeClock application

For each task that you want to keep track of, you create a new *job*, which consists of a title, in-use boolean, and a job number. To create a job, type a job number in the appropriate field and invoke the "New Job" command. This will create a form field for the new job; you can then fill in the name of the job in the text item. The tool allows a maximum of 20 possible jobs.

When you want to start tracking a particular job, you select the associated boolean field, and the tool starts keeping track of the time that you spend on that job. When you want to stop billing a job you can either turn off the boolean or select another job. In the figure above, the tool is billing Job 1, but not Job 2. The tool can only bill one job at a time; it assumes that you can only work on one task at a time.

You can delete an existing job by typing in the job number and selecting the "Delete Job" command. This will remove the deleted job from the display and delete all data associated with the job.

When you want to see the data, you can select the "Print Report" command to print a report of your activities from any date to any other date. "Print Report" creates a property sheet that contains from-date/to-date fields; simply fill in these fields

and select **Done** and a report will appear on the display. You can then copy this report into a document and print it.

Your assignment is to write the code to generate the form window for this tool. The first line of the form window will contain 3 commands and an integer (Print Report, DeleteJob, NewJob, and jobNumber.) These items are fairly straightforward and have corresponding enumerated items in the definitions module.

You also need to create 20 lines, one for each possible job. Each of these 20 lines should have a text item, boolean item, and integer item. You will have to calculate the itemKey for each of these items. In addition, some of these lines may need initial values, so you will need to check the context data to see if an item has initial values. If the user has created that particular job, then the item should have some initial values.

If a line does have corresponding values in the context data you should use those values when creating the form items. If a line does not have any values associated with it then you should not initialize those values and you should make the line invisible. Later, when the user wants to create a new job, our code will make these items visible.

The procedures that you need to implement are in TimeClockFormImplTemp.mesa. The comments in this module define what you need to do more completely.

You will also need the following modules:

- TimeClockDefs
- TimeClockFormImpl
- TimeClockMsgImpl
- TimeClockImpl
- TimeClockPSheetImpl
- TimeClock.config

Notes:

8. PROPERTY SHEETS

This chapter describes how to create property sheets, which are essentially specialized form windows; the material in this chapter is very closely tied to the material in the last chapter. However, property sheets also depend on information discussed later in the course, such as icons. Thus, there are some aspects of property sheets which we delay until later in the course. In particular, you will not be implement the aspects associated with icons until you have read chapter 15, *Icon Applications*.

Property sheets allow the user to inspect and modify the properties of various objects. For example, paragraph properties include margins, justification, and line spacing, and printer properties include number of copies and paper size. Figure 8.1 shows the property sheet for a mail inbasket.

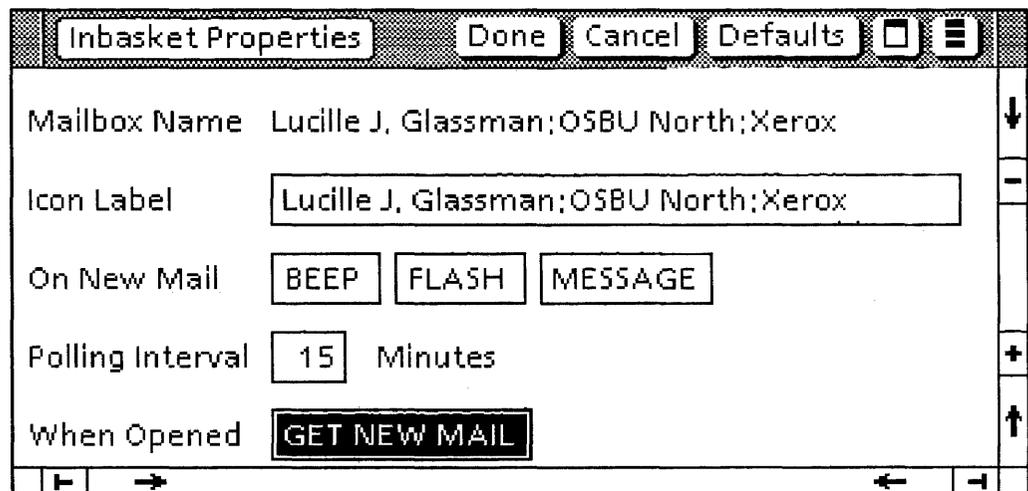


Figure 8.1: Property sheet

8.1 Creating a property sheet

To create a property sheet, you call `PropertySheet.Create`. Typically, you will call this procedure when the user selects an icon and presses `PROPS`, but we do not discuss how to implement this until chapter 15. Until then, you can call this procedure from anywhere else in your code; in the exercise for this chapter, we call it from a command in a tool's header.

```
PropertySheet.Create: PROCEDURE [  
  formWindowItems: FormWindow.MakeItemsProc,  
  menuItemProc: PropertySheet.MenuItemProc,  
  size: Window.Dims, --preferred size of property sheet  
  menuItems: PropertySheet.MenuItems ←  
    PropertySheet.propertySheetDefaultMenu,  
  title: XString.Reader ← NIL,  
  placeToDisplay: Window.Place ← PropertySheet.nullPlace,  
  formWindowItemsLayout: FormWindow.LayoutProc ← NIL,  
  windowAttachedto: StarWindowShell.Handle ← [NIL],  
  globalChangeProc: FormWindow.GlobalChangeProc ← NIL,  
  display: BOOLEAN ← TRUE,  
  clientData: LONG POINTER ← NIL,  
  afterTakenDown: PropertySheet.MenuItemProc ← NIL,  
  zone: UNCOUNTED_ZONE ← NIL]  
RETURNS [shell]: StarWindowShell.Handle];
```

Some of these parameters, such as **MakeItemsProc**, **LayoutProc**, and **globalChangeProc** are identical to the parameters to **FormWindow.Create**. When you are deciding what items to put in a property sheet, note that it is conventional to put adjectives, rather than commands. For example, "centered" is better than "center" and "justified" is better than "justify."

title is the property sheet title. This should include the word "properties" and be in all capitals. For example, INBASKET PROPERTIES would be better than Inbasket Properties.

The rest of this section provides some detail on the **menuItemProc** and **menuItems** parameters, and a complete example of a call to **Create**. For information on the other parameters, see the **PropertySheet** documentation in the *ViewPoint Programmer's Manual*.

8.1.1 Menultems

menuItems specifies the menu items (commands) that are displayed in the header of the property sheet:

```
PropertySheet.MenuItems: TYPE = PACKED ARRAY  
  PropertySheet.MenuItemType OF  
  PropertySheet.BooleanFalseDefault;
```

```
PropertySheet.MenuItemType: TYPE =  
  {done, apply, cancel, defaults, start, reset};
```

```
PropertySheet.BooleanFalseDefault: TYPE = BOOLEAN ← FALSE;
```

MenuItemType enumerates the possible commands; **MenuItems** specifies a subset of those commands.

The **PropertySheet** interface defines two common choices:

```
PropertySheet.propertySheetDefaultMenu:  
  PropertySheet.MenuItems = [  
    done: TRUE, apply: TRUE, cancel: TRUE];
```

```
PropertySheet.optionSheetDefaultMenu:  
  PropertySheet.MenuItems = [start: TRUE, cancel: TRUE];
```

You can use either of these, if you like. If you want a different subset of commands, you will have to use a record constructor to set the desired menu items to `TRUE`. See section 8.1.3 for an example of this.

8.1.2 MenuItemProc

ViewPoint will call the `menuItemProc` whenever the user selects one of the menu items in the header of the property sheet. A `menuItemProc` is of type `PropertySheet.MenuItemProc`:

```
PropertySheet.MenuItemProc: TYPE = PROCEDURE [
  shell: StarWindowShell.Handle,
  formWindow: Window.Handle,
  menuItem: PropertySheet.MenuItemType]
  RETURNS [ok: BOOLEAN ← FALSE];
```

`formWindow` is the main form window of the property sheet, and `menuItem` is the menu item that the user selected. Within this procedure, you implement the commands that are in the property sheet header. See the next section for an example.

8.1.3 Example of PropertySheet.Create

Here is an example of a call to create a property sheet. This code creates the property sheet illustrated in Figure 8.2:

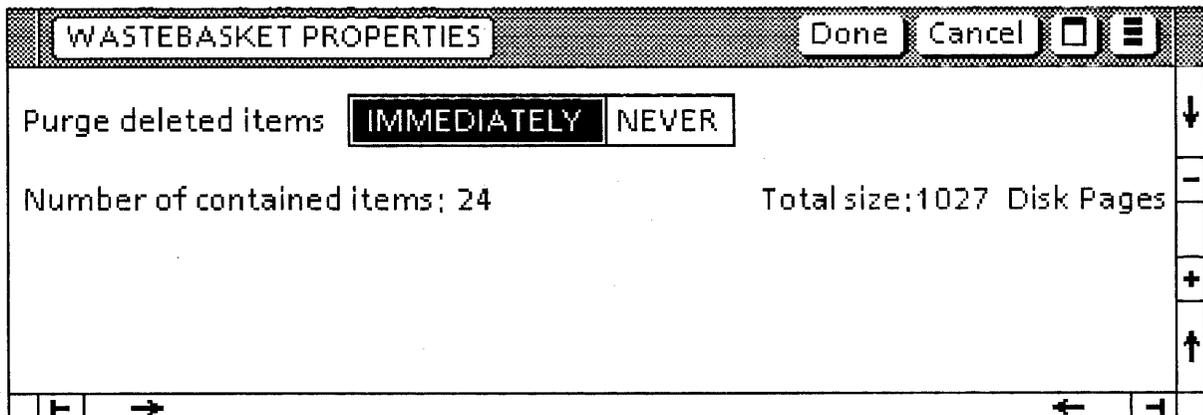


Figure 8.2: Property sheet for the Wastebasket icon

```

MakePropertySheet: PROC[...] = {
  title: XString.ReaderBody ← XString.FromSTRING [
    "WASTEBASKET PROPERTIES"L];
  << Create the property sheet. Have the commands Done,
  Cancel, and Defaults; have MyMenuItemProc implement
  those commands. Include a title. >>
  pSheetShell: StarWindowShell.Handle ← PropertySheet.Create[
    formWindowItems: MakeFWItems,
    menuItemProc: MyMenuItemProc,
    menuItems: [ done: TRUE, apply: FALSE, cancel: TRUE,
      defaults: TRUE, start: FALSE, reset: FALSE],
    title: @title,
    formWindowItemsLayout: DoLayout] };

```

--Omit the MakeItemsProc and the LayoutProc, since they are
-- just like those used with form windows.

```

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

```

<< Called when the user selects a command. If the command
is Done, then call ApplyAnyChanges to actually make the
changes; if Cancel, just return OK, since there is nothing to
change. If Defaults, then call a procedure to set defaults.>>

```

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

```

<< Update internal information based on the user's change to
the property sheet. This procedure is just a skeleton, because
the actual procedure relies on information presented in the
next few chapters. There is a complete example of this kind of
procedure in chapter 16, Icon Applications.>>

```

ApplyAnyChanges: PROC [fw:Window.Handle] RETURNS [ok:BOOL]
= {IF NOT FormWindow.HasAnyBeenChanged[fw] THEN
  RETURN [ok:TRUE];
  << check which items have changed. >>
  FOR myItem: Items IN Items DO
    itemKey: FormWindow.ItemKey = myItem.ORD;
    IF NOT FormWindow.HasBeenChanged [fw, itemKey]
    THEN LOOP;
    SELECT myItem FROM
    purgedeleteditems = > ...,
    ...; };
  ENDCASE;
  ENDOLOOP;
  RETURN [ok: TRUE];
};

```

--procedure called when user invokes defaults command

```

SetDefaults: PROC [window: Window.Handle] = {
  FormWindow.SetChoiceItemValue[
    window: window,
    item: Items.purgedeleteditems.ORD,
    newValue: 0, repaint: FALSE];
  FormWindow.Repaint>window: window];
};
}...

```

8.2 Linked property sheets

You can also create *linked property sheets*, which consist of several distinct property sheets that all share the same window. The user can only see one property sheet at a time; he selects which one to view from a choice item in an additional body window, called the *link window*. The link window remains visible at all times, while the main form window displays one of the possible choices.

The **Text Property Sheet** available with the document editor is an example of a linked property sheet. The link window contains choices for Character, Paragraph, and Tab Setting property sheets; selecting one of them displays the appropriate sheet. Figure 8.3 shows a generic linked property sheet with three possible property sheets: PSHEET1, PSHEET2, and PSHEET3.

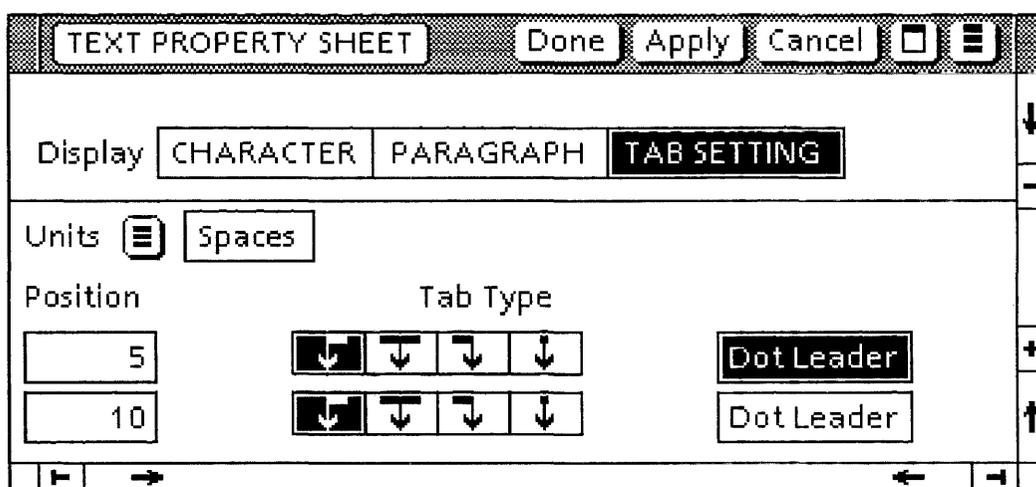


Figure 8.3: Linked property sheet

Note that you should only use linked property sheets when you have too much information to fit on a single property sheet. In general, a property sheet should occupy roughly $\frac{1}{4}$ of the screen. If you find that a property sheet is getting significantly larger than that, then you should use linked sheets or separate sheets or some other method of making the sheet smaller.

To create a linked property sheet, the first step is to write a **MakeItemsProc** and a **LayoutProc** for each of the individual property sheets. You do not call **Create** for each individual sheet, however. Instead, you call **PropertySheet.CreateLinked**. This procedure is just like **Create** except that it has two additional parameters: the **LayoutProc** and the **MakeItemsProc** for the link window:

```

PropertySheet.CreateLinked: PROCEDURE [
  formWindowItems: FormWindow.MakeItemsProc,
  menuItemProc: PropertySheet.MenuItemProc,
  size: Window.Dims, --preferred size of property sheet
  menuItems: PropertySheet.MenuItems ←
    PropertySheet.propertySheetDefaultMenu,
  title: XString.Reader ← NIL,
  placeToDisplay: Window.Place ← PropertySheet.nullPlace,
  formWindowItemsLayout: FormWindow.LayoutProc ← NIL,
  windowAttachedto: StarWindowShell.Handle ← [NIL],
  globalChangeProc: FormWindow.GlobalChangeProc ← NIL,
  display: BOOLEAN ← TRUE,
  linkWindowItems: FormWindow.MakeItemsProc,
  linkWindowItemsLayout: FormWindow.LayoutProc ← NIL
  clientData: LONG POINTER ← NIL,
  afterTakenDown: PropertySheet.MenuItemProc ← NIL,
  zone: UNCOUNTED_ZONE ← NIL]
RETURNS [shell: StarWindowShell.Handle];

```

The `formWindowItems` and the `formWindowItemsLayout` parameters should be the appropriate procedures for the property sheet that you want to display initially. `linkWindowItems` and `linkWindowItemsLayout` are the corresponding procedures for the *link window*.

The layout procedure for the link window is just like any other layout procedure.

The `linkWindowItems` procedure should create one item: the choice item that determines which property sheet is currently displayed. You create this item with a call to `MakeChoiceItem`. Recall from the previous chapter that one of the parameters to this procedure is a local change procedure that `ViewPoint` will call whenever the user selects a new choice from the choice item. This change procedure is responsible for swapping property sheets.

To swap property sheets, call `SwapFormWindows`:

```

PropertySheet.SwapFormWindows: PROCEDURE =
  shell: StarWindowShell.Handle, --the property sheet
  newFormWindowItems: FormWindow.MakeItemsProc,
  newFormWindowItemsLayout: FormWindow.LayoutProc ← NIL,
  apply: BOOLEAN ← TRUE,
  destroyOld: BOOLEAN ← TRUE,
  newMenuItemProc: PropertySheet.MenuItemProc ← NIL,
  newMenuItems: PropertySheet.MenuItems ← ALL[FALSE],
  newTitle: XString.Reader ← NIL,
  newGlobalChangeProc: FormWindow.GlobalChangeProc ← NIL,
  newAfterTakenDownProc: PropertySheet.MenuItemProc ←
    NIL,
  RETURNS [old: Window.Handle];

```

With the exception of `shell`, `apply`, and `destroyOld`, all of these parameters are the same as the parameters for a standard call to `PropertySheet.Create`.

`shell` is the property sheet. `apply` specifies whether you want to apply any changes to the old property sheet before you execute the swap.

`destroyOld` determines what happens to the old property sheet. If you don't destroy the old one, it is the return

parameter. You can then store this old sheet, and call `SwapExistingFormWindows` instead of `SwapFormWindows` the next time you want to create that property sheet. Whether you store the old property sheet is up to you; it is a simple trade-off of space for time. See the *ViewPoint Programmer's Manual* for the declaration of `SwapExistingFormWindows`.

Here is an example that creates a linked property sheet:

```
--the items in the link window, plus items in Sheet1.
Items: TYPE = {radix, BOOLEAN, choice, tag};

MakePropertySheet: PROC[...] = {
...
<< Create the linked sheet with a call to CreateLinked. Sheet1
is the sheet that is initially displayed. Omit the layout procs and
Makeltem procedures for the three linked property sheets,
since they are the same as in earlier examples. >>
pSheetShell: starWindowShell.Handle ←
  PropertySheet.CreateLinked [
    formWindowItems: MakeSheet1,
    menuItemProc: MakeMenuItems1,
    size: ...,
    linkWindowItems: MakeLinkWindowItems,
    linkWindowItemsLayout: NIL]; --Use default layout
                                -- for link window
  ];

<< This is the MakeltemsProc for the link window. It has just
one item, the choice item that determines which property
sheet is being displayed. Note the changeProc, which takes
care of switching the items. >>
MakeLinkWindowItems: FormWindow.MakeltemsProc = {

--declare necessary strings
sheet1, sheet2, sheet3, tag: XXstring.ReaderBody;
sheet1 ← XString.FromSTRING ["Sheet1"L];
sheet2 ← XString.FromSTRING ["Sheet2"L];
sheet3 ← XString.FromSTRING ["Sheet3"L];
tag ← XString.FromSTRING ["Current Sheet"L];

--set up the possible choices as the three strings
choices: ARRAY [0..3] OF FormWindow.ChoiceItem ← [
  [string[choiceNumber: 1, string: sheet1]],
  [string[choiceNumber: 2, string: sheet2]],
  [string[choiceNumber: 3, string: sheet3]]];

--Create an array of FormWindow.ChoiceItem. The tag will be
--Current Sheet, and the three choices will be Sheet1,
-- Sheet2, and Sheet3.
FormWindow.MakeChoiceItem [
  window:window,
  myKey:Items.radix.ORD, --items is the array of form items
  tag:@tag,
  values:DESCRIPTOR[choices],
  initChoice:1, --choiceNumber of first sheet displayed
  changeProc: ChangeFormWindow];
};
```

--Local change procedure that switches property sheets.

```
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;
```

*--the MakeItemsProc for the initial property sheet. Use
--messages this time.*

```
MakeSheet1: FormWindow.MakeItemsProc = {
  --make the boolean item
  FormWindow.MakeBooleanItem [
    window: window,
    myKey: Items.boolean.ORD,
    tag: Defs.Get[Defs.Key.tag],
    suffix: Defs.Get[Defs.Key.suffixBoolean],
    initBoolean: itemData.boolean,
    label: [string[
      (Defs..Get[Defs.Key.boolean])]];

  --make the choice item
  BEGIN
    choice1, choice2, choice3: XXstring.ReaderBody;
    choices: ARRAY [0..3] OF FormWindow.ChoiceItem ← [
      [string[choiceNumber: 1,
        string: Defs.Get[Defs.Key.psChoice1]],
      [string[choiceNumber: 2,
        string: Defs.Get[Defs.Key.psChoice2]],
      [string[choiceNumber: 3,
        string: Defs.Get[Defs.Key.psChoice3]]];
    FormWindow.MakeChoiceItem [
      window: window,
      myKey: Items.choice.ORD,
      tag: Defs.Get[Defs.Key.tagChoice],
      values: DESCRIPTOR [choices],
      initChoice: itemData.choice.ORD];
  END;

  --make the text item
  BEGIN
    FormWindow.MakeTextItem [
      window: window,
      myKey: Items.tag.ORD,
      tag: Defs.Get[Defs.Key.tagTag],
      width: 20,
      initString: @itemData.tag ];
  END;
};
```

8.3 Summary

To create a property sheet, you call `PropertySheet.Create` instead of `FormWindow.Create`. In addition to a `LayoutProc`, a `MakeItemsProc`, and a `GlobalChangeProc`, `PropertySheet.Create` has the following parameters:

- A `menultems` record, which specifies the items that are to appear in the header for the property sheet. The default is for Start and Cancel to appear.
- A `MenuItemsProc`, which is a call-back procedure to implement the commands in the property sheet header.

You can also create *linked property sheets*, which are held together by a *link window*. The link window is a form window with a choice item, which lists possible property sheets. To swap property sheets, you associate a change procedure with the choice item in the link window. The change procedure then calls `PropertySheet.SwapFormWindows` to do the swap.

For more information on the `PropertySheet` interface, see the `PropertySheet` chapter of the *ViewPoint Programmer's Guide*.

8.4 Exercise

The exercise for this chapter is an extension of the exercise for the last chapter. In the last chapter, you wrote the form window implementation for the Time Clock application; in this chapter, you need to write the property sheet implementation. If you didn't do the last exercise, you should go back and read the description of how the tool works.

Invoking the Print Report command creates the property sheet illustrated in Figure 8.4.

Figure 8.4: The Time Clock property sheet

Your assignment is to write the code to implement this property sheet. For a more complete description of what you need to do, see the module `TimeClockPSheetImplTemp.mesa`, which contains a template and comments for the code that you need to write.

Many applications require specialized interpretation of user input. For example, in a game that moves a piece through a maze, however, you might want mouse clicks or keystrokes to move the piece. This chapter describes the Terminal Interface Package (TIP), which provides basic user input facilities.

9.1 Overview

There are two named processes that respond to user actions: the *Stimulus* and the *Notifier*. The *Stimulus* watches the keyboard and mouse for user actions and enqueues them; its job is to ensure that no user actions are lost. The *Notifier* dequeues each action and associates it with a window. If the action is a mouse click, it goes to the window with the cursor; all other actions go to the window with the input focus.

Once it has determined the correct window for a user action, the *Notifier* decides how to interpret that action by checking for the 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 for each user action. There is typically a chain of *TIP tables* to handle various types of input.

The *Notifier* searches all *TIP tables* associated with a window until it finds a match or runs out of tables to check. If it doesn't find a match, it discards the action. If it finds a match, it passes the corresponding list of results to the application's *NotifyProc* procedure. The *NotifyProc* then executes some program action in response. Figure 9.1 illustrates this chain of events.

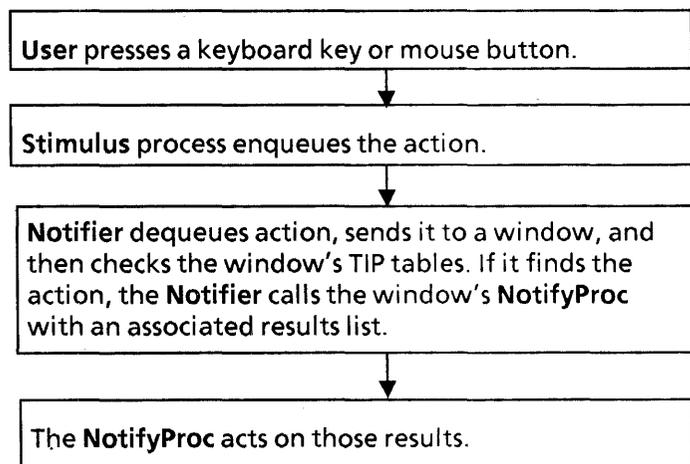


Figure 9.1: Path of user input

TIP tables and NotifyProcs thus control the way an application responds to user input; this chapter describes how to create and modify TIP tables and how to write NotifyProcs.

9.2 TIP tables

TIP tables define the way an application responds to user input. Since there can be a large number of TIP tables in the system, ViewPoint groups TIP tables into a structure based on the type of user action that they handle. The TIPStar interface defines this structure, which is based on the Placeholder:

```
TIPStar.Placeholder: TYPE = {mouseActions, keyOverrides,
    softKeys, keyboardSpecific, blackKeys, sideKeys,
    backstopSpecialFocus};
```

As their name implies, Placeholders are just categories, not actual TIP tables; they are effectively stacks onto which clients can add actual TIP tables. There can be a chain of TIP tables associated with any placeholder. Figure 9.2 illustrates the list of Placeholders, in the order in which they are checked.

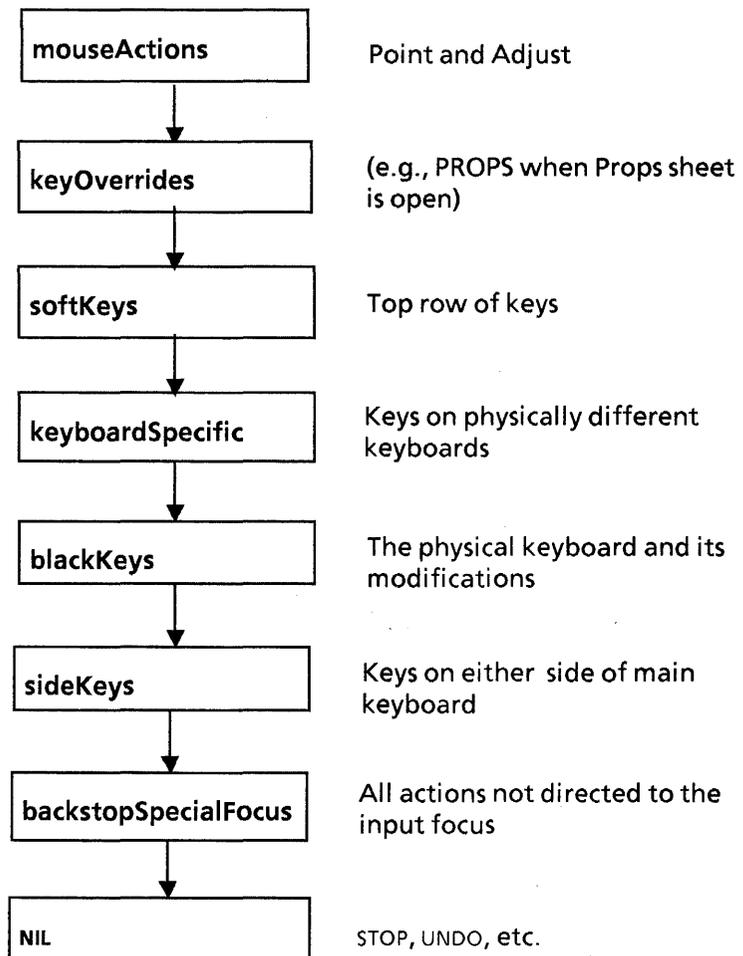


Figure 9.2: The Placeholder tables

Booting ViewPoint establishes an initial set of tables, called the *normal* tables. The new tables do not replace the placeholders;

they are added to the appropriate placeholder "stack", as illustrated in Figure 9.3

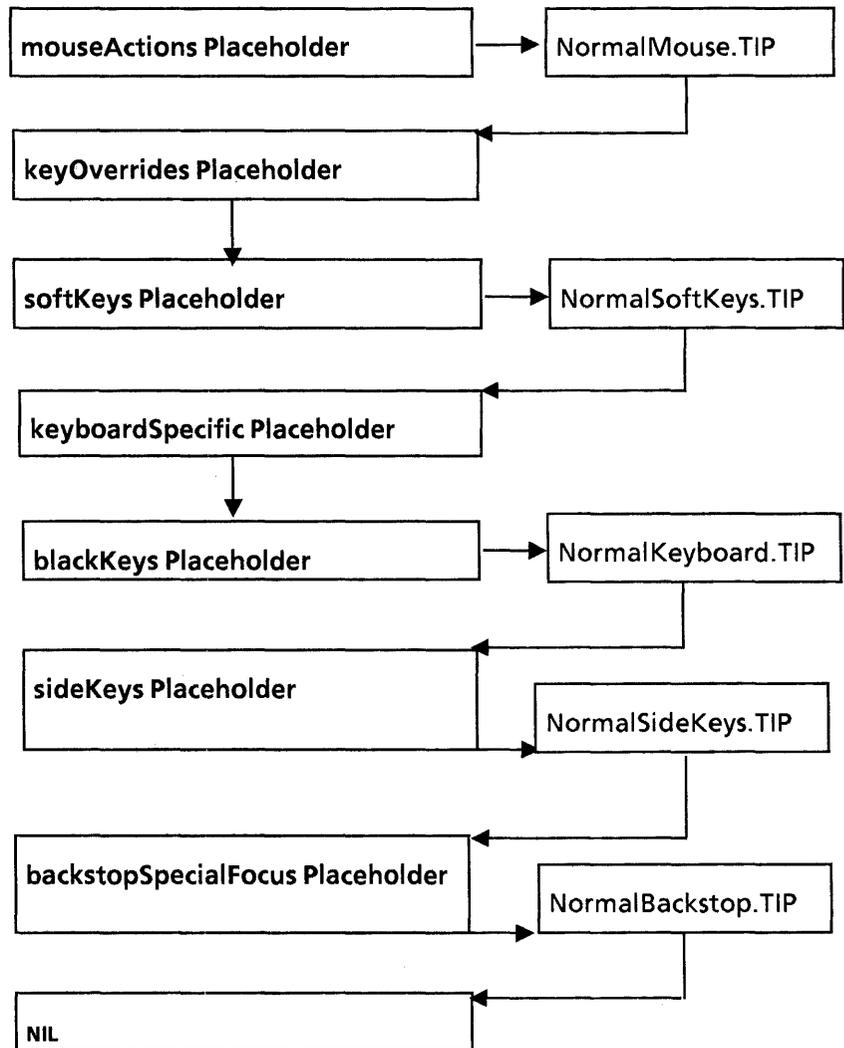


Figure 9.3: The normal tip tables

The normal TIP tables provide a standard interpretation of user actions. When you write an application, one choice is to have that application just use the standard TIP tables. However, if you want to change the interpretation of certain keystrokes or mouse actions, you can add a new table to one of these chains, as described later in the chapter. First, however, we discuss the syntax of a TIP table and the structure of a NotifyProc.

9.2.1 TIP table syntax

In its simplest form, a TIP table is a user-editable file with the extension ".TIP." TIP tables are stored in the system catalog.

The left hand side of a TIP table specifies *triggers* and *enablers*. 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. Here is part of the relevant syntax:

TriggerTerm :: = (Key | MOUSE | ENTER | EXIT) TimeOut

TimeOut :: = empty | BEFORE Number | AFTER Number

Key :: = KeyIdent UP | KeyIdent DOWN

Thus, the actions that can be in the left hand side of a TIP table are mouse movement (MOUSE), whether the mouse has entered or exited the window (ENTER and EXIT), time constraints, and key transitions. A **KeyIdent** can be any key or a mouse button. (See the TIP chapter of the *ViewPoint Programmer's Manual* for a complete list of the key names and for the complete syntax.) **Timeout** specifies a time interval within which the action must happen.

Enable actions are actions that have already happened, or that are in progress; enables are generally used to check the current state of a key. An **ENABLE** is thus similar to a **WHILE** statement.

The right hand side of a TIP table contains results, which are passed to the program's **NotifyProc**. We discuss results in the next section.

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;
  ENDCASE;
```

This TIP table matches the trigger action **Point 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 results are **COORDS** and **ShiftedDoubleClick**; otherwise, the results are **COORDS** and **NormalDoubleClick**. Similarly, If the right mouse button goes down less than 300 milliseconds after the left button, then the result is **PointAndAdjust**. If neither of these things happens after the left mouse button goes down, then the results are **COORDS** and **SimpleClick**.

9.2.2 Results

The results passed to a **NotifyProc** are structured into a linked list, of type **TIP.Results**:

TIP.Results: TYPE = LONG POINTER TO **TIP.ResultObject**;

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

For example, Figure 9.4 illustrates one possible chain of results for the TIP table discussed above.

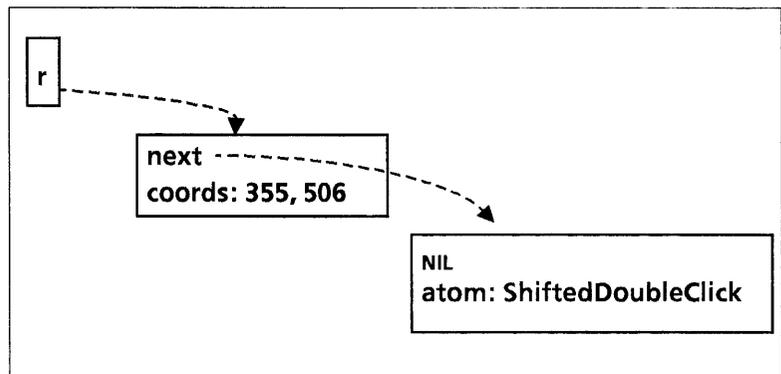


Figure 9.4: A possible results list

Each element of a results list is an object of type **ResultsObject**. Most of the variants of a **ResultObject** provide information about the current state of the keyboard or mouse. For example, **COORDS** is the current coordinates of the mouse, **key** represents the state of a particular key, and **time** measures time between user actions. (See the *ViewPoint Programmer's Manual* for a complete explanation of the variants.) These kinds of results are called information results, since they encapsulate information about the current state of the world.

However, a program also typically needs an action result, or some indication of what just happened. The most common way to convey this information is with an **atom**, which is essentially a unique string. For example, in the previous TIP table we used the atoms **ShiftedDoubleClick**, **NormalDoubleClick**, **PointAndAdjust**, and **SimpleClick**. Each of these are just terms that a particular program defines to represent a particular set of user actions.

Thus, a typical results list contains an atom and any necessary related information results. The convention is to pass "information" results first, and "action" results (atoms) last, so that you have the information result available when you implement the action. Thus, in the example above, we passed the information result **COORDS** before the atom. The next section contains an example that illustrates how the **NotifyProc** might use these results.

When you define an atom, you must create it and insure its uniqueness with a call to either **Atom.MakeAtom** or **Atom.Make**:

```
Atom.Make: PROCEDURE [pName: XString.Reader]
  RETURNS [atom: Atom.ATOM];
```

```
Atom.MakeAtom: PROCEDURE [pName: LONG STRING]
  RETURNS [atom: Atom.ATOM];
```

Make and **MakeAtom** both return an atom corresponding to the character string that you pass in; the only difference between them is the type of the argument that you pass in. The **Atom** interface will return the specified atom, creating a new one if necessary.

Here is an example that creates the atoms in the above table. Note that we allocate dynamically to keep the storage out of the global frame:

```
Atoms: PROGRAM = {
  z: UNCOUNTED ZONE = Heap.systemZone;
  atoms: LONG POINTER TO AtomRec ← NIL;
  AtomRec: TYPE = RECORD [
    PointAndAdjust, NormalDoubleClick, ShiftedDoubleClick,
    SimpleClick: Atom.ATOM];
  ...

  InitAtoms: PROCEDURE = {
    IF atoms = NIL THEN atoms ← z.NEW[AtomRec ← [
      PointAndAdjust:
        Atom.MakeAtom["PointAndAdjust"L],
      NormalDoubleClick:
        Atom.MakeAtom["NormalDoubleClick"L],
      ShiftedDoubleClick:
        Atom.MakeAtom["ShiftedDoubleClick"L],
      SimpleClick: Atom.MakeAtom["SimpleClick"L] ]];
    };
  ...
  InitAtoms;
};
```

9.3 The NotifyProc

When the Notifer process recognizes a user action in the left side of a TIP table, it passes the associated results list to a **NotifyProc**. 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];
```

Here is a possible **NotifyProc** for the above TIP table:

```

TIPMe: TIP.NotifyProc = {
  place: Window.Place;
  << Note the loop syntax here. This is the standard way to
  use a loop to go through a linked list. At the first iteration,
  the local variable input is set equal to the input parameter
  results. Each iteration of the loop looks at input.next until it
  reaches the end of the linked list.>>
  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;
  ENDLLOOP};

```

This procedure loops through the linked list of results. When it finds the information result `COORDS`, it stores the coordinates into the local variable `place`, and then loops. When it finds an action result (`atom`), it calls an appropriate procedure, passing in `place` when necessary. (This is why it is important to pass information results first from the TIP table; if you passed the `atom` first, the value of `place` would not be available.)

It is important to realize that the `NotifyProc` is called once for every successful match in the TIP table. The loops in the `NotifyProc` are there because the *results list* may have more than one element (e.g. `COORDS`, `NormalDoubleClick`), not because a series of user actions have been buffered.

9.4 Incorporating a new TIP table

If you want to write your own new TIP table, you also need to write code that makes your TIP table available to your application. This section describes the steps involved in that process.

9.4.1 Creating the compiled TIP table

The first step is to translate the text version of the TIP table into a program-readable "compiled" TIP table by calling `TIP.CreateTable`:

```

TIP.CreateTable: PROCEDURE [
  file: XString.Reader,
  z: UNCOUNTED_ZONE ← NIL,
  contents: XString.Reader ← NIL]
  RETURNS [table: TIP.Table];

```

```

TIP.Table: TYPE = LONG_POINTER TO TIP.TableObject;

```

```

TIP.TableObject: TYPE;

```

`CreateTable` generates a TIP table from the text file `file`. The storage for the table will come from `z`; if `z` is `NIL`, then the TIP implementation will use its own zone.

`contents` is the default contents of `file`. If `CreateTable` cannot read `file` or cannot parse `file` correctly, it will raise `InvalidTable`:

```
TIP.InvalidTable: SIGNAL [type: TIP.TableError,
    message: XString.Reader];
```

```
TIP.TableError: TYPE = {fileNotFound, badSyntax};
```

The `type` will be `badSyntax` if `CreateTable` could not parse the contents of `file`. `RESUME`ing the signal will cause TIP to write `contents` string in as the new contents of the file. If the `contents` string doesn't work either, then `CreateTable` will just return `NIL`, without raising the error again. Note, however, that the `file` parameter cannot initially be `NIL`, because TIP needs the name of a file to write the `contents` string into.

If the `contents` parameter is `NIL`, the `type` will be `fileNotFound`: the TIP file did not contain the correct information, and there is no backup in the `contents` string.

When `type = badSyntax`, the `message` parameter will contain the name of the bad TIP file.

Here is an example of calling `CreateTable`:

```
...
--declare strings for the title of the .TIP file and its contents
fileName: XString.ReaderBody ←
    XString.FromSTRING ["MyTipFile"L];
contents: XString.ReaderBody ← XString.FromSTRING["
SELECT TRIGGER FROM
    S Down = > TurnLeft;
    D Down = > TurnRight;
    K Down = > Forward;
    L Down = > Fire;
ENDCASE..."L];

--create the compiled version.
table: TIP.Table ← TIP.CreateTable[
    file: @fileName,
    contents: @contents! TIP.InvalidTable = > RESUME];

IF table = NILTHEN { --bad contents string
    error: XString.ReaderBody ← XString.FromSTRING [
        "Problem parsing "L];
    Attention.Post[@error];
    Attention.Post[@fileName] };
...

```

This example will parse the contents of `fileName` and generate the file `fileName.TIPC`. If there is something wrong with `file`, `CreateTable` will raise `InvalidTable`, which we `RESUME`. The `RESUME` writes the `contents` string into `file` and reparses it. If the `contents` string doesn't work either, then `CreateTable` does *not* raise the signal again; it just returns `NIL`. (This means that the `RESUME` will not cause an infinite loop.) Thus, we must check for `NIL` after the call to `CreateTable`. Since we provide a `contents` string, the error type `fileNotFound` will never be raised.

9.4.2 Associating tables and NotifyProcs

Once you have a table and a **NotifyProc**, you need to associate them with your application. (If you have just a **NotifyProc**, and don't need your own TIP table, you can just pass the **NotifyProc** as a parameter to **CreateBody**, and you don't need to worry about any of this.)

To associate a window, a table, and a **NotifyProc**, you call **SetTableAndNotifyProc**:

```
TIP.SetTableAndNotifyProc: PROCEDURE [
  window: Window.Handle,
  table: TIP.Table ← NIL,
  notify: TIP.NotifyProc ← NIL];
```

This procedure tells the **TIP** interface about the existence of your application's window, and associates a table and **NotifyProc** with it. If you want your application to use only your TIP table, and not any of the standard tables, you pass in your own table as **table**.

If, however, you want your application to recognize the standard TIP tables, as well as your own new special TIP table, then you should obtain the head of the list of standard tables, and use that value as **table**. To obtain the head of the table list (the **mouseActions** placeholder), call **TIPStar.NormalTable**:

```
TIPStar.NormalTable: PROCEDURE RETURNS [TIP.Table];
```

Calling **SetTableAndNotifyProc** with the head of the tables list associates your window with the standard set of TIP tables, and registers your **NotifyProc**. However, you still have to insert your TIP table somewhere in the tree of tables. To do this, you call **TIPStar.PushTable**:

```
TIPStar.PushTable: PROCEDURE [TIPStar.Placeholder, TIP.Table];
```

PushTable places the new table directly after the specified placeholder, without removing any of the existing tables. Figure 9.5 illustrates the effect of pushing a new table (**NewTableA.TIP**) onto the **mouseActions** placeholder.

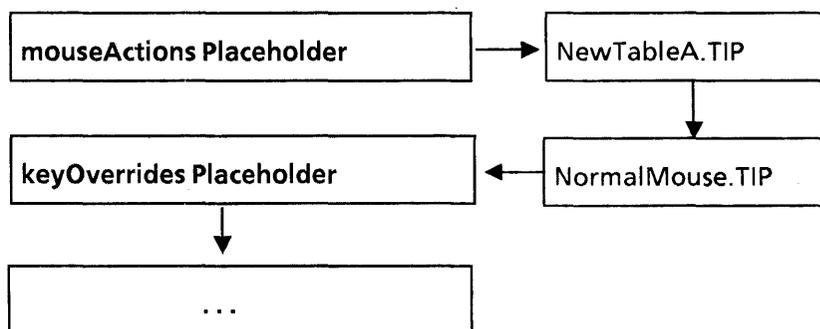


Figure 9.5: Pushing **NewTableA** onto **mouseActions**

Several different calls to **PushTable** will result in a stack of tables "hanging" from the Placeholder. The first table in the

chain will be the last one added. Figure 9.6 illustrates the effect of pushing a second new TIP table onto `mouseActions`.

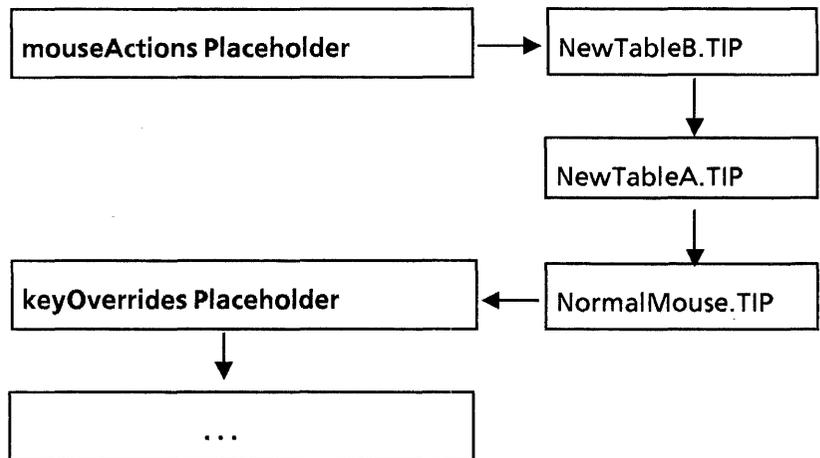


Figure 9.6: Pushing `NewTableB` onto `mouseActions`

Note: If you want to replace the old tables, rather than just add a new one, you can call `TIPStar.StoreTable` instead of `PushTable`. See the `TIPStar` chapter of the *ViewPoint Programmer's manual* for details.

The net effect of all of this is that your new table is not part of the standard TIP chain, and that your application is now a standard TIP client. When the user is not using your application, however, you should remove your TIP table from the tree so that your changes do not apply to all other applications as well. (Of course, if you want to change the interpretation of keystrokes for all applications, you can just call `PushTable` and leave it that way.)

You remove the table at the top of a particular placeholder with `PopTable`;

`TIPStar.PopTable`: PROCEDURE [`TIPStar.Placeholder`, `TIP.table`];

One standard approach is to call `PushTable` when the mouse enters your window, and `PopTable` when it exits the window; the example in the next section illustrates this.

9.2.3 Input focus

The Notifier directs mouse actions to the window containing the cursor, and keystrokes to the window containing the input focus. Thus, if you want your window to be able to accept keystrokes, you must make your application control the input focus by calling `SetInputFocus`:

```

TIP.SetInputFocus: PROCEDURE [
  w: Window.Handle,
  takesInput: BOOLEAN,
  newInputFocus: LosingFocusProc ← NIL,
  clientData: LONG POINTER ← NIL];
  
```

```
TIP.LosingFocusProc: TYPE = PROCEDURE [
  w: Window.Handle, data: LONG POINTER];
```

SetInputFocus makes your window the input focus; if you want your window to take type-in, you should set **takesInput** to TRUE. When you are about to lose the input focus, **ViewPoint** will call your **LosingFocusProc**; at this point, you can do any sort of clean up that you need to do, such as calling **TIPStar.PopTable**.

9.2.4 Example

Here are the relevant portions of a program that creates its own TIP table and **NotifyProc**.

```
TIPExampleImpl: PROGRAM ... = {
-- Declare global vars and types
zone: UNCOUNTED_ZONE ← Heap.Create [initial: 4];
context: Context.Type ← Context.UniqueType[];
atoms: LONG POINTER TO AtomRec ← NIL;
AtomRec: TYPE = RECORD [
  enter, exit, mouse, pointDown, pointMotion, pointUp:
  Atom.ATOM];
--initialization proc called from the mainline code
Init: PROC = {
  commandName: XString.ReaderBody ←
  XString.FromSTRING["TIPExample"L];
  Attention.AddMenuItem [
    MenuData.CreateItem [
      zone: sysZ,
      name: @commandName,
      proc: MenuProc] ];
  InitAtoms; };

--Initialize atoms; called from Init
InitAtoms: PROCEDURE = {
  IF atoms = NIL THEN atoms ← z.NEW[AtomRec ← [
    enter: Atom.MakeAtom["Enter"L],
    exit: Atom.MakeAtom["Exit"L],
    mouse: Atom.MakeAtom["Mouse"L],
    pointDown: Atom.MakeAtom["PointDown"L],
    pointMotion: Atom.MakeAtom["PointMotion"L],
    pointUp: Atom.MakeAtom["PointUp"L] ]];
};
```

```

MenuProc: MenuData.MenuProc = {
  name: XString.ReaderBody ←
    XString.FromSTRING["TIPExample"L];
  data: Defs.Data ← NIL;
  tipFile: XString.ReaderBody ←
    XString.FromSTRING["TIPExample.TIP"L];

  --This string represents the contents of tipFile.
  contents: XString.ReaderBody ← XString.FromSTRING["
  SELECT TRIGGER FROM
    MOUSE = > SELECT ENABLE FROM
      Point Down = > COORDS, PointMotion;
    ENDCASE;
    EXIT = > COORDS, Exit;
    ENTER = > COORDS, Enter;
    Point Down = > COORDS, PointDown;
    PointUp = > COORDS, PointUp;
  ENDCASE..."L];

  -- Create the TIP table
  table: TIP.Table ← TIP.CreateTable[
    file: @tipFile, contents: @contents!TIP.InvalidTable = >
      RESUME]];
  IF table = NIL THEN {
    error: XString.ReaderBody ←
      XString.FromSTRING["Bad syntax in TIP table"L];
    Attention.Post[@error];
    RETURN};

  -- Create the StarWindowShell, body window, etc.
  shell ← StarWindowShell.Create [name: @name];
  body: Window.Handle ← StarWindowShell.CreateBody [
    sws: shell,
    box: [ place: [0,0],
      dims: Defs.bodyWindowDims ]];
  --(Set up the form window, pop up menus, ; allocate
  -- context, etc.)

  << Make application a TIP client and have it use the
  standard tables. Any actions that newtable does not handle
  will thus be checked against normal TIP tables. >>
  TIP.SetTableAndNotifyProc [
    window: body,
    table: TIPStar.NormalTable[],
    notify: MyNotifyProc];

  StarWindowShell.Push [shell] };

```

```

-- Handle user input
MyNotifyProc: TIP.NotifyProc = {
  data: Defs.Data ← Defs.GetContext[parent];
  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
      pointMotion = > PointMotion[window, data, place];
      enter = > {TIP.SetInputFocus[
        w: window, takesInput: FALSE];
        TIPStar.PushTable[mouseActions, table]];
      exit = > {
        TIPStar.PopTable[mouseActions, table]];
      pointDown = >
        {data.oldCursorPos ← place;
        Defs.SelectItem[window, data]];
      pointUp = > PointUp[parent, data, place];
    ENDCASE;
  ENDCASE; -- WITH z: input
  ENDLOOP;
};

--The rest of the procedures that do the actual work
...

-- Main line code
Init[];
END.

```

9.5 Periodic notifiers

The Notifier process is important because it responds directly to the user. When the user invokes a command, and a process acts on that command, that process is “in the Notifier.” Only one process can be in the Notifier at a given time, and that process is guaranteed that the Notifier will not process another user action until it has completed.

This notification mechanism has some important consequences for program design.

First, if you will be processing a command that will take a long time to execute, you should FORK it from your **NotifyProc** to avoid tying up the Notifier. (If for some reason you must tie up the Notifier, you should turn the cursor into an hourglass to indicate this to the user.)

Second, you need to think carefully about which operations must be executed from the Notifier to guarantee that there is no interference. A good example of such an operation is setting the selection: when the user asks to “select” a certain object, the process that is responsible for implementing the selection (highlighting, etc.) must be guaranteed that no other user action (such as one that acts on the selection or changes the selection) can interfere.

When you are responding to a user action (in your **NotifyProc**), you are guaranteed to be in the Notifier, and you have nothing

to worry about. However, there are times when you are not processing a command, but you still want to have your action run from the Notifier to guarantee that there is no interference.

To be able to execute in the Notifier when you are not directly processing a user action, you can create a call back procedure that will be called from the Notifier at regular intervals:

```
TIP.CreatePeriodicNotify: PROCEDURE [
  window: Window.Handle,
  results: TIP.Results,
  milliSeconds: CARDINAL,
  notifyProc: TIP.NotifyProc ← NIL]
  RETURNS [TIP.PeriodicNotify];
```

```
TIP.PeriodicNotify: TYPE [1];
```

CreatePeriodicNotify registers a periodic notify procedure. The specified **notifyProc** is called from the Notifier with parameters **window** and **results** once every **milliSeconds** milliseconds, as long as no user action notifications are taking place. (If **notifyProc** is **NIL**, it defaults to the **NotifyProc** associated with **window**.) For example, suppose that for some reason you want to keep a count of the number of windows on the screen. When you want to examine or change this value, you must do so from the Notifier:

```
count: Atom.ATOM ← Atom.MakeAtom["UpdateCount"];
results: TIP.ResultsObject ← [
  next: NIL, body: atom[a: count]];
notifier: TIP.PeriodicNotify ← TIP.CreatePeriodicNotify[
  window: window,
  results: @results,
  milliSeconds: 20000];
```

<< The NotifyProc associated with the window. Since we didn't specify a NotifyProc in the call to CreatePeriodicNotifier, this one will be used. >>

```
MyNotifyProc: TIP.NotifyProc = {
  input: TIP.Results;
  FOR input ← results, results.next DO
    WITH z: input SELECT FROM
      atom = >
        IF z.a = count THEN { --do something
        ELSE { --do something else
      ENDCASE;
  ENDOLOOP;
```

This example creates a periodic notifier that will be called every 20,000 milliseconds. The first step is to declare a **ResultsObject** that contains only the atom **count**. Each time the **NotifyProc** is called, it looks at the value of the **count** and acts accordingly.

If you make a call to **CreatePeriodicNotify** with **milliSeconds = 0**, then the process runs once and destroys itself. This is known as a *kamikaze* notify proc.

9.6 User aborts

TIP also provides facilities for checking whether the user has aborted your application. One way to do this is to associate an **AttentionProc** with your window by calling **TIP.SetAttentionProc**:

```
TIP.SetAttention: PROC [
    window: Window.Handle,
    attention: TIP.AttentionProc];
```

```
TIP.AttentionProc: TYPE = PROC [window: Window.Handle];
```

An **AttentionProc** is called whenever the user presses the **STOP** key. (Note that it is not called from the Notifier.) You associate an **AttentionProc** with your window

If you don't associate an **AttentionProc** with your window, the system keeps a user abort flag that records whether the user has pressed the **STOP** key. You can check that flag at any time by calling **TIP.UserAbort**:

```
TIP.UserAbort: PROC [Window.Handle] RETURNS [BOOLEAN];
```

To check if the user has pressed the **ABORT** key over a particular window, pass in a handle to that window. To check if the user has pressed **ABORT** anywhere (a global abort), pass in **NIL**. For example, code to check for a global abort might look like this:

```
IF TIP.UserAbort[NIL] THEN GOTO GlobalAbort;
```

The abort flag for a window is cleared whenever a non-shift key goes down or whenever a notification is sent to the window. Once you have looked at the abort flag and acted on it, you should call **ResetUserAbort** to set the flag back to **FALSE**.

```
TIP.ResetUserAbort: PROC [Window.Handle];
```

9.7 Summary

The **TIP** interface provides facilities for translating user actions into program actions. The basic scheme is that the *Stimulus* process enqueues user actions and the *Notifier* process dequeues them and directs them to a window. The Notifier then looks up the action in a TIP table (or TIP tables) associated with the window. If it finds the action, it passes associated *results* to a *NotifyProc*, which acts on those results.

To write a new TIP table for an application, you must call **Tip.CreateTable** to create a program-readable version of the TIP file, **SetTableAndNotifyProc** to make your table a TIP client and set the *NotifyProc*, and **TIPStar.PushTable** to insert the new TIP table into the existing tree of TIP tables. Later, you should call **TIPStar.PopTable** to remove your new table from the tree.

TIP also provides a *periodic notification* mechanism, which allows you to provide a call-back procedure that the Notifier will call. This allows you to avoid multi-process interference. When you are designing your programs, you should think

about whether you need to use this technique to avoid multi-process interference.

TIP also provides many other user-input facilities; for more information about TIP, see the **TIP** and **TIPStar** chapters of the *ViewPoint Programmer's Manual*. Appendix A of the *ViewPoint Programmer's Manual* contains more information on the normal TIP tables.

9.8 Exercise

The exercise for this chapter is Tank, which plays the classic video game of Tank. To play this game, you first decide how many enemies you want to have by choosing a value from one to five from the Number of Enemies menu. (Note: depending on the size of your window, this command may appear in the header of the window or it may be in the auxiliary menu for the tool.) You can also increase or decrease the speed of the tanks by invoking Faster or Slower.

When you are ready to start playing, invoke **Start Game**. This command draws the "battlefield" and the tanks, as illustrated in Figure 9.7. (The octagonal tank is your tank; the others are enemy tanks.)

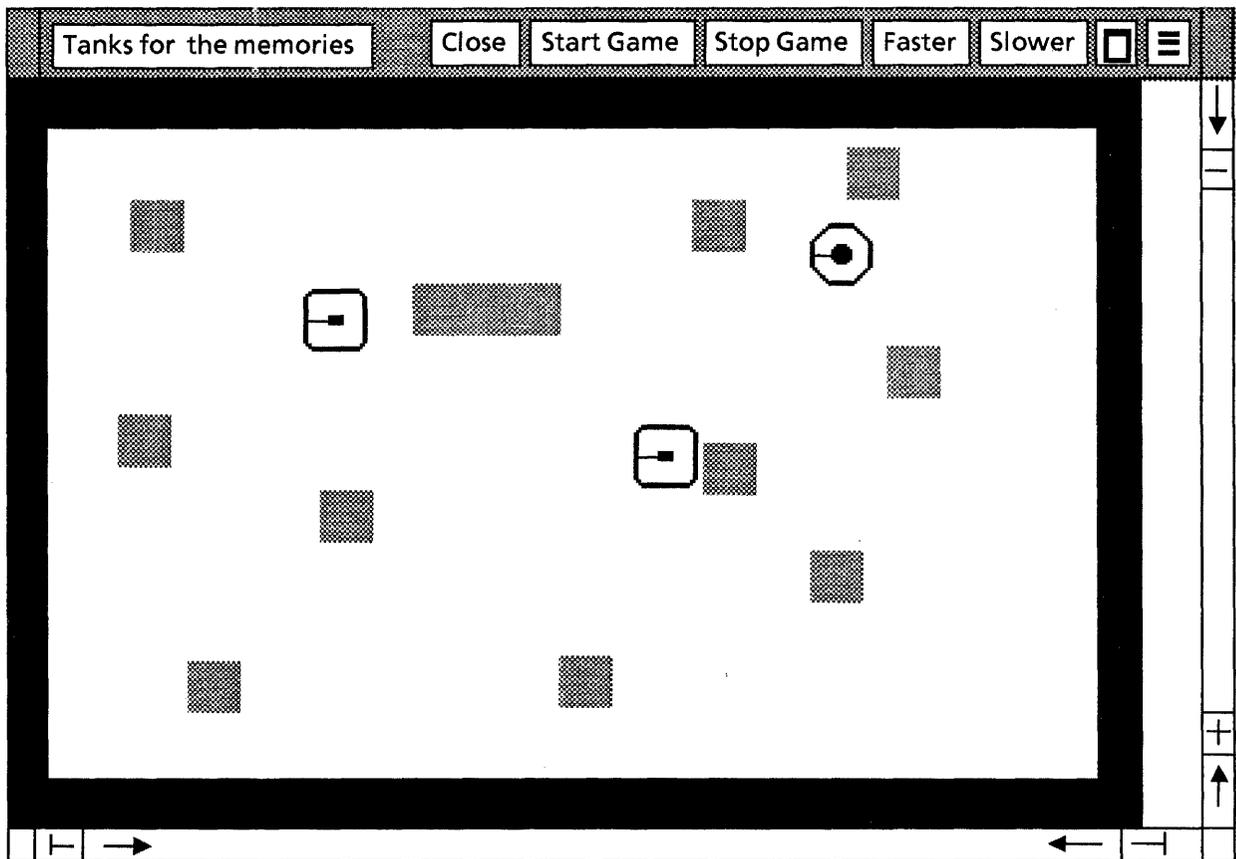


Figure 9.7: The Tank application

The battlefield is of fixed size and contains a tank representing the user, the tank(s) for the computer, and gray areas that

serve as barriers. Any tank can destroy a gray area by shooting at it. You can stop the game at any time by invoking **Stop Game**. Otherwise, the game is over when either you have destroyed all of the enemy's tanks or you have been destroyed by the enemy.

When you start the game, you have a short amount of time to move and or get the first shot in before the other tanks come to life. The enemy tanks will always move and shoot in your direction.

You can move your tank with the following keyboard keys:

s	= >	turn left (rotate counterclockwise)
d	= >	turn right (rotate clockwise)
k	= >	move forward
l	= >	shoot

Your assignment is to write the code for the following four procedures in TankTIPimplTemp:

InitAtoms initializes the atom used in the NotifyProc

LostInputFocus gets called when the tool window loses the input focus (let the user know this by posting a message in the Attention Window.)

MyNotifyProc interprets the atoms passed in and calls the appropriate procedures.

SetUpTipTable creates the TIP table for user actions.

You will also need the following modules:

- TankDefs
- TankGraphicImpl
- TankMsgImpl
- TankImpl
- Tank.config

Notes:

Every ViewPoint volume contains a tree-structured directory of files that you can manipulate with the ViewPoint filing system, called NSFiling. This chapter and the next three chapters discuss ViewPoint files in detail.

10.1 Content and attributes

ViewPoint files consist of two types of information: *content* and *attributes*. The *content* of a file is the actual data in the file. *Attributes* are pieces of information associated with the file that help identify the file, describe its structure or behavior, and record other information about the file. Figure 10.1 illustrates a typical file, with some content and four attributes.

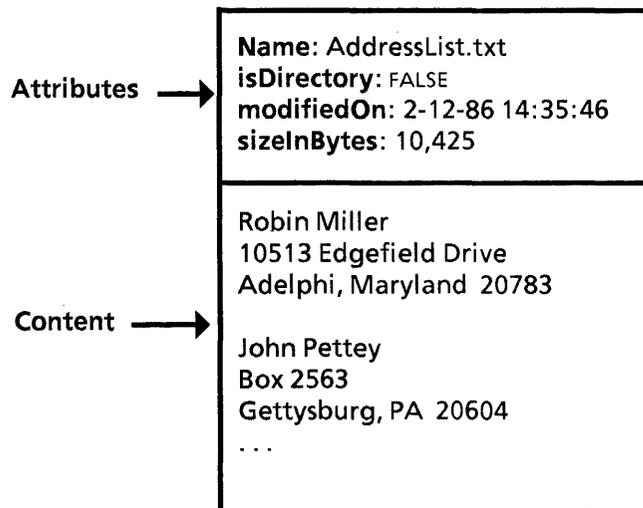


Figure 10.1: An NSFile

This chapter concentrates on specifying and accessing file *attributes*. The next chapter, *NSFiling*, discusses basic filing operations, such as naming, creating, opening, closing and deleting files. Chapter 12, *Streams*, and Chapter 13, *NSSegment*, discuss how to access the content of a file.

10.2 Interpreted and uninterpreted attributes

There are two major classes of attributes: *interpreted* and *uninterpreted*. Interpreted attributes are attributes that have a specific meaning to the file system; uninterpreted attributes are additional attributes that you define; they do not have a specific meaning to the file system.

The file system defines a wide range of interpreted attributes. Not all attributes apply to all files, but you can associate as many applicable attributes with a file as you like. Here is a brief summary of the major types of interpreted attributes:

Identity attributes serve to identify a file. Examples of identity attributes are **name**, **version**, **service**, and **file type**.

File attributes describe basic characteristics of a file. Examples of file attributes are **checksum**, **isDirectory**, and **parentID**.

Activity attributes record the date and time of significant events in the life of a file, and the name of the user on whose behalf the event occurred. Examples of activity attributes are **createdBy**, **filedOn**, **modifiedOn**, and **readBy**.

Size attributes record the size of a file. There are two possible size attributes: **sizeInPages** and **sizeInBytes**.

Access attributes specify the access restrictions of a file. To use these attributes, you make access lists and give a particular access, such as **fullAccess**, **noAccess**, or **readAccess**, to the list.

Directory attributes apply only to directory files. They describe useful characteristics of a directory. Examples of such attributes are **numberOfChildren**, **ordering**, and **defaultOrdering**.

The above list is not an exhaustive one, but it does cover the major kinds of attributes. In general, the file system maintains the value of interpreted attributes. For example, when a client opens a file with write access, the file system will update the **modifiedBy** attribute as a side-effect. (You can also explicitly change the value of an interpreted attribute via a procedure call; see Section 10.4.)

Uninterpreted attributes, on the other hand, do not have a specific meaning to the file system. You define and maintain your own uninterpreted attributes (also called *extended attributes*), depending on your application. For example, if you were implementing a mail system, you might want to use extended attributes to keep information like "sender," "time received," "return path" and the like. Section 10.3.2 discusses how to define extended attributes.

10.3 Specifying attributes

When you create a file, you specify the interpreted attributes that you want the file system to maintain for that file. The file system will maintain only the attributes you request; it does not automatically maintain all possible attributes for all files.

Attributes are specified with variant records; an **NSFile.Attribute** can take on a number of different values, depending on the variant that you select. When you create a file, you pass an **NSFile.AttributeList**, which is a descriptor for an array of variant records representing the attributes that you would like that file to have. Here are the relevant type declarations:

```

NSFile.Attribute: TYPE = MACHINE DEPENDENT RECORD [
  var: SELECT type: NSFile.AttributeType FROM
    fileID, parentID = > [value: NSFile.ID],
    checksum = > [value: CARDINAL],
    type = > [value: NSFile.Type],
    position = > [value: NSFile.Position],
    service = > [value: NSFile.Service],
    ordering = > [value: NSFile.Ordering],
    accessList, defaultAccessList = >
      [value: NSFile.AccessList],
    backedUpOn, createdOn, filedOn, modifiedOn, readOn
      = > [value: NSFile.Time],
    createdBy, filedBy, modifiedBy, readBy = >
      [value: NSFile.String],
    name, pathname = > [value: NSFile.String],
    childrenUniquelyNamed, isDirectory, isTemporary = >
      [value: BOOLEAN],
    version, numberOfChildren = > [value: CARDINAL],
    sizeInBytes, sizeInPages, subtreeSize, subtreeSizeLimit
      = > [value: LONG CARDINAL],
    extended = > [type: NSFile.ExtendedAttributeType,
      value: NSFile.Words],
  ENDCASE];

NSFile.AttributeList: TYPE = LONG DESCRIPTOR FOR ARRAY OF
  NSFile.Attribute;

```

Thus, you specify a set of attributes with an array of **Attribute** records, one for each attribute that you want your file to have.

10.3.1 Specifying interpreted attributes

Here is some code that specifies a set of interpreted attributes:

```

--declare the attributes of interest
nsFile: NSstring.String ←
  NSstring.StringFromMesaString["file"L];
myVersion: CARDINAL ← 1;
myType: NSFile.Type ← 110010;

--store them in an array of records
attributes: ARRAY [0..3] OF NSFile.Attribute ← [
  [name[nsFile]],
  [version[myVersion]],
  [type[myType]]];

```

This code will produce an array of three **Attributes**, as illustrated in Figure 10.2.

attributes	0	1	2
	name value: nsFile	version value: myVersion	type value: myType

Figure 10.2: ARRAY OF NSFile.Attributes

Later, you will need to create a **DESCRIPTOR** for this array (an **NSFile.AttributeList**), and pass the array descriptor to various procedures in the **NSFile** interface. We discuss how to use these attribute lists in the next chapter; for now you only need to worry about the syntax of specifying an attribute list. (If you are not comfortable with variant record syntax, you might want to review the appropriate chapter of the *Mesa Language Manual* or the *Mesa Course*.)

You should note the file type attribute. All **NSFiles** must have a file **type**, which is a **LONG CARDINAL**. Each distinct application has a distinct file type; files with similar function or multiple instances of an application use the same type. A file type is thus an important identity attribute. (You obtain a file type for a new application from the Xerox filing group.)

10.3.2 Specifying extended attributes

The last variant in the declaration of **NSFile.Attribute** is **extended**, which allows you to declare extended attributes. An extended attribute is a record with two fields: **type** and **value**.

A **type** is just a **LONG CARDINAL**:

```
NSFile.ExtendedAttributeType: TYPE = LONG CARDINAL;
```

A **type** is a unique identifier for a particular extended attribute. All attributes, both interpreted and uninterpreted, have a unique attribute type. An attribute type identifies a particular attribute, much as a file type identifies a particular file. You typically don't need to be aware of the type of an interpreted attribute, since interpreted attributes have names, but you do need to be aware of the attribute type for an extended attribute, since extended attributes do not have names. An attribute type is the only way to distinguish among different extended attributes.

The **value** of an extended attribute is the actual attribute information. The value is stored in an encoded form. **NSFile** provides procedures that encode/decode **BOOLEANS**, **LONG CARDINALS**, **INTEGERS**, **LONG INTEGERS**, **NSStrings**, and **NSFile.References**. Thus, when you want to store an extended attribute, you need to call the appropriate encoding procedure, and when you want to examine an extended attribute you need to call a decoding procedure. For example, the procedures to encode and decode **CARDINALS** look like this:

```
NSFile.Words: TYPE = LONG DESCRIPTOR FOR ARRAY OF UNSPECIFIED;
```

```
NSFile.EncodeCardinal: PROCEDURE [c: CARDINAL]  
  RETURNS [NSFile.Words];
```

```
NSFile.DecodeCardinal: PROCEDURE [NSFile.Words]  
  RETURNS [c: CARDINAL];
```

Here is a code segment that stores two extended attributes, a **LONG STRING** and a **CARDINAL**:

```

stringAttribute: NSFile.ExtendedAttributeType = 42332;
cardinalAttribute: NSFile.ExtendedAttributeType = 23231;
myString: NSString.String ←
  NSString.StringFromMesaString["aString"L];
myCardinal: CARDINAL ← 99; -- arbitrary values

```

--store the encoded values for string and cardinal into
--the records, and create an array of the records.

```

newAttributes: ARRAY [0..2] OF NSFile.Attribute ← [
  [extended
    type: stringAttribute,
    value: NSFile.EncodeString[myString]],
  [extended
    type: cardinalAttribute,
    value: NSFile.EncodeCardinal[myCardinal]]];

```

This code creates an array of attribute records, as illustrated in Figure 10.3. (Note that the figure shows the actual values for the attributes, whereas they are actually stored in an encoded form.)

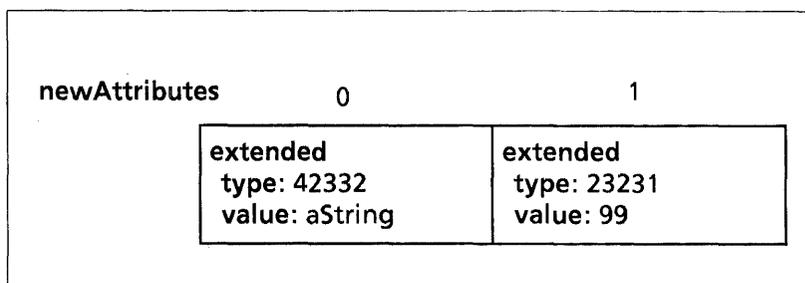


Figure 10.3: ARRAY OF NSFile.Attributes

The attribute types in this example are arbitrary. You can choose any value you like for an extended attribute type, as long as it does not conflict with any other attribute type (either interpreted or uninterpreted). Typically, you should consult other members of your group to see if they have a standard strategy for allocating such types. The range of interpreted attribute types is documented in the Filing Programmer's Manual. It is good practice to define extended attributes in an interface so that you can access them from other modules.

10.4 Getting interpreted attributes

The NSFile interface also provides procedures that allow you to inspect and modify attributes of a given file. To retrieve attributes, call NSFile.GetAttributes; to set new attributes, call NSFile.ChangeAttributes:

```

NSFile.GetAttributes: PROCEDURE [
  file: NSFile.Handle,
  selections: NSFile.Selections,
  attributes: NSFile.Attributes,
  session: NSFile.Session ← NSFile.nullSession];

```

```

NSFile.ChangeAttributes: PROCEDURE [
    file: NSFile.Handle,
    attributes: NSFile.AttributeList,
    session: NSFile.Session ← NSFile.nullSession];
    
```

file is a file handle, which is just a way to specify and access a particular file; we discuss file handles in the next chapter.

selections specifies the attributes of interest, and attributes provides the storage for those attributes. The following sections provide a detailed explanation of the selections, attributes, and session parameters, as well as a complete example illustrating both of these procedures.

10.4.1 Selections

The selections parameter is of type NSFile.Selections:

```

NSFile.Selections: TYPE = RECORD [
    interpreted: NSFile.InterpretedSelections ←
        NSFile.noInterpretedSelections,
    extended: NSFile.ExtendedSelections ←
        NSFile.noExtendedSelections];
    
```

```

NSFile.InterpretedSelections: TYPE = PACKED ARRAY
    NSFile.AttributeType OF NSFile.BooleanFalseDefault;
    
```

```

NSFile.AttributeType: TYPE = MACHINE DEPENDENT {checksum,
    childrenUniquelyNamed, createdBy, createdOn, fileID,
    isDirectory, isTemporary, modifiedBy, modifiedOn, name,
    numberOfChildren, ordering, parentID, position, readBy,
    readOn, sizeInBytes, type, version, accessList,
    defaultAccessList, pathname, service, backedUpOn, filedBy,
    filedOn, sizeInPages, subtreeSize, subtreeSizeLimit,
    extended};
    
```

```

NSFile.BooleanFalseDefault: TYPE = BOOLEAN ← FALSE;
    
```

This parameter specifies the attributes that you want to retrieve. As illustrated in Figure 10.4, a Selections record contains an array of booleans corresponding to the possible interpreted attributes and a descriptor for an array of extended attributes. (For now, ignore the extended attributes; section 10.5 discusses how to retrieve extended attributes.)

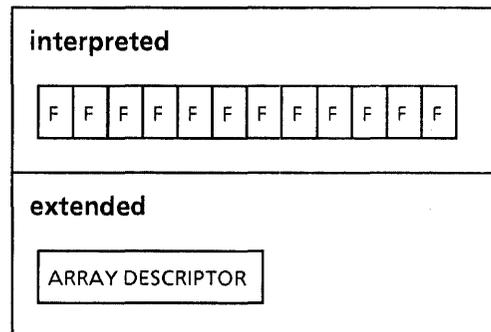


Figure 10.4: An NSFile.Selections record

To show interest in particular interpreted attributes, you set those attributes to `TRUE` in the array. `GetAttributes` will retrieve those attributes and store them in the `attributes` parameter.

10.4.2 Attributes

The `attributes` parameter to `GetAttributes` is of type `NSFile.Attributes`:

```
NSFile.Attributes: TYPE = LONG POINTER TO  
NSFile.AttributesRecord;
```

```
NSFile.AttributesRecord: TYPE = RECORD [  
fileID: NSFile.ID,  
service: NSFile.Service,  
name: NSFile.String,  
pathname: NSFile.String,  
version: CARDINAL,  
checksum: CARDINAL,  
type: NSFile.Type,  
isDirectory: BOOLEAN,  
isTemporary: BOOLEAN,  
parentID: NSFile.ID,  
position: NSFile.Position,  
backedUpOn: NSFile.Time,  
createdOn: NSFile.Time,  
filedOn: NSFile.Time,  
modifiedOn: NSFile.Time,  
readOn: NSFile.Time,  
createdBy: NSFile.String,  
filedBy: NSFile.String,  
modifiedBy: NSFile.String,  
readBy: NSFile.String,  
sizeInBytes: LONG CARDINAL,  
sizeInPages: LONG CARDINAL,  
accessList: NSFile.AccessList,  
defaultAccessList: NSFile.AccessList,  
ordering: NSFile.Ordering,  
childrenUniquelyNamed: BOOLEAN,  
subtreeSizeLimit: LONG CARDINAL,  
subtreeSize: LONG CARDINAL,  
numberOfChildren: CARDINAL,  
extended: NSFile.ExtendedAttributeList];
```

The purpose of the `attributes` record is to provide storage for the attributes being retrieved. `GetAttributes` will copy the specified attributes into your `attributes` record; you can then do anything you like with those attributes.

The `attributes` parameter to `SetAttributes` is of type `NSFile.AttributeList`, which is a descriptor for an array of attributes. (See section 10.3.) Be careful to distinguish among these data types, which all have similar names.

10.4.3 Sessions

Before it can use the file system, a client must log on via the Authentication Service. (Note that it is actually the user who is authenticated, not the client.) Once the client has been

authenticated, the file system establishes a *session*, which is identified by a session handle. The file system returns the session handle to the client, which can then use the session handle to identify itself in future calls to the file system.

Logging in to ViewPoint establishes a default session. The filing system will use this session in all subsequent filing operations unless you specifically request a different session handle. In the declaration of `GetAttributes`, `session` has a default value of `nullSession`. This default is really the default session, and not a null session. In this chapter we use only the default session handle; for more information on sessions, see the *Filing Programmer's Manual*.

10.4.4 Storage management

Another thing you need to know about `GetAttributes` is that it allocates storage from the system zone. Thus, you need to call `NSFile.FreeAttributes` after a call to `GetAttributes` to release that storage. The example in the next section illustrates this.

10.4.5 Example of `GetAttributes` and `SetAttributes`

Here is an example that obtains a file's attributes and then changes the file's name and version number.

```
newVersion: CARDINAL;
newName: NSString.String ←
    NSString.StringFromMesaString["newNameForFile"L];
fileHandle: NSFile.Handle ← --GetFileHandleSomeHow;

--storage for old attributes being retrieved
myAttributes: NSFile.AttributesRecord;
--storage for new attributes
attributeArray: ARRAY [0..2] OF NSFile.Attribute;
--create a selections record and set the ones we want to TRUE
selections: NSFile.Selections = [interpreted: [
    name: TRUE, version: TRUE]];

NSFile.GetAttributes[
    file: fileHandle,
    selections: selections, -- select name and version only
    attributes: @myAttributes]; -- retrieved attributes

--Put the new attributes in attributeArray
newVersion ← myAttributes.version + 1;
attributeArray ← [name[newName], version[newVersion]];
--store the new attributes
NSFile.ChangeAttributes[file: fileHandle,
    attributes: DESCRIPTOR[attributeArray]];

NSFile.FreeAttributes[@myAttributes]; --important!
```

The first step is to set up the parameters to `GetAttributes`. `myAttributes` is the storage for the attributes being retrieved, and `selections` specifies the attributes of interest. Figure 10.5 illustrates these data structures.

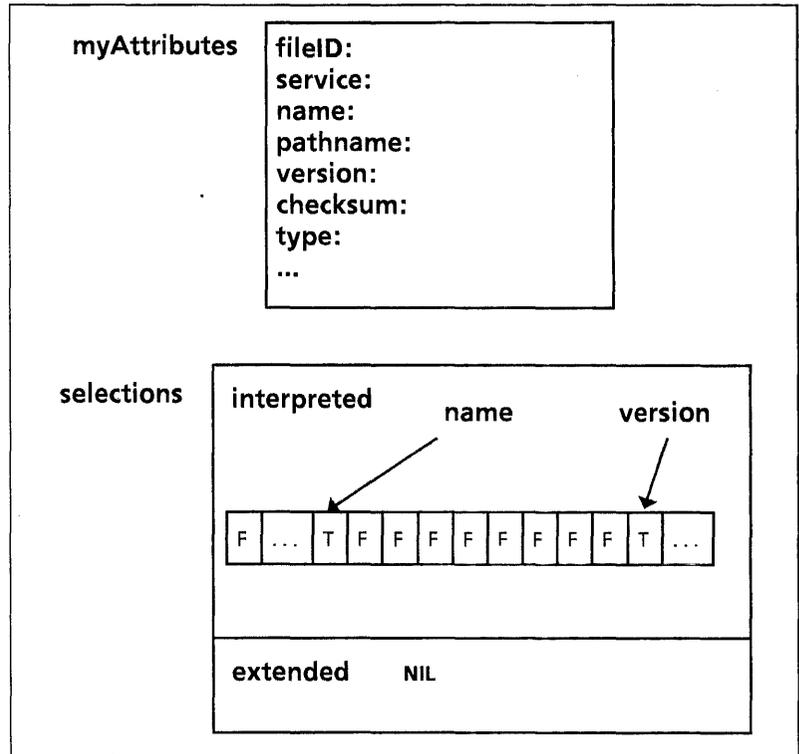


Figure 10.5: selections and myAttributes

The call to **GetAttributes** will store the specified attributes in **myAttributes**, as illustrated in Figure 10.6.

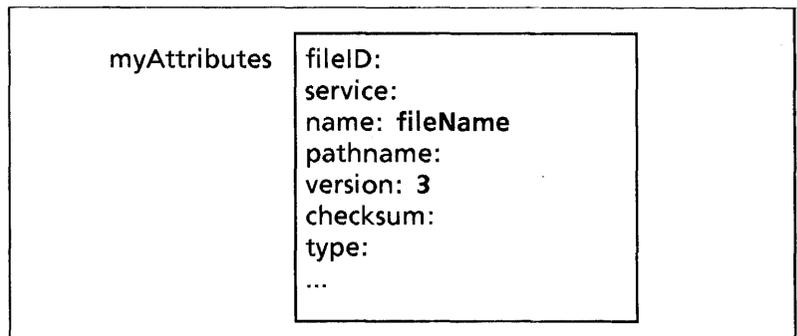


Figure 10.6: After the call to **GetAttributes**

The next step is to look at the old version number, update it, and then store new attributes with a call to **SetAttributes**. The **attributes** parameter to **SetAttributes** is a descriptor for the array of attributes illustrated in Figure 10.7.

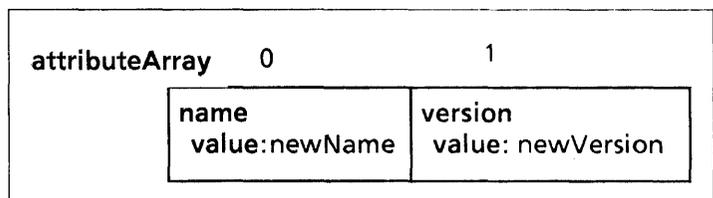


Figure 10.7: attributeArray

The final step is to call `FreeAttributes`. Although `myAttributes`, `selections`, and `attributeArray` all come from the local frame and will go away when the procedure returns, `GetAttributes` allocates string items from the `systemZone` and places just a pointer to the string in the `AttributesRecord`. Thus, you need to call `FreeAttributes` to free this storage from the `systemZone`.

10.5 Getting extended attributes

You can also call `GetAttributes` to retrieve extended attributes. With extended attributes, the `selections` parameter is a descriptor for an array of attribute types, rather than an array of booleans. (See §10.4.1 for the declaration of `Selections`.)

```
NSFile.ExtendedSelections: TYPE = LONG DESCRIPTOR FOR ARRAY
    CARDINAL OF NSFile.ExtendedAttributeType;
```

Looking at the value of the attributes once you have retrieved them is also a little more complicated with extended attributes, since the values are encoded. You need to search for the desired extended attribute type and then call a decoding procedure to extract the corresponding value. For example:

```
stringAttribute: NSFile.ExtendedAttributeType = 42332;
cardinalAttribute1: NSFile.ExtendedAttributeType = 23231;
cardinalAttribute2: NSFile.ExtendedAttributeType = 2222;
myString: NSstring.String;
firstCardinal, secondCardinal: CARDINAL;
```

```
--used for specifying selections of interest
attr: ARRAY [0..3) OF NSFile.ExtendedAttributeType ←
    [stringAttribute, cardinalAttribute1, cardinalAttribute2];
```

```
--storage for attributes being returned
attributes: NSFile.AttributesRecord;
```

```
fileHandle: NSFile.Handle ← --Get File Handle Somehow;
NSFile.GetAttributes[
    file: fileHandle,
    selections: [extended: DESCRIPTOR[attr]],
    attributes: @attributes];
```

```
-- get the values from the extended attributes array by
--searching for for the attribute type.
FOR c: CARDINAL IN [0..LENGTH[attributes.extended]) DO
    SELECT attributes.extended[c].type FROM
    stringAttribute => myString ←
        NSFile.DecodeString[attributes.extended[c].value
            !Courier.Error => CONTINUE] -- decoding error
    cardinalAttribute1 => firstCardinal ←
        NSFile.DecodeCardinal[attributes.extended[c].value
            !Courier.Error => CONTINUE];
    cardinalAttribute2 => secondCardinal ←
        NSFile.DecodeCardinal[attributes.extended[c].value
            !Courier.Error => CONTINUE];
    ENDCASE;
ENDLOOP;
```

The first step is to create the array `attr`, which contains the attribute types for the extended attributes of interest, and an `attributes` record, which provides storage for the attributes.

Next, we call `GetAttributes`, passing in a descriptor for the array of attribute types, and the `attributes` record. The call to `GetAttributes` will store the specified attributes in the `attributes` record, as illustrated in Figure 10.8.

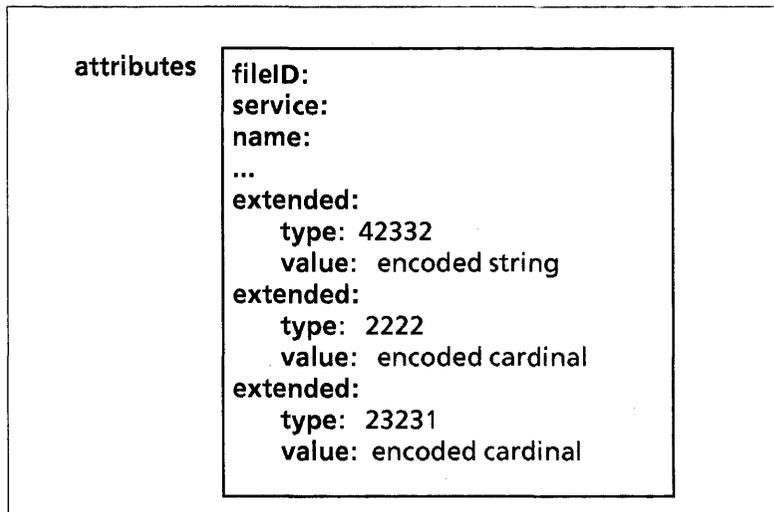


Figure 10.8: Attributes record

Since the value field is encoded, you cannot directly access the value of an extended attribute; you need to use decoding procedures. Thus, the next step is to loop through the extended attributes in `attributes` and match the attribute type to `stringAttribute`, `cardinalAttribute1`, or `cardinalAttribute2`. If there is a match, call the appropriate decoding procedure to extract the actual value of the extended attribute. (Note: The loop is necessary because NSFiling does not guarantee that the attributes will be stored in the order specified in `selections`.)

You should note the catch phrase for `courier.Error` in the decoding procedures. This catch phrase is used to catch several errors, such as using a decoding procedure that is not appropriate for the encoded information. To protect yourself, you should always catch this error and `CONTINUE`.

10.6 Summary

NSFiling attributes contain various pieces of information about a file. Attributes are distinct from content, which is the actual data in a file. The file system defines a large number of interpreted attributes and also allows you to define your own *extended* attributes.

Attributes are represented by variant records. To specify a set of attributes, you create an array of variant records: each record represents one attribute. You can then use this array to change the attributes for an existing file, to identify an existing file or create a new file. This chapter described how to set new

attributes for an existing file; the next chapter describes how to create new files and find existing files.

You can specify extended attributes by defining an attribute type and an associated value. If you use extended attribute types, you have to call encoding and decoding procedures to store and record the value of the attribute.

If you want to access or change a file's attributes, you can call `NSFile.GetAttributes`. To call this procedure, you need to pass a `selections` parameter that specifies the attributes of interest, and an `attributes` parameter that provides storage for the attributes being retrieved.

10.7 Exercise

The exercise for this chapter is an application called Music Man. This application allows you to write music by adding notes to a scale. When run, this application registers the command "Music Man" in the Attention Menu; invoking this command displays the window illustrated in Figure 10.10.

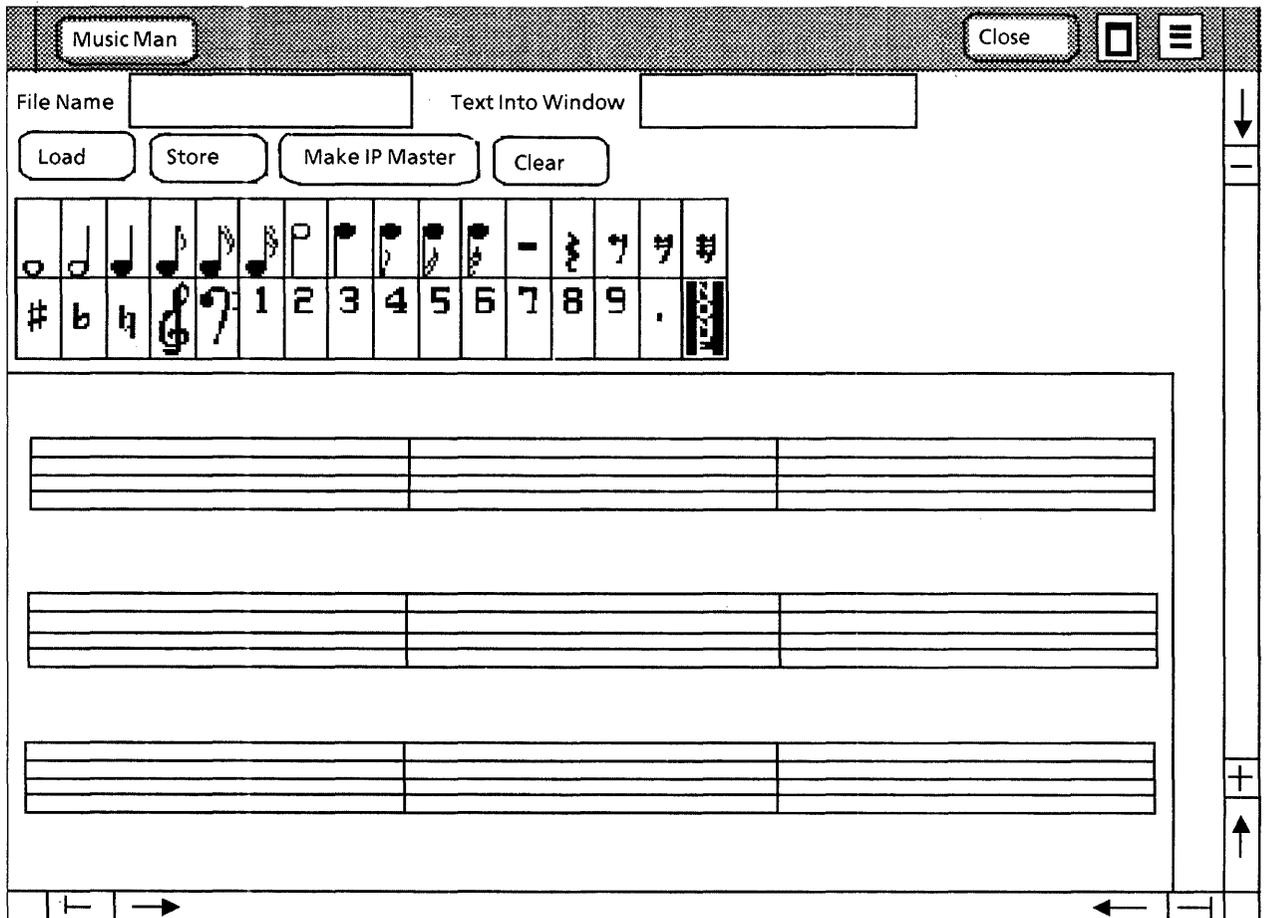


Figure 10.9: The Music Man application

To add a note, select the desired note, hold down Point, and choose a location for the note. The note will track the cursor, so you can move the mouse until you reach the right place on the

staff. Releasing Point will add the note to the staff and include it in the application's data structure.

To delete notes, select the "none" choice, move the cursor over the desired measure and begin clicking Point. This will highlight the notes one at a time. Pressing the DELETE key when a note is video-inverted will remove the note.

After you have finished editing a page of music you can create an Interpress master from the display or you can store the music in a file. To make an Interpress master, specify a file and select the "Make IP Master" command. To store your music, specify a file and select "Store"; to load an existing file select "Load". The file that you load can be either on the desktop or in the system folder. If the file is on the desktop, you can select it instead of typing the file name.

We have written most of the code for this application. You need to write procedures that store and retrieve extended attributes. The extended attributes for this application contain the number of notes (or other objects) contained in each measure of music. The procedures that you will implement are called **GetExtendedAttributes** and **StoreExtendedAttributes**.

GetExtendedAttributes takes a file handle and returns the extended attributes in an array. All of the extended attributes are **CARDINALS** and their **NSFile.ExtendedAttributeTypes** are specified by the subrange **ExtendedSubrange**. Thus you should get all the extended attributes in the range **ExtendedSubrange**, decode them as **CARDINALS** and return them in the specified array.

We have written some of the code in **StoreExtendedAttributes** (so you do not have to understand the details of the data structure). You just need to encode the count for each measure and associate it with the proper extended attribute type (this is well documented within the procedure). Lastly, you need to call **NSFile.ChangeAttributes** to store the new extended values in the file.

These two procedures are in the module **MusicFileImplExerciseA.mesa**. This file also contains some comments with a more detailed explanation of the code that you have to write.

To run the program, you will need several other modules, all of which are stored on the course directory and listed in the configuration file, **Music.config**. You should not need to modify any of these other modules.

Notes:

This chapter discusses how to perform basic filing operations such as creating, opening, closing and deleting files. It covers only operations on the files themselves, not the content of the files. The next two chapters, *Streams* and *NSSegment*, discuss how to access the content of a file.

11.1 Naming files

Before you can operate on a file, you must first have a way to identify that file. You can identify a ViewPoint file *by reference, by attributes, or by path name*.

The most common way to locate a file is by *reference*. An `NSFile.Reference` is a unique identifier for a file: it includes the name and network location of the file service containing the file, and a unique identifier for the file on that volume:

```
NSFile.Reference: TYPE = RECORD [  
    fileID: NSFile.ID,  
    service: NSFile.Service];
```

```
NSFile.Service: TYPE = LONG POINTER TO NSFile.ServiceRecord;
```

```
NSFile.ServiceRecord: TYPE = RECORD [  
    name: NSName.NameRecord,  
    systemElement: NSFile.SystemElement];
```

A *service* specifies the physical location of a file; that is, the machine where the file resides. A *fileID* is a unique identifier for a file on a particular machine. Section 11.3, *Opening remote files*, provides more information on *Service*. For now, you just need to be aware that a *Reference* uniquely specifies a file.

The second way to name a file is *by attributes*. To name a file using attributes, you must create an array of attributes that contains at least the file's name and version number. (For more information on specifying attributes, see Chapter 10, *NSFile Attributes*.)

The third common way to identify a file is *by path name*. A path name is a hierarchical list of directories in the path to the file, such as `Working/Current/temp.mesa`. The path name may be relative to a specified starting directory or to the root file.

The following sections contain examples of each of these methods.

11.2 Opening local files

If you want to be able to do anything useful with a file, such as examine its contents, change its attributes, modify its contents, or delete it, you first need to *open* the file. When you open a file, the file system gives you a *handle* to that file. A handle is effectively just a pointer to a particular file, although the actual structure of a handle is private to the implementation:

`NSFile.Handle: TYPE = [2];`

Having a handle marks the file as "in use," so that other processes are aware that you are using the file, but does not necessarily restrict other processes from using the file simultaneously.

When you acquire a file, you can also associate *controls* with your file handle. Controls are essentially ways to describe your use of the file. There are three types of controls: *locks*, *timeouts*, and *access*.

A *lock* is a restriction on the ways that other sessions can use a file. For example, you might wish to specify that no other client can read or modify the file while you are using it. Note that a lock only restricts file access *through other sessions*, not through other handles in the same session; this is not a complete locking mechanism. You can open a file several times within a single session, thereby creating several distinct handles. The default lock is *none*, which only prevents other processes from deleting the file.

A *timeout* control specifies a length of time that you are willing to wait to acquire the file before the attempt times out. NSFiling defines a `defaultTimeout`.

An *access* control determines the access that a particular file handle allows; this can be *read*, *write*, *owner*, *add*, *remove*, or some combination of the above. The default is `fullAccess`, which sets all the accesses to `TRUE`.

The following sections discuss various ways to open files, but do not show examples of using controls, since the defaults are sufficient for most uses. For more information on controls, see the *Filing Programmer's Manual*.

11.2.1 NSFile.Open

The most general way to open a file is by calling `NSFile.Open`:

```
NSFile.Open: PROCEDURE [  
  attributes: NSFile.AttributeList, --name file by attributes  
  directory: NSFile.Handle ← NSFile.nullHandle,  
  controls: NSFile.Controls ← [],  
  session: NSFile.Session ← NSFile.nullSession]  
  RETURNS [file: NSFile.Handle];
```

The `attributes` parameter identifies the file that you wish to open; you are thus naming the file by attributes. The `directory` parameter specifies a directory in which to start searching for

the file. If you don't know or don't care where the search should begin, you can leave this defaulted to the null handle. In the example below, the directory is passed in as a parameter.

`controls` and `session` have default values; all the examples in this chapter use these default values. See the *Filing Programmer's Manual* if you want more information on either of these parameters.

Here is an example of calling `Open`:

```

OpenFile: PROCEDURE[directory: NSFile.Handle,
  name: XString.ReaderBody,
  version: CARDINAL]
  RETURNS [file: NSFile.Handle] =

  BEGIN
    nsName: NSString.String ←
      XString.NSStringFromReader [@name, sysZ];

    --specify the attributes used to name the file
    --need at least name and version
    attributes: ARRAY [0..2] OF NSFile.Attribute ← [
      [name[nsName]],
      [version[version]] ];

    --open the file
    file ← NSFile.Open[attributes: DESCRIPTOR [attributes],
      directory: directory];

    --perform operations on the file...then free storage and file
    NSString.FreeString[sysZ, nsName];
    NSFile.Close[directory];
  END;

```

This example creates an attributes record that contains the name and version, and uses that attributes record to identify the file in the call to `NSFile.Open`.

Once the file is open, you can perform operations on it. Before exiting the procedure, the final step is to free the storage from the system zone and close the file. (Section 4 discusses closing files in more detail.)

11.2.2 NSFile.OpenByReference

If you have a reference to a file, you can use `NSFile.OpenByReference` instead of `NSFile.Open`:

```

NSFile.OpenByReference: PROCEDURE [
  reference: NSFile.Reference,
  controls: NSFile.Controls ← [],
  session: NSFile.Session ← NSFile.nullSession]
  RETURNS [NSFile.Handle];

```

`OpenByReference` is less general than `Open`, but is somewhat simpler to use if you already have a reference to the file.

11.2.3 OpenByName

`NSFile.OpenByName` is another alternative to `Open`:

```
NSFile.OpenByName: PROCEDURE [  
  directory: NSFile.Handle,  
  path: NSFile.String,  
  controls: NSFile.Controls ← [],  
  session: NSFile.Session ← NSFile.nullSession]  
RETURNS [NSFile.Handle];
```

`OpenByName` opens a descendant of a `directory` with a given path name. If `directory` is null, the path name is assumed to be relative to the root file. For example:

```
BEGIN  
  nsName: NSString.String ← NSString.StringFromMesaString [  
    "Blackjack/temp/BlackJack.df"];  
  file ← NSFile.OpenByName [path: nsName];  
  NSFile.Close [directory];  
  NSString.FreeString [nsName];  
END;
```

This fragment will open a file called *BlackJack.df* on the subdirectory *temp*, on the directory *blackjack*. Your path name can include as many subdirectories as you like; you must separate directories with the / character.

11.2.4 Catalog.Open

You can also open files with the `Catalog` interface. A *catalog* is a file that is a direct descendant of the root file, so a catalog is just a special case of a directory. The `Catalog` interface provides procedures for creating and opening catalogs and files contained in catalogs.

The two most commonly used catalogs are the *system catalog* and the *prototype catalog*. The system catalog contains TIP files, program bcds, font files, and any other files that you bring over from XDE. The prototype catalog contains blank copies of icons that the user can copy onto his desktop.

The two relevant open operations are `Catalog.Open`, and `Catalog.GetFile`. These procedures open a catalog and open a file within a catalog, respectively.

```
Catalog.Open: PROCEDURE [  
  catalogType: NSFile.Type,  
  session: NSFile.Session ← NSFile.nullSession]  
RETURNS [catalog: NSFile.Handle];
```

```
Catalog.GetFile: PROCEDURE [  
  catalogType: NSFile.Type ← BWSFileTypes.systemFileCatalog,  
  name: XString.Reader,  
  readOnly: BOOLEAN ← FALSE,  
  session: NSFile.Session ← NSFile.nullSession,  
RETURNS [file: NSFile.Handle];
```

You can use these procedures to operate on any catalog: the system catalog, the prototype catalog, or any other catalog. If you want to use one of the well-known catalogs, you will have to get the file type for that catalog from the **BWSFileTypes** interface. The example below uses the system catalog:

```

GetFileFromCatalog: PUBLIC PROCEDURE [name: XString.Reader]
RETURNS [file: NSFile.Handle] =
BEGIN
    catalogType: NSFile.Type ← BWSFileTypes.systemFileCatalog;
    file ← Catalog.GetFile [catalogType: catalogType,
        name: name,
        readonly: TRUE]; -- if file not found nullHandle returned
END;

```

This example obtains the file type for the system catalog, and then calls **GetFile** to search for a file with the given name. If the file is in the catalog, **GetFile** returns a handle to it; if the file is not in the catalog, **GetFile** returns a null handle. Note that with **Catalog** operations, you pass in the file name directly, without creating an attributes list.

If the catalog does not exist, then **Catalog** will raise a filing error. ViewPoint will catch the error and post a message in the Attention window. If this isn't how you would like to handle this situation, you can explicitly call **catalog.Open** to make sure that the catalog type is valid and if it is not valid, then handle the error yourself.

11.3 Opening remote files

Opening a remote file (a file on another machine) is not much more complicated than opening a local file. To open a remote file, you get a file handle to the machine on which the file resides, and then specify that file handle as the directory when you ask to open the file. To get a handle to the machine, call **Open**, specifying the service attribute to identify the machine. The service attribute is of type **Service**:

```

NSFile.Service: TYPE = LONG POINTER TO ServiceRecord;

```

```

NSFile.ServiceRecord: TYPE = RECORD [
    name: NSName.NameRecord,
    systemElement: NSFile.SystemElement ←
        NSFile.nullSystemElement];

```

```

NSName.NameRecord: TYPE = RECORD [
    org: NSName.Organization,
    domain: NSName.Domain,
    local: NSName.Local];

```

```

NSFile.SystemElement: TYPE = System.NetworkAddress;

```

Thus, to uniquely specify a remote service, you need both the name of that service and its network address. (The reason for this is that there might be more than one service on a given machine.)

If you do not specify an address for the system element, the file system will look up the name of the service in the Clearinghouse to get its network address.

To minimize repetitive lookups, it maintains a cache of service records with network addresses. If you want to get such a reference from the cache, you can call `MakeReference`:

```
NSFile.MakeReference: PROCEDURE [
  fileID: NSFile.ID,
  service: NSFile.Service ← NSFile.nullService]
  RETURNS [reference: NSFile.Reference];
```

Section 11.8 has an example of this procedure.

Here is an example that opens a file on a remote machine:

```
hostHandle: NSFile.Handle ← NSFile.nullHandle;
```

```
<< host is an XString.ReaderBody obtained from the user to
specify the machine of choice. If host is not NIL, then get a
handle to it; otherwise, use the default value (nullHandle),
which specifies the root directory of the local volume.>>
```

```
IF NOT XString.Empty[@host] THEN {
  serviceRec: NSFile.ServiceRecord;
  << array of attributes for calling Open. This example
specifies only the service attribute.>>
  attribute: ARRAY [0..1] OF NSFile.Attribute ←
    [service[@serviceRec]];
```

```
<< get host name into proper format (first convert from
XString to NSString, and then from NSString to
NSName)>>
```

```
nsName: NSName.Name ← NSName.NameFromString[
  z: Heap.systemZone,
  s: XString.NSStringFromReader
    [@host, Heap.systemZone]];
```

```
--set up the name field of the service record
--use the default network address
serviceRec.name ← nsName ↑ ;
```

```
<< open host machine. Note that with remote operations,
you need to catch Courier.Error.>>
```

```
hostHandle ← NSFile.Open[DESCRIPTOR[attribute]
  !NSFile.Error = > CONTINUE;
  Courier.Error = > CONTINUE]];
```

The point of this code is to get a handle to the remote machine whose name is `host`. If `host` is not an empty string, call `Open`, identifying the file by attributes. In this case, we use only one attribute to identify the file: the service attribute. The service attribute includes only the name of the service; we use the default address, which means that the file system will obtain the address from the Clearinghouse.

Once `hostHandle` has been initialized, you can use it as the `directory` parameter to other NSFiling procedures.

11.4 Closing files

When you are finished with a file, you must close it. Once you have closed a file, your handle is no longer valid, and other clients may move or delete the file.

```
NSFile.Close: PROCEDURE [  
    file: NSFile.Handle,  
    session: NSFile.Session ← NSFile.nullSession];
```

If the file handle is invalid, the **NSFile** implementation will raise **NSFile.Error**. After calling **Close**, you should set your file handle to **NIL** to ensure that you don't try to use it again.

11.5 Creating files

The **NSFile** and **Catalog** interfaces also provide procedures to create new files. When you create a file, you do not necessarily associate any contents with the file; instead, you typically associate content with the file later. NSFiling itself does not care about the contents of the file; it allocates storage for the content, but does not manipulate the content. (See the next chapter, *Streams*, for information on putting content in files. For now, we limit the discussion to creating the file itself.)

You can also create files by other means, such as copying existing files. For more information on other methods, see the *Filing Programmer's Manual*.

11.5.1 NSFile.Create

The most general way to create a new file or directory is by calling **NSFile.Create**. The file that you create can either be temporary or permanent (contained in a directory).

```
NSFile.Create: PROCEDURE [  
    directory: NSFile.Handle,  
    attributes: NSFile.AttributeList ← NSFile.nullAttributeList,  
    controls: NSFile.Controls ← [],  
    session: NSFile.Session ← NSFile.nullSession]  
RETURNS [file: NSFile.Handle];
```

Note that you must first open the directory in which the file is to be located. Specifying a null **directory** will create a temporary file. Here is an example of a procedure that creates a file and returns a handle to that file.

```
CreateFile: PUBLIC PROCEDURE [  
  catalogType: NSFile.Type,  
  name: XString.Reader,  
  type: NSFile.Type, -- type of file being created  
  isDirectory: BOOLEAN ← FALSE,  
  size: LONG CARDINAL ← 0]  
  RETURNS [file: NSFile.Handle ← NSFile.nullHandle] =  
  
BEGIN  
  catalog: NSFile.Handle ← Catalog.Open [catalogType];  
  nsName: NSString.String ←  
    XString.NSStringFromReader [name, zone];  
  
  attributes: ARRAY [0..4] OF NSFile.Attribute ← [  
    [name[nsName]],  
    [isDirectory[isDirectory]],  
    [sizeInBytes[size]],  
    [type[type]]];  
  IF catalog = NSFile.nullHandle THEN RETURN;  
  file ← NSFile.Create [  
    directory: catalog,  
    attributes: DESCRIPTOR [attributes]! NSFile.Error = >  
    {HandleError[error]; CONTINUE}}];  
  NSFile.Close [catalog];  
  NSString.FreeString [zone, nsName];  
END;
```

This procedure opens the catalog in which the file is to be located, creates an attributes record, and then creates a file in the specified directory with the specified attributes.

One thing you should notice from this example is the `sizeInBytes` attribute. This attribute is optional, but is very important; it specifies the size of the initial chunk of storage for the file. If you do not include this attribute, the file will grow as it is written; this can scatter the file's pages over the disk, and can result in very long access times. Thus, you should always make a good guess as to the final size of your file and include that value in your attributes.

Trying to create a file that already exists will generate an error. If this happens, you should catch the signal and create a new file with a higher version number. (Section 11.9 provides more information on `NSFile` errors.)

11.5.2 Catalog.Create

The `Catalog` interface provides an easy way for you to create a file that is a descendant of a particular catalog:

```
Catalog.CreateFile: PROCEDURE [  
  catalogType: NSFile.Type ← BWSFileTypes.systemFileCatalog,  
  name: XString.Reader,  
  type: NSFile.Type,  
  isDirectory: BOOLEAN ← FALSE,  
  size: LONG CARDINAL ← 0,  
  session: NSFile.Session ← NSFile.nullSession,  
  RETURNS [file: NSFile.Handle];
```

Note that with `CreateFile`, you just pass name, type, `isDirectory` and `size` as parameters; you don't specify file attributes in an

AttributeList. You are limited to the attributes listed in the procedure declaration, however. (If you want additional attributes, you will have to set them later, after you have created the file.) As you can see from the type declaration, if you don't specify a catalog, the default is to use the system catalog. Once again, you should include the **size** parameter to avoid fragmentation.

11.6 Deleting files

NSFile.Delete deletes an existing file, removes any association with a directory, and frees the resources allocated to the file. If the deleted file is a directory, then all descendants are deleted too.

```
NSFile.Delete [
  file: NSFile.Handle,
  session: NSFile.Session ← NSFile.nullSession];
```

To correctly delete a file, there can be no other handles attached to the file. If an error occurs during the Delete process, the file handle will remain valid; otherwise, the handle becomes invalid.

11.7 Finding files

Suppose you want to locate all files whose name contains a particular string or all files created by a particular person. In such cases, you can use **NSFile.Find**:

```
NSFile.Find: PROCEDURE [
  directory: NSFile.Handle,
  scope: NSFile.Scope ← [],
  controls: NSFile.Controls ← [],
  session: NSFile.Session ← NSFile.nullSession]
RETURNS [file: NSFile.Handle];
```

scope is the most important parameter here; **scope** is how you specify the subset of files that you are interested in:

```
NSFile.Scope: TYPE = RECORD [
  count: CARDINAL ← LAST[CARDINAL],
  direction: NSFile.Direction ← forward,
  filter: NSFile.Filter ← NSFile.nullFilter,
  ordering: NSFile.Ordering ← NSFile.nullOrdering,
  depth: CARDINAL ← 1];
```

count, **direction**, **ordering**, and **depth** are relatively straightforward; **filter** is the hard part.

count is the maximum number of files that you are interested in. The file system will continue to search until it has found a match, until it has searched **count** files, or until it has run out of files to search. The default value considers all possible files.

direction is the direction of the search: you can either start at the end of the directory and work back to the beginning, or vice versa.

ordering is the order in which you want to search the directory. **nullOrdering** defaults to the directory's ordering attribute. (Directories can have complicated orderings; see section 6.3.6 of the Filing Programmer's Manual for details.) Your other choice for **ordering** is **defaultOrdering**; no other ordering schemes are implemented. **defaultOrdering** specifies alphabetical order by file name.

depth is the number of levels (subdirectories) in the file system hierarchy that you want the search to encompass.

filter is the real basis of the scope. In most cases, the filter is the only part of the scope that changes. The filter may be an arbitrarily complex relational expression, as described in the next section.

11.7.1 Filters

NSFile.Filters allow you to construct a relational expression made up of attributes and the relations "and," "or," and "not". Here is the declaration of **Filter**:

```
NSFile.Filter: TYPE = MACHINE DEPENDENT RECORD [  
  var: SELECT type: FilterType FROM  
    less, lessOrEqual, equal, notEqual, greaterOrEqual, greater  
      = > [attribute: NSFile.Attribute,  
          interpretation: NSFile.Interpretation ← none],  
    matches = > [attribute: NSFile.Attribute],  
    and, or = > [list: LONG DESCRIPTOR FOR ARRAY OF NSFile.Filter],  
    not = > [filter: LONG POINTER TO NSFile.Filter],  
    none, all = > [],  
  ENDCASE];
```

You can create various combinations of these filters, depending on the search criteria that you are interested in. For example, if you wanted to find a file whose name was "Training Notes," with a version number less than 3, that was not created by anyone with a first name of "Jim," then you would make the following filter:

```
[and [  
  [equal [[name ["Training Notes"]]],  
  [less [[version [3]]],  
  [not [ [matches [[createdBy ["Jim *"]]] ] ]  
]]
```

The Mesa code for this filter would look something like this:

```
createdByFilter: NSFile.Filter ←  
  [matches [[createdBy ["Jim  
  *"]]]];  
  
filterArray: ARRAY [0..3] OF NSFile.Filter ← [  
  [equal [[name ["Training Notes"]]],  
  [less [[version [3]]],  
  [not [@createdByFilter]] ];  
  
filter: NSFile.Filter ← [and[DESCRIPTOR[filterArray]]];
```

This filter will search for files whose `name` attribute is "training notes," whose `version` attribute is less than 3, and whose `createdBy` attribute does not pass `createdByFilter`.

Notice that the `not` operation requires a long pointer to a `Filter`, so you need to first create the "inner" filter, and then put a pointer to it in the "outer" filter.

You can also use the two wildcards `*` and `#` in your matches expression. The `*` character matches zero or more characters within a string attribute; `#` matches any single character.

11.7.2 NSFile.Find

Once you specify a `scope`, you can call `Find` to locate and open a file in a particular directory. Here is an example that finds any file whose name matches a specified string:

```
FindFile: PROCEDURE [file: XString.Reader,
                    zone: UNCOUNTED_ZONE, dir: NSFile.Handle]
  RETURNS [handle: NSFile.Handle ← NSFile.nullHandle] =

BEGIN
  nsSource: NSSString.String ← XString.NSSStringFromReader[
    file, zone];

  -- if no directory is specified use the root file
  IF dir = NSFile.nullHandle THEN dir ←
    NSFile.OpenByReference[reference: NSFile.nullReference];

  handle ← NSFile.Find[
    directory: dir,
    scope: [filter: [matches[[name[nsSource]]]]]! NSFile.Error = >
CONTINUE];
  NSSString.FreeString[Defs.zone, nsSource];
END;
```

This example opens the specified directory. If there is no directory, then it uses the root directory. Next, it searches the open directory for all files whose name attribute matches the file parameter.

11.8 Listing files

You can use `NSFile.List` to enumerate files in a directory and return the desired attributes of each listed file:

```
NSFile.List: PROCEDURE [
  directory: NSFile.Handle,
  proc: NSFile.AttributesProc,
  selections: NSFile.Selections,
  scope: NSFile.Scope ← [],
  clientData: LONG POINTER ← NIL,
  session: NSFile.Session ← NSFile.nullSession];

NSFile.AttributesProc: TYPE = PROCEDURE [
  attributes: NSFile.Attributes, clientData: LONG POINTER]
  RETURNS [continue: BOOLEAN ← TRUE];
```

`directory` is the directory that you want to search. `proc` is a call back procedure that you want the file system to call when it finds an appropriate file. For each file that matches the specified scope, the file system will call `proc` with `attributes`, which is a pointer to an attributes record.

`clientData` is a `LONG POINTER` to information of your choice. If there is any additional information that you want available when `proc` is called, you can pass it in via `clientData`.

Here is an example of using `List`:

<<The file system will call Enumerate when it finds a file that matches the search criteria. When called, Enumerate will open the file by reference. If there are problems, abort this enumeration and continue the search with other files.

```
Enumerate: NSFile.AttributesProc = {
  ref: NSFile.Reference ← NSFile.MakeReference[
    fileID: attributes.fileID! NSFile.Error = > GOTO Exit];
  handle: NSFile.Handle ←
    NSFile.OpenByReference[ref! NSFile.Error = > GOTO Exit];
  ...
  -- do something interesting with file handle
  NSFile.Close[handle! NSFile.Error = > CONTINUE];
  EXITS Exit = > NULL;
};
```

--This is the procedure that does the actual search.

```
ListDirectories: PROCEDURE[
  handle: NSFile.Handle, file: XString.Reader] = {
  nsSource: NSString.String ← XString.NSStringFromReader[
    file, Defs.zone];

  --interested in all files whose name matches nsSource
  scope: NSFile.Scope ←
    [filter: [matches[[name[nsSource]]]],direction:backward];
  -- get the ID of the file being enumerated
  selections: NSFile.Selections;
  selections.interpreted[fileID] ← TRUE;

  -- if no directory specified use the root
  IF handle = NSFile.nullHandle THEN handle ←
    NSFile.OpenByReference[reference: NSFile.nullReference];

  -- call List. This will result in a call to Enumerate for each file
  --that matches the specified scope
  NSFile.List[
    directory: handle,
    proc: Enumerate,
    selections: selections,
    scope: scope!NSFile.Error = > CONTINUE];

  -- free all allocated data and close the directory
  NSString.FreeString[Defs.zone, nsSource];
  NSFile.Close[handle! NSFile.Error = > CONTINUE];
};
```

The parameters to `ListDirectories` are a handle to a directory, and the name of a file. `ListDirectories` sets up a filter to match all files whose name matches the file parameter, and then calls `NSFile.List`. `List` calls `Enumerate` for each file that matches the

specified scope; Enumerate can then perform any desired operations on those files.

11.9 Errors

When working with the NSFiling system, you need to concern yourself with the errors `NSFile.Error` and `Courier.Error`. `NSFile.Error` has a parameter that indicates the type of error:

```
NSFile.Error: ERROR [error: NSFile.ErrorRecord];
```

```
NSFile.ErrorRecord: TYPE = RECORD [
  SELECT errorType: NSFile.ErrorType FROM
  access = > [problem: NSFile.AccessProblem],
  attributeType, attributeValue = > [
    problem: NSFile.ArgumentProblem,
    type: NSFile.AttributeType,
    extendedType: NSFile.ExtendedAttributeType ←
      LAST[NSFile.ExtendedAttributeType];
  authentication = > [
    problem: NSFile.AuthenticationProblem],
  clearingHouse = > [problem: NSFile.ClearinghouseProblem],
  connection = > [problem: NSFile.ConnectionProblem];
  controlType, controlValue = > [
    problem: NSFile.ArgumentProblem,
    type: NSFile.ControlType],
  handle = > [handle: NSFile.HandleProblem],
  insertion = > [problem: NSFile.InsertionProblem],
  range = > [problem: NSFileArgumentProblem],
  scopeType, scopeValue = > [
    problem: NSFile.ArgumentProblem,
    type: NSFile.ScopeType],
  service = > [problem: NSFile.ServiceProblem],
  session = > [problem: NSFile.SessionProblem].
  space = > [problem NSFile.SpaceProblem],
  transfer = > [problem: NSFile.TransferProblem],
  undefined = > [problem: NSFile.UndefinedProblem],
  ENDCASE];
```

Within an Error, there are various classes of errors. The error classes are Access errors, Argument errors, Authentication errors, Clearinghouse errors, Connection errors, Handle errors, Insertion errors, Service errors, Range errors, Session errors, Space errors, Transfer errors, and Undefined errors.

Within each of these classes, there are a number of specific errors. For example, in the case of an access error, there is a parameter `problem` that indicates the exact nature of the problem. This problem is of type `AccessProblem`:

```
NSFile.AccessProblem: TYPE = MACHINE DEPENDENT {
  accessRightsInsufficient(0), accessRightsIndeterminate(1),
  fileChanged(2), fileDamaged(3), fileInUse(4),
  fileNotFound(5), fileOpen(6), fileNotLocal(7)};
```

Each of the other classes of errors has a similar list of specific errors; consult the *Filing Programmer's Manual* for a complete list. When working with filing operations, you need to catch `NSfile.Error`, select the error class and then select the actual error from within the class. For example:

```

NSFileCatchError: PROCEDURE [error: NSFile.ErrorRecord] =
BEGIN
  WITH myError: error SELECT FROM
    access = > SELECT myError.problem FROM
      fileChanged = > <<File changed error>>
      fileInUse = > <<File in use error>>
      fileNotFound = > <<File not found error>>
    ENDCASE;
  handle = >
    SELECT myError.problem FROM
      invalid = > <<Invalid handle error>>
      obsolete = > <<Obsolete handle error>>
    ENDCASE;
  ENDCASE = > <<Undefined error>>
END;

```

If you are working with remote files, you also need to be concerned with `Courier.Error`. Like `NSFile.Error`, this error has a parameter that indicates the exact nature of the problem:

```
Courier.Error: ERROR [errorCode: Courier.ErrorCode];
```

```

Courier.ErrorCode: TYPE = {
  transmissionMediumHardwareProblem,
  transmissionMediumUnavailable,
  transmissionMediumNotReady, noAnswerOrBusy,
  noRouteToSystemElement, transportTimeout,
  remoteSystemElementNotResponding,
  noCourierAtRemoteSite, tooManyConnections,
  invalidMessage, noSuchProcedureNumber,
  returnTimedOut, callerAborted,
  unknownErrorInRemoteProcedure, streamNotYours,
  truncatedTransfer, parameterInconsistency,
  invalidArguments, noSuchProgramNumber,
  protocolMismatch, duplicateProgramExport,
  noSuchProgramExport, invalidHandle, noError};

```

For more information on any of these error codes, see Section 6.6.4.1 of the *Pilot Programmer's Manual*.

11.10 Summary

To open a file so that you can read or write to it, you generally use `NSFile.Open`. Alternatives to `Open` are `NSFile.OpenByReference`, `NSFile.OpenByName`, and `Catalog.Open`.

To close a file, use `NSFile.Close`.

To create a file, use `NSFile.Create`. If you want to create a file in a catalog, you can use `Catalog.Create` instead.

To delete a file, call `NSFile.Delete`.

To find or list files matching particular criteria, call `NSFile.Find` or `NSFile.List`.

There is a complete filing example at the end of Chapter 13, *NSSegments*. This example illustrates a full range of filing operations, covering the information in Chapters 10 through 13.

For complete documentation on the **NSFile** interface, see the *Filing Programmer's Manual*.

11.11 Exercise

The exercise for this chapter builds on the exercise for the previous chapter. You don't need to have done the earlier exercise to do this one, but you do need to have read the description of the user interface. If you didn't do the last exercise, you should go back now and read the description of how the tool operates.

For this exercise, you need to implement the two procedures **GetSelectedFile** and **OpenTypedfile**.

GetSelectedFile should open the currently selected file and return an **NSFile.Handle** to it. If there are any problems, you should print an error message to the user and raise the signal **FileProblem**.

The second procedure, **OpenTypedFile**, is more complicated. In this procedure, you receive the name of a file and an operation (**saveData**, **loadData**, or **saveIP**) as parameters. You need to do the following:

- Check to see if the file is on the desktop or in the system catalog. If it is, then just return a handle to it. (See the **StarDesktop** interface for more information on manipulating the desktop.)
- If the file isn't in either the system catalog or on the desktop, you need to decide whether to create it or return an error. If the operation is **loadData**, then you are trying to read the contents of the file. Since the file doesn't exist, you should print an error message and raise the signal **FileProblem**.

If the operation is not **loadData**, then you need to create a new file on the desktop. To create the file, you need to determine the file type by checking the operation argument again. If **operation** is **saveData**, then use the type **SimpleDocumentType**; otherwise use **IPMasterType**. The reason for having two file types is that this procedure is called to create files both for storing data and creating Interpress masters.

The procedures that you need to modify are in **MusicFileImplExerciseB.mesa**. This is the same module that you worked with in the last exercise. This time, however, we provide the attributes code, and you must write the code that manipulates the files. The comments in this module describe exactly what you need to do.

Notes:

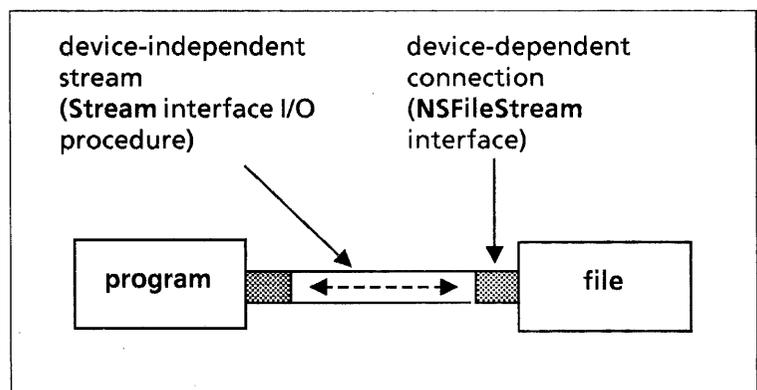
The last chapter discussed how to create, open, close, and delete files, but did not discuss how to access the actual contents of the file. The two most common ways for a program to access the contents of a file are attaching a *stream* to the file, and *mapping* the contents of the file to virtual memory. This chapter focusses on how to attach a stream to a file; the next chapter discusses mapping the file to virtual memory.

12.1 Overview

A *stream* is an abstraction for accessing data sequentially: one byte at a time, one word at a time, and so on. Streams are *device-independent*: you can use streams to access data on various different devices, such as local disk files, floppies and tape drives. For the purposes of this chapter, however, a stream is a connection from a program to a local disk file. If you want to know how to use streams to access other kinds of devices, see the *Pilot Programmer's Manual*.

To use a stream to access the contents of a file, you need to set up a connection between your program and the file. Creating the stream is *device-dependent*; you need to use a procedure that knows the details of file I/O. The ViewPoint `NSFileStream` interface provides this procedure.

Once the stream is established, however, you can use procedures from the Pilot `Stream` interface to do I/O. These procedures are device-independent; you use the same I/O procedures regardless of the type of the device to which the stream is attached. Figure 12.1 illustrates this idea.



12.1 A stream to a file

The rest of this chapter describes how to set access the contents of a file using the facilities of `NSFileStream` and `Stream`.

12.2 Creating a stream

To open a stream to a file, the first step is to call `NSFileStream.Create` :

```
NSFileStream.Create: PROCEDURE [
    file: NSFile.Handle,
    closeOnDelete: BOOLEAN ← TRUE,
    options: Stream.InputOptions ← Stream.defaultInputOptions,
    session: NSFile.Session]
    RETURNS [fileStream: NSFileStream.Handle];
```

`file` is a handle to the file to which you want to attach the stream; you can get the file handle with any of the methods described in the previous chapter. Note that `file` must be a handle to a local file; you cannot use `NSFileStream` to access remote files.

`closeOnDelete` specifies whether the file is to be automatically closed when the stream is deleted. `options` controls various aspects of the stream. `session` is as described in Chapter 10, *NSFiling Operations*.

`Create` creates a stream to the specified file and returns a handle to that stream. The handle is of type `NSFileStream.Handle`, which is equivalent to a `Stream.Handle`:

```
NSFileStream.Handle: TYPE = RECORD [Stream.Handle];
```

```
Stream.Handle: TYPE = LONG POINTER TO Stream.Object;
Stream.Object: TYPE = RECORD [
    options: Stream.InputOptions,
    getByte: Stream.GetByteProcedure,
    putByte: Stream.PutByteProcedure,
    ...];
```

A `Stream.Object` defines the mechanisms for data transfer to and from the particular device for which the stream was created. The `Object` contains specific procedures to do I/O; the `Stream` implementation will call these procedures when the client requests a read or write. (The next section covers stream I/O in detail.) You don't need to know anything about the `Object` however; the stream handle is all you need.

12.3 Stream I/O

Once you have created a stream to a file, you can use the facilities of the `Stream` interface to read and write to the file. Note that controls acquired when creating the file handle apply when you access the file via the file stream, and are used to determine your access to the file.

The basic output operations for streams are `PutByte`, `PutWord`, and `PutBlock`. (`PutBlock` is discussed in the next section.) The procedures are declared as follows:

```
Stream.PutByte: PROCEDURE [sH: Stream.Handle,
    byte: Stream.Byte];
```

```
Stream.PutWord: PROCEDURE [sH: Stream.Handle,
    word: Stream.Word];
```

These procedures are straightforward; you provide a stream handle and a piece of data, and the procedure puts that data to the appropriate stream.

The basic input procedures are `GetByte`, `GetWord`, and `GetBlock`. (`GetBlock` is discussed in the next section.)

```
Stream.GetByte: PROCEDURE [sH: Stream.Handle]
    RETURNS [byte: Stream.Byte];
```

```
Stream.GetWord: PROCEDURE [sH: Stream.Handle]
    RETURNS [word: Stream.Word];
```

These procedures return the next byte or word (respectively) from the specified stream.

When it reaches the end of an input file, the `Stream` interface raises the signal `EndOfStream`. You need to catch this signal and react accordingly. For example:

```
ch: Stream.Byte;
DO -- copy the file input to the file output
    ch ← Stream.GetByte[input!Stream.EndOfStream = > EXIT];
    Stream.PutByte[output,ch];
ENDLOOP;
```

This example gets a byte from `input` and puts it to `output` until it reaches the end of the input file. At that point, the `Stream` interface raises the `EndOfStream` signal, which is caught inside the loop. The catch phrase exits the loop, and control will resume at the statement following the loop.

12.3.1 Example of I/O

Here is an example of a procedure that creates streams on two files and then copies the contents of one file to the other. After the copy is complete, the streams are deleted, thus closing the files. (Deleting a stream automatically closes the file to which the stream is attached, unless you change the value of the `closeOnDelete` boolean parameter of `NSFileStream.Create`.)

```
Copy: PROCEDURE [from, to: NSFile.Handle] = {
    --set up the streams with NSFileStream
    input: NSFileStream.Handle ← NSFileStream.Create[file: from];
    output: NSFileStream.Handle ← NSFileStream.Create[file: to];

    -- after Streams are created you can use Pilot Stream operations
    ch: Stream.Byte;
    DO -- copy the file
        ch ← Stream.GetByte[input!Stream.EndOfStream = > EXIT];
        Stream.PutByte[output,ch];
    ENDLOOP;

    Stream.Delete[input]; --deletes stream and closes file
    Stream.Delete[output] };
```

12.3.2 Block I/O

Stream blocks allow you to transfer arbitrary data structures.

Stream.Block: TYPE = Environment.Block;

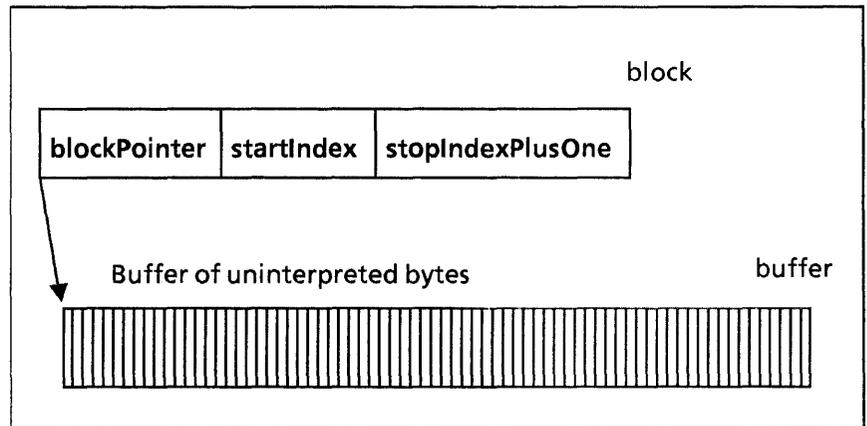
Environment.Block: TYPE = RECORD [
blockPointer: LONG POINTER TO PACKED ARRAY [0..0)
 OF Environment.Byte,
startIndex, **stopIndexPlusOne:** CARDINAL];

Stream.GetBlock: PROCEDURE [sH: Stream.Handle,
block: Stream.Block]
 RETURNS [bytesTransferred: CARDINAL,
why: Stream.CompletionCode,
sst: Stream.SubSequenceType];

Stream.PutBlock: PROCEDURE [sH: Stream.Handle,
block: Stream.Block,
endRecord: BOOLEAN ← FALSE];

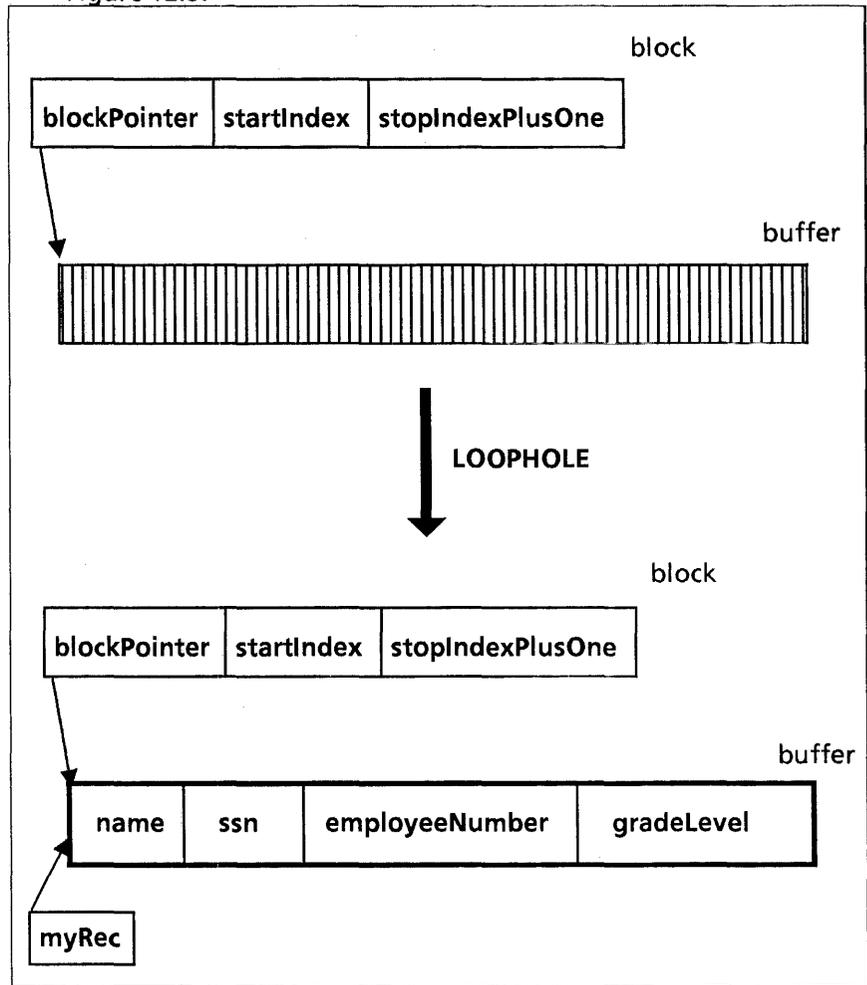
Stream.CompletionCode: TYPE = {normal, endRecord,
 sstChange, endOfStream, attention, timeout};

To use **GetBlock**, you must first declare a **Block**, which includes a pointer to a buffer (array) of bytes, and information on where to start and stop in the array of bytes. You then call **GetBlock**, passing in a stream handle and your **Block**. **GetBlock** will retrieve bytes from the file and store them into the buffer specified by **block**, as illustrated in Figure 12.2.



12.2 After a **GetBlock** operation

Once you have the array of bytes, you can use Mesa's LOOPHOLE operation to operate on the bytes as if they were a data structure. The LOOPHOLE operation allows you to circumvent Mesa's type checking; you can use LOOPHOLE to treat any type as if it were another, as long as the two types occupy the same number of words. Thus, you can loophole the block pointer into a pointer to another record structure, as illustrated in Figure 12.3.



12.3 Using stream block operations

Here is a code fragment that corresponds to the above figure:

```

EmployeeInfo: TYPE = MACHINE DEPENDENT RECORD[
  name(0): PACKED ARRAY[0..30] OF CHARACTER,
  ssn(15): LONG CARDINAL,
  employeeNumber(17): LONG CARDINAL,
  gradeLevel(19): CARDINAL];

--size of record in bytes
RecordSize = SIZE [EmployeeInfo] * Environment.BytesPerWord;
inputStream, outputStream: Stream.Handle ← NIL;

myRec: LONG POINTER TO EmployeeInfo;
--declare a buffer that has same number of bytes as the record
buffer: PACKED ARRAY[0..RecordSize] OF Environment.Byte;
--create a stream block with pointer to your buffer
block: Stream.Block ← [@buffer, 0, RecordSize];

--need to initialize stream handles here by calling
--NSFileStream.Create
...

completionCode: Stream.CompletionCode ← normal;
UNTIL completionCode = endOfStream DO
--get a block
  [block.stopIndexPlusOne, completionCode,] ←
    Stream.GetBlock[inputStream, block];
--lay template on data
  myRec ← LOOPHOLE[block.blockPointer];
--modify record
  myRec.gradeLevel ← myRec.gradeLevel + 1;
--put modified record to output file
  Stream.PutBlock[sH: outputStream, block: block,
    endRecord: TRUE]; -- ensure block is sent before return
ENDLOOP;

```

In this example, we declare a buffer, make a `Stream.Block` that has a pointer to that buffer, and then call `GetBlock`, passing in our input stream and the block. `GetBlock` fills the buffer with bytes taken from the stream. We then use `LOOPHOLE` to access the block pointer as if it were a pointer to our record structure. We can then use the record pointer to modify the bytes as if they were a record of type `EmployeeInfo`. (Note that we never actually declare a record of type `EmployeeInfo`; instead, we use the buffer of bytes as if it were the record.) When we have finished modifying the record, we use `PutBlock` to write the buffer to another stream.

You should notice that `GetBlock` normally uses the completion code `endOfStream` instead of signalling `endOfStream`. To cause `GetBlock` to raise the signal, you can call `Stream.SetInputOptions` to set `signalEndOfStream` in `inputOptions` to `TRUE`. (Recall from section 11.2 that `NSFileStream.Create` has an options parameter that specifies certain characteristics of the stream. `Stream.SetInputOptions` is a procedure that allows you to change the characteristics of a stream once it has been created.)

12.3.3 Random access

Although streams are generally used for sequential access, `NSFileStream` supports random access to the contents of local files. Note, however, that you should not do a lot of random access with streams. If you need to do a lot of random access, you should use mapping instead of streams. (The next chapter discusses this method.)

To get to a particular position you call `Stream.SetPosition`. Similarly, you can determine the current location within a file by using `Stream.GetPosition`:

`Stream.Position`: TYPE = LONG CARDINAL;

`Stream.GetPosition`: PROCEDURE [sH: Stream.Handle] RETURNS [position: Stream.Position];

`Stream.SetPosition`: PROCEDURE [sH: Stream.Handle, position: Stream.Position];

In both of the procedure declarations, the `position` parameter is the byte-index of the next data in the stream to be read or written, where the first byte in the file has the index 0. Here is some code to illustrate the use of these procedures:

`Advance`: PROCEDURE [stream: Stream.Handle, advanceAmount: Stream.Position] RETURNS [newPos: Stream.Position] =

BEGIN

position: Stream.Position ← Stream.GetPosition [sH: stream];

newPos ← position + advanceAmount;

stream.SetPosition [sH: stream, position: newPos];

END;

12.3.4 Miscellaneous operations

`NSFileStream` also provides procedures that allow you to obtain a count of the data bytes in a file stream, or set the length of data bytes in the file stream:

`NSFileStream.GetLength`: PROCEDURE [fileStream: NSFileStream.Handle] RETURNS [lengthInBytes: LONG CARDINAL];

`NSFileStream.SetLength`: PROCEDURE [fileStream: NSFileStream.Handle, lengthInBytes: LONG CARDINAL];

Note that the length returned by `GetLength` is not necessarily equal to the size of the underlying file. For example, if you are appending data to the stream, the actual size of the file may be smaller than the number of bytes in the stream. See the *Filing Programmer's Manual* for details.

Note also that you can use `SetLength` to make the length of the file stream either larger or smaller.

Another `NSFileStream` procedure that you might find useful is `FileFromStream`:

```
NSFileStream.FileFromStream: PROCEDURE [  
    fileStream: NSFileStream.Handle]  
RETURNS [file: NSFile.Handle];
```

This procedure returns a copy of the file handle to which the stream is attached. It is a copy of the handle used to acquire the stream; it is valid only during the session during which you acquired the stream.

12.4 Deleting streams

Since a stream is a connection between a program and a device, the program should never terminate without telling the device that the connection is no longer open. For every stream you create, you must call `stream.Delete` to close the stream when you are finished with it.

```
Stream.Delete: PROCEDURE [sH:Stream.Handle];
```

After closing the stream, you should always set the stream handle variable to `NIL`, to ensure that you don't accidentally try to use it later on. (`Delete` invalidates the stream handle, but does not set it to `NIL`.)

12.5 Summary

Streams provide sequential access to data. You can use streams to access various devices, but this chapter only discussed how to use them to access the contents of local disk files.

To create a stream, you need to use a procedure that knows something about the device to which you are attaching a stream. To attach a stream to a local disk file, you use `NSFileStream.Create`.

Once you have created a stream, you perform I/O with the `Stream` procedures `PutByte`, `PutWord`, `PutBlock`, `GetByte`, `GetWord`, and `GetBlock`.

To do random access, use `stream.GetPosition` and `stream.SetPosition`.

When you are through, call `Stream.Delete` to delete the stream.

12.6 Exercises

The tool for this chapter is the Stream Tool, which allows you to load a file (simple document) into the tool window, or load the contents of the window into a file.

To load a file into the tool, select the file and invoke `Load`. To store the current contents of the window into a file, select the

destination file and invoke Save. You can use standard editing techniques to change the contents of the window.

You must write the commands Load and Save. The procedures that you need to change are in the module StreamToolImplTemp.mesa. You will also need the following modules:

StreamToolDefs
StreamToolImpl
StreamToolImplTemp
StreamToolMsgImpl
StreamTool.config

Notes:

The last chapter discussed how to access the contents of a file with a stream; this chapter discusses how to access the contents of a file by mapping the file to virtual memory.

Streams are an easy way to access a file sequentially. For some applications, however, you want to access a file via a data structure rather than sequentially. For example, if you have a file that contains a binary tree of data, you probably need to access that data as a tree and not as a sequence of bytes. For this kind of application, you should modify the file by mapping it to virtual memory rather than using a stream.

This chapter describes how to use the facilities of the Services **NSSegment** interface and the underlying Pilot **Space** interface to map files to virtual memory. You should be familiar with the basic idea of virtual memory before you read this chapter.

13.1 Definition of terms

Page: One page is `Environment.wordsPerPage` (256) words or `Environment.bytesPerPage` (512) bytes.

13.2 Virtual memory overview

Since no machine ever has enough real memory, virtual memory has become the standard way to combine the resources of the disk drive with the resources of real memory. The idea is to create the illusion of an environment that has the size of the disk and the speed of main memory.

The underlying premise is that you don't have to have all of a program in real memory to execute the program. Rather, the parts of the program that are currently executing are in real memory, and the rest of it is "in virtual memory." Everything that is "in" virtual memory must be backed by a page on the disk; the disk is the real location of the information.

Since there are more pages in virtual memory than can fit in real memory, there will come a point when a program tries to access a page that is not currently in real memory. At this point, the operating system will *swap out* pages that are no longer needed and *swap in* the necessary page and any other pages of the containing *swap unit*. (A swap unit is just a way to group pages so that they will be swapped in and out together.)

The operating system handles all swapping; you never know whether your code is in real memory or not. However, swapping pages in and out takes more time than just loading a program into real memory and executing it. Virtual memory

allows you to execute more programs, but each individual program may be somewhat slowed by swapping.

13.3 Mapping

Every piece of virtual memory that contains useful information must have a backing file on the disk. Typically the backing file for a piece of virtual memory is anonymous, but it is possible to set up a mapping between a specific file and a specific piece of virtual memory. Thus, you can get a pointer to a piece of virtual memory that is backed by your file, modify the virtual memory, and then write the information back out to the file on the disk.

The simplest view of the relationship between an interval of virtual memory and its backing store is that the virtual memory and the file always contain the same data. This isn't really the way it works, however, since changes to the virtual memory interval aren't immediately reflected in the backing file.

A page that has been changed in memory but not in the backing file is "dirty." The operating system writes dirty pages to the backing file before it swaps the page out of virtual memory, but you should not count on this for backup since you have no way of knowing when or if it will happen. Instead, you can explicitly write dirty pages to the disk with procedure calls, either in the middle of modifications (as an intermediate backup) or when you are through making changes.

The rest of this chapter describes how to map a file to virtual memory, how to modify it, and how to write the modified contents of virtual memory back to the file.

13.3.1 Setting up the mapping

To map a file to virtual memory, you call `NSSegment.Map`:

```
NSSegment.Map: PROCEDURE [  
  origin: NSSegment.Origin,  
  access: NSFile.Access ← NSFile.ReadAccess,  
  usage: Space.Usage ← Space.unknownUsage,  
  life: Space.Life ← alive, --or dead  
  swapUnits: Space.SwapUnitOption ←  
    Space.defaultSwapUnitOption,  
  session: NSSegment.Session ← NSSegment.nullSession]  
RETURNS [mapUnit: Space.Interval];
```

```
NSSegment.Origin: TYPE = RECORD [  
  file: NSFile.Handle,  
  base: NSSegment.PageNumber,  
  count: NSSegment.PageCount,  
  segment: ID ← NSSegment.defaultID];
```

```
Space.Interval: TYPE = RECORD [  
  pointer: LONG POINTER,  
  count: Environment.PageCount];
```

As illustrated in Figure 13.1, **Map** takes a specified portion of a file, copies it to virtual memory, and returns a pointer to the appropriate pages in virtual memory. (The next section provides detail on the `SpaceInterval` that **Map** returns.)

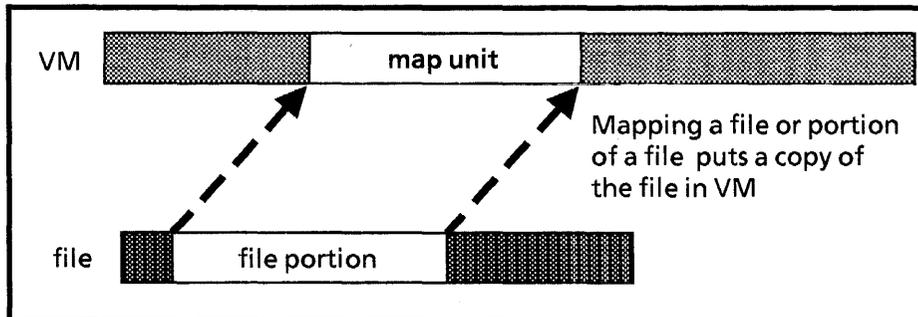


Figure 13.1 Mapping a file to virtual memory

The **origin** record specifies the location of the information that you want to map. Within an **origin**, **file** specifies the file of interest, **base** is the page in that file where you want to start processing, and **count** is the number of pages of interest.

The **segment** field allows you to specify the portion of a file that you are interested in. Every file is composed of **segments**. There must be at least one segment, the default segment, in a file. Although the **NSegment** interface currently supports multi-segment files, their use is not recommended. Thus, you should always leave the **segment** field defaulted. Figure 13.2 illustrates an **origin** record.

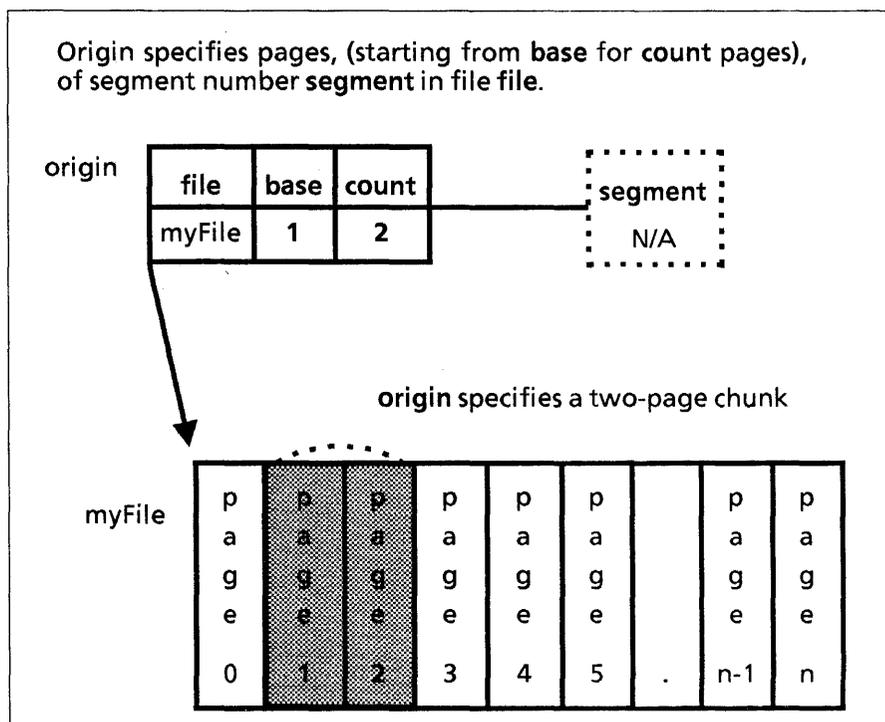


Figure 13.2 An `NSegment.Origin`

The **access** parameter to **Map** determines whether you are allowed to write in the mapped space after you map a file. You must have at least **read** access, or you will raise `NSFile.Error`.

Note that the default access is read (even if you have write access to the file), so you will have to change this if you wish to write the file.

`usage` provides a hint as to how the space will be used. This data will be made available to tools like the debugger and the performance management package. The `Space` interface defines an interval of usages that is further managed by the `SpaceUsage` interface. See the *Pilot Programmer's Manual* for details.

`life` specifies whether or not the initial contents of the backing file are useful. `alive` specifies that a swap unit initially contains useful data; `dead` specifies that it does not. See the *Pilot Programmer's Manual* for more details.

The `swapUnit` parameter specifies the size of the swap units that Pilot should use when your program takes a page fault. If you don't specify a size, it will be defaulted. See the *Pilot Programmer's Manual* for details of default swap units.

`session` is as described in Chapter 10, *NSFile Attributes*.

Here is an example of calling `Map`:

```
-- set up to retrieve attributes.
selections: NSFile.Selections;
attributes: NSFile.AttributesRecord;

--open the file
file: NSFile.Handle ← NSFile.OpenByReference[fileReference];

--retrieve the sizeInPages attribute
selections.interpreted[sizeInPages] ← TRUE;
NSFile.GetAttributes[file, selections, attributes];

-- map file to VM so we can write to it. Specify entire file as
--origin; allow write access.
space: Space.Interval ← NSSegment.Map[
    origin: [file:file, base: 0, count: attributes.sizeInPages],
    access: NSFile.fullAccess];
```

This example maps an entire file. The first step is to retrieve the `sizeInPages` attribute to find out how big the file is. The second step is to map the file, using the `sizeInPages` information to specify that we are mapping the entire file.

13.3.2 Accessing the file

`Map` returns a `Space.Interval`, which describes a sequence of pages in the virtual memory. An `Interval` contains a pointer to the first page and a count of the number of pages:

```
Space.Interval: TYPE = RECORD[
    pointer: LONG POINTER, count: Environment.PageCount];
```

Since you have a pointer to the starting address of the interval, you can impose any structure on your interval of virtual memory by creating a variable that is a pointer to a data object

and then assigning to that pointer the address of the segment. (This typically involves a LOOPHOLE operation.) For example:

```
--declare a record type with a LONG POINTER and two BOOLS
RecType: TYPE = RECORD [
  someString: XString.Reader,
  isEnglish: BOOLEAN,
  isAlphabetized: BOOLEAN];

RecPointer: TYPE = LONG POINTER TO MyRec;

--map the file....
space: Space.Interval ← NSSegment.Map[...];

-- use somePtr to access the contents of the file as if
--it were a variable of type RecType. Notice that we never
--actually declare a variable of type RecType; we just use the
--virtual memory as if it were a variable of RecType.
somePtr: RecPointer ← LOOPHOLE[space.pointer];
somePtr.isEnglish ← FALSE;
...
```

This example sets up a record structure with three fields, maps a file to virtual memory, and then access the virtual memory as if it were a variable of type `RecType`. Figure 13.3 illustrates this idea.

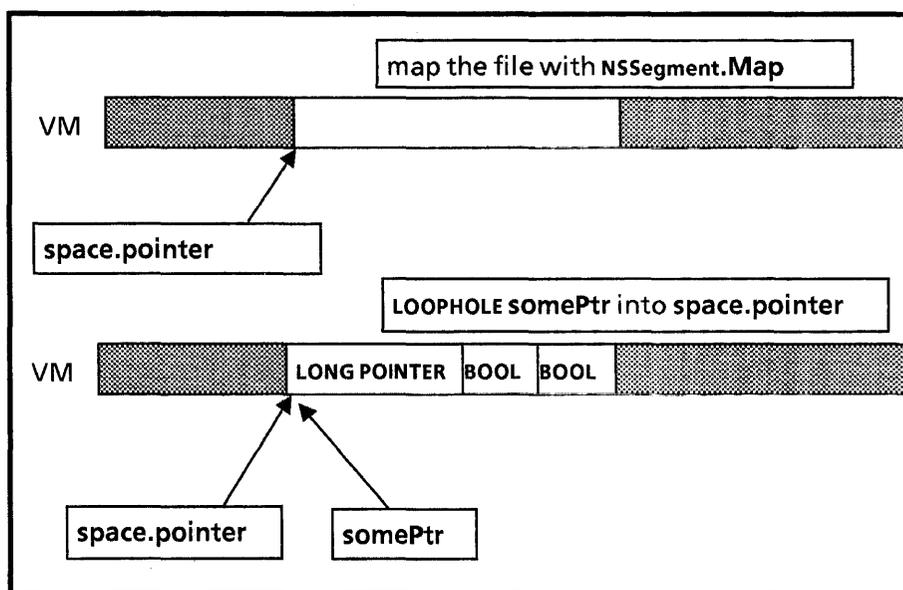


Figure 13.3: Mapping a file and then LOOPHOLING

13.3.3 Updating the file

When you are finished with a mapped space, you should call `Space.Unmap`:

```
Space.Unmap: PROCEDURE [
  pointer: LONG POINTER,
  returnWait: Space.ReturnWait ← wait]
  RETURNS [nil: LONG POINTER];
```

`pointer` denotes your virtual memory interval; `returnWait` specifies whether Pilot backs up dirty pages before returning to the client. (`wait` is currently the only value implemented; you should always just default this parameter.) `Unmap` returns a `NIL` pointer; the idea is that you should do something like this:

```
myPointer ← space.Unmap[myPointer];
```

`Unmap` will write any dirty pages, and free the virtual memory that the space occupies, as illustrated in Figure 13.4. You should call `Unmap` regardless of whether you have made any changes.

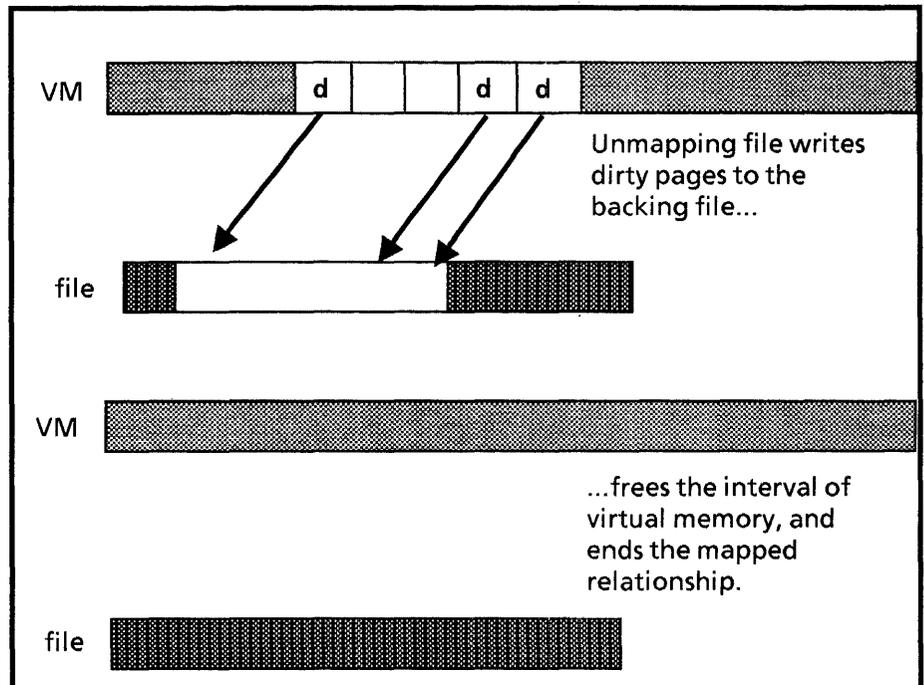


Figure 13.4: Unmapping

You can also back up dirty pages manually, with the procedure `space.ForceOut`:

```
space.ForceOut: PROCEDURE [interval: space.Interval];
```

You can call this procedure at any time to force your changes to be written to the disk. How often you call `ForceOut` is a judgment call based on the importance of the data that you are manipulating and the probability of a machine failure.

13.3.4 Complete mapping example

Here is a program fragment that keeps a record of the ingredients that are used to make various kinds of drinks. It maps the data file to virtual memory in order to add a new drink.

```

NSSegmentExample: PROGRAM IMPORTS NSFile, NSSegment,
Space =
BEGIN

--max number of drinks that can be on file
maxNumberOfDrinks: CARDINAL = 255;
--max number of ingredients in a given drink
numOfIngredients: CARDINAL = 32;
--A drink
Drink: TYPE = PACKED ARRAY [0..numOfIngredients) OF BOOL];

--the main data structure. Count keeps track of the number
--of drinks currently on file; bitmap is an array of drinks,.
ContentBitMap: TYPE = RECORD [
    count: CARDINAL, -- number of drinks in file
    bitmap: ARRAY [0..maxNumberOfDrinks) OF Drink;
BitMapPtr: TYPE = LONG POINTER TO ContentBitMap;

...
--This procedure is called when it is time to add a new drink
--to the database
WriteContents: PUBLIC PROC[
    drink: Drink ,
    fileReference: NSFile.Reference] = {

--get the size of the file in pages. If the file is not an exact
--multiple of the page size, include an extra partial page.
sizeInPages: LONG CARDINAL ←
    SIZE[ContentBitMap] / Environment.wordsPerPage;
IF SIZE[ContentBitMap] MOD Environment.wordsPerPage # 0
    THEN sizeInPages ← sizeInPages + 1;

--open the file
file: NSFile.Handle ← NSFile.OpenByReference[fileReference];

-- map file to VM so we can write to it. Specify entire file as
--origin; allow write access.
space: Space.Interval ← NSSegment.Map[
    origin: [file:file, base: 0, count: sizeInPages],
    access: NSFile.fullAccess];

-- use contentPtr to access the contents of the file as if
--it were a variable of type ContentBitMap
contentPtr: BitMapPtr ← LOOPHOLE[space.pointer];

--set index to be the next available slot for new drink
index: CARDINAL ← contentPtr.count;
--update count of drinks. Raise a signal if we're out
--of room.
contentPtr.count ← index + 1;
    IF contentPtr.count > maxNumberOfDrinks
        THEN SIGNAL FileFull;
-- write ingredients of new drink to mapped contents file
FOR i: CARDINAL IN [0..numOfIngredients) DO
    contentPtr.bitmap[index][i] ← drink[i];
ENDLOOP;

-- unmap the interval, thus writing changes back to the file
space.pointer ← Space.Unmap[space.pointer!
    Space.Error = > CONTINUE];
NSFile.Close[file:file]);
END...

```

The **WriteContents** procedure is called with two arguments: a new drink, and a reference to the file where the data is stored. **WriteContents** should add the new drink to the file.

The first step is to map the file to virtual memory. To do this, we get a handle to the file, and set up an origin that represents the entire file. We then call **NSSegment.Map** to map the file to virtual memory.

Once we have the interval of virtual memory, we **LOOPHOLE space.pointer** into a pointer to our record structure. We can then access the virtual memory as if it were a variable of type **ContentBitMap**. We add the new drink to the file, and then call **Unmap** to write the changes back out to the file. Finally, we call **NSFile.Close**.

13.4 CopyIn/ CopyOut

NSSegment.CopyIn and **NSSegment.CopyOut** allow you to copy from virtual memory to a file and vice versa. Thus, for example, you can map one file to virtual memory and then use the **CopyOut** operation to copy the information to a second file. Figure 13.5 illustrates this idea.

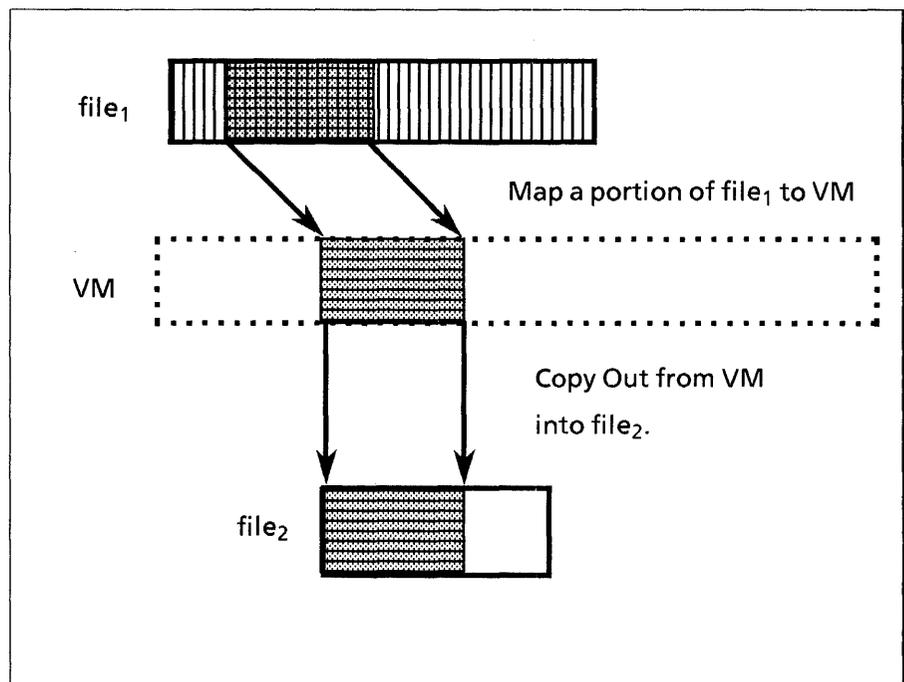


Figure 13.5: The **CopyOut** operation

An example of a situation in which you would want to use one of the Copy procedures is in copying portions of files. You might first map a file to a space and then call **CopyOut** to copy part of its contents into another file.

Here is some code that uses **CopyOut** to copy one entire file to another:

```

Copy: PROCEDURE[fromFile, toFile: NSFile.Handle]
RETURNS[copied: BOOLEAN ← FALSE] =
BEGIN
  ENABLE {NSFile.Error = > GOTO Exit;
  Space.Error = > GOTO Exit;};
  BEGIN
    pages: NSSegment.PageCount;
    length: NSSegment.PageCount ← NSSegment.GetSizeInPages[
      file: fromFile];  --length of original file
    fromOrigin: NSSegment.Origin ← [
      file: fromFile,
      base: 0,
      count: length];
    toOrigin: NSSegment.Origin ← [
      file: toFile,
      base: 0,
      count: length];

    --map original file to virtual memory
    space: Space.Interval ← NSSegment.Map [
      origin: fromOrigin]; --default access is read
    NSSegment.SetSizeInPages[file: toFile, pages: length];

    --copy from first file to second file
    pages ← NSSegment.CopyOut[
      pointer: space.pointer, origin: toOrigin];
    IF pages = length THEN copied ← TRUE;
    space.pointer ← Space.Unmap[space.pointer];
  END;
  EXITS Exit = > RETURN;
END;

```

13.5 Summary

To establish a mapping between a file and an interval of virtual memory, call `NSSegment.Map`. `Map` takes several parameters, the most important of which are `origin`, and `access`. `origin` specifies the portion of a file that you are interested in, and `access` specifies whether or not you can write the space. `Map` returns a `Space.Interval`, which describes a portion of virtual memory.

You can then use `LOOPHOLE` to access that interval as if it were the data structure of your choice. When you are through making changes, you need to call `Space.Unmap` to write your changes out to the file. If you want to save your information at an intermediate point, you can call `space.ForceOut`.

You can also use `NSSegment.CopyIn` and `NSSegment.CopyOut` to copy data from one file into another via virtual memory. For example, you could first map a file to a space and then call `CopyOut` to copy part of its contents into another file.

For more information on the topics in this chapter, consult the `Space` chapter of the *Pilot Programmer's Manual* and the `NSSegment` chapter of the *Filing Programmer's Manual*.

Finally, here is the complete filing example that we promised you. The code creates the tool illustrated in Figure 13.6

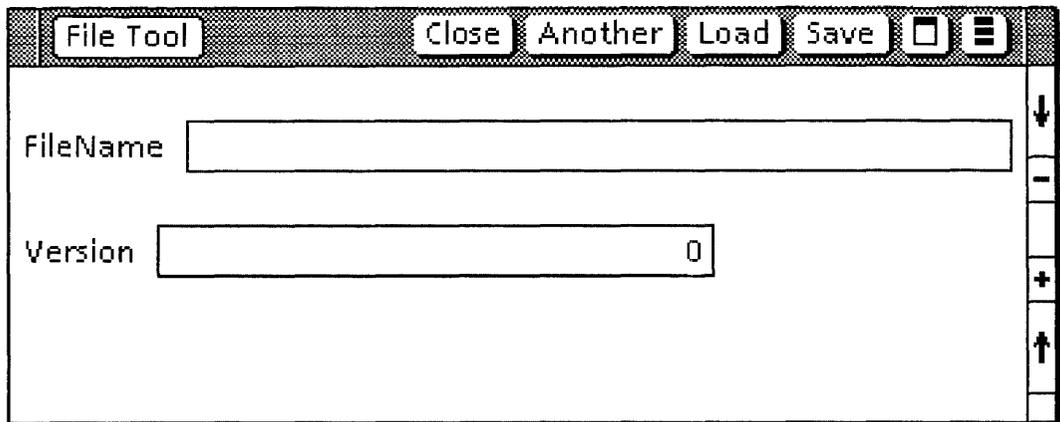


Figure 13.6: The File Tool

With this tool, the user can create a new file in the system catalog by filling in a name and version, typing the desired contents of the file in the space below the version field, and then invoking Save. Similarly, he can load an existing file with the Load command or delete a file with the Destroy command.

DIRECTORY

Atom,
 Attention,
 BWSFileTypes,
 Context,
 FormWindow,
 Catalog,
 Environment,
 Heap,
 MenuData,
 NSFile,
 NSSegment,
 NSSString,
 NSFileStream,
 Process,
 Space,
 StarWindowShell,
 Stream,
 TIP,
 Window,
 XChar,
 XString;

BWSFileTool: PROGRAM

```
IMPORTS Attention, Catalog, Context, FormWindow, Heap,
MenuData, NSFile, NSFileStream, NSSegment, Space,
StarWindowShell, Stream, XString SHARES XString =
```

BEGIN

```
fileType: NSFile.Type = 100101; -- arbitrary
Items: TYPE = {nameItem, versionItem, textItem};
context: Context.Type ← Context.UniqueType[];
bodyWindowDims: Window.Dims = [1000, 1000];
```

```
Data: TYPE = LONG POINTER TO DataObject;
DataObject: TYPE = RECORD [
  fileHandle: NSFile.Handle ← NSFile.nullHandle,
  space: Space.Interval ← Space.nullInterval,
  string: XString.ReaderBody ← NULL];
```

--Procedures

-- delete the specified file

```
Destroy: MenuData.MenuProc = {
  noName: XString.ReaderBody ←
    XString.FromSTRING ["No name specified"L];
  name: XString.ReaderBody;
  fileName: NSFile.Handle;
  body: Window.Handle ←
    StarWindowShell.GetBody[[window]];
```

--get name and version from form window

```
name ← FormWindow.GetTextItemValue[
  body, 0, Heap.systemZone];
IF XString.Empty[@name] THEN
  {Attention.Post[@noName]; RETURN;};
```

--get a handle to file and then delete it.

```
fileName ← Catalog.GetFile[name: @name
  ! NSFile.Error = > GOTO Exit];
NSFile.Delete[fileName! NSFile.Error = > GOTO Exit2];
```

--set the file name and contents to NIL in form window.

```
FormWindow.SetTextItemValue[
  body, Items.nameItem.ORD, NIL];
FormWindow.SetTextItemValue[
  body, Items.textItem.ORD, NIL];
```

EXITS

```
Exit = > {
  error: XString.ReaderBody ←
    XString.FromSTRING ["NSFile create error"L];
  Attention.Post[@error]; RETURN;};
Exit2 = > {
  error2: XString.ReaderBody ←
    XString.FromSTRING ["NSFile Delete error"L];
  Attention.Post[@error2]; RETURN; } };
```

-- return the context for this window body

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

--register "File Tool" command in Attention Menu

```
Init: PROC = {
  fileTool: XString.ReaderBody ←
    XString.FromSTRING["File Tool"L];
  Attention.AddMenuItem [MenuData.CreateItem [
    zone: Heap.systemZone,
    name: @fileTool,
    proc: MenuProc] ]};
```

```

-- get the contents of the file by mapping the file to VM
-- and then LOOPHOLING a pointer to the characters.
-- Display contents of file in window.
Load: MenuData.MenuProc = {
  body: Window.Handle ←
    StarWindowShell.GetBody[[window]];
  data: Data ← GetContext[body];
  sizeInBytes: LONG CARDINAL ← 0;
  deleted: XString.ReaderBody ←
    XString.FromSTRING ["File Deleted"L];
  name: XString.ReaderBody;
  version: LONG INTEGER;
  myBlock: Environment.Block ← [NIL, 0, 0];

  BEGIN ENABLE {
    Space.Error = >
      {error: XString.ReaderBody ←
        XString.FromSTRING ["Space Error"L];
        Attention.Post[@error];
        GOTO Exit;};
    NSFile.Error = > {NSFileError[error]; GOTO Exit};
    XString.InvalidNumber, XString.Overflow = >
      {error: XString.ReaderBody ←
        XString.FromSTRING ["Space Error"L];
        Attention.Post[@error];
        GOTO Exit} };

    --get name and version from input in form window
    name ← FormWindow.GetTextItemValue[
      body, Items.nameItem.ORD, Heap.systemZone];
    version ← FormWindow.GetIntegerItemValue[
      body, Items.versionItem.ORD];

    --open file
    data.fileHandle ← Open[name, version];
    IF data.fileHandle = NSFile.nullHandle THEN RETURN;

    --map entire file to virtual memory
    data.space ← NSSegment.Map[
      origin: [
        data.fileHandle,
        0,
        NSSegment.GetSizeInPages[data.fileHandle],
        swapUnits: [uniform[4]]];

    --access virtual memory as if it is a Environment.block
    sizeInBytes ←
      NSSegment.GetSizeInBytes[data.fileHandle];
    myBlock.stopIndexPlusOne ← CARDINAL[sizeInBytes];
    myBlock.blockPointer ← data.space.pointer;
    data.string ← XString.FromBlock[block: myBlock];

    --display contents of file in form window and then close
    --everything up.
    FormWindow.SetTextItemValue[
      body, Items.textItem.ORD, @data.string];
    data.space.pointer ←
      space.Unmap[data.space.pointer];
    NSFile.Close[data.fileHandle];
    EXITS Exit = > RETURN;
  END };

```

```

-- Build the window and menu commands
MenuProc: MenuData.MenuProc = {
  another: XString.ReaderBody ←
    XString.FromSTRING["Another"L];
  load: XString.ReaderBody ←
    XString.FromSTRING["Load"L];
  save: XString.ReaderBody ←
    XString.FromSTRING["Save"L];
  destroy: XString.ReaderBody ←
    XString.FromSTRING["Destroy"L];
  fileTool: XString.ReaderBody ←
    XString.FromSTRING["File Tool"L];

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

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

--add commands
z: UNCOUNTED_ZONE ← StarWindowShell.GetZone [shell];
items: ARRAY [0..4] OF MenuData.ItemHandle ← [
  MenuData.CreateItem [
    zone: z, name: @another, proc: MenuProc],
  MenuData.CreateItem [
    zone: z, name: @load, proc: Load],
  MenuData.CreateItem [
    zone: z, name: @save, proc: Save],
  MenuData.CreateItem [
    zone: z, name: @destroy, proc: Destroy]];

--create menu of commands
myMenu: MenuData.MenuHandle =
  MenuData.CreateMenu [
    zone: z,
    title: NIL,
    array: DESCRIPTOR [items]];

--add commands to header
StarWindowShell.SetRegularCommands [
  sws: shell, commands: myMenu];

-- Allocate data and "hang it off the body window"
Context.Create [ type: context,
  data: Heap.systemZone.NEW[DataObject ← []],
  proc: Context.SimpleDestroyProc,
  window: body];

--make the body window a form window
FormWindow.Create [window:body,
  makeItemsProc:MakeFormItems,
  zone:Heap.systemZone];

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

```

--MakeItemsProc for form window. Note that there are
 --three fields: a text field for the name of the file, an
 --integer item for the version, and another text item for
 --the contents of the file. The contents text item does not
 --have a box around it, so it is not visible on the screen.

```
MakeFormItems: FormWindow.MakeItemsProc = {
  nameTag: xString.ReaderBody ←
    xString.FromSTRING ["FileName" L];
  versionTag: xString.ReaderBody ←
    xString.FromSTRING ["Version" L];
```

```
FormWindow.MakeTextItem[
  window: window,
  myKey: Items.nameItem.ORD,
  tag: @nameTag, width: 300];
```

```
FormWindow.MakeIntegerItem[
  window: window,
  myKey: Items.versionItem.ORD,
  tag: @versionTag, width: 200];
```

```
FormWindow.MakeTextItem[
  window: window,
  boxed: FALSE,
  myKey: Items.textItem.ORD,
  width: 600] };
```

-- print NSFile error messages

```
NSFileError: PROCEDURE [error: NSFile.ErrorRecord] =
BEGIN
```

```
  accessError: xString.ReaderBody ←
    xString.FromSTRING ["accessError" L];
  attributeTypeError: xString.ReaderBody ←
    xString.FromSTRING ["attributeTypeError" L];
  handleError: xString.ReaderBody ←
    xString.FromSTRING ["handleError" L];
  spaceError: xString.ReaderBody ←
    xString.FromSTRING ["spaceError" L];
  undefinedError: xString.ReaderBody ←
    xString.FromSTRING ["undefinedError" L];
```

```
  WITH myError: error SELECT FROM
    access = > Attention.Post[@accessError];
    attributeType, attributeValue = >
      Attention.Post[@attributeTypeError];
    handle = > Attention.Post[@handleError];
    space = > Attention.Post[@spaceError];
    undefined = > Attention.Post[@undefinedError];
```

```
  ENDCASE;
```

```
END;
```

14. SELECTION REQUESTORS

This chapter discusses how to use the **Selection** interface, which defines the abstraction of the user's current selection.

There are two kinds of programs that use the **Selection** interface: *requestors* and *managers*. Requestors are programs that want to obtain the value of the current selection; managers are programs that own and change the current selection. For example, the desktop implementation is a selection *manager*: when the user selects an icon, the desktop implementation must implement that selection. A printer icon, on the other hand, is a requestor: it needs to know the value of the currently selected icon so that it can print the correct file, but it does not change the value of that selection.

This chapter covers only requestors; for information on selection managers you will have to consult the **Selection** chapter of the *ViewPoint programmer's Manual*.

14.1 Converting the selection

When the user selects something and then asks an application to operate on that selection, the application needs to get the value of the current selection. An application typically can't act on all possible selections, however. For example, if the user selects something, presses **COPY**, and then selects a printer icon, the printer application can only print the selection if it is a document, a folder, an interpress master, or the like. If the selection is a paragraph of text, or another printer icon, then the printer cannot accept the selection. Similarly, a document editor might only be interested in the selection if it is a string.

Thus, a selection requestor is only interested in the selection if it can obtain that selection in a particular format. To request the selection as a particular data type, you call **Selection.Convert**:

```
Selection.Convert: PROCEDURE [  
    target: Selection.Target,  
    zone: UNCOUNTED_ZONE ← NIL]  
    RETURNS [value: Selection.Value];
```

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

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

```
Selection.nullValue: Selection.Value = [value: NIL, ...];
```

Convert is a request to produce the selection as an object of **TYPE target**. (Note that the **target** is just the type, not the actual data object.)

When a requestor calls **Convert**, the **Selection** implementation will call the selection manager to obtain the selection in the desired format. The selection manager determines whether or not it supports conversion to a particular target; most managers only support a limited number of targets. For example, if the selection is a text string, the manager might implement conversion to **string** and perhaps to **integer**, but probably not to **file** or **fileType**.

If the conversion is possible, the return parameter **value.value** will be a **LONG POINTER TO** the converted selection. (There are also other fields in **value**, but they are for the use of selection managers only. You don't have to worry about them for now.) If the conversion is not possible, **Convert** will return **nullValue**.

Notice that **target** is an open-ended enumeration of data types. The **Selection** interface defines a number of common **targets**, and the values that each returns. If the target type that you need is not among those defined in this interface, you can define a new **target** with a call to **selection.UniqueTarget**. See the *ViewPoint Programmer's Manual* for details.

Here is a list of the most frequently used **targets** and the results that they return; for more information on any of the values in **Selection.Target**, see the **Selection** chapter of the *ViewPoint Programmer's Manual*:

window	returns a Window.Handle for the window that contains the current selection.
shell	returns a StarWindowShell.Handle for the shell that contains the current selection.
string	returns an XString.Reader representing the text of the current selection.
length	returns a LONG POINTER TO LONG CARDINAL containing the number of characters in the selection.
integer	returns a LONG POINTER TO LONG INTEGER containing the selection as a number.
interpressMaster	returns a Stream.Handle to an interpress master.
file	returns a LONG POINTER TO NSFile.Reference for the file associated with the selection.
fileType	returns a LONG POINTER TO NSFile.Type for the file associated with the selection.

Here is an example of calling **Selection.Convert** with a target type of **string**:

```

streamHandle: Stream.Handle ← --GetStreamToSomeFile--;
xfo: XFormat.Object ← XFormat.StreamObject[streamHandle];
-- Convert returns nullValue if manager can't convert selection;
-- nullValue has a value.value of NIL.
savedString: Selection.Value ← Selection.Convert[string];
IF savedString.value = NIL THEN {
    Stream.Delete[streamHandle]; RETURN};
XFormat.Reader[@xfo, LOOPHOLE[
    savedString.value, XString.Reader]];
Stream.Delete[streamHandle];
Selection.Free[@savedString];

```

This example sets up a stream to a file, creates an **XFormat** object that contains the stream, and then calls **Convert** to try to convert the selection to a string. If the conversion is not possible, the procedure just returns. If the conversion is possible, it sends the string to the stream, and then returns, deleting the stream and calling **Selection.Free** on the way out.

You must call **selection.Free** when you are through with the selection; this allows the selection manager to deallocate any storage that it may have allocated:

```
Selection.Free: PROCEDURE [v: Selection.ValueHandle];
```

Note: To guarantee that the user cannot alter the selection while you are reading it, you should only read the selection from within the Notifier. If you are not responding to a user command when you need to get the value of the selection, you should use a periodic notifier. See Chapter 9, TIP, for more information on the Notifier and periodic notifiers.

14.2 Resource management

An important rule for requestors is that the selection belongs to the manager, not the requestor. **Convert** returns a *read-only value*: you can look at the data in *value*, but you cannot modify it. If you want to use the selection later, or pass it to another process, you must first copy the selection using **Selection.Copy**, **Selection.Move**, or **Selection.CopyMove**:

```
Selection.Copy: PROCEDURE [
    v: Selection.ValueHandle,
    data: LONG POINTER] = INLINE {CopyMove[v,copy, data]};
```

```
Selection.Move: PROCEDURE [
    v: Selection.ValueHandle,
    data: LONG POINTER] = INLINE {CopyMove[v,move, data]};
```

```
Selection.CopyMove: Selection.ValueCopyMoveProc;
```

```
Selection.ValueCopyMoveProc: TYPE = PROCEDURE [
    v: Selection.ValueHandle, op: CopyOrMove,
    data: LONG POINTER];
```

```
Selection.CopyOrMove: TYPE = {copy, move};
```

```
Selection.ValueHandle: LONG POINTER TO Selection.Value;
```

The meaning of **data** depends on the target. It is often used as the storage for the destination of the copy. See the *ViewPoint*

Programmer's Manual for the exact meaning of data for a particular target.

Here is an example of calling **Copy**:

```
GetSelectedFile: PROC RETURNS [stream: Stream.Handle ← NIL] = {
  element: Selection.Value ← Selection.Convert [target: file];
  IF element.value = NIL THEN SIGNAL FileProblem
  ELSE BEGIN --conversion was successful
    ref: LONG POINTER TO NSFile.Reference;
    handle: NSFile.Handle;
    dir: NSFile.Reference ←
      StarDesktop.GetCurrentDesktopFile[];
    Selection.Copy[@element, @dir];
    ref ← element.value;
    handle: NSFile.Handle ← NSFile.OpenByReference [
      reference: ref ↑ !NSFile.Error = > SIGNAL FileProblem];
    stream ← NSFileStream.Create[handle];
    Selection.Free[@element];
  END;
};
```

This example obtains the value of the currently selected file. If the selection cannot be converted to a file, then raise a signal indicating that there is a problem. If there is no problem, then the next step is to copy the selection.

When the selection is a file, the data parameter to **Copy** should be a **LONG POINTER TO NSFile.Reference** to the directory in which you want to store the file. (Remember, you need to consult the *ViewPoint Programmer's Manual* to find out what data should be for a given selection type.)

In this case, we call **StarDesktop.GetCurrentDesktopFile** to get a reference to the current desktop, and then call **Copy**, passing in the desktop reference as the data parameter. Once we have copied the selection, we open the file, get a stream to the file, call **Selection.Free**, and then return.

If the selection is a string, you can use standard **String** routines, such as **XString.CopyReader** or **XString.CopyToNewWriterBody** or the like, to copy the string instead of using **Selection.Copy**. For example:

```
GetString: PUBLIC PROCEDURE [] RETURNS
  [string: XString.ReaderBody] = {
  value: Selection.Value ← Selection.Convert[string];
  string ← IF value.value = NIL
    THEN GetConstantString[]
    ELSE XString.CopyReader[LOOPHOLE
      [value.value, XString.Reader], sysZ] ↑ ;
  Selection.Free[@value];
};
```

This example gets the currently selected string by calling **Convert** with a target type of **string**. If the return value is **NIL** (the conversion did not succeed), then call the procedure **GetConstantString** to get a default constant value. Otherwise, call **CopyReader** to copy the string, call **Free** to allow the manager to deallocate storage, and then return.

Note that in the case of the target type **stream**, calling **Copy** is particularly important. In this case, you cannot even read the stream without copying it, because reading the stream alters the stream state and thus alters the **stream.Object** to which the **stream.Handle** points. Thus, you must copy the stream handle before using the stream. (Even after you have copied the stream, you can only read it; the stream handle has read-only access.) Once you have copied the handle, you are responsible for the stream and you must call **stream.Delete** when you are through with it.

In all cases you must call **selection.Free** regardless of whether you copy the selection. If you do copy the selection, you own the resources, and you should eventually free them. If you don't copy the selection, the manager owns the resources but you should still call **Free** to let him know that he can free the resources.

14.3 Can you convert the selection?

Since all selections do not convert to all target types, you might want to find out whether a particular conversion is possible, and possibly get an indication of how difficult the conversion would be before you actually do the conversion. This is useful in cases where the conversion is expensive (for example, when the target type is an interpress master), and you want to find out whether or not the conversion is possible before you waste too much time. To find out this information, you can call **Selection.CanYouConvert**.

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

The **enumeration** parameter allows you to ask if the manager supports enumerating the selection (see the next section.)

CanYouConvert returns a **BOOLEAN** specifying whether the manager supports conversion to the specified type. Note that this is just an indication of whether the manager believes it is possible; it is not a guarantee that the conversion will actually work. The manager might still run into some unexpected difficulties, such as running out of disk space or the like.

If you want a specific indication of how hard the conversion will be, instead of an indication of whether it is possible, you can call **HowHard** directly:

```
Selection.HowHard: PROCEDURE [
  target: Selection.Target,
  enumeration: BOOLEAN ← FALSE]
  RETURNS [difficult: Selection.Difficulty];
```

```
Selection.Difficulty: TYPE = {easy, moderate, hard, impossible};
```

Here is an example of calling **CanYouConvert**. This type of procedure is generally called from the **canYouTakeSelection** arm of a **Containeer.GenericProc**. This code is interested in the selection only if it is an integer or a string. If the selection

manager does not implement conversion to any of these types, then this application cannot accept the selection.

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

Note also that there is a `Selection.Query` procedure, which effectively allows you to ask multiple `CanYouConvert` questions at the same time. See the *ViewPoint Programmer's Manual* for details.

14.4 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 (such as a long string). In particular, there is a limit on the size of string that `Convert` can return. A call to `Convert[string]` never produces a string longer than `Selection.maxStringLength` (200) characters; if the selection is a longer string you will have to treat it as a sequence of strings.

A requestor can convert each part of such a selection individually by calling `Selection.Enumerate`.

```
Selection.Enumerate: PROCEDURE [
  proc: Selection.EnumerationProc,
  target: Selection.Target,
  data: Selection.RequestorData ← NIL,
  zone: UNCOUNTED_ZONE ← NIL]
  RETURNS [aborted: BOOLEAN]

Selection.EnumerationProc: TYPE = PROCEDURE [
  element: Selection.Value,
  data: Selection.RequestorData]
  RETURNS [stop: BOOLEAN ← FALSE];

Selection.RequestorData: TYPE = LONG_POINTER;
```

Calling `Enumerate` will result in a call to `proc` for each piece of the selection. Each time it calls `proc`, the `Selection` implementation will pass `data` as a parameter. This `data` is for your own use; it can be `NIL` if you like.

Calling `Enumerate` is thus roughly equivalent to calling `Convert` for each individual piece. (If the manager can't convert the selection to the specified `target`, `proc` won't be called.) If you want to stop the enumeration, you can set `stop` to be `TRUE` in your `proc`.

Here is an example of using `Enumerate`:

```

Absorb: PROCEDURE[data: Containee.DataHandle]
RETURNS[absorbed: BOOLEAN ← FALSE] =
BEGIN
  --This is the enumeration proc. It will be called once for
  --each piece of the selection. It appends element.value to
  --the end of the stream.
  AbsorbString: Selection.EnumerationProc = BEGIN
    XFormat.Reader[@xfo, LOOPHOLE[element.value]];
    Selection.Free[@element];
  END;

  --set up a format object whose output sink is a stream
  xfo: XFormat.Object;
  fileStream: NSFileStream.Handle ← GetStream [data];
  --set current position to be the end of the file
  Stream.SetPosition[
    fileStream, NSFileStream.GetLength[fileStream]];
  xfo ← XFormat.StreamObject [fileStream];
  --if the manager can convert the selection to a series of strings,
  --then call Selection.Enumerate; if it can be converted to a
  --single string, call AbsorbString directly.
  IF Selection.CanYouConvert [string, TRUE] THEN
    [] ← Selection.Enumerate[AbsorbString, string, NIL]
  ELSE {
    v: Selection.Value = Selection.Convert[string];
    IF v.value # NIL THEN [] ← AbsorbString[ v, NIL];
  }
  Stream.Delete[fileStream];
END;

```

The main code of **Absorb** starts by setting up a format object with a stream as its output sink. The next step is to call **CanYouConvert** to find out if it is possible to get the selection as a simple string. If it is, then call the procedure **AbsorbString** to add the string to the stream.

If isn't possible, call **CanYouConvert** again with enumeration **TRUE** to find out if it can be converted to a series of strings. If it can, call **Enumerate**. This will result in a call to **AbsorbString** for each string in the sequence. **AbsorbString** can then add the strings to the stream one at a time.

14.5 Summary

Applications that need to obtain the value of the current selection are *requestors*; applications that own and control the current selection are *managers*. This chapter discussed only requestors.

The principle action that requestors perform is asking for the selection in a particular form by calling **Convert**. The **Selection** interface defines a number of common target types for the selection; you can also define your own target. When a requestor calls **Convert**, the **Selection** interface in turn calls the selection *manager* to determine whether the conversion is possible. If the conversion is possible, **Convert** returns a long pointer to the selection. If the conversion is not possible, **Convert** returns **nullValue**.

The value that **Convert** returns belongs to the manager, not the requestor. If you want to use the value of the current selection after you leave the Notifier, or if you want to pass it

to another process, you must first copy the selection with **Copy**, **Move**, or **CopyMove**.

When you are through with the selection, you should call **Free** to free the resources associated with the selection. You need to do this regardless of whether or not you have copied the selection.

If you want to find out how difficult a particular conversion will be before you actually attempt it, you can call **CanYouConvert** or **HowHard**. These procedures provide information on whether the selection manager believes the conversion is possible; they do not produce a guarantee that the conversion will be successful.

You may also have to obtain the selection as a series of objects rather than a single object. For example, if the user copies a folder to the printer, the printer implementation will need to call **Enumerate** to get an interpress master for each of the documents in the folder. This procedure is often called when a call to **CanYouConvert** says that the manager supports conversion to the specified type, but a call to **Convert** fails.

14.6 Exercise

The exercise for this chapter is the Checkers Tool, which is illustrated in Figure 14.1.

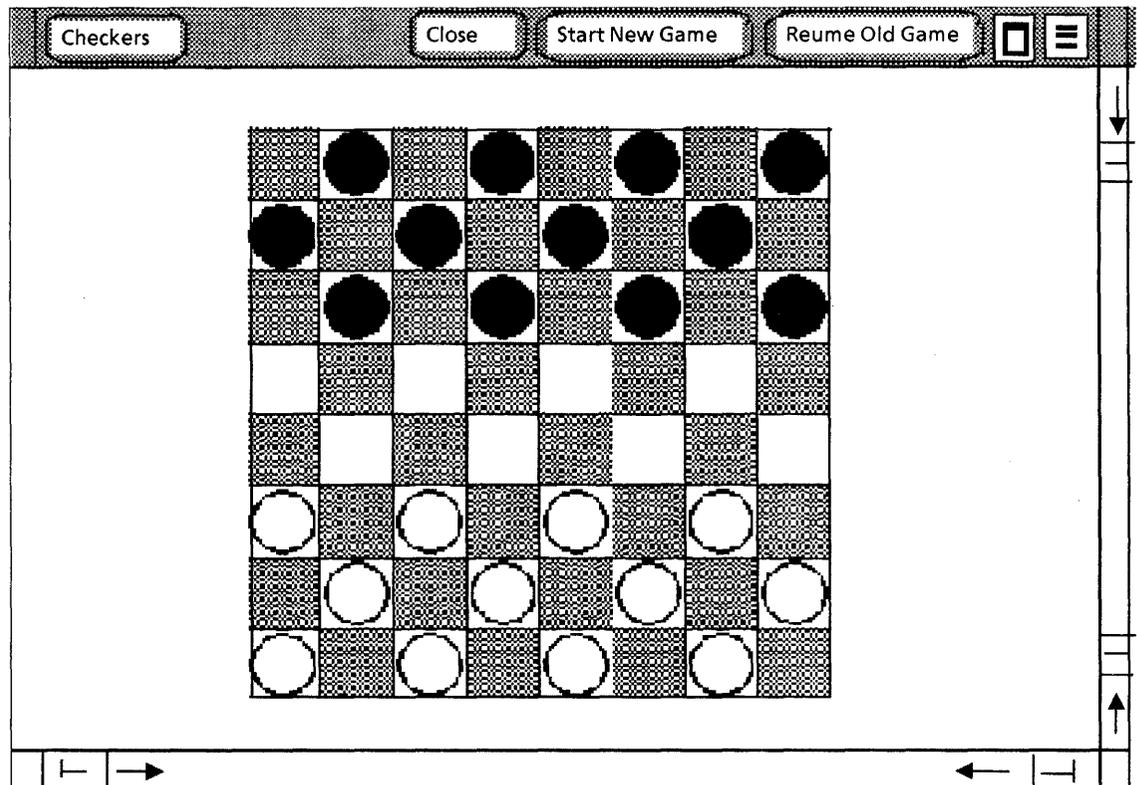


Figure 14.1: The checkers tool

To move a checker, you select the desired checker with the mouse and then select a new location for that checker. When

you select a new square, the checker is erased from the previous square and drawn in the new square.

If you don't finish a game, you can store it in a file by selecting a file on the desktop and invoking "StoreGame." To resume an old game, select the file that contains the game and invoke "ResumeOldGame." If the selection contains several files, the tool will enumerate the available files, and you can choose the game that you want to play from the Attention window.

Standard checkers rules apply. The tool checks the legality of moves, although it does not keep track of whose turn it is.

Your assignment is to write the code for the procedures **ResumeOldGame** and **GetSelectedFile** in the module **CheckersSelectionImplTemp**.

ResumeOldGame gets the current selection, converts it into a file, and reads the data in that file. **GetSelectedFile** is called when the tool needs a stream handle to a file; it converts the current selection to a file.

You will also need the following modules:

- CheckersDefs
- CheckersImpl
- CheckersMsgImpl
- CheckersBitmapImpl
- Checkers.config

Notes:

This chapter describes how to write an application that runs from an icon instead of from a command in the Attention Menu.

15.1 Overview

Icons on the desktop are part of what is referred to as the “user illusion.” The user sees various icons, such as printers and documents, and associates those icons with certain functions and files. He can open a document, copy a document to the printer, look at properties of the document, and so forth.

From the client’s point of view, however, an icon is a picture that represents a file. The desktop itself is a directory with children; there is an icon for each child of the desktop. Every application that uses icons must *register* with the desktop by specifying the file type on which it operates and providing procedures to display the icon on the screen and implement operations on that icon.

When the user first logs in, the desktop enumerates its children and calls the appropriate application to display an icon on the desktop. When the user selects an icon and wants to operate on it, the desktop determines the file type for that icon, associates the file type with an application, and then calls that application to perform the operation.

To write an icon application, you need to register with the desktop and write the appropriate call back procedures.

15.2 Registering with the desktop

To register an application with the desktop, call `Containeer.SetImplementation`:

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

```
Containeer.Implementation: TYPE = RECORD [
    implementors: LONG POINTER ← NIL,
    name: XString.ReaderBody ← XString.nullReaderBody,
    smallPictureProc: Containeer.SmallPictureProc ← NIL,
    pictureProc: Containeer.PictureProc ← NIL,
    convertProc: Selection.ConvertProc ← NIL,
    genericProc: Containeer.GenericProc ← NIL];
```

SetImplementation associates an **Implementation** record with a file type, and returns the **Implementation** previously associated with that file type.

implementors is a pointer for client-specific data; you can store the data of your choice here. **name** is a name for the objects on which the application operations, such as "spreadsheet." You can ignore both of these fields for now.

pictureProc is a call back procedure that displays the icon picture on the screen. Section 15.2.1 discusses **PictureProcs**.

smallPictureProc is similar to **pictureProc**, except that it creates a miniature version of the icon picture. Section 15.2.2 discusses **SmallPictureProcs**.

genericProc is a call back procedure that determines how the icon implements the "generic" operations **COPY**, **MOVE**, **OPEN**, and **PROPS**. Section 15.2.3 discusses **GenericProcs**.

convertProc is a call back procedure that determines whether the manager can convert the file to a particular type. This is important when you are managing the selection. For now, just use **Containeer.DefaultFileConvertProc**, which implements conversion to the types **file** and **fileType**. See the *ViewPoint Programmer's Manual* for more information.

Here is an example of calling **SetImplementation**:

--register application with desktop. Called from mainline code.

--myFileType is a global variable

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

This example declares the local variable **newImpl**, and then calls **GetImplementation** to store the existing implementation in **newImpl**. It then allocates the variable **oldImpl** (a global **LONG POINTER TO Containeer.Implementation**), and stores the old implementation in this variable as well. At this point, **newImpl** and **oldImpl** contain the same information.

The next step is to change the **convertProc**, **genericProc**, **SmallPictureProc**, and **pictureProc** in **newImpl**, and then store the new implementation. Notice that we discard the results of the call to **SetImplementation**; since we have already retrieved the value of the old implementation.

newImpl is a local variable. It doesn't need to be global, because you can always get this information with a call to **GetImplementation**. **oldImpl** is a global variable because you will need to access it from another procedure. (There is an example in the next section that shows why you need **oldImpl**.) **oldImpl** is a pointer to an **Implementation** rather than an **Implementation** to minimize storage in the global frame.

15.2.1 PictureProcs

The `pictureProc` field of an `Implementation` is a call back procedure that displays an icon on the screen. There are two possible ways to define an icon picture: you can include a `pictureProc` in your `Implementation` or you can use *icon files*. An icon file associates an icon bitmap with a file type; when the application is loaded, the icon file in the application folder is opened to read the bitmap and display the icons. Chapter 16, *Application Folders*, discusses how to use icon files.

A `pictureProc` is of type `Containeer.PictureProc`:

```
Containeer.PictureProc: TYPE = PROCEDURE [
  data: Containeer.DataHandle,
  window: Window.Handle,
  box: Window.BOX,
  old, new: Containeer.PictureState];
```

```
Containeer.DataHandle: TYPE = LONG POINTER TO Containeer.Data;
```

```
Containeer.Data: TYPE = RECORD [
  reference: NSFile.Reference ← NSFile.nullReference];
```

```
Containeer.PictureState: TYPE = {garbage, normal, highlighted,
  ghost, reference, referenceHighlighted};
```

`old` and `new` describe the state of the icon. An icon generally has slightly different pictures for different states, such as when the icon is selected (**highlighted**) or open (**ghost**.) Figure 15.1 illustrates different states of the document icon.

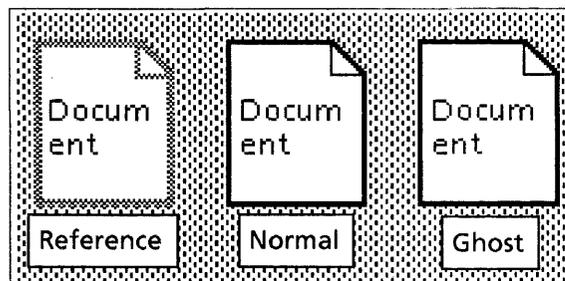


Figure 15.1 Icons

`window` and `box` describe the area of the screen where the icon is to appear; you should pass these parameters to the `Display` routine that you call to actually display the icon.

`data` allows you to distinguish among many files of the same file type. You can create slightly different icons for various files associated with an application. For example, all documents associated with the document editor have the same file type (and hence the same icon shape), but generally different names and contents. Thus, the document editor includes the name of the file on each icon.

To get the name of the file from the `NSFile.Reference`, you don't need to use `NSFile` routines. Instead, you should use `Containeer.GetCachedName`:

```

Containeer.GetCachedName: PROCEDURE [
  data: Containeer.DataHandle]
  RETURNS [name: XString.ReaderBody, ticket: Containeer.Ticket];

```

```

Containeer.Ticket: TYPE[2];

```

data is the data parameter to the PictureProc. The return parameters are the file name and a Ticket. A Ticket ensures that no other process can change the name of the file while you are looking at it. When you are through, you need to return the ticket:

```

Containeer.ReturnTicket: PROCEDURE [ticket: Containeer.Ticket];

```

Here is a PictureProc that puts the name of the file on the icon:

--This procedure describes the actual bitmap.

```

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

```

--This procedure paints the name of the file on the icon

```

PaintIconName: PROC [window: window.Handle,
  box, textBox: Window.Box, name: XString.Reader] = {
  [] ← SimpleTextDisplay.StringIntoWindow [
    string: name,
    window: window,
    place: textBox.place,
    lineWidth: textBox.dims.w ] };

```

```

PictureProc: Containeer.PictureProc = {

```

--textBox describes where the title is to appear. textBox is

--relative to box, which is in turn relative to window.

```

  textBox: Window.Box ← [[x:7, y:10],[w:55, h:36]];
  name: XString.ReaderBody;
  ticket: Containeer.Ticket;
  IF new = garbage THEN RETURN;
  box.dims ← [64,64]; --size of the icon

```

--Retrieve the name of the file whose reference is data.

```

  [name, ticket] ← Containeer.GetCachedName[data];

```

<<Display the icon. Call InitBigPicture to display the icon, and PaintIconName to add the name of the file. >>

```

SELECT old FROM
  garbage,ghost = > {
    Display.Bitmap[
      window:window,
      box: box,
      bitmapBitWidth: 64,
      address: [myIconPicture, 0, 0];
      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;

```

--return the ticket from the call to GetCachedName
 Containeer.ReturnTicket[ticket] };

InitBigPicture is the procedure that contains the actual bits for the icon picture. See Appendix B, Icon Editor, to find out how to generate the bits for your desired picture.

Inside the **PictureProc**, the first step is to call **GetCachedName** to retrieve the name of the file. The next step is to display the icon picture with a call to **Display.Bitmap** and to display the name of the icon with a call to **PaintIconName**. If the icon state is **highlighted**, invert the icon picture; if the icon state is **ghost**, paint a white version of it. After it paints the icon, it returns the **Ticket** that it got in the call to **GetCachedName**, and then returns.

15.2.2 SmallPictureProcs

A **SmallPictureProc** is similar to a **PictureProc**, except that it defines a small (13 X 13 pixels) version of the icon. For example, this tiny icon will be used when the icon is inside a folder. When you open a document inside a folder, the title for the folder will display the tiny icon for both the folder and the document, as illustrated in Figure 15.2.



15.2: Document header with tiny icons

A **smallPictureProc** is of type **Containeer.SmallPictureProc**:

```
Containeer.SmallPictureProc: TYPE = PROCEDURE [
  data: Containeer.DataHandle ← NIL,
  type: NSFile.Type ← Containeer.ignoreType,
  normalOrReference: Containeer.PictureState]
  RETURNS [smallPicture: XString.Character];
```

To create a tiny icon, you create the bitmap and then create a new character out of that bitmap. You create a new character by calling **SimpleTextFont.AddClientDefinedCharacter**:

```
SimpleTextFont.AddClientDefinedCharacter: PROCEDURE [
  width, height: CARDINAL,
  bitsPerLine: CARDINAL,
  bits: LONG POINTER,
  offsetIntoBits: CARDINAL ← 0,
  RETURNS [XString.Character];
```

This procedure takes a bitmap and creates a character out of it. Here is a code fragment that illustrates how to do a tiny icon:

```

smallPic: XString.Character ← XString.nullChar;
SmallPicture: Containee.SmallPictureProc = {RETURN [smallPic]};

--Call this procedure directly from mainline code, or from
--SetImplementation (which is called from mainline.)
InitSmallPicture: PROCEDURE = {
  bits: ARRAY [0..13] OF Word ← [177777B, . . .];
  IF smallPic = XString.nullChar THEN
    smallPic ← SimpleTextFont.AddClientDefinedCharacter[
      width:13,
      height: 13,
      bitsPerLine 16,
      bits: @bits ]};

```

You can also initialize the bitmap directly in your `SmallPictureProc`, if you like.

15.2.3 Generic procs

When the user selects an icon, the desktop implementation determines the icon's file type, calls `GetImplementation` to find the `Implementation` associated with that file type, and then calls the `genericProc` in the `Implementation`. The `genericProc` thus implements various icon operations: for example, it determines what happens when the user `MOVE` or `COPYS` something to your application.

A `genericProc` is of type `Containee.GenericProc`:

```

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

```

```

Containee.ChangeProc: TYPE = PROCEDURE [
  changeProcData: LONG POINTER ← NIL,
  data: Containee.DataHandle,
  changedAttributes: NSFile.Selections ← [],
  noChanges: BOOLEAN ← FALSE];

```

`atom` specifies which operation to perform. Notice that the return result is unspecified; different operations return different results. Here is a list of the most common atoms and the results that you should return for each:

CanYouTakeSelection

The user has selected something, pressed `COPY` or `MOVE`, and then selected your file. You should determine whether you can accept the selection (by calling `Selection.CanYouConvert`, `selection.HowHard` or the like) and return a `LONG POINTER TO BOOLEAN` indicating whether the conversion is possible.

Open

The user has selected your icon and invoked `OPEN`. You should create a `StarWindowShell` and return a handle to it.

Props

The user has selected your icon and invoked PROPS. You should create a property sheet and return a handle to it.

TakeSelection

The user has selected something, pressed MOVE, and then selected your icon. You should implement the operation (how you do this depends on the application) and then return a LONG POINTER TO BOOLEAN indicating whether the operation was successful.

TakeSelectionCopy

This atom is just like TakeSelection, except that the user pressed COPY instead of MOVE.

Here is a simple example of a GenericProc:

```
GenericProc: Containee.GenericProc = {
  SELECT atom FROM
    open = > RETURN[
      Defs.MakeShell[data, changeProc, changeProcData]];
    props = > RETURN[
      Defs.MakePropertySheet[
        data, changeProc, changeProcData]];
  ENDCASE = > RETURN[oldImpl.genericProc[atom, data,
    changeProc, changeProcData] ];
```

Note the ENDCASE, which calls the old implementation for any atoms that the SELECT statement doesn't handle. (This is why you need to store the old Implementation when you call SetImplementation.) In most cases, the old implementation will be the default implementation supplied by Containee, which displays an appropriate message to the user.

Notice also that we pass changeProc and changeProcData as parameters to MakeShell and MakePropertySheet. The next section discusses ChangeProcs in detail.

15.2.3.1 ChangeProc

Depending on which operation you are performing, executing the GenericProc can potentially change the value of some of the file's attributes. This is not a problem for the application itself: if it does something that alters an attribute, it is responsible for updating the attributes accordingly. There are times, however, when a change to an application's file affects other applications.

For example, suppose the user presses PROPS on a document in a folder, uses the property sheet to change the name of the document, and then selects Done. The document application can respond to the change in its implementation of the Done command, but the folder must also recognize the change so that it can update the name of the document inside the folder.

A ChangeProc is a procedure that a "container" (such as a folder) supplies to a "containee" (such as a document). The containee is expected to call the ChangeProc to allow the container to recognize changes. Note that the containee implementation does not provide the ChangeProc, and never even sees it; the containee just has to call the ChangeProc.

In this case, the icon application is the "containeer," and the desktop is the "container." Thus, the desktop implementation supplies a **ChangeProc** as a parameter to every **GenericProc**; the application must in turn call that **ChangeProc**.

Since the purpose of a **ChangeProc** is to allow the container to act on changes, you might think that you only need to call the **ChangeProc** if you change attributes. However, calling the **ChangeProc** also allows the container to deallocate the **changeProcData**. Thus, you must call the **ChangeProc** regardless of whether you have changed any attributes.

Therefore, you need to ensure that every arm of your **GenericProc** results in a call to the **ChangeProc**. If the operation can't affect any attributes, you can call the **ChangeProc** immediately, before you implement the operation. If the operation can potentially change attributes, however, you need to call the **ChangeProc** after the operation completes. This sometimes requires some careful planning to make sure that you have access to the **ChangeProc** when you need it.

For example, consider the following **GenericProc**:

```
GenericProc: Containeer.GenericProc = {
  SELECT atom FROM
    open = > RETURN[
      Defs.MakeShell[data, changeProc, changeProcData]];
    props = > RETURN[
      Defs.MakePropertySheet[
        data, changeProc, changeProcData]];
    canYouTakeSelection = > {
      changeProc[changeProcData: changeProcData,
        noChanges: TRUE];
      RETURN @[false]};
  ENDCASE = > RETURN[oldImpl.genericProc[atom, data,
    changeProc, changeProcData] ];
```

This **GenericProc** calls the **ChangeProc** directly from the **canYouTakeSelection** atom, since executing that arm cannot change any attributes. Note that the **@[false]** value returned is a pointer to a global variable **false**, which is a boolean declared to be **FALSE**. Using a local variable is not good enough, since the storage must exist after return from this procedure.

The **open** and **props** branches, however, can both potentially change attributes, so we pass **changeProc** and **changeProcData** to **MakeShell** and **MakePropertySheet**, which should call **changeProc** when the operation completes.

This is harder than it sounds, however. In the case of **MakePropertySheet**, you really need to call the **ChangeProc** when the user invokes Done or Cancel, not when you have just put the property sheet on the screen. Similarly, in the case of **MakeShell**, you need to call the **ChangeProc** after the shell is put away, not when it is first created.

For example, here is a program fragment that shows how **MakePropertySheet** should handle the **changeProc**:

--Globals, including record to contain change procedure

```
DataObject: TYPE = RECORD [
  fh: NSFile.Handle,
  changeProc: Containee.ChangeProc ← NIL,
  changeProcData: LONG POINTER ← NIL];
Data: TYPE = LONG POINTER TO DataObject;
zone: UNCOUNTED ZONE ← Heap.Create[initial:1];
```

<< This procedure is called from the GenericProc, with parameters changeProc, changeProcData, and data. It allocates a record for the changeProc, and calls PropertySheet.Create, passing the data record as clientData. >>

```
MakePropertySheet: PUBLIC PROC [
  data: Containee.DataHandle,
  changeProc: Containee.ChangeProc ← NIL,
  changeProcData: LONG POINTER ← NIL]
  RETURNS [pSheetShell: StarWindowShell.Handle] = {
--allocate data. Note that fh is an open file handle.
myData: Data ← zone.NEW [DataObject ← [
  fh: NSFile.OpenByReference[data.reference],
  changeProc: changeProc,
  changeProcData: changeProcData];
```

<< Call Create. Note the clientData parameter, which is the ChangeProc. >>

```
pSheetShell ← PropertySheet.Create [
  formWindowItems: MakeItems,
  menuItemProc: MenuItemProc,
  menuItems [done: TRUE, cancel: TRUE],
  size: size,
  title: @title,
  placeToDisplay: placeToDisplay,
  formWindowItemsLayout: FormWindow.defaultLayoutProc,
  display: FALSE,
  clientData: myData] };
--See chapter 7, Form Windows
MakeItems: FormWindow.MakeItemsProc = {...};
```

<< Called when user invokes Done or Cancel. clientData is the myData record allocated in MakePropertySheet. To use this data, you need to declare a local variable and then assign clientData to it. >>

```
MenuItemProc: PropertySheet.MenuItemProc = {
  myData: Data ← clientData;
  SELECT menuItem FROM
  done = > {
    ok ← ApplyAnyChanges[formWindow, myData].ok;
    zone.FREE[@myData];
    NSFile.Close[myData.fh];
    RETURN [ok];
```

--call the change procedure even though there are no
--changes

```
cancel = > {
  data: Containee.Data ← NSFile.GetReference[myData.fh];
  IF myData.changeProc # NIL THEN myData.changeProc [
    changeProcData: myData.changeProcData,
    data: @data,
    changedAttributes: [],
    noChanges: TRUE];
  NSFile.Close[myData.fh];
  zone.FREE[@myData];
  RETURN [ok: TRUE];
  ENDCASE = > RETURN [ok: FALSE] };
```

```

--Called when the user invokes Done. Updates actual attributes
ApplyAnyChanges: PROC [fw: Window.Handle, myData: Data]
  RETURNS [ok: BOOLEAN] = {
    attrList: ARRAY [0..1] OF NSFile.Attribute;
    changedAttributes: NSFile.Selections ← [];
    ctChangedAttrs: CARDINAL ← 0;

    --if nothing's been changed, call ChangeProc and return
    IF NOT FormWindow.HasAnyBeenChanged[fw] THEN {
      IF myData.changeProc # NIL THEN
        myData.changeProc
          changeProcData: myData.changeProcData,
          noChanges: TRUE];
      RETURN [ok: TRUE] };

    <<Something has been changed. Loop through to find
    out what has changed, update the attributes, and then call
    the changeProc...>>
    FOR myItem: Items IN Items DO
      itemKey: FormWindow.ItemKey = myItem.ORD;
      --if this item hasn't changed, then loop
      IF NOT FormWindow.HasBeenChanged [fw, itemKey]
        THEN LOOP;
      --the item has changed, so update appropriate attributes
      SELECT myItem FROM
        name = > {
          rb: XString.ReaderBody ←
            FormWindow.LookAtTextItemValue [fw, itemKey];
          ns: NSString.String ← XString.NSStringFromReader [
            @rb, localZone];
          FormWindow.DoneLookingAtTextItemValue [
            fw, itemKey];
          attrList[ctChangedAttrs] ← [name[ns]];
          changedAttributes.interpreted[name] ← TRUE; };
        ENDCASE;
      ctChangedAttrs ← ctChangedAttrs + 1;
    ENDOOP;

    --if any attributes have changed, then store new attributes
    --and then call change proc
    IF ctChangedAttrs > 0 THEN {
      data: Containee.Data ← [ NSFile.GetReference [myData.fh] ];
      NSFile.ChangeAttributes [
        myData.fh, DESCRIPTOR [@attrList, ctChangedAttrs]];
      NSFile.ClearAttributeList [
        DESCRIPTOR[@attrList, ctChangedAttrs]];
      --call change proc
      IF myData.changeProc # NIL THEN
        myData.changeProc[
          myData.changeProcData, @data, changedAttributes];
    }

    --no attributes have changed, but still have to call
    changeProc
    ELSE
      IF myData.changeProc # NIL THEN
        myData.changeProc[
          changeProcData:myData.changeProcData,
          noChanges: TRUE];
      RETURN [ok: TRUE];
    };
  };

```

The `PROPS` arm of the `genericProc` calls `MakePropertySheet`, passing in `changeProc` and `changeProcData`. `MakePropertySheet` creates the property sheet but does not implement the Done and Cancel commands, so you need to pass the `changeProc` on to the procedure that will be in control when the user finishes making changes and invokes Done.

To do this, allocate a record that contains the change procedure. The storage for this record should come from a heap, not from a local or global frame. (The local frame doesn't work because the storage isn't permanent enough; the global frame doesn't work because there may be more than one property sheet open for a given application.)

Once you have stored the `changeProc` in the record, you can pass a pointer to that record as the `clientData` parameter to `PropertySheet.Create`. You should also pass `MenuItemProc` as the procedure to be called when the user invokes Done or Cancel; `clientData` will be a parameter to this procedure.

Inside the `MenuItemProc`, store `clientData` into a variable of type `Data`. (Mesa's type checking prevents you from accessing it directly.) If the command was Cancel, call the `changeProc` and return. If the command was Done, call `ApplyAnyChanges`, which figures out if there were any changes and acts accordingly. Notice that we call the `changeProc` in all cases.

The `MakeShell` procedure will be somewhat similar. You need to pass the `changeProc` to `MakeShell`, but you should call it from your `TransitionProc`. (The potential changes to the file will occur after `MakeShell` has completed.) The standard way to handle this is to pass the `changeProc` to `MakeShell`, and then store it in the shell's context. You can then retrieve the context from the `TransitionProc` and call the `changeProc` from there.

15.3 The Prototype interface

When you write an icon application you don't place the icon directly on the desktop; that is the user's prerogative. Instead, you put your icon in the Prototype folder.

You create and manipulate prototype files using the ViewPoint **Prototype** interface. Its main procedures are **Find** and **Create**:

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

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

type, **subtype**, and **version** uniquely identify a given prototype file. **subtype** distinguishes objects of the same **type**; **version** helps determine if the prototype is current.

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`:

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

The first step is to call **Find** to see if the file already exists. If not, (**Find** returns `NSFile.nullReference`), then call **Create**, which creates the file, opens it, and returns a file handle. In this example, we just close the file immediately, since we don't need the open file handle for anything.

15.4 Summary

To write an icon application, you need to do the following :

- Write a **GenericProc** to implement `MOVE`, `COPY`, etc. Make sure that you call the **ChangeProc** (sooner or later) from each arm of the **GenericProc**.
- Initialize the atoms you recognize in the **GenericProc**.
- Write a **PictureProc** to display the icon, and optionally a **SmallPictureProc** for the tiny version of the icon.
- Call `Containeer.SetImplementation` to register the application with the desktop. Your **Implementation** should include at least a **GenericProc** and a **PictureProc**.
- Put a copy of the icon in the Prototype folder using either `Prototype.Find` or `Prototype.Create`.

Here is a program fragment that illustrates these steps:

```
--global data
smallPic: XString.Character ← XString.nullChar;
oldImpl: LONG POINTER TO Containeer.Implementation;

--called from the mainline code to put icon in prototype folder
FindOrCreateIconFile: PROCEDURE = {
    mh: XMessage.Handle = Defs.GetMessageHandle[];
    name: XString.ReaderBody ← XMessage.Get [
        mh, Defs.MessageKey.prototypeFileName.ORD];
    version: CARDINAL ← 1;

    IF Prototype.Find[type, version] = NSFile.nullReference
    THEN NSFile.Close[Prototype.Create[
        name:@name, type:type, version:version]] };
```

--implement OPEN and PROPS; all other atoms go to the ENDCASE.

```
GenericProc: Containee.GenericProc = {
  SELECT atom FROM
  open = > RETURN[Defs.MakeShell[
    data, changeProc, changeProcData]];
  props = > RETURN[
    Defs.MakePropertySheet[
    data, changeProc, changeProcData]];
  ENDCASE = > RETURN[oldImpl.genericProc[
    atom, data, changeProc, changeProcData] ]};
```

--called from mainline code to initialize atoms

```
InitAtoms: PROC = {
  open: Atom.ATOM ← Atom.MakeAtom["Open"L];
  props: Atom.ATOM ← Atom.MakeAtom["Props"L];
```

```
InitBigPicture: PROC = { iconPicture ← ...--set up bitmap};
```

```
InitSmallPicture: PROCEDURE = {
  bits: ARRAY [0..13] OF Word ← [177777B, ...];
  IF smallPic = XString.nullChar THEN
    smallPic ← SimpleTextFont.AddClientDefinedCharacter[
      width:13,
      height: 13,
      bitsPerLine 16,
      bits: @bits] };
```

--draw the icon

```
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] };
```

--register application with the desktop

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

```
SmallPicture: Containee.SmallPictureProc = {RETURN [smallPic]};
```

```
--mainline code
InitAtoms[];
FindOrCreateIconFile[];
InitBigPicture[];
InitSmallPicture[];
SetImplementation[];
```

15.5 Exercise

The exercise for this chapter is a Tic-Tac-Toe game, illustrated in Figure 15.3.

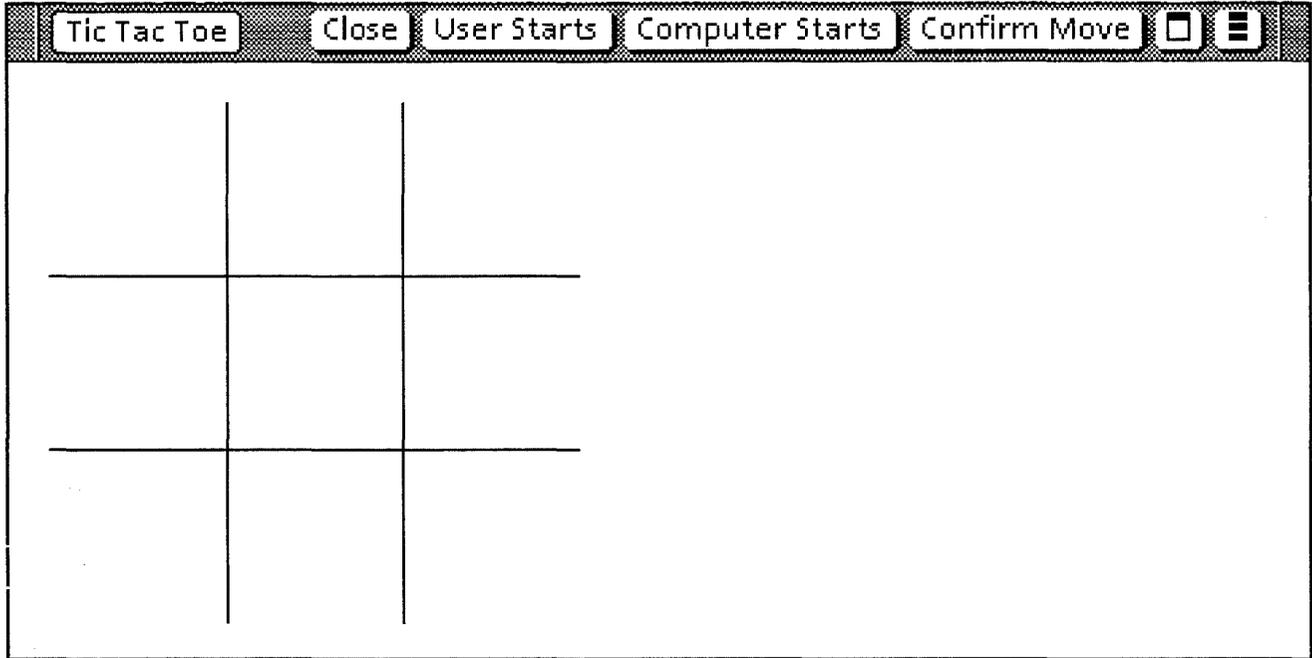


Figure 15.3 The Tic Tac Toe application

To start a game, select either **Computer Starts** or **User Starts**. If you select **Computer Starts**, the program will make a random move on the board and wait for you to respond. If you select **User Starts**, the program waits for you to make the first move.

To make a move, select a box and invoke **Confirm Move**. You cannot select a square that is already occupied. You can start a new game at any time with **User Starts** or **Computer Starts**.

The module `TicTacToeToolImplTemp.mesa` implements the tool code and the user commands. The application currently runs from the Attention menu; you should modify this module so that the tool will run from an icon on the desktop. This will involve writing the procedures `GenericProc`, `FindOrCreateIconFile`, `InitBigPicture`, `PaintIconName`, and `SetImplementation` and modifying the `MakeShell` procedure. You will also need the following modules:

- | | |
|---------------|------------------|
| TicTacToeDefs | TicTacToeMsgImpl |
| TicTacToeImpl | TicTacToe.config |

Once you have an application running to your satisfaction, it is time to package it as a finished application. Packaging consists of taking auxiliary data out of the code modules and putting it in separate data files. For example, if you have designed a new icon for your application (other than the ones in Standard.icons), you should put the bitmap for that icon in an icon file. If your application posts messages you should put the messages in a separate message file rather than in an implementation module. This approach has the advantage that it allows you to modify the data files without recompiling the application. This is particularly useful for applications that are potentially multinational.

Once you have moved the data into separate data files, you need to group the data files and the object files together into a single coherent object, so that the user doesn't have to concern himself with obtaining all the component files. ViewPoint provides the notion of an *application folder* for this purpose.

16.1 Building an application folder

An *application folder* consists of object files and associated data files. Note that the term application folder is a bit of a misnomer, since an application folder is not the same thing as a standard folder on the desktop (i.e., they have different file types.) An application folder must have at least one object file, and may have any or all of the following data files:

- A message file
- An icon file
- One or more TIPC files
- A keyboard file (if the application uses a keyboard that is not in the standard keyboard file)

An application folder can also include other private data files, such as translation tables or the like. For example, if your application translates ASCII to some other code, such as EBCDIC, you might wish to include a translation table for this purpose.

This chapter discusses only how to build the application folder once you have the data files; it does not discuss how to generate the data files. The appendices of this manual describe some tools that are available for creating message and icon files; Chapter 9, *TIP*, discusses .TIPC files.

16.1.1 Application description files

In addition to the actual files that constitute the application, an application folder must contain an *Application Description File (ADF)*. This file describes the components of the application; having a separate description file means that you don't need to hardcode the names of the data files into the application.

When you write an application, you should include the capability of having the data files either bound in an application folder or stored in the system folder. Typically, developers store their data files in the system folder during development (and use the *WorkstationProfile* as an ADF), and then create their own application folder as the final step. The examples later in this chapter show how to write code that allows either of these approaches.

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.

The loading 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; the loader starts applications in increasing priority number order. If your application has no dependencies, use a priority of zero (the default.)

The easiest way to create an ADF is to copy the *WorkstationProfile*, modify it, and then rename it. The syntax for an ADF follows that of an option file. (Other examples of option files are the *Workstation Profile* and the *User Profile*.)

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 TIP file
Priority: 0
```

Only the section name and *bcd* entry are mandatory. The section name (**SampleApplication**) is the *internal name* of the application; the code for the application uses this name to reference the application folder. The internal name never changes. The application also has an *external name*, which appears on the folder. The external name can be changed, making multilingual conversion easier and more complete. An external name for an ADF is typically something like **ApplicationName.adf**.

Figure 16.1 illustrates the complete syntax for an ADF:

```

<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>

```

Figure 16.1 ADF syntax

Note that the **<Entry Identifier>**s can be any identifier, 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**.

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

16.2 Modifying the code

Once you have created your data files and written an ADF for them, you must change your code to access the data files. Typically, you start with the internal name of the application, and then you need to perform the following steps.

1. Call **ApplicationFolder.FromName** to get an **NSFile.Reference** to the application folder. This procedure returns **NSFile.nullReference** if it can't find the specified folder.

```

ApplicationFolder.FromName: PROCEDURE [
    internalName: Xstring.Reader]
    RETURNS [applicationFolder: NSFile.Reference];

```

2. Open the folder with **NSFile.OpenByReference**.
3. Get a reference to the ADF within the specified folder with a call to **ApplicationFolder.FindDescriptionFile**:

```

ApplicationFolder.FindDescriptionFile: PROCEDURE [
    applicationFolder: NSFile.Handle]
    RETURNS [descriptionFile: NSFile.Reference];

```

4. Parse the ADF to find the name of the data file that you are interested in. To do this, call `OptionFile.GetStringValue`:

```
OptionFile.GetStringValue: PROCEDURE [  
    section, entry: XString.Reader,  
    callBack: PROCEDURE [value: XString.Reader],  
    index: CARDINAL ← 0,  
    file: NSFile.Reference ← NSFile.nullReference];
```

`GetStringValue` calls `callBack` with the value of a string entry. `section` is the internal name of the ADF that you want to read; `entry` is the specific entry that you are interested in. If the entry is there, `GetStringValue` will call back to `callBack`, passing in the value of that entry. `file` is a reference to the file that contains the specified ADF.

The following sections show examples of these steps for retrieving a message file, a TIP file, and an icon file from an ADF.

16.2.1 Message Files

Whenever an application uses messages, it must provide a way to access the message handle. Recall that when you use a message implementation, you call `XMessage.AllocateMessages` and `XMessage.RegisterMessages` to retrieve the messages. (See Chapter 3, *Strings and Messages*, if you want to review message bcds.) If your messages are in a file, however, you retrieve them using `XMessage.MessageFromFile` or `XMessage.MessagesFromReference`.

```
XMessage.MessagesFromFile: PROCEDURE [  
    fileName: LONG STRING,  
    clientData: XMessage.ClientData,  
    proc: XMessage.DestroyMsgsProc]  
    RETURNS [msgDomains: XMessage.MsgDomains];
```

```
XMessage.MessagesFromReference: PROCEDURE [  
    file: NSFile.Reference,  
    clientData: XMessage.ClientData,  
    proc: XMessage.DestroyMsgsProc]  
    RETURNS [msgDomains: XMessage.MsgDomains];
```

```
XMessage.MsgDomains: TYPE = LONG DESCRIPTOR FOR ARRAY OF  
XMessage.MsgDomain;
```

```
XMessage.MsgDomain: TYPE = RECORD [  
    applicationName: XString.ReaderBody,  
    handle: XMessage.Handle];
```

`MessagesFromFile` gets the messages from the file named `fileName` in the system folder, while `MessagesFromReference` gets the messages from the file whose reference is `file`. When you are through with the messages, you must call `FreeMsgDomainsStorage`:

```
XMessage.FreeMsgDomainsStorage: PROCEDURE [msgDomains:  
XMessage.MsgDomains];
```

Here is an example of how to write code that uses a message file. The code reads the ADF to find the name of the message file, and then uses **XMessage** routines to access the messages.

-- File: *SampleMsgFileImpl.mesa*
 -- Copyright (C) 1985 by Xerox Corporation. All rights reserved.

DIRECTORY

... = {

-- Data

h: XMessage.Handle ← NIL;

localZone: UNCOUNTED_ZONE ← Heap.systemZone;

-- Procedures

DeleteMessages: PROC [clientData: XMessage.ClientData] =
 {};

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

<< *This procedure is called from mainline code. Its job is to call MessagesFromReference to initialize the value of h so that other procedures can call GetMessageHandle and be able to access the messages. The hard part is getting the file parameter to MessagesFromReference. To do this, first call FromName to get a reference to the folder for the application, and then call GetMessageFileRef to find the message file within that folder.*

>>

```
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];
};
```

<< This procedure is called from *InitMessages*. It declares the internal procedure *FindMessageFileFromName*, and then initializes *folderHandle* and *adf*. If there is no ADF for the application, use the system folder and the *WorkStationProfile*; otherwise, call *FindDescriptionFile* to find the appropriate ADF. The final step is to call *GetStringValue* to get the name of the message file. *GetStringValue* results in a call back to *FindMessageFile*, which finally gets a handle to the message file.>>

```

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: NSSString.String ← XString.NSSStringFromReader
      [r: value, z: localZone];
    msgFileHandle: NSFile.Handle ← NSFile.nullHandle;
    -- do NSFile.Find 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}};
    -- No message file
    IF msgFileHandle = NSFile.nullHandle THEN ERROR;
    msgFile ← NSFile.GetReference [msgFileHandle];
    NSFile.Close [msgFileHandle];
    NSSString.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[];

}.

```

The mainline code calls `InitMessages`, which calls `XMessage.MessagesFromReference` to obtain the messages for the application. To call `MessagesFromReference`, however, it needs a reference to the file containing the messages. To get this reference, it calls `ApplicationFolder.FromName` to get a reference to the folder and then calls `GetMessageFileRef`.

`GetMessageFileRef` checks to see whether folder is `nullReference`. If it is, it sets `folder` to the system catalog, and `adf` to the workstation profile. If there is an ADF for the application, however, it calls `ApplicationFolder.FindDescriptionFile` to get a reference to the ADF (within the specified folder.)

Once `folderHandle` and `adf` are initialized, the next step is to call `OptionFile.GetStringValue` to parse the ADF. If there is a `messages` entry in the option file, this procedure results in a call back to `FindMessageFileFromName`. This procedure just gets a handle to the file that contains the messages; this handle (`msgFile`) is returned up to `InitMessages` to be the file parameter to `MessagesFromReference`.

Once you have edited the message implementation to use message files, you still need to use the *Message Tools* to actually create the messages file. See Appendix C, *Message Tools*, for information on how to use these tools to create a message file.

16.2.2 Private TIP file

This example shows how to write code that puts a TIP file in the ADF instead of putting it directly in the code. Note that the actual data file that you put in the ADF should be a `.TIPC` file, but that you should name it `.TIP` in the ADF entry.

```
sampleTIPTable: TIP.Table ← NIL;
```

<< This procedure is called from the mainline code. It calls `AppendTIPFileName` to get the complete path name for the TIP file, and then calls `CreateTable` to generate the runtime TIP table for the program's use. >>

```
InitTIPTable: PROCEDURE = {
--separator is a / character, used for separating directories
  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];
};
```

<< The job of this procedure is to get a full path name for the TIP file and store it in the writer parameter. This involves parsing the ADF. >>

```

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

    --call back procedure that is called if there is a TIP entry in
    --the ADF. Adds the name of the ADF to the path name in
    --writer.
    AppendName: PROCEDURE [value: XString.Reader] = {
        XString.AppendReader [to: writer, from: value];
    };

--If there is no application folder, then use a default,
--hard-coded TIP file name.
    IF folderRef = NSFile.nullReference THEN {
        XString.AppendSTRING [
            writer, "SampleBWSApplication.TIP"L];
        RETURN};

    -- ELSE (there is an ADF, so parse it to get the
    -- name of the TIP file.
    folderHandle ← NSFile.OpenByReference [folderRef];
    --put the name of the folder in the writer
    AppendFolderName [folderHandle, writer];
    --add a directory separator
    XString.AppendChar [to: writer, c: separator];
    --parse the ADF to get the name of the TIP entry. If the entry
    --is there, this results in a call to AppendName, which adds
    --the name of the file to create a full path name.
    OptionFile.GetStringValue [
        section: @internalName,
        entry: @tipFile,
        callBack: AppendName,
        file: ApplicationFolder.FindDescriptionFile [folderHandle]];
    NSFile.Close [folderHandle];
};

--stick the name of the folder in writer.
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];
};

```

This example is somewhat similar to the message file example. The basic goal is to obtain the name of the TIP file from the application folder. To do this, we call `AppendTIPFileName`, which gets the name of the folder by calling `GetAttributes`,

and then gets the name of the file from the ADF. These two names are concatenated into the `pathName` variable, which is then used to create the runtime TIP table.

If there is no application folder, the default hard-coded value of `SampleBWSApplication.TIP` will be used. You should only use hard-coded information during development.

16.2.3 Private icons file

If you want your application to use an icons file, you need to modify your `SetImplementation` procedure so that it doesn't use a `PictureProc`. When you use application folders, you don't have to worry about finding the icon file; if you register the file type that you are interested in, ViewPoint will locate the icon file when the user loads the application and associate the application and its icon by type.

Here is an example of a `SetImplementation` procedure that uses an icon file instead of a `PictureProc`. Notice that there is no mention of the `PictureProc` or `SmallPictureProc`.

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

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

16.3 Create the application folder

Once you have created all your data files, and modified your code so that it uses the data files, you still need to actually create the application folder. To do this, you need to:

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

Folder → Application

Application → Folder

The first item takes a regular 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 16.2 illustrates the steps of building an application folder.

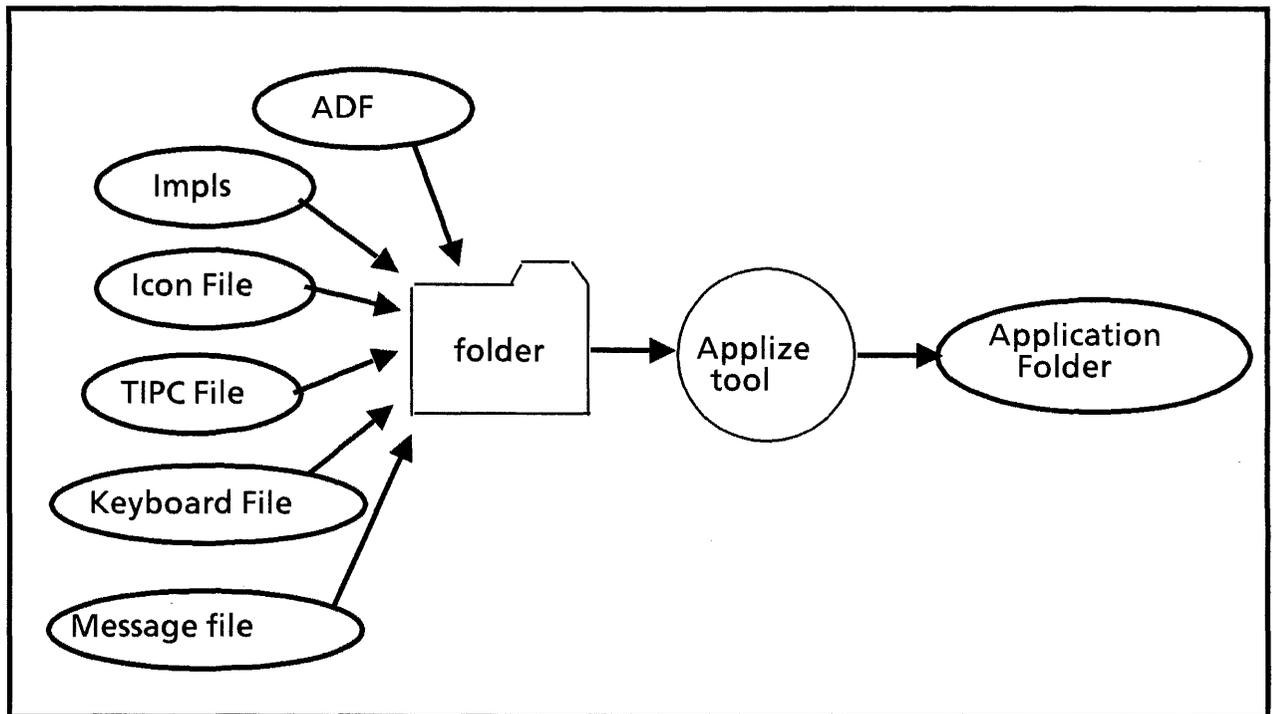


Figure 16.2 Building an application folder

16.4 Summary

To create an application folder, you need to do the following:

- Build the data files for the application
- Write an application description file
- Change code so that it accesses the data files
- Integrate the components into an application folder

The hardest part is writing the code that accesses the data files. Starting with the internal name of the folder, you need to do the following:

- Get a reference to the folder with specified internal name. (`ApplicationFolder.FromName`)
- Open the folder
- Get a reference to the ADF within that folder (`ApplicationFolder.FindDescriptionFile`)
- Parse the ADF to get the name of your desired data file. (`OptionFile.GetStringValue`)

16.5 Exercise

The assignment for this chapter is to turn the Black Book application into an application folder. Black Book was the exercise for Chapter 13, NSSegment; if you have not done this exercise, you will need to go back and read the description in Chapter 13. Also, if you do not have a working version of this application, then you will need to retrieve our version from the solutions for Chapter 13.

You need to create a messages file, an icon file, and an ADF, modify the code to reference the new data files, and then run the Applize tool to create an application folder.

Notes:

This chapter introduces the document interfaces, which are a group of interfaces that enable you to create or read the contents of ViewPoint documents. The principle interface in this group is `DocInterchangeDefs`, which supports creating and reading basic document structures such as text, fields, headings/ footings, frames, and formatting characters.

Although `DocInterchangeDefs` includes the facilities for adding frames to documents, it does not include the facilities for creating the contents of frames; there are separate interfaces for dealing with graphics, tables, and the like. We discuss these frame content interfaces in the next chapter.

17.1 Creating documents

To create a new ViewPoint document, you call `DocInterchangeDefs.StartCreation`. This creates a document whose only contents are a single page format character and a single new paragraph character.

```
DocInterchangeDefs.StartCreation: PROC [
  paginateOption: DocInterchangeDefs.PaginateOption ←
    compress,
  wantHeadingHandles, wantFootingHandles: BOOL ← FALSE,
  initialFontProps: FontPropsDefs.ReadOnlyProps ← NIL,
  initialParaProps: ParaPropsDefs.ReadOnlyProps ← NIL,
  initialPageProps: DocPagePropsDefs.ReadOnlyProps ← NIL]
  RETURNS [
    doc: DocInterchangeDefs.Doc,
    docIzn: InstanceDefs.Izn,
    leftHeading, rightHeading: DocInterchangeDefs.Heading,
    leftFooting, rightFooting: DocInterchangeDefs.Footing];

DocInterchangeDefs.PaginateOption: TYPE = {
  none, simple, compress};

DocInterchangeDefs.DOC: TYPE = LONG POINTER TO
  DocInterchangeDefs.DocObject;

DocInterchangeDefs.DocObject: TYPE;

InstanceDefs.Izn: TYPE = RECORD[UNSPECIFIED];
```

`paginateOption` specifies the type of pagination that will occur when you finish the document. `compress` pagination is full pagination. `simple` pagination provides the same external appearance as `compress` pagination, but leaves the internal structure of the document less compact. `none` leaves the document without any pagination at all. If your document will be longer than a few pages, you should use some form of pagination, or performance will be very slow.

wantHeadingHandles and **wantFootingHandles** specify whether the document will have headings and footings. If you specify true for either of these parameters, **StartCreation** will return handles to the headings and footings. Like the document itself, the headings and footings will be initially blank; the next section discusses how to add content to the document and its headings.

initialFontProps, **initialParaPros**, and **initialPageProps** indicate the initial properties for the document. If you do not specify any properties, **StartCreation** will use the document default properties. See Section 17.2 for more information on properties.

StartCreation returns a **Doc** handle, a **doclzn**, and handles for headings and footings.

The **doclzn** is a storage space that holds various "instances" (objects) within the document. You can just elide this value. The heading and footing handles will be NIL unless you specified TRUE for the corresponding **want*Handle** parameter. If you have a valid heading or footing handle, you must later free it; see section 17.1.1.4 for details.

The **Doc** handle represents the new document, which does not yet have any contents. Thus, the next step is to pass this handle to the **Append*** procedures described below, which allow you to add various kinds of information to the document.

17.1.1 Adding information to a document

Once you have a document handle, the next step is to add information to the document with various **Append*** procedures: **AppendAnchoredFrame**, **AppendChar**, **AppendColumnBreak**, **AppendField**, **AppendNewParagraph**, **AppendPageBreak**, **AppendPFC** (Page Format Character), or **AppendText**. (All of these procedures are in the **DocInterchangeDefs** interface.)

Each of these procedures appends the specified text or formatting character to the existing text in the document. You thus add all desired information to the document sequentially. Some of the objects within a document, such as page format characters and fields, can themselves have contents. Thus, the process of adding information to a document can be recursive. Figure 17.1 illustrates the kinds of information that you can add to a document.

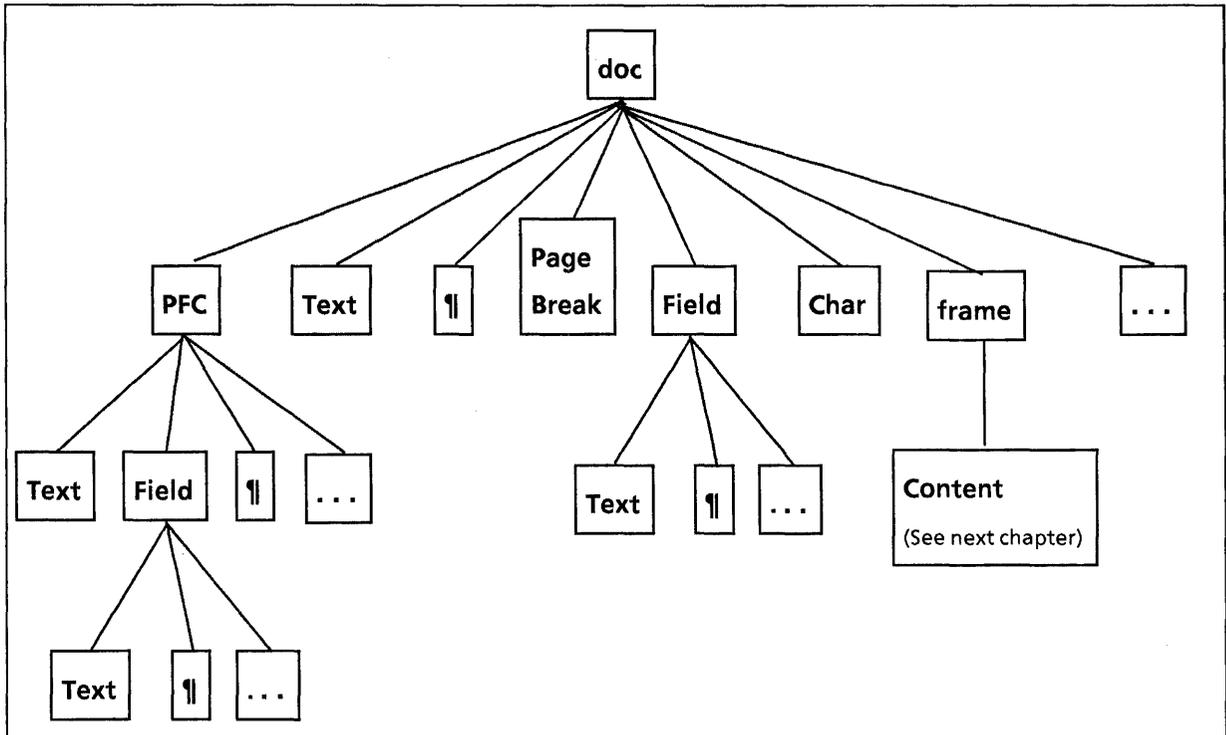


Figure 17.1: Appending to a document

17.1.1.1 Adding text

The routines that append textual information—**AppendChar**, **AppendField**, **AppendNewParagraph**, and **AppendText**—take a **DocInterchangeDefs.TextContainer** and a piece of data as parameters, and append the data to the text container.

A **TextContainer** is any object that can contain text: a caption, document, field, heading, or footing:

```

DocInterchangeDefs.TextContainer: TYPE = RECORD [
  var: SELECT type: * FROM
    caption = > [h: DocInterchangeDefs.Caption],
    doc = > [h: DocInterchangeDefs.Doc],
    field = > [h: DocInterchangeDefs.Field],
    heading = > [h: DocInterchangeDefs.Heading],
    footing = > [h: DocInterchangeDefs.Footing],
  ENDCASE];
  
```

The individual types are all opaque: for example, here are the declarations of **caption** and **doc**:

```

DocInterchangeDefs.Caption: TYPE =
  LONG POINTER TO DocInterchangeDefs.CaptionObject;

DocInterchangeDefs.CaptionObject: TYPE;

DocInterchangeDefs.Doc: TYPE =
  LONG POINTER TO DocInterchangeDefs.DocObject;

DocInterchangeDefs.DocObject: TYPE;
  
```

Thus, you add information to a header within a document the same way that you add information to the document itself: the **TextContainer** that you pass to **AppendChar** can be a heading,

a document, or any of the other variants. Note that you get the different types of text containers from different routines: `doc`, `heading`, and `footing` come from `StartCreation`; `caption` and `field` from `AppendAnchoredFrame` and `AppendField`, respectively. See below for more information on these two procedures. Figure 17.2 illustrates adding information to a header within a document.

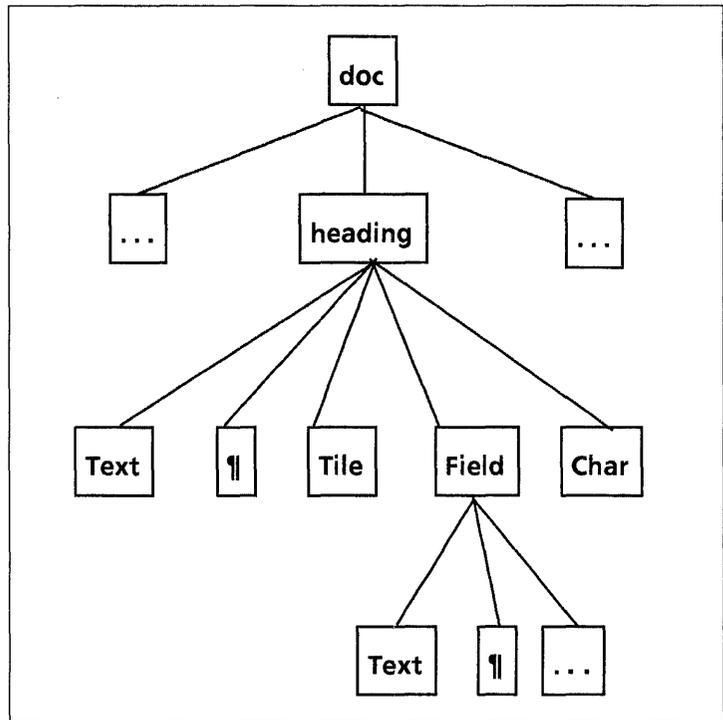


Figure 17.2: Adding text to a header

Note that all `TextContainers` always contain at least one `newParagraph` character; you don't have to provide the initial paragraph character. Also note that most of the `Append` routines allow you to specify new properties for the information you want to append. If you default this information, the new information will inherit the properties of the preceding paragraph or character, as appropriate. See Section 17.2 for a more complete discussion of the various properties.

As an example of one of these procedures, here is the declaration of `AppendChar`:

```

DocInterchangeDefs.AppendChar: PROC [
  to: DocInterchangeDefs.TextContainer,
  char: xChar.Character,
  fontProps: FontPropsDefs.ReadOnlyProps ← NIL,
  nToAppend: CARDINAL ← 1];

```

`AppendChar` appends one or more copies of the text character `char` to the specified `TextContainer`. `nToAppend` specifies the number of copies of the character that are to be appended; `fontProps` specifies the character properties. `AppendText` is similar, except it takes an `xstring.Reader` as a parameter instead of an `xChar.Character`.

AppendField is slightly different, because it returns a text container that you can use in other calls to **Append*** routines:

```
DocInterchangeDefs.AppendField: PROC [
  to: DocInterchangeDefs.TextContainer,
  fieldProps: FieldPropsDefs.ReadOnlyProps,
  fontProps: FontPropsDefs.ReadOnlyProps ← NIL]
  RETURNS [field: DocInterchangeDefs.Field];
```

AppendNewParagraph is straightforward:

```
DocInterchangeDefs.AppendNewParagraph: PROC [
  to: DocInterchangeDefs.TextContainer,
  paraProps: ParaPropsDefs.ReadOnlyProps ← NIL];
```

The only part of this that is slightly tricky is the syntax for specifying the variant. For example, here is a code fragment to create a document and add some text to it:

```
doc : DocInterchangeDefs.DOC ← DocInterchangeDefs.StartCreation [];
DocInterchangeDefs.AppendText[
  to: [doc[h: Doc]],
  text: @text,
  textEndContext: xstring.unknownContext];
```

AppendText appends an **xString.Reader** to the document, assuming that the variable **text** contains some string. This is a very incomplete fragment, but it does illustrate how to convert a document handle to a **doc** variant of a **TextContainer**.

17.1.1.2 Adding formatting information

The remaining **Append** procedures—**AppendAnchoredFrame**, **AppendColumnBreak**, **AppendPageFormatCharacter**, and **AppendPageBreak**—take only a **Doc**, and not a general purpose **TextContainer**. These procedures append characters that can appear in a document, but not in other **TextContainers** such as headings and footings.

AppendColumnBreak, **AppendPFC**, and **AppendPageBreak** each take a document and some properties as parameters, and append the specified character with the specified properties to the document.

Like **AppendField**, **AppendPFC** returns a text container: this allows you to call **Append*** routines recursively to add text and formatting information to PFC headers if you like. It will return **NIL** headings and footings unless you specify **TRUE** for one of the **want*Handle** parameters.

```
DocInterchangeDefs.AppendPFC: PROC [
  to: DocInterchangeDefs.DOC,
  pageProps: DocPagePropsDefs.ReadOnlyProps,
  wantHeadingHandles, wantFootingHandles: BOOL ← FALSE,
  fontProps: FontPropsDefs.ReadOnlyProps ← NIL]
  RETURNS [
  leftHeading, rightHeading: DocInterchangeDefs.Heading,
  leftFooting, rightFooting: DocInterchangeDefs.Footing];
```

AppendPFC appends a page format character to the main document text.

17.1.1.3 Adding frames

AppendAnchoredFrame appends an anchored frame such as a graphics or bitmap frame to a document.

```
DocInterchangeDefs.AppendAnchoredFrame: PROC [  
  to: DocInterchangeDefs.Doc,  
  type: DocInterchangeDefs.AnchoredFrameType,  
  anchoredFrameProps: DocFramePropsDefs.ReadOnlyProps,  
  content: InstanceDefs.Instance ← InstanceDefs.InstanceNil,  
  wantTopCaptionHandle,  
  wantBottomCaptionHandle,  
  wantLeftCaptionHandle,  
  wantRightCaptionHandle: BOOL ← FALSE,  
  anchorFontProps: FontPropsDefs.ReadOnlyProps ← NIL]  
RETURNS [  
  anchoredFrame: InstanceDefs.Instance,  
  topCaption, bottomCaption,  
  leftCaption, rightCaption: DocInterchangeDefs.Caption];
```

```
DocInterchangeDefs.AnchoredFrameType: TYPE =  
  MACHINE DEPENDENT {  
    nil(0), bitmap,  
    cuspButton, equation, graphics, IMG, table, text,  
    firstAvailable, lastAvailable(255)};
```

AppendAnchoredFrame appends the anchored frame **type** with properties **anchoredFrameProps** to the document to. **content** is a pointer to the contents of the frame. **DocInterchangeDefs** does not provide the facilities for creating the contents of frames; instead, you will have to use specialized interfaces such as **GraphicsInterchangeDefs** or **TableInterchangeDefs** to create the contents of the frame, and then call **AppendAnchoredFrame** to add that frame and its contents to the document. See the next chapter for details; for now, just assume that you have a pointer to the contents of the frame.

The **want*CaptionHandle** parameters indicate whether you want the frame to have captions. If you indicate **TRUE** for any of these parameters, the procedure will return a valid caption handle, which you can then use as a text container in other calls to **Append** routines. If you specify **TRUE**, and receive a valid caption handle, you must later free the storage for that handle, as described in the next section.

17.1.1.4 Storage management

With all of the **Append** procedures, you must manage the storage for the property records or other data structures that you pass in, except for handles obtained from the interface itself. The storage for the properties must remain valid during the call to **Append***; after **Append*** returns, you can free it.

Also, you are responsible for freeing any non-NIL handles obtained from any **Append** routines, or from **StartCreation**, with a call to an appropriate **Release*** routine. This applies to caption handles, field handles, and heading/footings. Here are the declarations of the relevant **Release** routines:

```
DocInterchangeDefs.ReleaseCaption: PROC [
    captionPtr: LONG POINTER TO DocInterchangeDefs.Caption];
```

```
DocInterchangeDefs.ReleaseField: PROC [
    fieldPtr: LONG POINTER TO DocInterchangeDefs.Field];
```

```
DocInterchangeDefs.ReleaseHeading: PROC [
    headingPtr: LONG POINTER TO DocInterchangeDefs.Heading];
```

```
DocInterchangeDefs.ReleaseFooting: PROC [
    footingPtr: LONG POINTER TO DocInterchangeDefs.Footing];
```

After you call **Release***, the handle will be invalid. To help prevent use of an invalid handle, the **Release*** routines take a pointer to the handle, and set the handle itself to NIL. (This is similar to Mesa's **FREE** operation.)

17.1.2 Finalizing document

When you have added all the necessary information to a document, you must call **DocInterchangeDefs.FinishCreation** to finalize the document and release the **Doc** handle. **FinishCreation** returns an **NSFile.Handle** to the newly-created document, and a status. The document that **FinishCreation** provides will be in paginated form if you so specified in **StartCreation**.

```
DocInterchangeDefs.FinishCreation: PROC [
    docPtr: LONG POINTER TO DocInterchangeDefs.Doc]
    RETURNS [
        docFile: NSFile.Handle,
        status: DocInterchangeDefs.FinishCreationStatus];
```

```
DocInterchangeDefs.FinishCreationStatus: TYPE = {ok,
    okButNotEnoughDiskSpaceToPaginate,
    okBuNotEnoughVMToPaginate,
    okButUnknownPaginateProblem, unknownProblem};
```

This document file is temporary, and will be deleted when you reboot. To make the file permanent, you must move it to the current user desktop with **NSFile.Move**, followed by a call to **StarDesktop.AddReference** to put the icon on the display. To do this, you must first get a reference to the file and to the current desktop. Here is a fragment to illustrate this; there is a complete example at the end of this section:

```
doc File: NSFile.Handle ← DocInterchangeDefs.FinishCreation[...];
--get reference to file
refDoc: NSFile.Reference ← NSFile.GetReference[docFile];
--get reference to desktop
refDT: NSFile.Reference ← StarDesktop.GetCurrentDesktopFile[];
--open file
fileDT: NSFile.Handle ← NSFile.OpenByReference[refDT];
--move file to desktop
NSFile.Move[docFile, fileDT];
NSFile.Close[fileDT];
NSFile.Close[docFile];
StarDesktop.AddReferenceToDesktop[refDoc];
```

If you want the opportunity to abort creation under certain circumstances, you can use `FinishCreationWithAbortProc` instead of `FinishCreation`:

```
DocInterchangeDefs.FinishCreationWithCheckAbortProc: PROC [  
    docPtr: LONG POINTER TO DocInterchangeDefs.DOC,  
    checkAbortProc: DocumentDefs.CheckAbortProc,  
    clientData: LONG POINTER ← NIL]  
    RETURNS [  
        docFile: NSFile.Handle,  
        status: DocInterchangeDefs.FinishCreationStatus,  
        aborted: BOOLEAN];
```

```
DocumentDefs.CheckAbortProc: PROC [clientData: LONG POINTER]  
    RETURNS [abort: BOOLEAN];
```

`FinishCreationWithCheckAbortProc` provides the ability to abort the document creation. `DocInterchangeDefs` will call you `checkAbortProc` just before it creates the document; if it returns `TRUE`, the process will be aborted. At that point, you should call `AbortCreation`.

```
DocInterchangeDefs.AbortCreation: PROC [docPtr: LONG POINTER TO  
DocInterchangeDefs.DOC];
```

`AbortCreation` aborts document creation and deallocates the storage associated with that document.

17.2 Properties

Each of the objects in a document has associated properties; there is a separate interface for each of these possible types of properties. `DocFramePropsDefs` describes the properties of an anchored frame within a document; `DocPageProps` describes the properties of a page format character, `ParaPropsDefs` describes paragraph properties, `FieldPropsDefs` describes a field, and `FontPropsDefs` describes font properties.

Each of these interfaces contains the following three types:

```
Props: TYPE = LONG POINTER TO PropsRecord;
```

```
ReadOnlyProps: TYPE = LONG POINTER TO READONLY PropsRecord;
```

```
PropsRecord: TYPE = RECORD [  
    ...];
```

The `PropsRecord` contains various fields for the particular properties. We include the declarations each of the `PropsRecords` here, but not all of the subsidiary types that they reference, since that would take up too much space. To see the full declarations, consult the appropriate chapter of the *ViewPoint Programmer's Manual*.

17.2.1 Anchored frame properties

```
DocFrameProps.PropsRecord: TYPE = RECORD [
  borderStyle: BorderStyle ← TRASH,
  borderThickness: CARDINAL ← TRASH,
  frameDims: DocFrameProps.FrameDims ← TRASH,
  fixedWidth, fixedHeight: BOOL ← TRASH,
  span: Span ← TRASH,
  verticalAlignment: DocFrameProps.VerticalAlignment ← TRASH,
  horizontalAlignment: DocFrameProps.HorizontalAlignment
  ← TRASH,
  topMarginHeight, bottomMarginHeight,
  leftMarginWidth, rightMarginWidth: CARDINAL ← TRASH];
```

17.2.2 Font properties

```
FontPropsDefs.PropsRecord: TYPE = MACHINE DEPENDENT RECORD [
  fontDesc(0:0..31): FontPropsDefs.FontDescription,
  offset(2:0..15): INTEGER ← TRASH,
  foregroundBackground(3:0..1):
    FontPropsDefs.ForegroundBackground,
  nUnderlines(3:2..3): CARDINAL[0..3] ← TRASH,
  strikeout(3:4..4): BOOLEAN ← TRASH;
  placement(3:5..7): FontPropsDefs.Placement ← TRASH,
  unused3(3:8..15): PACKED ARRAY [8..15] OF [0..1] ← ALL[0];
```

17.2.3 Page properties

```
DocPagePropsDefs.PropsRecord: TYPE = MACHINE DEPENDENT RECORD [
  pageDims(0:0..31): DocPagePropsDefs.PageDims ← TRASH,
  topMarginHeight(2:0..15): CARDINAL ← TRASH,
  bottomMarginHeight(3:0..15): CARDINAL ← TRASH,
  leftMarginWidth(4:0..15): CARDINAL ← TRASH,
  rightMarginWidth(5:0..15): CARDINAL ← TRASH,
  startingPageSide(6:0..1): DocPagePropsDefs.PageSide ← TRASH,
  bindingMarginWidth(6:2..15): CARDINAL[0..16383] ← TRASH,
  nColumns(7:0..6): CARDINAL[1..127] ← TRASH,
  balancedColumns(7:7..7): BOOL ← TRASH,
  unused7(7:8..15): PACKED ARRAY [8..15] OF [0..1] ← ALL[0],
  columnSpacing(8:0..15): CARDINAL ← TRASH,
  startingPageNumber(9:0..15): CARDINAL ← TRASH,
  pageNumberFormat(10:0..2):
    DocPagePropsDefs.NumberFormat ← TRASH,
  restartPageNumbering(10:3..3): BOOL ← TRASH,
  unused10(10:4..15): PACKED ARRAY [4..15] OF [0..1] ← ALL[0],
  startingLineNumber(11:0..15): CARDINAL ← TRASH,
  lineNumberInterval(12:0..10): CARDINAL[0..2047] ← TRASH,
  lineNumberFormat(12:11..13): NumberFormat ← TRASH,
  lineNumberLocation(12:14..15):
    DocPagePropsDefs.LineNumberLocation ← TRASH,
  headingStartsOnThisPage(13:0..0): BOOL ← TRASH,
  headingSameOnLeftRightPages(13:1..1): BOOL ← TRASH,
  footingStartsOnThisPage(13:2..2): BOOL ← TRASH,
  footingSameOnLeftRightPages(13:3..3): BOOL ← TRASH,
  unused13(13:4..15): PACKED ARRAY [4..15] OF [0..1] ← ALL[0];
```

17.2.4 Field properties

```
FieldPropsDefs.PropsRecord: TYPE = RECORD [  
  language: MultiNational.Language ← TRASH,  
  length: CARDINAL ← TRASH,  
  required: BOOLEAN ← TRASH,  
  skipIf: FieldPropsDefs.SkipIfChoiceType ← TRASH,  
  stopOnSkip: BOOLEAN ← TRASH,  
  type: FieldPropsDefs.FieldChoiceType ← TRASH,  
  fillInRule,  
  description,  
  format,  
  name,  
  range,  
  skipIfField: xString.ReaderBody ← TRASH,  
  fillInRuleRuns: FontRunDefs.FontRuns ← TRASH];
```

17.2.5 Utilities for getting and setting properties

`DocInterchangeDefs` also provides some routines to get and set properties easily. The following three routines create properties records with "reasonable" default values:

```
DocInterchangeDefs.GetFontPropsDefaults: PROC [  
  props: FontPropsDefs.Props];
```

```
DocInterchangeDefs.GetPagePropsDefaults: PROC [  
  props: DocPagePropsDefs.Props];
```

```
DocInterchangeDefs.GetParaPropsDefaults: PROC [  
  props: ParaPropsDefs.Props];
```

To set properties, you can use `SetCurrentParagraphProps`:

```
DocInterchangeDefs.SetCurrentParagraphProps: PROC [  
  textContainer: DocInterchangeDefs.TextContainer,  
  paraProps: ParaPropsDefs.ReadOnlyProps];
```

You can call `SetCurrentParagraphProps` at any time, with any `TextContainer` as an argument. If you call it repeatedly, only the most recent call will remain in effect.

`SetCurrentParagraphProps` affects the entire current paragraph, including any text that you append later. The properties also affect all subsequent paragraphs unless you override the properties with new ones passed to `AppendNewParagraph`, or by another call to `SetCurrentParagraphProps`.

Note, however, that you must be careful when calling `SetCurrentParagraphProps` on an empty text container. The algorithm that `DocInterchangeDefs` uses for adding the initial new paragraph properties is to check before doing an `Append*`, and add a paragraph character if there is not already one there. Thus, calling `SetCurrentParagraphProps` before calling any `Append*` routines will result in an error, since there is not yet a paragraph character in the text container.

17.3 Example

Here is a simple example that creates a new document and puts a few words in it. Note that it is just a command procedure, and not a complete example:

```

--called when user invokes some command
MakeDoc: MenuData.MenuProc = {
  doc: DocInterchangeDefs.Doc ← NIL;
  heading: DocInterchangeDefs.Heading ← NIL;
  fontProps: FontPropsDefs.PropsRecord;
  pageProps: DocPagePropsDefs.PropsRecord;
  paraProps: ParaPropsDefs.PropsRecord;
  status: DocInterchangeDefs.FinishCreationStatus;
  docFile: NSFile.Handle;
  --strings for doc. and header contents.
  text: XString.ReaderBody ← XString.FromSTRING[
    "But if the while I think on thee, dear friend,
    All losses are restored and sorrows end."L];
  headerText: XString.ReaderBody ← XString.FromSTRING[
    "Shakespeare"L];

  --get default properties
  DocInterchangeDefs.GetFontPropsDefaults[@fontProps];
  DocInterchangeDefs.GetParaPropsDefaults[@paraProps];
  DocInterchangeDefs.GetPagePropsDefaults[@pageProps];
  --create new document with headings and no footings.
  --elide docizn, footing, and second heading, since left
  --and right headings will be the same
  [doc, , heading, , ] ← DocInterchangeDefs.StartCreation[
    paginateOption: compress,
    wantHeadingHandles: TRUE,
    wantFootingHandles: FALSE,
    initialFontProps: @fontProps,
    initialParaProps: @paraProps,
    initialPageProps: @pageProps];
  --add text to document and header
  DocInterchangeDefs.AppendText[
    to: [doc[h: doc]],
    text: @text,
    textEndContext: xstring.unknownContext];
  DocInterchangeDefs.AppendText[
    to: [heading[h:leftHeading]],
    text: headerText,
    textEndContext: xstring.unknownContext];
  --free header and then finish up
  DocInterchangeDefs.ReleaseHeading[@heading];
  [docFile, status] ← DocInterchangeDefs.FinishCreation[@doc];
  IF status # ok THEN UserTerminal.BlinkDisplay[]
  ELSE { --copy document to desktop
    refDoc: NSFile.Reference ← NSFile.GetReference[docFile];
    refDT: NSFile.Reference ←
      StarDesktop.GetCurrentDesktopFile[];
    fileDT: NSFile.Handle ← NSFile.OpenByReference[refDT];
    NSFile.Move[docFile, fileDT];
    NSFile.Close[fileDT];
    NSFile.Close[docFile];
    StarDesktop.AddReferenceToDesktop[refDoc];
  };
};

```

17.4 Enumerating documents

To read (Enumerate) the contents of an existing ViewPoint document, the first step is to call **Open**, which opens the document and returns a **Doc** handle for that document.

```
DocInterchangeDefs.Open: PROC [
  docFileRef: NSFile.Reference,
  password: XString.Reader ← NIL]
  RETURNS [
    doc: DocInterchangeDefs.Doc,
    status: DocInterchangeDefs.OpenStatus];
```

```
DocInterchangeDefs.OpenStatus: TYPE = {
  ok, badSeal, cantOpenIzn, incompatible,
  notLocal, outOfDiskSpaceForBackup,
  outOfDiskSpaceToUpgrade, outOfVMToUpgrade,
  unknownProblem, outOfVMToOpen, accessConflict,
  invalidPassword};
```

password is currently ignored.

Once you have a handle to the document, the next step is to call **Enumerate**, passing in the **Doc** and an **EnumProcs** record. The **EnumProcs** record contains a set of callback procedures, one for each of the following structures: {anchored frame, column break, field, new paragraph, page break, page format character, text, tile}.

```
DocInterchangeDefs.Enumerate: PROC [
  textContainer: DocInterchangeDefs.TextContainer,
  procs: DocInterchangeDefs.EnumProcs,
  clientData: LONG POINTER ← NIL]
  RETURNS [dataSkipped: BOOL];
```

```
DocInterchangeDefs.EnumProcs: TYPE = LONG POINTER TO
  DocInterchangeDefs.EnumProcsRecord;
```

```
DocInterchangeDefs.EnumProcsRecord: TYPE = RECORD [
  anchoredFrameProc:
    DocInterchangeDefs.AnchoredFrameProc ← NIL,
  columnBreakProc: DocInterchangeDefs.ColumnBreakProc ← NIL,
  fieldProc: DocInterchangeDefs.FieldProc ← NIL,
  newParagraphProc:
    DocInterchangeDefs.NewParagraphProc ← NIL,
  pageBreakProc: DocInterchangeDefs.PageBreakProc ← NIL,
  pfcProc: DocInterchangeDefs.PFCProc ← NIL,
  textProc: DocInterchangeDefs.TextProc ← NIL,
  tileProc: DocInterchangeDefs.TileProc ← NIL];
```

Enumerate proceeds sequentially from the beginning of the document: as it comes to different structures within the document, it calls the appropriate callback procedures (which you have to write.) If you don't supply a procedure for some type of object, **Enumerate** will ignore all objects of that type.

Each of the call back procedures takes as parameters the properties of the structure and its content when appropriate. For example, here are the declarations of two of the procedures, one with content and one without. Since the

others are quite similar, we won't include all the declarations; see the *ViewPoint Programmer's Manual* for a complete list.

```
DocInterchangeDefs.ColumnBreakProc: TYPE = PROC [
  clientData: LONG POINTER,
  fontProps: FontPropsDefs.ReadonlyProps]
  RETURNS [stop: BOOL ← FALSE];
```

```
DocInterchangeDefs.TextProc: TYPE = PROC [
  clientData: LONG POINTER,
  fontProps: FontPropsDefs.ReadonlyProps,
  text: XString.Reader,
  textEndContext: XString.Context]
  RETURNS [stop: BOOL ← FALSE];
```

`clientData` is the `clientData` that you passed to `Enumerate`.

The data handle (header, caption, etc.) supplied to you in the call-back is `readonly` and is valid only during the call-back's invocation; you should not try to free this handle. It is possible for such a handle to be `NIL`; a `NIL` handle means that the corresponding object has no text content.

The storage for the properties passed to these procedures is also temporary; you must explicitly copy any properties that you want to save.

Each of the call back procedures returns a boolean value `stop`; if any one of the procedures returns `stop = TRUE`, the enumeration will terminate. If `stop` is never `TRUE`, the enumeration will continue to the end of the document.

Note that the enumeration does include the default paragraph and page format characters supplied with the `TextContainer`. Thus, when copying a document into a new document, you should be careful to avoid copying the default paragraph and page format properties, since that would cause duplication.

Document enumeration can be recursive, just like document creation. For example, if there is a page format character in the document, then you can use `Enumerate` recursively to parse the contents of that page format character.

When the enumeration is complete, you should call `Close` to free all associated data structures and close any open file handles to the document. `Close` sets `docPtr` ↑ to `NIL`.

```
DocInterchangeDefs.Close: PROC [
  docPtr: LONG POINTER TO DocInterchangeDefs.Doc];
```

17.5 Summary

Creating a new document requires the following steps:

- Call `DocInterchangeDefs.StartCreation` to get a doc handle
- Pass that doc handle to `DocInterchangeDefs.Append*` to add information to the document
- Call `DocInterchangeDefs.Release*` to release any valid caption, heading, footing, or field handles.
- Call `DocInterchangeDefs.FinishCreation` to complete the document
- Call `NSFile.Move` to move the file to the desktop and make it permanent and then `StarDesktop.AddReference` to display the new icon on the desktop.

Enumerating a document involves the following steps:

- Call `DocInterchangeDefs.Open` to get a document handle
- Call `DocInterchangeDefs.Enumerate`, passing in the document handle and a record of call back procedures to enumerate the various items within the document
- Call `DocInterchangeDefs.Close` to close the document.

17.6 Example: copying a file

Here is an example of both enumeration and creation. This program adds the command `DocEx` to the Attention Window. When called, this command checks to see if the current selection is a document. If it is, then the program enumerates the contents of that document and copies the information into a new document.

<< A `DICtxtHandle` is passed as `clientData` to procs called by `DocInterchangeDefs.Enumerate`. The record contains handles to the new document, and the old document. `ignoreNewPar` and `ignorePFC` allow you to avoid duplicating the initial page format character and initial new paragraph character. >>

`DICtxtHandle: TYPE = LONG POINTER TO DICtxt;`

`DICtxt: TYPE = RECORD [
 sourceDoc, targetDoc: DocInterchangeDefs.Doc,
 ignoreNewPar, ignorePFC: BOOLEAN];`

`TabStopsHandle: TYPE = LONG POINTER TO TabStops;`

`TabStops: TYPE = RECORD [
 list: SEQUENCE length: CARDINAL OF ParaPropsDefs.TabStop];`

`z: UNCOUNTED_ZONE = Heap.systemZone;`

`diEnumProcs: DocInterchangeDefs.EnumProcs ← NIL;`

```

<< This is the command procedure. It copies the contents of
the currently selected document to a new document. >>
MakeDoc: MenuData.MenuProc = {
  --get reference to selected file
  IF Selection.CanYouConvert[file] THEN {
    selValue: Selection.Value ← Selection.Convert[file];
    docFileRef: NSFile.Reference = LOOPHOLE[
      selValue.value, LONG POINTER TO NSFile.Reference] ↑ ;

    --open source document
    sourceDOC: DocInterchangeDefs.Doc;
    openStatus: DocInterchangeDefs.OpenStatus;
    [sourceDoc, openStatus] ←
      DocInterchangeDefs.Open[docFileRef];

    IF openStatus = ok THEN {
      --declare some variables
      targetDoc: DocInterchangeDefs.Doc;
      diCtxt: DICtxt;
      docFile: NSFile.Handle;
      refDoc, refDt: NSFile.Reference;
      fileDt: NSFile.Handle;
      tabStops: TabStopsHandle;
      fontProps: FontPropsDefs.PropsRecord;
      paraProps: ParaPropsDefs.PropsRecord;
      pageProps: DocPagePropsDefs.PropsRecord;
      --get props from source document, and create new doc
      --with those props
      GetInitialDocProps[
        docFileRef, @sourceDoc, @fontProps, @paraProps,
        @pageProps, @tabStops];
      paraProps.tabStops ←
        IF tabStops = NIL THEN DESCRIPTOR[NIL, 0]
        ELSE DESCRIPTOR[@tabStops.list[0], tabStops.length];
      targetDoc ← DocInterchangeDefs.StartCreation[
        paginateOption: simple, initialFontProps: @fontProps,
        initialParaProps: @paraProps,
        initialPageProps: @pageProps].doc;
      IF tabStops # NIL THEN Z.FREE[@tabStops];
      diCtxt ← [sourceDoc, targetDoc, TRUE, TRUE];

      --start enumeration
      [] ← DocInterchangeDefs.Enumerate[
        [doc[h: sourceDoc]], diEnumProcs, @diCtxt];
      --enumeration done. Close source doc; create new doc,
      make it permanent, and display it on desktop.
      DocInterchangeDefs.Close[@sourceDoc];
      docFile ← DocInterchangeDefs.FinishCreation[
        @targetDoc].docFile;
      refDoc ← NSFile.GetReference[docFile];
      refDt ← StarDesktop.GetCurrentDesktopFile[];
      fileDt ← NSFile.OpenByReference[refDt];
      NSFile.Move[docFile, fileDt]; -- put new doc on Desktop
      NSFile.Close[fileDt];
      NSFile.Close[docFile];
      StarDesktop.AddReferenceToDesktop[refDoc];
    }
  }
  ELSE UserTerminal.BlinkDisplay[];
}
ELSE UserTerminal.BlinkDisplay[];
}; -- MakeDoc

```

<<The call back procedures for enumeration. They all just add the specified structure to the new document.>>

<<Add new paragraph to new document. If it is the first new paragraph character, then ignore it, since new document will already have one.>>

AppendNewParToTargetDoc:

```
DocInterchangeDefs.NewParagraphProc = {
  diCtxt: DICtxtHandle = clientData;
  IF diCtxt.ignoreNewPar THEN diCtxt.ignoreNewPar ← FALSE
  ELSE DocInterchangeDefs.AppendNewParagraph[
    [doc[h: diCtxt.targetDoc]], paraProps, fontProps]; };
```

--Append page break to new document

AppendPageBreakToTargetDoc:

```
DocInterchangeDefs.PageBreakProc = {
  diCtxt: DICtxtHandle = clientData;
  DocInterchangeDefs.AppendPageBreak[
    diCtxt.targetDoc, fontProps];};
```

<<Add page format character to new document. If it is the first format character, then ignore it, since new document will already have one.>>

```
AddPFCToTargetDoc: DocInterchangeDefs.PFCProc = {
  diCtxt: DICtxtHandle = clientData;
  IF diCtxt.ignorePFC THEN diCtxt.ignorePFC ← FALSE
  ELSE [] ← DocInterchangeDefs.AppendPFC[
    to: diCtxt.targetDoc, pageProps: pageProps,
    fontProps: fontProps];};
```

--Append text to new document

```
AppendTextToTargetDoc: DocInterchangeDefs.TextProc = {
  diCtxt: DICtxtHandle = clientData;
  DocInterchangeDefs.AppendText[
    [doc[h: diCtxt.targetDoc]],
    text,
    textEndContext,
    fontProps];};
```

--Copy the font, para, and page props of source document.

```
GetInitialDocProps: PROC[... ] = {...};
```

<<Allocate enumProcs record, and add command to attention menu. EnumProcs record is only interested in new paragraphs, page breaks, page format characters, and text; it ignores all other structures.>>

```
Init: PROC = {
  name: XString.ReaderBody ← XString.FromSTRING["DocEx"L];
  diEnumProcs ← z.NEW[
    DocInterchangeDefs.EnumProcsRecord ← [
      anchoredFrameProc: NIL,
      columnBreakProc: NIL,
      fieldProc: NIL,
      newParagraphProc: AppendNewParToTargetDoc,
      pageBreakProc: AppendPageBreakToTargetDoc,
      pfcProc: AppendPFCToTargetDoc,
      textProc: AppendTextToTargetDoc];
  Attention.AddMenuItem[
    MenuData.CreateItem[z, @name, MakeDoc];};
```

```
Init[];
}.
```

17.7 Exercise

The Form letter application takes a template document and name file as arguments and produces form letters. The template has VP fields that contain keywords; when the application finds one of these keywords, it substitutes the corresponding information from the data file. For example, if a field in the template contained the keyword "LAST" then the resulting document might contain the name "Smith" from the data file. Figure 17.3 illustrates a data file and Figure 17.4 illustrates a template.

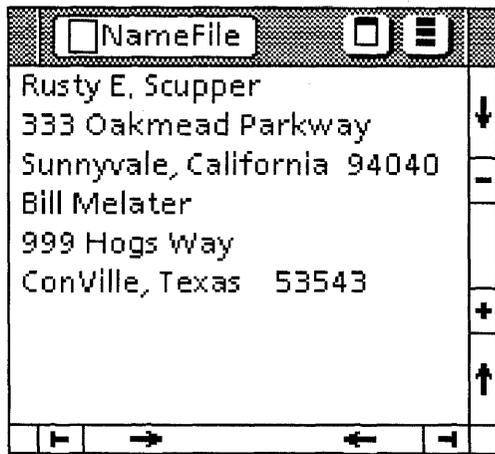


Figure 17.3: Data file

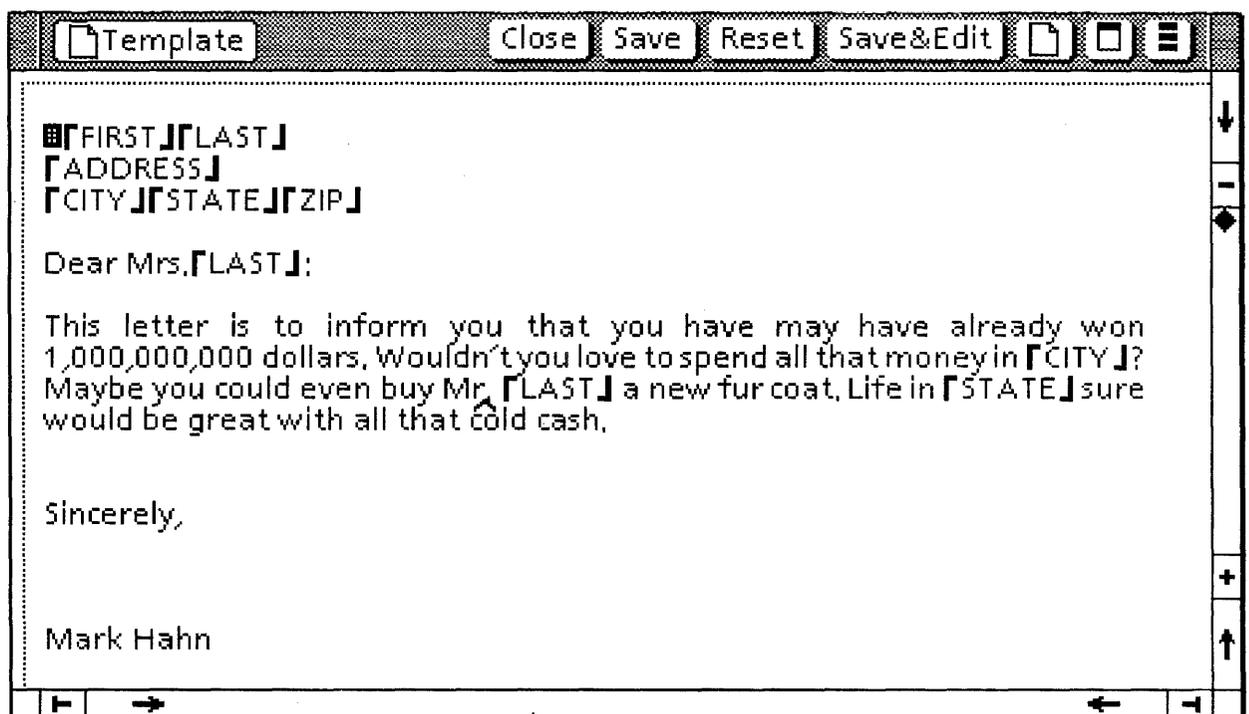


Figure 17.4: Template document

To use the program, you select a template file and a data file and drop them onto the Form letter maker icon, shown in

Figure 17.5. The program will parse the name file, merge its information with the template file, and create a new document on the desktop for each person in the name file.

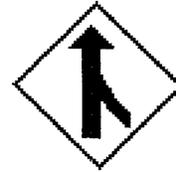


Figure 17.5: Form
letter icon

Your assignment is to write the code to merge the template with the information in the name file and generate new documents. You will need the following files to complete this assignment:

FormLetter.config,
FormLetterDefs.mesa,
FormLetterImpl.mesa,
FormLetterMsgImpl.mesa,
FormLetterDocImplTemplate.mesa.

The procedures that you need to implement are all in FormLetterDocImplTemplate.

This chapter describes how to create and enumerate graphics within graphics frames, using the facilities of the **GraphicsInterchangeDefs** interface. **GraphicsInterchangeDefs** is meant to be used in conjunction with **DocInterchangeDefs**; you need to be familiar with the material in the previous chapter before you read this chapter.

There are also similar interfaces for manipulating tables and charts, which we do not discuss in this course. For information on creating and reading tables within documents, see the **TableInterchangeDefs** chapter in the *ViewPoint Programmer's Manual*. For information on creating and enumerating charts, see the **ChartDataInstallDefs** chapter in the *ViewPoint Programmer's Manual*.

18.1 Creating graphics

To create new graphics, the first step is to call **StartGraphics**, which creates a new graphics frame and returns a graphics **Handle** for it.

Once you have a **Handle**, you can pass that **Handle** to various **Add*** routines to add new graphics objects, such as curves, rectangles, bitmaps, and text frames, to the graphics frame.

When you are through adding graphics, the final step is to call **FinishGraphics**, which returns an object of type **InstanceDefs.Instance**. Typically, you will then pass that handle to **DocInterchangeDefs.AppendAnchoredFrame** to add the frame and its contents to a document.

The following sections discuss each of these steps in detail.

18.1.1 Start routines

To create new graphics objects, you must first call **GraphicsInterchangeDefs.StartGraphics** to get an anchored frame handle:

```
GraphicsInterchangeDefs.StartGraphics: PROC [
  doc: DocInterchangeDefs.Doc]
  RETURNS [h: GraphicsInterchangeDefs.Handle];
```

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

```
GraphicsInterchangeDefs.Object: TYPE;
```

StartGraphics creates a new graphics frame, taking the storage from **doc**. **StartGraphics** returns a **Handle**, which is a pointer to an opaque type that contains, among other things, a *graphics container*. A graphics container is just an object that can contain graphic objects: a graphics container can be an anchored graphics frame, a nested graphics frame, a cusp button within a graphics frame, or another similar construct, such as a chart.

There are also similar routines to create nested frames within an anchored frame: **StartGraphicsFrame** initializes a nested frame within an anchored frame; **StartCluster** initializes a cluster of graphic objects within a graphics frame:

```
GraphicsInterchangeDefs.StartCluster: PROC [
  h: GraphicsInterchangeDefs.Handle,
  box: GraphicsInterchangeDefs.Box]
  RETURNS [ch: GraphicsInterchangeDefs.Handle];
```

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

```
GraphicsInterchangeDefs.Place: TYPE = RECORD [x, y: INTEGER];
```

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

StartCluster initializes a cluster of graphics objects within the graphics frame **h**. **box** describes the size and location of the cluster relative to the anchored frame; **place** and **dims** are in micras. (2540 micras = 1 inch.)

```
GraphicsInterchangeDefs.StartGraphicsFrame: PROC [
  h: GraphicsInterchangeDefs.Handle,
  box: GraphicsInterchangeDefs.Box,
  frameProps: GraphicsInterchangeDefs.FrameProps,
  wantTopCaptionHandle,
  wantBottomCaptionHandle,
  wantLeftCaptionHandle,
  wantRightCaptionHandle: BOOLEAN ← FALSE]
  RETURNS [
    gfh: Handle, topCaption, bottomCaption,
    leftCaption, rightCaption: DocInterchangeDefs.Caption];
```

StartGraphicsFrame creates a new nested nonanchored frame within the anchored frame **h**. Again, **box** describes the location of the nested frame.

want*CaptionHandle indicates whether you want the frame to have the corresponding captions. If you pass in **TRUE** for any of these values, **StartGraphicsFrame** will return a valid caption handle; you can then use **DocInterchangeDefs** routines to add text to the caption. If you pass in **TRUE**, and receive a caption handle, you must eventually free that caption with **DocInterchangeDefs.ReleaseCaption**. See Section 17.1.1.4 for more details.

frameProps are the properties for the nested frame. Note that **StartGraphics** does not have a corresponding parameter to specify the properties for the anchored frame; you set these properties when you add the frame to a document. See Section 17.1.1.3 for details.

```
GraphicsInterchangeDefs.FrameProps: TYPE =
    LONG POINTER TO GraphicsInterchangeDefs.FramePropsRec;
```

```
GraphicsInterchangeDefs.FramePropsRec: TYPE = RECORD [
    brush: GraphicsInterchangeDefs.Brush,
    expandRight, expandBottom: BOOLEAN,
    margins: ARRAY GraphicsInterchangeDefs.Side OF CARDINAL,
    captionContent: ARRAY Side OF DocInterchangeDefs.Caption
];
```

```
GraphicsInterchangeDefs.Brush: TYPE = RECORD [
    wthbrush: CARDINAL,
    stylebrush: GraphicsInterchangeDefs.StyleBrush];
```

```
GraphicsInterchangeDefs.StyleBrush: TYPE = MACHINE DEPENDENT {
    invisible(0), solid(1), dashed(2), dotted(3), double(4),
    broken(5), (15)};
```

```
GraphicsInterchangeDefs.Side: TYPE = {top, bottom, left, right};
```

brush describes the properties of the lines that make up the frame. The brush width is in micras. The standard brush widths on the property sheet are roughly multiples of 35: 35, 71, 106, 141, 176, and 212. You should use one of these widths.

expandRight, **expandBottom** indicate whether the frame should expand automatically when the user puts in more information; these correspond to the entries on the property sheet.

margins are the frame margins, in points.

captionContent is an array of captions for the frame. Note that this parameter is only interesting during enumeration, since you add the caption content *after* you create the frame.

Here is a code fragment that creates the graphics frames shown in Figure 18.1:

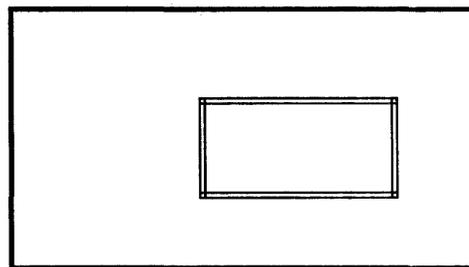


Figure 18.1: Graphics frames

```

doc: DocInterchangeDefs.Doc ← NIL;
graphics: InstanceDefs.Instance;
anchoredFrame, nestedFrame: GraphicsInterchangeDefs.Handle;
bCaption: DocInterchangeDefs.Caption; --for bottom caption
content: XString.ReaderBody ←
  XString.FromSTRING["Figure 18.1: Graphics frames"L];
--set up properties for both frames and para props for caption
nestedProps: GraphicsInterchangeDefs.FramePropsRec ← [
  brush: [wthbrush: 106, stylebrush: double],
  expandRight: FALSE, expandBottom: FALSE,
  margins: [0, 0, 0, 0],
  captionContent: [NIL, NIL, NIL, NIL]];
anchoredProps: DocFramePropsDefs.PropsRecord ← [
  borderStyle: solid,
  borderThickness: 2,
  frameDims: [w: 176, h: 101],
  fixedWidth: TRUE,
  fixedHeight: TRUE,
  span: fullColumn,
  verticalAlignment: floating,
  horizontalAlignment: right,
  topMarginHeight: 0,
  bottomMarginHeight: 0,
  leftMarginWidth: 0,
  rightMarginWidth: 100]; --points; 72 points to the inch
captionProps: ParaPropsDefs.PropsRecord;
DocInterchangeDefs.GetParaPropsDefaults[@captionProps];
captionProps.basicProps.preLeading ← 10; --points
captionProps.basicProps.paraAlignment ← center;

--create doc, then anchored frame, then nested frame. box
--dimensions are in micras; 2450 micras to an inch
[doc, , , ] ← DocInterchangeDefs.StartCreation[];
anchoredFrame ← GraphicsInterchangeDefs.StartGraphics[doc];
[nestedFrame, , ] ← GraphicsInterchangeDefs.StartGraphicsFrame[
  h: anchoredFrame,
  box: [place: [x: 2540, y: 1270], dims: [w: 2540, h: 1270]],
  frameProps: @nestedProps];
--now finish nested frame and anchored frame, and pass
--content of anchored frame to AppendFrame
GraphicsInterchangeDefs.FinishGraphicsFrame[nestedFrame];
graphics ← GraphicsInterchangeDefs.FinishGraphics[
  anchoredFrame];
[, , bCaption, , ] ← DocInterchangeDefs.AppendAnchoredFrame[
  to: doc,
  type: graphics,
  anchoredFrameProps: @anchoredProps,
  content: graphics,
  wantBottomCaptionHandle: TRUE];
--add new par. with new props. Just changing props won't
--work, because there is not yet a para. char in the caption.
DocInterchangeDefs.AppendNewParagraph[
  to: caption:[h: bCaption]],
  paraProps: @captionProps];
DocInterchangeDefs.AppendText[
  to: [caption[h: bCaption]],
  text: @content,
  textEndContext: XString.unknownContext,
  fontProps: @fontProps];
DocInterchangeDefs.ReleaseCaption[@bCaption];
--finish document
[docFile] ← DocInterchangeDefs.FinishCreation[@doc];

```

18.1.2 Add routines

After calling a **Start*** routine to initialize a graphics container, the next step is typically to call various **Add*** routines to add information to the graphics container. The **Add*** routines all add a specified object to a specified place in the graphics frame.

Note that we have not included the declarations of all possible graphics routines; for a complete list, check the **GraphicsInterchangeDefs** documentation in the *ViewPoint Programmer's Manual*.

18.1.2.1 Lines

AddLine is a basic example of an **Add** routine:

```
GraphicsInterchangeDefs.AddLine: PROC [
    h: GraphicsInterchangeDefs.Handle,
    box: GraphicsInterchangeDefs.Box,
    lineProps: GraphicsInterchangeDefs.LineProps];

GraphicsInterchangeDefs.LineProps: TYPE =
    LONG POINTER TO LinePropsRec;

GraphicsInterchangeDefs.LinePropsRec: TYPE = RECORD [
    brush: GraphicsInterchangeDefs.Brush,
    lineEndNW: GraphicsInterchangeDefs.LineEnd,
    lineEndSE: GraphicsInterchangeDefs.LineEnd,
    lineEndHeadNW: GraphicsInterchangeDefs.LineEndHead,
    lineEndHeadSE: GraphicsInterchangeDefs.LineEndHead,
    direction: GraphicsInterchangeDefs.LineDirection
    ];

GraphicsInterchangeDefs.LineEnd: TYPE = MACHINE DEPENDENT
    {flush(0), square(1), round(2), arrow(3), (7)};

GraphicsInterchangeDefs.LineEndHead: TYPE = MACHINE DEPENDENT
    {none(0), h1(1), h2(2), h3(3), (15)};

GraphicsInterchangeDefs.LineDirection: TYPE = MACHINE DEPENDENT
    {WE(0), NS(1), NwSe(2), SwNe(3)};
```

AddLine adds a line to the graphics container at location **box.place**. Thus, **box** is the parameter that describes the location of the line; you specify a line by specifying the box in which the line should fit.

lineEnd* describe the properties of the ends of the curve. **lineEndNW** describes the end that is in the West, North, or North-West; **lineEndSE** describes the end that is in the East, South, or South-East. (Note that West and East take precedence, so an end in the SW is considered the NW end. See the example below.)

If **lineEnd = arrow**, then **lineEndHead** describes the type of arrow: see Figure 18.2. If **lineEnd ≠ arrow**, then **lineEndHead** is **none**.

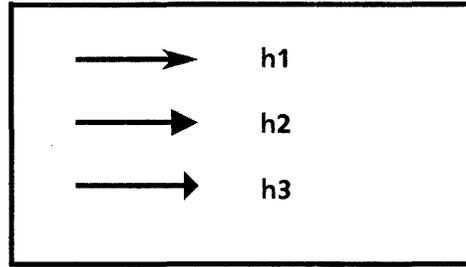


Figure 18.2: Arrowheads

wthbrush specifies the width of the line. As with frames, the standard widths are 35, 71, 106, 141, 176, and 212.

Here is a fragment that creates the graphics frame in Figure 18.3. Note that we have omitted the document creation code, which is the same as in the last example.

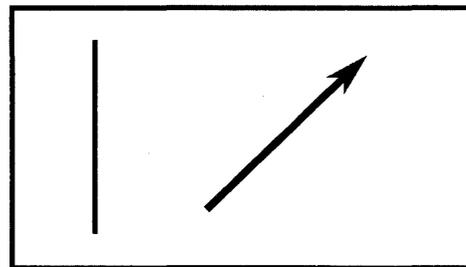


Figure 18.3: Lines

```

--set up line properties
lineProps: GraphicsInterchangeDefs.LinePropsRec ← [
  brush: [wthbrush: 71, stylebrush: solid],
  lineEndNW: square,
  lineEndSE: square,
  lineEndHeadNW: none,
  lineEndHeadSE: none,
  direction: NS];
...
--start graphics frame
anchoredFrame ← GraphicsInterchangeDefs.StartGraphics[doc];
--add first line
GraphicsInterchangeDefs.AddLine [
  h: anchoredFrame,
  box: [place:[x: 1129, y: 494],
    dims:[w:0, h:2540]],
  lineProps: @lineProps];
--change properties
lineProps ← [[106, solid], square, arrow, h1, none, SwNe];
--add second line
GraphicsInterchangeDefs.AddLine [
  h: anchoredFrame,
  box: [place:[x: 2646, y: 635],
    dims:[w:2117, h:2081]],
  lineProps: @lineProps];
--finish up
graphics ← GraphicsInterchangeDefs.FinishGraphics[
  anchoredFrame];

```

18.1.2.2 Rectangles and ellipses

```
GraphicsInterchangeDefs.AddRectangle: PROC [
  h: GraphicsInterchangeDefs.Handle,
  box: GraphicsInterchangeDefs.Box,
  rectangleProps: GraphicsInterchangeDefs.RectangleProps]
```

```
GraphicsInterchangeDefs.RectangleProps: TYPE =
  LONG POINTER TO GraphicsInterchangeDefs.RectanglePropsRec;
```

```
GraphicsInterchangeDefs.RectanglePropsRec: TYPE = RECORD [
  brush: GraphicsInterchangeDefs.Brush,
  shading: GraphicsInterchangeDefs.Shading];
```

```
GraphicsInterchangeDefs.Shading: TYPE = RECORD [
  gray: GraphicsInterchangeDefs.Gray,
  textures: GraphicsInterchangeDefs.Textures];
```

```
GraphicsInterchangeDefs.Gray: TYPE = MACHINE DEPENDENT{
  none(0), gray25(1), gray50(2), gray75(3), black(4), (15)};
```

```
GraphicsInterchangeDefs.Textures: TYPE = PACKED ARRAY
  GraphicsInterchangeDefs.Texture OF BOOLEAN;
```

```
GraphicsInterchangeDefs.Texture: TYPE = MACHINE DEPENDENT{
  vertical(0), horizontal(1), nwse(2), swne(3),
  polkadot(4), (11)};
```

AddRectangle adds the rectangle whose shape is specified by **box.dims** to the graphics container at location **box.place**. **AddEllipse** is just like **AddRectangle**, except that it creates curved lines rather than straight lines. **box.dims** determine the size and shape of the ellipse; **box.place** determines its location relative to the frame.

```
GraphicsInterchangeDefs.AddEllipse: PROC [
  h: GraphicsInterchangeDefs.Handle,
  box: GraphicsInterchangeDefs.Box,
  ellipseProps: GraphicsInterchangeDefs.EllipseProps];
```

```
GraphicsInterchangeDefs.EllipseProps: TYPE = LONG POINTER TO
  GraphicsInterchangeDefs.EllipsePropsRec;
```

```
GraphicsInterchangeDefs.EllipsePropsRec: TYPE = RECORD [
  brush: GraphicsInterchangeDefs.Brush,
  shading: GraphicsInterchangeDefs.Shading];
```

For example, here is a call that creates the ellipse in Figure 18.4:

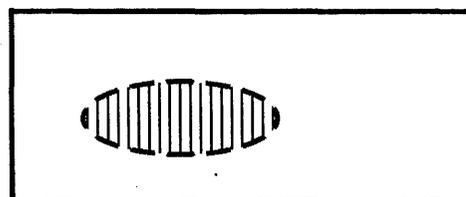


Figure 18.4: Ellipse

```
--set up the ellipse props
ellipseProps: GraphicsInterchangeDefs.EllipsePropsRec ←
  [brush: [wthbrush: 71, stylebrush:dashed],
   shading: [gray:none,
             textures: [TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
                       FALSE, FALSE, FALSE, FALSE]]];
--add the ellipse to the graphics frame
GraphicsInterchangeDefs.AddEllipse [
  h: anchoredFrame,
  box: [place:[x: 1000, y: 1000], dims:[w:2540, h:1000]],
  ellipseProps: @ellipseProps];
```

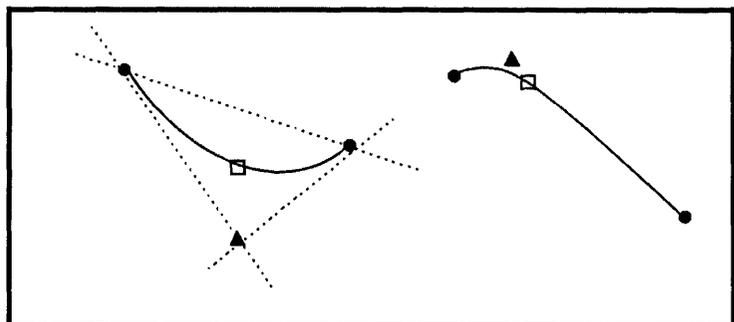
18.1.2.3 Curves

```
GraphicsInterchangeDefs.AddCurve: PROC [
  h: GraphicsInterchangeDefs.Handle,
  box: GraphicsInterchangeDefs.BOX,
  curveProps: GraphicsInterchangeDefs.CurveProps];

GraphicsInterchangeDefs.CurveProps: TYPE = LONG POINTER TO
GraphicsInterchangeDefs.CurvePropsRec;
GraphicsInterchangeDefs.CurvePropsRec: TYPE = RECORD [
  brush: GraphicsInterchangeDefs.Brush,
  lineEndNW: GraphicsInterchangeDefs.LineEnd,
  lineEndSE: GraphicsInterchangeDefs.LineEnd,
  lineEndHeadNW: GraphicsInterchangeDefs.LineEndHead,
  lineEndHeadSE: GraphicsInterchangeDefs.LineEndHead,
  direction: GraphicsInterchangeDefs.LineDirection,
  placeNW, placeApex, placeSE, placePeak:
  GraphicsInterchangeDefs.Place];
```

In `AddCurve`, `placeNW`, `placeApex`, `placeSE`, and `placePeak` are the four points that define the curve, relative to `box` (and not the frame itself.) The *apex* of a curve is the intersection of the tangents; the *peak* is the "highest" point on the curve, where highest is defined as the farthest from the straight line that connects the endpoint.

Figure 18.5 illustrates these four points for two different curves; the triangle marks the apex, the square marks the peak, and the circles mark the endpoints. The dotted lines on the lefthand curve indicate the lines used for determining the peak and apex. Also note that curves always paint clockwise, so you must make sure that the **NW** endpoint precedes the **SW** endpoint when tracing the curve clockwise.



18.5: Defining curves

direction is ignored; you should always set this to `WE`.

box specifies the location of the curve relative to the graphics frame. The function of **box.dims** is slightly different than in the previous **Add*** routines, however. Rather than defining the shape of the curve, **box.dims** specifies the part of the curve that is visible. Thus, if you define a curve that is larger than **box**, only the part of the curve that fits within **box.dims** will appear.

Here is an example that paints the three curves shown in Figure 18.6. Notice that the box for the first curve is not large enough to contain the entire curve, so only the endpoint and the portion that fits within the box appear on the screen. However, the curves *do* print properly, so the printed version of this document will contain the entire curve.



18.6: Curves

```

--set up some initial curve properties
curveProps: GraphicsInterchangeDefs.CurvePropsRec ←
  [brush: [wthbrush: 71, stylebrush:dashed],
   lineEndNW: square,
   lineEndSE: square,
   lineEndHeadNW: none,
   lineEndHeadSE: none,
   direction: WE,
   placeNW: [0,71],
   placeApex: [1199,2681],
   placeSE: [1870, 0],
   placePeak: [1023, 1129]];
...
--after setting things up, add a curve to the graphics frame
GraphicsInterchangeDefs.AddCurve [
  h: anchoredFrame,
  box: [place:[x: 1305, y: 988], dims:[w:1870, h:671]],
  --671 is too small for curve
  curveProps: @curveProps];

--change the curve props and add another curve
curveProps ← [[141, solid], square, square, none, none, WE,
  [1305,1235], [0,212], [1976,0], [917,423]];
GraphicsInterchangeDefs.AddCurve [
  h: anchoredFrame,
  box: [place:[x: 2716, y: 2822], dims:[w:1976, h:1235]],
  curveProps: @curveProps];

--and again: change props and add third curve
curveProps ← [[212, solid], square, square, none, none, WE,
  [0,0], [247,1094],[494,1799],[247,953]];
GraphicsInterchangeDefs.AddCurve [
  h: anchoredFrame,
  box: [place:[5826,1588], dims:[494,2000]],
  curveProps: @curveProps];

```

18.1.2.4 Text frames

There are also a number of **Add*** routines to add various types of frame objects to the graphics container. For example, here is the declaration of **AddTextFrame**:

```
GraphicsInterchangeDefs.AddTextFrame: PROC [
  h: GraphicsInterchangeDefs.Handle,
  box: GraphicsInterchangeDefs.Box,
  frameProps: GraphicsInterchangeDefs.FrameProps,
  wantTextHandle,
  wantTopCaptionHandle,
  wantBottomCaptionHandle,
  wantLeftCaptionHandle,
  wantRightCaptionHandle: BOOLEAN ← FALSE]
RETURNS [
  text: Text, topCaption, bottomCaption,
  leftCaption, rightCaption: DocInterchangeDefs.Caption];
```

```
GraphicsInterchangeDefs.Text: TYPE = LONG POINTER TO
GraphicsInterchangeDefs.TextObject; -
```

```
GraphicsInterchangeDefs.TextObject: TYPE;
```

AddTextFrame adds a text frame to the specified graphics container. **frameProps** and **want*CaptionHandle** are as described in section 18.1.1. If you specify **wantTextHandle = TRUE**, **AddText** will return a handle to a text frame. Once you have the handle to the text frame, you can call any of the **Append*ToText** routines below to add text to the text frame.

```
GraphicsInterchangeDefs.AppendCharToText: PROC [
  to: GraphicsInterchangeDefs.Text,
  char: XChar.Character,
  fontProps: FontPropsDefs.ReadOnlyProps ← NIL,
  nToAppend: CARDINAL ← 1];
```

```
GraphicsInterchangeDefs.AppendFieldToText: PROC [
  to: GraphicsInterchangeDefs.Text,
  fieldProps: FieldProps,
  fontProps: FontPropsDefs.ReadOnlyProps ← NIL]
RETURNS [field: DocInterchangeDefs.Field];
```

```
GraphicsInterchangeDefs.AppendNewParagraphToText: PROC [
  to: GraphicsInterchangeDefs.Text,
  paraProps: ParaPropsDefs.ReadOnlyProps ← NIL,
  fontProps: FontPropsDefs.ReadOnlyProps ← NIL,
  nToAppend: CARDINAL ← 1];
```

```
GraphicsInterchangeDefs.AppendTextToText: PROC [
  to: GraphicsInterchangeDefs.Text,
  text: XString.Reader,
  textEndContext: XString.Context,
  fontProps: FontPropsDefs.ReadOnlyProps ← NIL];
```

These routines are just like the **Append** routines in **DocInterchangeDefs**; see Section 17.1.1 for more information.

If you receive a valid text handle from **AddTextFrame**, you must eventually call **ReleaseText** to return the storage:

```
GraphicsInterchangeDefs.ReleaseText: PROC [
    textPtr: LONG POINTER TO GraphicsInterchangeDefs.Text];
```

ReleaseText releases handles obtained from **AddText**. Like Mesa's **FREE** operator, these routines take a pointer to the object to be freed, and set the handle itself to **NIL**. Thus, after a call to **ReleaseText**, text will be **NIL**.

Here is a code fragment to create the text frame shown in Figure 18.7:

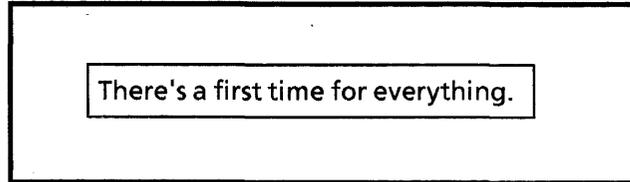


Figure 18.7: Text frame

```
text: GraphicsInterchangeDefs.Text ← NIL;
content: XString.ReaderBody ← XString.FromSTRING[
    "There's a first time for everything."L];
--set up text frame props
textframeProps: GraphicsInterchangeDefs.FramePropsRec ←
    [brush: [35, solid],
     expandRight: TRUE,
     expandBottom: TRUE,
     margins: [0,0,0,0],
     captionContent: [NIL,NIL,NIL,NIL]];
--set up character props for content of text frame
fontProps: FontPropsDefs.PropsRecord;
DocInterchangeDefs.GetFontPropsDefaults[@fontProps];
fontProps.fontDesc.pointSize ← 10;

...
--after setting up graphics frame, add text frame to it
[text] ← GraphicsInterchangeDefs.AddTextFrame [
    h: anchoredFrame,
    box: [place:[x: 1058, y: 847],
         dims:[w:3563, h:706]],
    frameProps: @textFrameProps,
    wantTextHandle: TRUE];
--initialize contents of text frame
GraphicsInterchangeDefs.AppendTextToText[
    to: text,
    text: @content,
    textEndContext: XString.unknownContext,
    fontProps: @fontProps];
--release text handle
GraphicsInterchangeDefs.ReleaseText[@text];
...--finish graphics frame and document
```

18.1.3 Finish routines

When you are through adding things to a new graphics frame, the last step is to call a **Finish** routine:

```
GraphicsInterchangeDefs.FinishCluster: PROC [
    ch: GraphicsInterchangeDefs.Handle];
```

```
GraphicsInterchangeDefs.FinishGraphics: PROC [h: Handle] RETURNS [
    graphics: InstanceDefs.Instance];
```

```
GraphicsInterchangeDefs.FinishGraphicsFrame: PROC [
    gfh: GraphicsInterchangeDefs.Handle];
```

ch, h, and gfh are the handles obtained from the corresponding Start routines. Typically you will pass the InstanceDefs.Instance returned by FinishGraphics to DocInterchangeDefs.AppendAnchoredFrame.

18.2 Reading graphics

You can also use GraphicsInterchangeDefs to read the contents of graphics frames. To read a graphics frame, you start by calling GraphicsInterchangeDefs.Enumerate, which takes as parameters a graphics container and a record of call back procedures, one for each of the kinds of things that might be in the graphics container: bitmap frame, cusp button, cluster, curve, ellipse, form field, frame, image, line, point, rectangle, text, triangle, other.

Enumerate proceeds through the contents of the graphics container, calling the appropriate procedure for each object that it encounters. If you don't provide a procedure for a particular type of object, the enumeration will ignore objects of that type.

```
GraphicsInterchangeDefs.Enumerate: PROC [
    doc: DocInterchangeDefs.Doc,
    graphicsContainer: InstanceDefs.Instance,
    procs: GraphicsInterchangeDefs.EnumProcs,
    clientData: LONG POINTER ← NIL]
    RETURNS [dataSkipped: BOOLEAN];
```

```
GraphicsInterchangeDefs.EnumProcs: TYPE = LONG POINTER TO
    GraphicsInterchangeDefs.EnumProcsRecord;
```

```
GraphicsInterchangeDefs.EnumProcsRecord: TYPE = RECORD [
    bitmapProc: GraphicsInterchangeDefs.BitmapProc ← NIL,
    buttonProc: GraphicsInterchangeDefs.ButtonProc ← NIL,
    clusterProc: GraphicsInterchangeDefs.ClusterProc ← NIL,
    curveProc: GraphicsInterchangeDefs.CurveProc ← NIL,
    ellipseProc: GraphicsInterchangeDefs.EllipseProc ← NIL,
    formFieldProc: GraphicsInterchangeDefs.FormFieldProc ← NIL,
    frameProc: GraphicsInterchangeDefs.FrameProc ← NIL,
    imageProc: GraphicsInterchangeDefs.ImageProc ← NIL,
    lineProc: GraphicsInterchangeDefs.LineProc ← NIL,
    otherProc: GraphicsInterchangeDefs.OtherProc ← NIL,
    pointProc: GraphicsInterchangeDefs.PointProc ← NIL,
    textFrameProc: GraphicsInterchangeDefs.TextFrameProc ← NIL,
    triangleProc: GraphicsInterchangeDefs.TriangleProc ← NIL];
```

Each enumeration procedure takes parameters that describe the properties of the object. These properties are temporary, which means that you shouldn't try to release the storage associated with them. It also means that you must explicitly copy any properties that you wish to save, since they will be destroyed after the procedure returns. You

In the case of a cluster, or nested graphics frame within an anchored frame, you can recursively call `Enumerate` to get the contents of the nested object.

Here are the declarations of some of the enumeration procedures; see the *ViewPointProgrammer's Manual* for the complete documentation:

```
GraphicsInterchangeDefs.ClusterProc: TYPE = PROC [
  clientData: LONG POINTER,
  graphicsContainer: InstanceDefs.Instance,
  box: GraphicsInterchangeDefs.Box]
  RETURNS [stop: BOOLEAN ← FALSE];
```

```
GraphicsInterchangeDefs.CurveProc: TYPE = PROC [
  clientData: LONG POINTER,
  box: GraphicsInterchangeDefs.Box,
  curveProps: GraphicsInterchangeDefs.CurveProps]
  RETURNS [stop: BOOLEAN ← FALSE];
```

```
GraphicsInterchangeDefs.EllipseProc: TYPE = PROC [
  clientData: LONG POINTER,
  box: GraphicsInterchangeDefs.Box,
  ellipseProps: GraphicsInterchangeDefs.EllipseProps]
  RETURNS [stop: BOOLEAN ← FALSE];
```

```
GraphicsInterchangeDefs.FrameProc: TYPE = PROC [
  clientData: LONG POINTER,
  graphicsContainer: InstanceDefs.Instance,
  box: GraphicsInterchangeDefs.Box,
  frameProps: FrameProps]
  RETURNS [stop: BOOLEAN ← FALSE];
```

```
GraphicsInterchangeDefs.LineProc: TYPE = PROC [
  clientData: LONG POINTER,
  box: GraphicsInterchangeDefs.Box,
  lineProps: GraphicsInterchangeDefs.LineProps]
  RETURNS [stop: BOOLEAN ← FALSE];
```

```
GraphicsInterchangeDefs.PointProc: TYPE = PROC [
  clientData: LONG POINTER,
  box: GraphicsInterchangeDefs.Box,
  pointProps: GraphicsInterchangeDefs.PointProps]
  RETURNS [stop: BOOLEAN ← FALSE];
```

```
GraphicsInterchangeDefs.RectangleProc: TYPE = PROC [
  clientData: LONG POINTER,
  box: GraphicsInterchangeDefs.Box,
  rectangleProps: GraphicsInterchangeDefs.RectangleProps]
  RETURNS [stop: BOOLEAN ← FALSE];
```

```
GraphicsInterchangeDefs.TextFrameProc: TYPE = PROC [
  clientData: LONG POINTER,
  box: GraphicsInterchangeDefs.Box,
  frameProps: GraphicsInterchangeDefs.FrameProps,
  content: GraphicsInterchangeDefs.Text]
  RETURNS [stop: BOOLEAN ← FALSE];
```

```
GraphicsInterchangeDefs.TriangleProc: TYPE = PROC [
  clientData: LONG POINTER,
  box: GraphicsInterchangeDefs.Box,
  triangleProps: GraphicsInterchangeDefs.TriangleProps]
  RETURNS [stop: BOOLEAN ← FALSE];
```

18.2.1 Enumerating text frames

There is also a related enumerator, `EnumerateText`, that takes as parameters a text frame and a record of procedures to handle the various kinds of information that can be in a text frame: fields, new paragraphs, and text.

```
GraphicsInterchangeDefs.EnumerateText: PROC [  
  text: GraphicsInterchangeDefs.Text,  
  procs: GraphicsInterchangeDefs.TextEnumProcs,  
  clientData: LONG POINTER ← NIL]  
  RETURNS [dataSkipped: BOOLEAN];
```

```
GraphicsInterchangeDefs.TextEnumProcs: TYPE = LONG POINTER TO  
GraphicsInterchangeDefs.TextEnumProcsRecord;
```

```
GraphicsInterchangeDefs.TextEnumProcsRecord: TYPE = RECORD [  
  fieldProc: DocInterchangeDefs.FieldProc ← NIL,  
  newParagraphProc:  
    DocInterchangeDefs.NewParagraphProc ← NIL,  
  textProc: DocInterchangeDefs.TextProc ← NIL];
```

`EnumerateText` enumerates the contents of a text frame, calling the client-supplied `EnumProcs` as appropriate.

18.3 Summary

Creating new graphics and inserting them in a document involves the following steps:

1. Call `DocInterchangeDefs.StartCreation` to get a document handle (`doc`).
2. Call `StartGraphics[doc]` to get an anchored frame handle (`h`).
3. Call `Add*[h]` to add graphics to the anchored frame.
4. Call `FinishGraphics[h]` to complete the anchored frame and get an object of type `InstanceDefs.Instance` (`graphics`).
5. Call `DocInterchangeDefs.AppendAnchoredFrame[graphics]`, optionally receiving caption handles in return.
6. Call `DocInterchangeDefs.Append*` to add information to the captions. (Optional.)
7. Call `DocInterchangeDefs.FinishCreation[@doc]`.

Enumerating the contents of an existing graphics frame involves the following steps:

1. Call `DocInterchangeDefs.Open` to open the document
2. Call `DocInterchangeDefs.Enumerate`, passing in a `DocInterchangeDefs.AnchoredFrameProc` to handle any graphics frames within the document.
3. Within the `DocInterchangeDefs.AnchoredFrameProc`, call `GraphicsInterchangeDefs.Enumerate`, passing in procedures for any graphic objects of interest.
4. Call `DocInterchangeDefs.Close`.

A. PROGRAMMING IN VIEWPOINT

This appendix describes some miscellaneous information that you will need to know about running and debugging a new ViewPoint application. You need to have finished the XDE tutorials before you read this appendix.

If you have never run a ViewPoint application before, you should read this appendix before you start the course. If you have, you might want to skim this anyway.

A.1 The programming cycle

When you program in the Xerox Development Environment (XDE), there are two possible "target environments": XDE and ViewPoint. When you want to write new applications for XDE itself, you write the application in CoPilot, test it in Tajo and debug in CoPilot until it works, and then eventually run it in CoPilot alongside other XDE tools.

When you program for ViewPoint, you write the code in CoPilot, test it in ViewPoint, debug in CoPilot, and then eventually run it in ViewPoint alongside other applications.

When you are testing a new Viewpoint application, you need to copy the code to ViewPoint and then run it there. There are two ways to do this: you can copy the code from XDE to ViewPoint with CommandCentral, or you can put the code in a remote file drawer from XDE and then retrieve it from ViewPoint. During development, you can use either method. Once you are through with your application, however, you should put it on a file server so that others can access it. (See the *XDE User's Guide* for more information on CommandCentral, the debugger, the editor, the compiler, and the binder.)

One thing to remember when you are testing an application is that loading an application does not produce any visible results. As discussed in Chapter 2, User interface, an application can run either from a command in the Attention Menu or from an icon on the desktop. If the application runs from a command in the Attention Menu, you need to bring up that menu before you see the command. If the application runs from an icon, you will have to open the Prototype folder and copy the icon onto your desktop. Applications do not place icons directly on the desktop: that is the user's prerogative.

A.2 The Workstation Profile

As mentioned in the introduction, you need to have a Workstation Profile on your machine. This file specifies

whether or not you are a "developer"; being a developer gives you certain privileges that customers do not have. Your WorkstationProfile should look like this:

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

```
[Application Loader]
Developer: TRUE
```

You might also want to have a UserProfile, which allows you to set defaults for various tools. The course does not depend on one, however.

A.3 The SystemFolder

Every file on your ViewPoint system must be either on the desktop or in the *system directory*. 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 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. You can then copy those files onto your desktop, or onto the loader, as described in the next section.

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

A.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 object 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. (Chapter 16, Application folders, discusses application folders in detail.)

This distinction is important because the Loader looks for the following entry in 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.

A.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.

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.

Notes:

The icon editor is a tool that allows you to create icon files for inclusion in an application folder.

B.1 Getting started

To use the icon editor, make sure that you have the files **BWSIconEditor.bcd** and **Standard.icons** in your system folder. If they are not there, you can add them either by copying them from a file drawer, or by running them from Command Central using the `/-e` switch.

Once you have these two files in the system folder, you need to do the following:

1. Open your System folder and copy **Standard.icons** to your desktop. **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 copy of **Standard.icons** that is on your desktop 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.)

The basic idea is that you create a new icon by modifying an existing icon. Once you have the new icon, you can use it for any application. There is nothing to prevent you from modifying **Standard.icons** directly, without copying it; copying the file just protects you from accidentally overwriting an existing icon.

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. (See the *ViewPoint Series Reference Library* for more information on icons.)

The next section describes the available editing commands.

5. When you have finished editing the file, invoke the Save command. You then have a file called **new.icons** that

contains your new icon. You can delete all the other icons in the file, and then include **new.icons** in an application folder, as described in Chapter 16, Application Folders.

B.2 Editing the 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 editing.
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**.

This chapter describes tools for creating, modifying, and translating *message files*. There are three related tools: the Message Master File Creation tool, the Message Master Editor, and the Message Runtime File Creation tool.

The *Message Master File Creation Tool* takes a message bcd and generates a *Message Master* file. A Message Master file contains the original text, a translation of that text, and additional information for the translator. You can then modify or translate those messages with the *Message Master Editor*.

Finally, once you have finished editing your messages, you need to use the *Message Runtime File Creation Tool*. This tool builds a Message Runtime file from a Message Master file. The runtime file ("compiled version") contains information for a running application; it cannot be edited.

C.1 Message Master File Creation Tool

To create a message master file, the first step is to run `MasterFileCreate.bcd`; this will create an icon identified by the words "Msg Master Maker." Next, copy the file containing your message information to this icon. (The file with the message information is the compiled version of the message implementation; it can be either a single file or a folder with several files. See Chapter 3, 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, 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 C.1 illustrates this option sheet.

Message Master Creation Tool	
Start	Cancel
Application Name:	The name of the application
Version:	3.3i
Language:	US

Figure C.1 Message Master Creation Tool

To create 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 identified by the name of the application, as illustrated in Figure C.2.

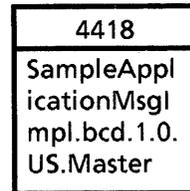


Figure C.2 Master File icon

The name of the Message Master file will be:

<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.

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 3, Strings and Messages, 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.

The message master file contains an untranslated version of all the messages. Once you have this file, you can either edit the messages with the Message Master Editor, or you can go directly to the last step, creating a runtime ("compiled") version of the file. Typically, you won't need to edit the Message Master file when you first create it, but you may later need to changes the messages and create a new runtime message file. Section C.3 discusses the Message Master Editor; Section C.4 discusses the Message Runtime File Creation Tool.

C.2 Message File property sheet

There is also a property sheet associated with each Message Master file. 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 C.3.

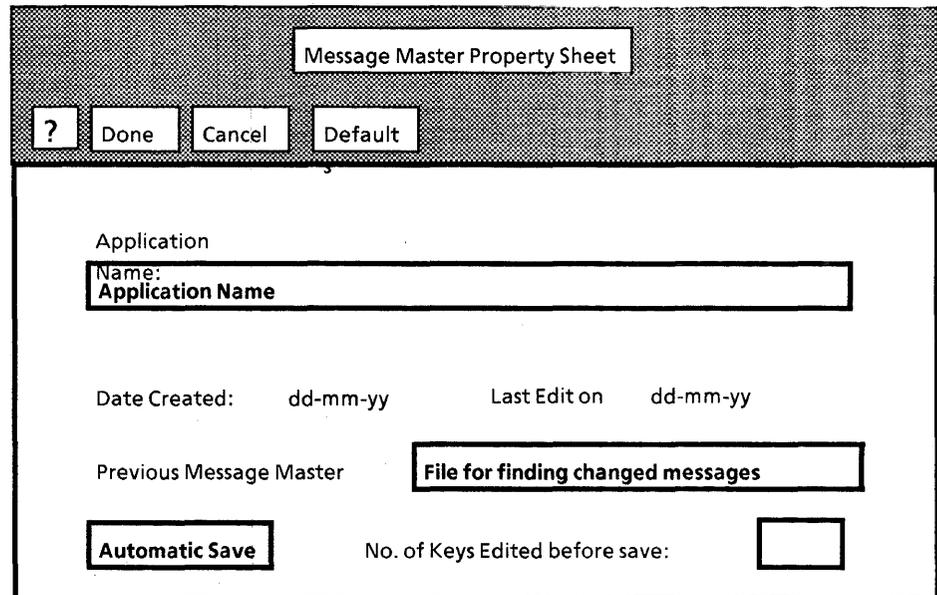


Figure C.3 Message Master property sheet

You can use this property sheet to change the application name, frequency of automatic save, and so on. For more information on any of the fields in this property sheet, see the complete Message Tools documentation.

C.3 Message Master Editor

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 C.4.

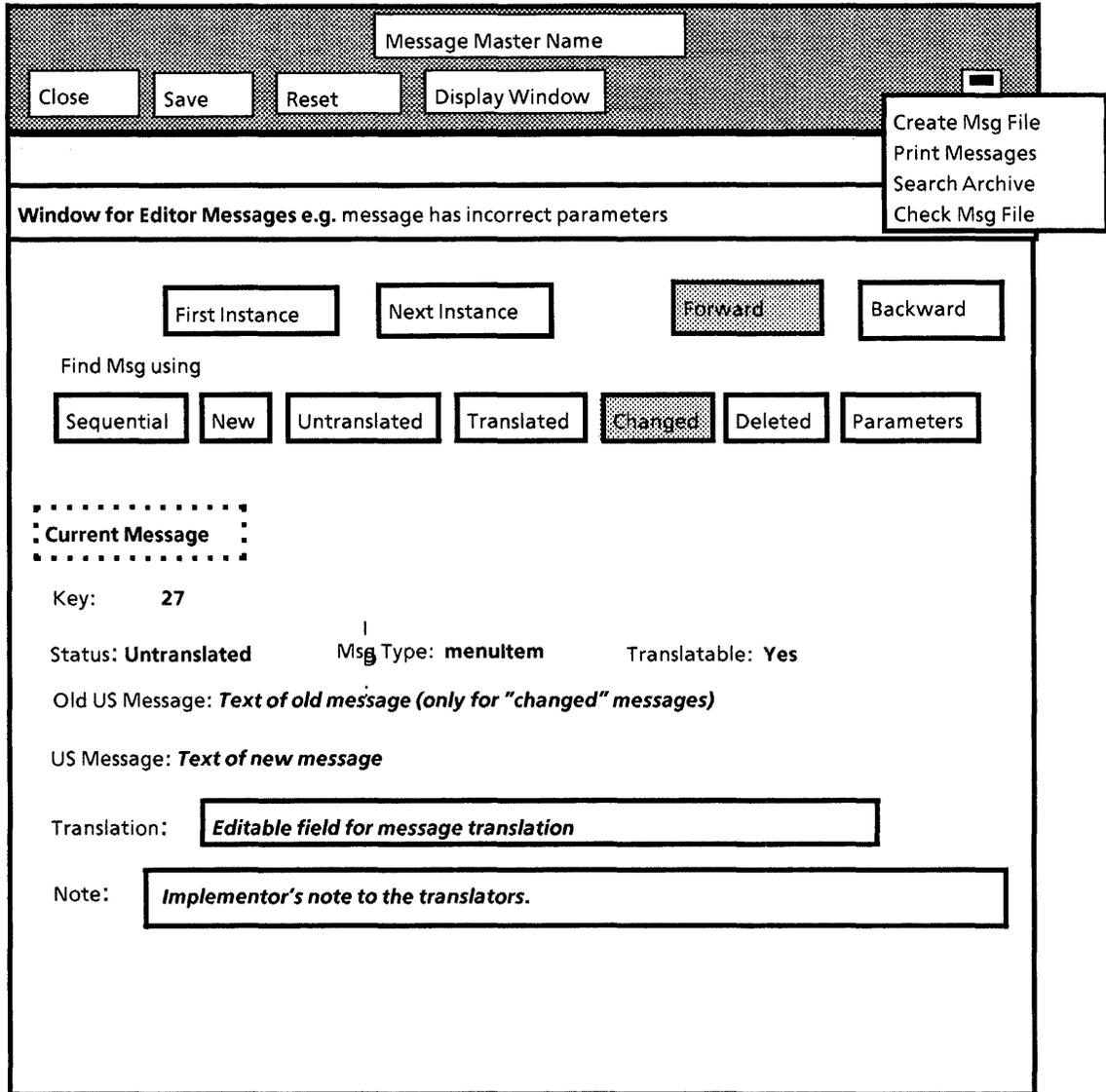


Figure C.4 Message Master Editor window

Using this window, you can edit the current message, and change the current message using various search criteria. For example, you can examine and potentially modify any new messages, or all untranslated messages, or the like.

C.3.1 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.

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. See Section C.2.2 for details.

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.

C.3.2 Search parameters

You can also search by parameter. When you select the parameters option, a section on search parameters will appear, as illustrated in Figure C.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**).

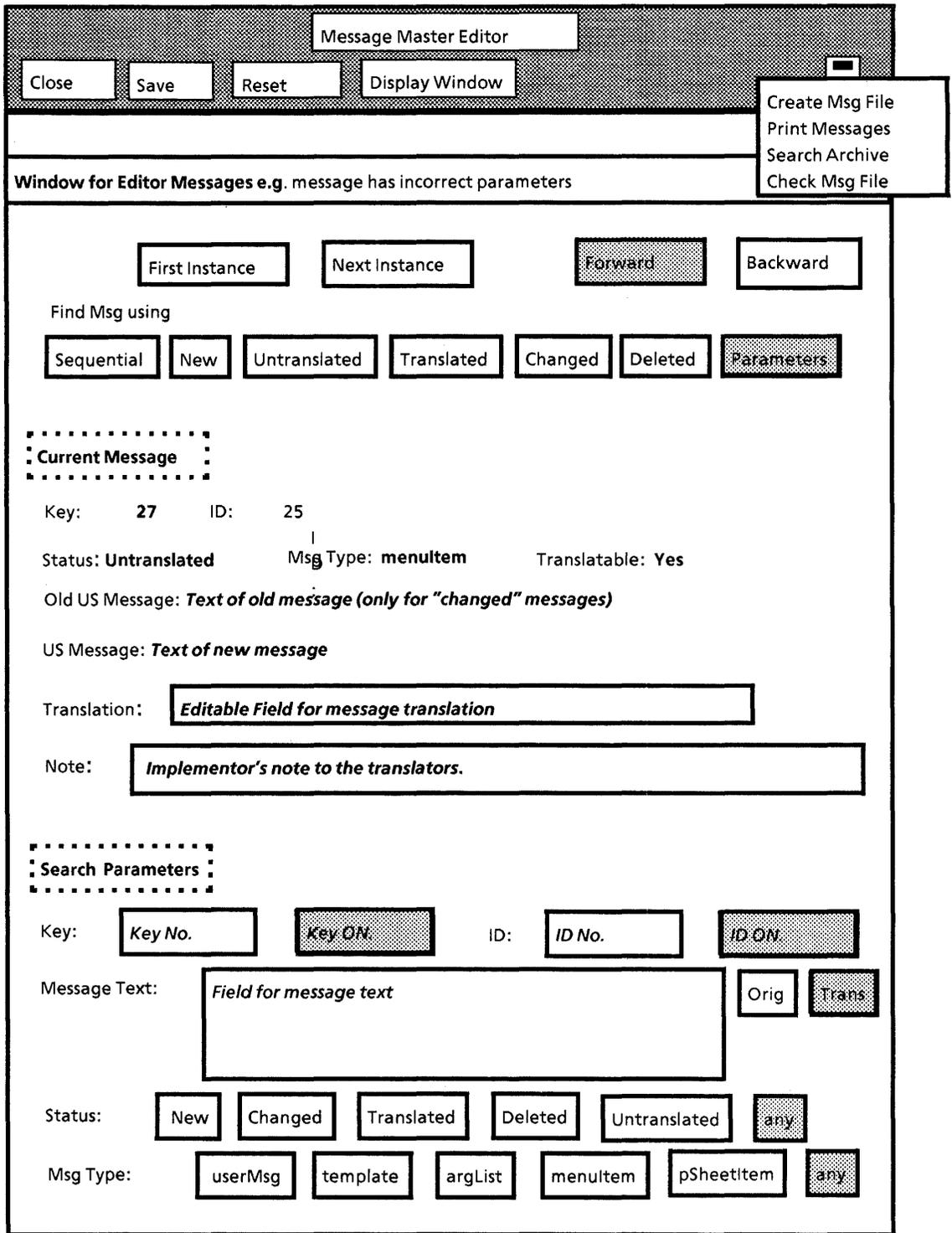


Figure C.5 Searching by parameter

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.

When a search is successful, a message appears with information about the message. See the complete Messate Tools documentation for more information.

C.3.3 Closing, saving, and resetting

Once you have edited a message 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 command.)
- 2 Use the Automatic Save function in the Message File property sheet. See section C.2 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.

You can also use the Reset command to restore a file to its last saved state. If you have changed the file, you must confirm the command (by reselecting the Reset command.)

C.3.4 Printing message files

To obtain hardcopies of information contained within the Message Master, use the **Print Messages** command in the auxiliary menu of the Message Editor. This command displays a list of print options, as illustrated in Figure C.6.

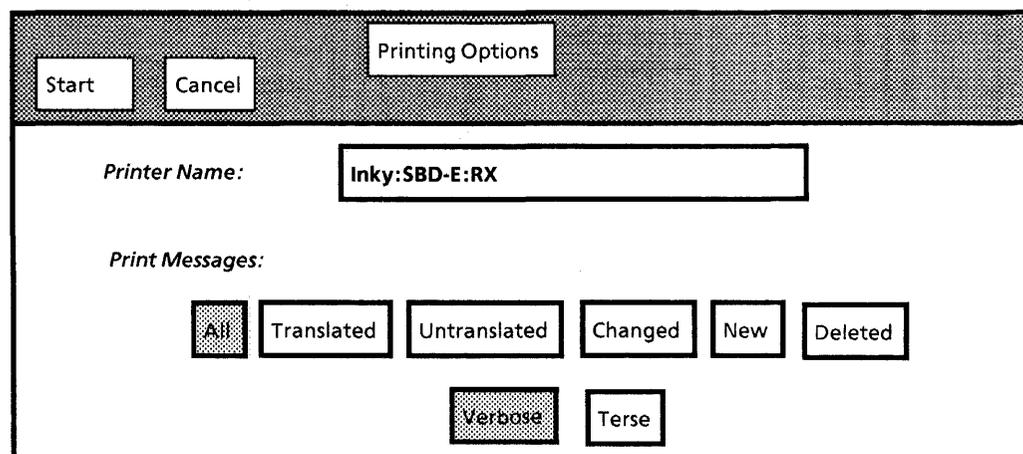


Figure C.6 Print options

Selecting **Start** produces an Interpress master and sends it to the printer specified in the **Printer Name** field. You can print all messages or a category of messages. (The category can be translated, untranslated, changed, new, or deleted.)

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

C.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. You can run this program either from the auxiliary menu of the Message Master Editor, or as a tool in its own right. In either case, you need to run the program `RuntimeFileCreate.bcd`.

To run this tool from the Message Master Editor, select the Create Message File command from the auxiliary menu. This command creates the window illustrated in Figure C.7.

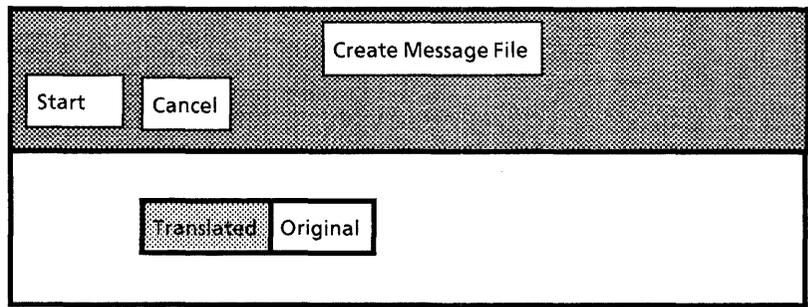
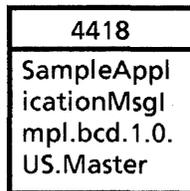


Figure C.7 Using the Create Message File command

This window allows you to specify the type of text (original or translated) in the Runtime Message file. If you specify **Original**, the Runtime Message file will contain only the original text. If you specify **Translated**, the command Runtime Message file will contain 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 C.8, this icon is just like the message master icon, except that its name ends in `.Runtime` instead of `.Master`. You cannot perform any operations on the runtime file.

Master File Icon



Runtime File Icon

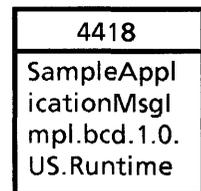


Figure C.8 The resulting icon

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 on the desktop.

Notes:

.autorun applications, A-3
 .TIPC files, 9-8

A

aborting an application, 9-15
 aborting document creation, 17-8
 access attributes, 10-2
 access control (of files), 11-2
 activity attributes, 10-2
 ADF, 16-2
 example, 16-2
AdjustProc, 6-10
 anchored frames
 adding to documents, 17-6
 properties of, 17-8
 apex
 of a curve, 18-8
 application
 internal name for, 16-2
 vs. application folders, A-2
 Application Description File, 16-2
 application folders, 16-1
 creating, 16-9
 summary, 16-10
 Application Loader, A-2
ApplicationFolder interface
 FindDescriptionFile (Def), 16-3
 FindDescriptionFile (Ex), 16-6
 FromName (Def), 16-3
 FromName (Ex), 16-5, 16-8
 applications
 loading, A-2
 loading priority for, 16-2
 packaging, 16-1
 running, A-1
 Applize tool, 16-9
 atom, 9-5
 creation of (Ex), 9-6
Atom interface
 Make, 9-5
 MakeAtom (Def), 9-6
 MakeAtom (Ex), 9-6, 9-11
Attention interface
 AddMenuItem (Def), 4-1, 4-2
 AddMenuItem (Ex), 4-11
 Clear, 3-15
 ClearSticky, 3-15
 Post (Def), 3-14
 Post (Ex), 3-14, 4-5
 PostAndConfirm, 3-14
 PostSticky, 3-14
Attention Menu
 adding a command to, 4-1

Attention Window, 2-4
 clearing messages, 3-15
 posting messages to, 3-14
 attribute types, 10-4, 10-5
 attributes (of files)
 access, 10-2
 activity, 10-2
 changing, 10-5
 directory, 10-2
 example, 10-8
 extended, 10-2
 file, 10-2
 identity, 10-2
 interpreted, 10-1
 of files, 10-1
 retrieving uninterpreted, 10-10
 size, 10-2
 specifying, 10-2
 uninterpreted, 10-1
 autorun applications, A-3

B

backing file, 13-2
 bad phosphor list, 6-3
 body windows, 4-5
 display of, 6-11
 boolean
 in form window, 7-2
BWSFileTypes, 11-4
 SystemFileCatalog, 13-15

C

call back procedure, 4-1
CanYouTakeSelection, 15-6
Catalog interface
 CreateFile (Def), 11-8
 CreateFile (Ex), 13-16
 GetFile (Def), 11-4
 GetFile (Ex), 11-5, 13-11
 Open (Def), 11-4
 Open (Ex), 13-15
 catalogs, 11-4
 change procedure
 in form window, 7-9
 changed boolean
 for form window item, 7-9
ChangeProcs
 with icon applications, 15-7
 Character Code Standard, 3-1
 character set, 3-1
 characters
 defining new, 15-5
 representation of, 3-1

choice item
 creating, 7-5
 in form window, 7-2
 clipping (of windows), 6-3
 commands
 in form window, 7-1 - 7-3
 in StarWindowShell, 4-6
Containee interface
ChangeProc (Def), 15-6
ChangeProc (Ex), 15-7
Data, 15-3
DataHandle, 15-3
DefaultFileConvertProc(Def), 15-2
DefaultFileConvertProc (Ex), 15-13
GenericProc (Def), 15-6
GenericProc (Ex), 15-7, 15-13
GetCachedName (Def), 15-3
GetCachedName (Ex), 15-4, 15-13
GetImplementation, 15-2, 15-13, 16-9
Implementation (Def), 15-1
Implementation (Ex), 15-13
PictureProc (Def), 15-3
PictureProc (Ex), 15-4, 15-13
PictureState 15-3
ReturnTicket (Def), 15-4
ReturnTicket (Ex), 15-4, 15-13
SetImplementation (Def), 15-1
SetImplementation (Ex), 15-2, 15-13, 16-9
SmallPictureProc (Def), 15-5
SmallPictureProc (Ex), 15-6, 15-13
Ticket, 15-4
 content
 of files, 10-1
Context interface, 5-1
Acquire, 5-3
Create (Def), 5-2
Create (Ex), 5-3, 5-5
Data, 5-2
DestroyProcType, 5-2
Find, 5-3
NopDestroyProc, 5-3
SimpleDestroyProc (Def), 5-3
SimpleDestroyProc (Ex), 5-6
UniqueType, 5-2, 5-4
 context
 full example, 5-4
 retrieving, 5-3
 control point, 2-3
 controls
 for file access, 11-2
 COORDS, 9-4
 COPY
 implementing, 15-6
Courier interface
Error, 10-10, 11-14
ErrorCode, 11-14
 current selection, 14-1 (*See selection*)
Cursor interface
Set, 6-9
Store, 6-9
 curves
 adding to graphics frame, 18-8
 apex of, 18-8
 peak of, 18-8

D
 decimal item (in form window), 7-2
 default session, 10-8
 deleting files, 11-8
 desktop, 2-1
 adding files to, 17-7
 directory attributes, 10-2
 Directory icon, 2-6
 dirty page, 13-2
 updating, 13-6
Display interface, 6-5
Bitmap (Def), 6-5
Bitmap (Ex), 6-7, 6-9
Black, 6-5
Gray, 6-7
Line, 6-7
White, 6-9
 display procedure, 6-1, 6-3
 displaying on the screen, 6-1
 text, 6-4
 windows, 4-10
DocFrameProps interface
PropsRecord, 17-8
DocInterchangeDefs interface, 17-1
AbortCreation, 17-8
AnchoredFrameType, 17-6
AppendAnchoredFrame (Def), 17-6
AppendAnchoredFrame (Ex), 18-4
AppendChar, 17-4
AppendField, 17-5
AppendNewParagraph, 17-5
AppendPFC, 17-5
AppendText, 17-5, 17-11
Caption, 17-3
CheckAbortProc, 17-8
Close, 17-13
ColumnBreakProc, 17-12
Doc, 17-1
Enumerate (Def), 17-12
Enumerate (Ex), 17-15
EnumProcs, 17-12
FinishCreation (Def), 17-7
FinishCreation (Ex), 17-7, 17-11
FinishCreationStatus, 17-7
FinishCreationWithCheckAbortProc, 17-8
GetFontPropsDefaults, 17-10
GetPagePropsDefaults, 17-10
GetParaPropsDefaults (Def), 17-10
GetParaPropsDefaults (Ex), 18-4
Open (Def), 17-12
Open (Ex), 17-15
OpenStatus, 17-12
PaginateOption, 17-1
ReleaseCaption (Def), 17-7
ReleaseCaption (Ex), 18-4
ReleaseField, 17-7
ReleaseFooting, 17-7
ReleaseHeading (Def), 17-7
ReleaseHeading (Ex), 17-11
SetCurrentParagraphProps, 17-10
StartCreation (Def), 17-1
StartCreation (Ex), 17-11
TextContainer, 17-3
TextProc, 17-12

DocPagePropsDefs interface**PropsRecord**, 17-9

documents

- aborting creation of, 17-8
- accessing, 17-1
- adding anchored frames, 17-6
- adding formatting information, 17-5
- adding information to, 17-2
- adding page format characters, 17-5
- adding text, 17-3
- adding text (ex), 17-5
- creating, 17-1
- creating heading (Ex), 17-11
- creating headings, 17-2
- creation (Ex), 17-11
- enumerating, 17-12
- enumeration example, 17-15
- example of copying, 17-14
- field properties, 17-10
- finalizing, 17-7
- font properties, 17-8
- frame properties, 17-8
- getting properties, 17-10
- getting properties (Ex), 17-11
- initial pagination of, 17-1
- initial paragraphs, 17-4
- making files permanent, 17-7
- page properties, 17-9
- properties of, 17-8
- reading contents of, 17-12
- setting properties, 17-10
- setting props for empty paragraph, 17-10

E

ellipses

- adding to graphics frame, 18-7

enable actions

- in TIP tables, 9-4

encoding uninterpreted attributes, 10-4

ENTER, 9-4

Environment interface**BitAddress**, 6-5**Block (Def)**, 12-4**Block (Ex)**, 13-12**bytesPerPage**, 13-1**wordsPerPage**, 13-1

example

- of accessing mapped files, 13-5
- of ADF, 16-2
- of all filing operations, 13-9
- of appending text, 17-5
- of catching filing errors, 11-13
- of **Containeer.ChangeProc**, 15-7
- of copying files, 17-14
- of copying selection, 14-4
- of creating atoms, 9-6
- of creating nested graphics frames, 18-3
- of creating text frame, 18-11
- of document creation, 17-11
- of end of stream, 12-3
- of enumerating selection, 14-6
- of file creation, 11-7
- of generic proc, 15-7
- of implementing property sheet commands, 15-7

of listing files, 11-12

of making icon applications, 15-12

of mapped files, 13-6

of mapping files, 13-4

of obtaining selection, 14-2

of opening files, 11-3

of opening remote file, 11-6

of putting icon in Prototype folder, 15-12

of registering icon with desktop, 15-2

of retrieving extended attributes, 10-10

of setting filing attributes, 10-8

of stream blocks, 12-6

of stream IO, 12-3

of TIP tables, 9-11

of uninterpreted attributes, 10-4

of using icons file, 16-9

of using TIP files, 16-7

of **XFormat**, 3-9of **XMessage** implementation module, 3-13

EXIT, 9-4

extended attributes, 10-2

(See *uninterpreted attributes*)

external name, 16-2

F**FieldPropsDefs** interface**PropsRecord**, 17-10

field properties, 17-10

file attributes, 10-2

file handle, 11-2

file system, 10-1

file type, 10-4

files

access control, 11-2

adding to desktop, 17-7

closing, 11-7

complete example, 13-9

contents of, 12-1

controls for, 11-2

creating, 11-7

deleting, 11-8

error handling for, 11-12

IO, 12-2

listing, 11-9

making permanent, 17-7

mapping (Ex), 13-4

mapping to virtual memory, 13-2

opening, 11-2

references to, 11-1

remote, 11-5

segments of, 13-3

temporary, 11-7

unmapping, 13-5

fixed layout, 7-8

flexible layout, 7-6

font properties, 17-8

FontPropsDefs interface**PropsRecord**, 17-8

form items, 2-4, 7-1

creating, 7-3

getting and setting values, 7-13

key for, 7-3

form windows, 2-4, 7-1

destroying, 7-13

format procedures, 3-8
FormWindow interface
 AppendItem (Def), 7-7
 AppendItem (Ex), 7-8, 7-12
 AppendLine, 7-7
 Bitmap, 7-5
 ChangeReason, 7-9
 ChoiceIndex, 7-5
 ChoiceItem, 7-5
 ChoiceItems, 7-5
 CommandProc, 7-4
 Create (Def), 7-2
 Create (Ex), 7-10
 DefaultLayout, 7-6
 defaultLineHeight, 7-7
 Destroy, 7-13
 DestroyItem, 7-13
 DestroyItems, 7-13
 GetBooleanItemValue, 7-13
 GlobalChangeProc, 7-9
 HasAnyBeenChanged, 7-9
 HasAnyBeenChanged (Ex), 8-3
 HasBeenChanged, 7-9
 HasBeenChanged (Ex), 8-3
 InsertItem, 7-7
 InsertLine, 7-7
 LayoutProc (Def), 7-6
 LayoutProc (Ex), 7-8, 7-12
 MakeBooleanItem, 7-10
 MakeChoiceItem (Def), 7-5
 MakeChoiceItem (Ex), 7-11
 MakeCommandItem (Def), 7-3
 MakeCommandItem (Ex), 7-11
 MakeItemsProc (Def), 7-2
 MakeItemsProc (Ex), 7-10
 MakeTextItem (Def), 7-4
 MakeTextItem (Ex), 7-11
 OutlineOrHighlight, 7-6
 ResetAllChanged, 7-9
 ResetChanged, 7-9
 SetBooleanItemValue, 7-13
 SetItemBox, 7-8
 SetTabStops, 7-12
 SetTextItemValue, 13-11
 TabStops, 7-7
 TabStops (Ex), 7-12
frame properties, 17-8

G

graphics container, 18-2
graphics frames
 adding text frames, 18-10
 creating ellipses, 18-7
 creating, 18-1
 creating captions, 18-2
 creating lines, 18-5
 creating rectangles, 18-5
 finalizing, 18-11
 inserting curves, 18-8
 nested, 18-2
 reading, 18-12
 setting properties, 18-2

GraphicsInterchangeDefs interface, 18-1

AddCurve, 18-8
AddEllipse, 18-7
AddLine, 18-5
AddRectangle, 18-6
AddTextFrame, 18-10
AppendCharToText, 18-10
AppendFieldToText, 18-10
AppendNewParagraphToText, 18-10
AppendTextToText, 18-10
Box, 18-2
Brush, 18-3
ClusterProc, 18-12
CurveProc, 18-12
CurveProps, 18-8
CurvePropsRec, 18-8
Dims, 18-2
EllipseProc, 18-12
EllipseProps, 18-7
EllipsePropsRec, 18-7
Enumerate, 18-12
EnumerateText, 18-14
EnumProcs, 18-12
EnumProcsRecord, 18-12
FinishCluster, 18-11
FinishGraphics, 18-12
FinishGraphics (Ex), 18-4
FinishGraphicsFrame, 18-12
FrameProc, 18-12
FrameProps, 18-3
FramePropsRec, 18-3
Gray, 18-7
LineDirection, 18-5
LineEnd, 18-5
LineEndHead, 18-5
LineProc, 18-12
LineProps, 18-5
LinePropsRec, 18-5
Place, 18-2
PointProc, 18-12
RectangleProc, 18-12
RectanglePropRec, 18-6
RectangleProps, 18-6
ReleaseText, 18-11
Shading, 18-7
StartCluster, 18-2
StartGraphics, 18-1
StartGraphics (Ex), 18-4
StartGraphicsFrame, 18-2
StartGraphicsFrame (Ex), 18-4
StyleBrush, 18-3
Text, 18-10
TextEnumProcs, 18-14
TextEnumProcsRecord, 18-14
TextFrameProc, 18-12
Texture, 18-7
Textures, 18-7
TriangleProc, 18-12

H

headings
 creating, 17-2
 creating (Ex), 17-11

- I
- icon applications
 - full example, 15-12
 - summary, 15-12
- icon editor, B-1
- icon files
 - creating, B-1
 - example of using, 16-9
- icons, 2-1, 15-1
 - displaying, 15-3
 - putting in Prototype folder, 15-11
 - registering with desktop, 15-1
 - states, 15-3
 - tiny version, 15-5
- identity attributes, 10-2
- input focus
 - setting, 9-10
- integer item
 - in form window, 7-2
- internal name, 16-2
- interpreted attributes, 10-1
 - changing, 10-5
 - retrieving, 10-5
 - specifying, 10-3
- invalid list
 - of window, 6-1
- invisible applications, A-3
- IO (to files), 12-2

- K
- kamikaze periodic notifier, 9-14

- L
- LimitProc, 6-10
- lines (in graphics frame), 18-5
- link window, 8-4
- linked property sheets, 8-5
- loading priority, 16-2
- lock
 - for file access, 11-2
- LOOPHOLE, 12-5, 12-6, 13-5, 13-7

- M
- managers (selection), 14-1
- mapped files, 13-2
 - accessing, 13-4
 - example, 13-4, 13-6
 - unmapping, 13-5
 - updating, 13-5
- MasterFileCreate.bcd, C-1
- menu, 2-3
 - pop up, creation of, 4-9
- MenuData interface
 - ArrayHandle, 4-8
 - CreateItem (Def), 4-1
 - CreateItem (Ex), 4-8
 - CreateMenu (Def), 4-8
 - CreateMenu (Ex), 4-8, 4-12
 - Item, 4-1
 - ItemHandle, 4-1
 - MenuProc, 4-2
- message files, C-1
 - creating, C-1
 - editing, C-3
 - printing, C-7
 - property sheet, C-3
 - runtime, C-8
 - searching, C-4
- message keys, 3-11
- Message Master Editor, C-3
- Message Master File Creation Tool, C-1
- MessageFileTool.bcd, C-3
- messages
 - definitions module, 3-11
- MOUSE, 9-4
- MOVE
 - implementing, 15-6

- N
- normal TIP tables, 9-2
- Notifier, 9-1, 9-13
 - and selection, 14-3
- NotifyProc, 9-1, 9-6
 - (Ex), 9-12
- NSFile interface
 - AccessProblem, 11-13
 - Attribute, 10-2
 - AttributeList, 10-2, 10-7
 - Attributes, 10-7
 - AttributesProc, 11-11
 - AttributesRecord, 10-7
 - AttributeType, 10-6
 - BooleanFalseDefault, 10-6
 - ChangeAttributes (Def), 10-5
 - ChangeAttributes (Ex), 10-8
 - ClearAttributes, 16-8
 - Close (Def), 11-7
 - Close (Ex), 11-3
 - Create, 11-7
 - DecodeCardinal (Def), 10-4
 - DecodeCardinal (Ex), 10-10
 - DecodeString, 10-10
 - Delete (Def), 11-9
 - Delete (Ex), 13-11
 - EncodeCardinal, 10-4
 - EncodeString, 10-5
 - Error (Def), 11-13
 - Error (Ex), 13-14
 - ErrorRecord, 11-13
 - ErrorRecord (Ex), 11-14, 13-14
 - ExtendedAttributeType, 10-4
 - ExtendedSelections, 10-10
 - Filter, 11-9
 - Find (Def), 11-9
 - Find (Ex), 11-11, 16-6
 - FreeAttributes, 10-8
 - GetAttributes (Def), 10-5
 - GetAttributes (Ex), 10-8, 10-10, 13-4, 16-8
 - Handle, 11-2
 - InterpretedSelections, 10-6
 - List, 11-11
 - MakeReference, 11-6
 - nullSession, 10-8
 - Open (Def), 11-2
 - Open (Ex), 11-3, 11-6, 13-15
 - OpenByName, 11-4
 - OpenByReference (Def), 11-3
 - OpenByReference (Ex), 11-11, 11-12, 13-4

- Reference, 11-1
- Scope (Def), 11-9
- Scope (Ex), 11-12
- Selections, 10-5
- Service, 11-1, 11-5
- ServiceRecord, 11-1, 11-5
- SystemElement, 11-5
- Words, 10-4
- NSFileStream interface, 12-1
 - Create (Def), 12-2
 - Create (Ex), 12-3, 13-16
 - FileFromStream, 12-8
 - GetLength, 12-7
 - Handle, 12-2
 - SetLength, 12-7
- NSFiling, 10-1
- NSName interface
 - NameRecord, 11-5
- NSSegment interface
 - CopyIn, 13-8
 - CopyOut, 13-8
 - GetSizeInPages, 13-9, 13-12
 - Map (Def), 13-2
 - Map (Ex), 13-4, 13-7, 13-9, 13-12
 - Origin, 13-2
 - SetSizeInPages, 13-9
- O**
- opening files, 11-2
 - example, 11-3
- option files, 16-2
- OptionFile interface
 - GetStringValue (Def), 16-4
 - GetStringValue (Ex), 16-6, 16-8
 - GetWorkstationProfile, 16-6
- overlapping windows, 2-3
- P**
- packaging applications, 16-1
- page, 13-1
 - dirty, 13-2
- page format characters, 17-5
- page properties, 17-9
- path name, 11-1
- peak
 - of a curve, 18-8
- periodic notifiers, 9-13
 - kamikaze, 9-14
- pop-up menu, 2-3
 - creation of, 4-9
- properties
 - getting and setting, 17-10
 - of documents, 17-8
 - setting for empty paragraph, 17-10
- property sheets, 2-5, 8-1
 - creating, 8-3
 - linked, 8-5
- PropertySheet interface
 - BooleanFalseDefault, 8-2
 - Create (Def), 8-1
 - Create (Ex), 8-4
 - CreateLinked, 8-5
 - MenuItem, 8-2
 - MenuItemProc, 8-3
 - MenuItemType, 8-2
 - optionSheetDefaultMenu, 8-2
 - propertySheetDefaultMenu, 8-2
 - SwapExistingFormWindows, 8-7
 - SwapFormWindows, 8-6
- Prototype interface
 - Create, 15-11
 - Find, 15-11
- prototype catalog, 11-4
- Prototype folder, 2-6
- R**
- readers, 3-2
 - accessing contents of, 3-4
 - creating, 3-4
 - vs. readerBodies, 3-4
- rectangles
 - adding to graphics frame, 18-5
- reference (to file), 11-1
- remote files, 11-5
- requestors (selection), 14-1
- results (in TIP table), 9-4
- Runtime File Creation Tool, C-8
- Runtime Message Files, C-8
- RuntimeFileCreate.bcd, C-8
- S**
- segments (of files), 13-3
- Selection interface
 - CanYouConvert (Def), 14-5
 - CanYouConvert (Ex), 14-6, 14-7
 - Convert (Def), 14-1
 - Convert (Ex), 14-2, 14-4
 - Copy 14-3
 - CopyMove, 14-3
 - CopyOrMove, 14-3
 - Difficulty, 14-5
 - Enumerate, 14-6
 - EnumerationProc, 14-6
 - Free, 14-3, 14-5
 - HowHard, 14-5
 - maxStringLength, 14-6
 - Move, 14-3
 - nullValue, 14-1
 - RequestorData, 14-6
 - Target, 14-1
 - UniqueTarget, 14-2
 - Value, 14-1
 - ValueCopyMoveProc, 14-3
 - ValueHandle, 14-3
- selection
 - copying (Ex), 14-4
 - copying streams, 14-5
 - defining new target types for, 14-2
 - enumerating, 14-6
 - example, 14-2
 - is conversion possible?, 14-5
 - managers, 14-1
 - monitoring, 14-3
 - Notifier and, 14-3
 - obtaining, 14-1
 - requestors, 14-1
 - target types for, 14-2
 - session, 10-8

- SimpleTextDisplay interface
 - StringIntoWindow, 6-4
 - SimpleTextFont interface
 - AddClientDefinedCharacter (Def), 15-5
 - AddClientDefinedCharacter (Ex), 15-6, 15-13
 - SIZE (Ex), 12-6, 13-7
 - size attributes, 10-2
 - Space interface
 - Error, 13-7, 13-12
 - ForceOut, 13-6
 - Interval, 13-2
 - ScratchMap, 15-4
 - Unmap (Def), 13-5
 - Unmap (Ex), 13-6, 13-7, 13-12
 - StarDesktop interface
 - AddReferenceToDesktop, 17-7
 - GetCurrentDesktop, 14-4
 - StarWindowShell interface
 - AddPopupMenu, 4-9
 - AdjustProc, 6-11
 - Create (Def), 4-3
 - Create (Ex), 4-4, 4-6, 4-11
 - CreateBody (Def), 4-5
 - CreateBody (Ex), 4-6, 4-11
 - GetAdjustProc, 6-11
 - GetLimitProc, 6-11
 - GetZone, 4-8, 4-11
 - Handle, 4-3
 - IsCloseLegalProc, 4-4
 - LimitProc, 6-10
 - Pop, 4-10
 - Push, 4-12
 - SetAdjustProc, 6-11
 - SetLimitProc, 6-11
 - SetRegularCommands (Def), 4-8
 - SetRegularCommands (Ex), 4-8, 4-12
 - StandardLimitProc, 6-11
 - State, 4-4
 - TransitionProc, 4-4
 - When, 6-11
 - StarWindowShell
 - adding commands to (Ex), 4-8
 - creating, 4-2
 - creation of (Ex), 4-11
 - Stimulus, 9-1
 - Stream interface, 12-1
 - Block, 12-4
 - Delete (Def), 12-8
 - Delete (Ex), 12-3, 13-16
 - EndOfStream, 12-3
 - GetBlock (Def), 12-4
 - GetBlock (Ex), 12-6
 - GetByte, 12-3
 - GetPosition, 12-7
 - GetWord, 12-3
 - Handle, 12-2
 - Object, 12-2
 - Position, 12-7
 - PutBlock (Def), 12-4
 - PutBlock (Ex), 12-6, 13-16
 - PutByte, 12-2
 - PutWord, 12-3
 - SetInputOptions, 12-6
 - SetPosition, 12-7
 - streams, 12-1
 - as target of selection, 14-5
 - block example, 12-6
 - counting bytes of, 12-7
 - creating, 12-2
 - deleting, 12-8
 - example of end of stream, 12-3
 - IO example, 12-3
 - miscellaneous operations on, 12-7
 - random access, 12-7
 - strings, 3-1
 - swap unit, 13-1
 - swapping, 13-1
 - syntax
 - for ADF, 16-2
 - of TIP tables, 9-3
 - system catalog, 11-4
 - system directory, A-2
 - SystemFolder, A-2
 - SystemFolder application, 2-7
- T**
- tagonly item
 - in form window, 7-2
 - TakeSelection, 15-7
 - TakeSelectionCopy, 15-7
 - temporary file, 11-7
 - Terminal Interface Package
 - see TIP, 9-1
 - text frames
 - adding contents to, 18-10
 - creating, 18-10
 - reading contents of, 18-14
 - releasing storage, 18-11
 - text item
 - creating, 7-4
 - in form window, 7-2
 - tiled windows, 2-3
 - timeout
 - for file access, 11-2
 - TIP** interface
 - AttentionProc, 9-15
 - CreatePeriodicNotify, 9-14
 - CreateTable (Def), 9-7
 - CreateTable (Ex), 9-8, 9-12
 - InvalidTable (Def), 9-8
 - InvalidTable (Ex), 9-12
 - LosingFocusProc, 9-10
 - NotifyProc (Def), 9-6
 - NotifyProc (Ex), 9-6, 9-12
 - PeriodicNotify, 9-14
 - ResetUserAbort, 9-15
 - ResultObject, 9-4
 - Results, 9-4
 - SetAttention, 9-15
 - SetInputFocus, 9-10
 - SetTableAndNotifyProc (Def), 9-9
 - SetTableAndNotifyProc (Ex), 9-12
 - Table, 9-7
 - TableError, 9-8
 - TableObject, 9-7
 - UserAbort, 9-15
 - TIP files
 - in application folder, 16-7

TIP tables, 9-1
 associating with windows, 9-9
 creating (Ex), 9-8
 example of, 9-4, 9-11
 incorporating, 9-7
 normal, 9-2
 NotifyProcs and, 9-9
 results lists, 9-4
 syntax of, 9-3

TIPC files, 9-8

TIPStar interface
 NormalTable, 9-9
 Placeholder, 9-2
 PopTable, 9-10
 PushTable, 9-9

trigger action, 9-3

type checking
 circumventing, 12-5

U

uninterpreted attributes, 10-1
 encoding and decoding, 10-4
 retrieving, 10-10
 specifying, 10-4

user abort, 9-15

user actions, 9-1

User Profile
 window entries for, 2-3

V

virtual memory overview, 13-1

W

window, 2-2
 adding commands to, 4-6
 creating, 4-2

Window interface
 Box, 4-6, 6-2, 7-8
 Clarity, 6-2
 Dims, 4-6, 6-2, 7-8
 DisplayProc, 6-3
 EnumerateInvalidBoxes, 6-3
 InvalidateBox (Def), 6-2
 InvalidateBox (Ex), 6-2, 6-8
 Place, 4-6, 6-2, 7-8
 Validate (Def), 6-2
 Validate (Ex), 6-2, 6-8
 ValidateTree, 6-2

windows
 changing defaults for, 2-3
 clipping, 6-3
 controlling size of, 6-10
 creation of (Ex), 4-11
 displaying information in, 6-1
 displaying on screen, 4-10
 overlapping mode, 2-3
 storing data with, 5-1
 tiled mode, 2-3
 validating and invalidating, 6-1

WorkstationProfile, A-2

writers, 3-2
 allocating, 3-5
 editing, 3-6
 expanding, 3-6

X

XChar, 3-1

Xerox Character Code Standard, 3-1

XFormat interface, 3-6
 Char, 3-9
 Character, 3-1
 CharRep, 3-1
 ClientData, 3-7
 Decimal, 3-9
 example, 3-9
 format procedures, 3-8
 FormatObject, 3-8
 FormatProc, 3-7
 Handle, 3-7
 Object (Def), 3-7
 Object (Ex), 3-9
 Reader, 3-9
 StreamObject, 3-8
 StreamProc, 3-8
 String, 3-9
 WriterObject, 3-9

XMessage interface
 AllocateMessages (Def), 3-11
 AllocateMessages (Ex), 3-12, 3-13
 client module, 3-14
 definitions module, 3-11
 FreeMsgDomainsStorage, 16-4
 Get, 3-13
 Handle, 3-11
 implementation module, 3-11
 Messages, 3-12
 MessagesFromFile, 16-4
 MessagesFromReference, 16-4
 MsgDomain, 16-4
 MsgDomains, 16-4
 MsgEntry, 3-12
 MsgKey, 3-12
 Object, 3-11
 RegisterMessages, 3-12

XString interface
 AppendReader, 16-8
 Byte, 3-2
 Bytes, 3-5
 ByteSequence, 3-2
 Context, 3-2
 CopyReader, 14-4
 ExpandWriter, 3-6
 First, 3-4
 FromBlock, 13-12
 FromNSString, 3-4
 FromSTRING, 3-4
 InvalidNumber, 13-12
 Lop, 3-4
 NewWriterBody, 3-5
 NthCharacter, 3-4
 Overflow, 13-12
 Reader, 3-2
 ReaderBody, 3-2
 ReaderFromWriter, 3-4
 ReadOnlyBytes, 3-2
 Writer, 3-5
 WriterBody, 3-5, 3-6
 WriterBodyFromNSString, 3-6