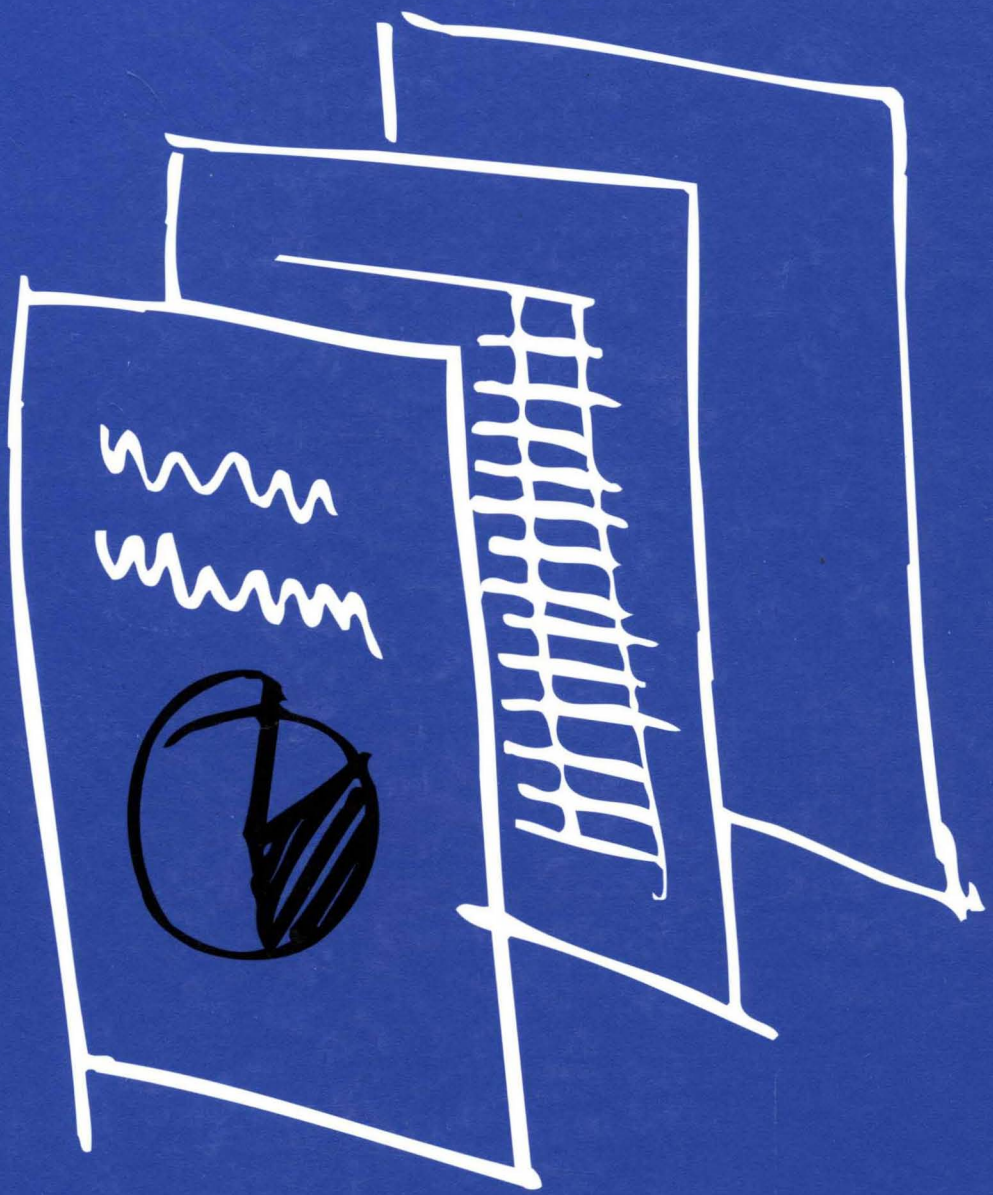
GO Corporation

GO Technical Library

PenPoint™ Application Writing Guide



PenPoint

PenPoint™

PenPoint™ Application Writing Guide

GO CORPORATION

GO TECHNICAL LIBRARY

. . . . . .

PenPoint Application Writing Guide provides a tutorial on writing PenPoint applications, including many coding samples. This is the first book you should read as a beginning PenPoint applications developer.

PenPoint Architectural Reference Volume I presents the concepts of the fundamental PenPoint classes. Read this book when you need to understand the fundamental PenPoint subsystems, such as the class manager, application framework, windows and graphics, and so on.

PenPoint Architectural Reference Volume II presents the concepts of the supplemental PenPoint classes. You should read this book when you need to understand the supplemental PenPoint subsystems, such as the text subsystem, the file system, connectivity, and so on.

PenPoint API Reference Volume I provides a complete reference to the fundamental PenPoint classes, messages, and data structures.

PenPoint API Reference Volume II provides a complete reference to the supplemental PenPoint classes, messages, and data structures.

PenPoint User Interface Design Reference describes the elements of the PenPoint Notebook User Interface, sets standards for using those elements, and describes how PenPoint uses the elements. Read this book before designing your application's user interface.

PenPoint Development Tools describes the environment for developing, debugging, and testing PenPoint applications. You need this book when you start to implement and test your first PenPoint application.

PenPoint™

PenPoint™ Application Writing Guide

GO CORPORATION

GO TECHNICAL LIBRARY

Addison-Wesley Publishing Company

Reading, Massachusetts ◆ Menlo Park, California ◆ New York
Don Mills, Ontario ◆ Wokingham, England ◆ Amsterdam
Bonn ◆ Sydney ◆ Singapore ◆ Tokyo ◆ Madrid ◆ San Juan
Paris ◆ Seoul ◆ Milan ◆ Mexico City ◆ Taipei

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial capital letters.

The authors and publishers have taken care in preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The following are trademarks of GO Corporation: GO, the PenPoint logo, the GO logo, ImagePoint, PenPoint, GrafPaper, TableServer, BaseStation, EDA, MiniNote, MiniText, and DrawingPaper.

Words are checked against the 77,000 word Proximity/Merriam-Webster Linguibase, ©1983 Merriam Webster. ©1983. All rights reserved, Proximity Technology, Inc. The spelling portion of this product is based on spelling and thesaurus technology from Franklin Electronic publishers. All other products or services mentioned in this document are identified by the trademarks or service marks of their respective companies or organizations.

Warranty Disclaimer and Limitation of Liability

GO CORPORATION MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT, REGARDING PENPOINT SOFTWARE OR ANYTHING ELSE. **GO Corporation does not warrant, guarantee, or make any representations regarding the use or the results of the use of the PenPoint software, other products, or documentation in terms of its correctness, accuracy, reliability, currentness, or otherwise. The entire risk as to the results and performance of the PenPoint software and documentation is assumed by you. The exclusion of implied warranties is not permitted by some states. The above exclusion may not apply to you.**

In no event will GO Corporation, its directors, officers, employees, or agents be liable to you for any consequential, incidental, or indirect damages (including damages for loss of business profits, business interruption, loss of business information, cost of procurement of substitute goods or technology, and the like) arising out of the use or inability to use the documentation or defects therein even if GO Corporation has been advised of the possibility of such damages, whether under theory of contract, tort (including negligence), products liability, or otherwise. Because some states do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitations may not apply to you. GO Corporation's total liability to you from any cause whatsoever, and regardless of the form of the action (whether in contract, tort [including negligence], product liability or otherwise), will be limited to $50.

U.S. Government Restricted Rights

The PenPoint documentation is provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in FAR 52.227–19 (Commercial Computer Software—Restricted Rights) and DFAR 252.227–7013 (c) (1) (ii) (Rights in Technical Data and Computer Software), as applicable. Manufacturer is GO Corporation, 919 East Hillsdale Boulevard, Suite 400, Foster City, CA 94404.

ISBN 0–201–60857–X

123456789–AL–9695949392

First Printing, March 1992

PENPOINT APPLICATION WRITING GUIDE
CONTENTS

Chapter 1 / Introduction

The PenPoint™ operating system is an object-oriented, multitasking operating system that is optimized for pen-based computing. Writing applications for the PenPoint operating system will present you with some new challenges. However, PenPoint contains many features that make application development far easier than development in most other environments.

One feature that makes application development easier is the PenPoint Application Framework, which eliminates the need to write "boilerplate" code. In other operating systems, programmers must write code to perform housekeeping functions, such as application installation, input and output file handling, and so on. These are provided automatically by the PenPoint Application Framework.

As another example, PenPoint provides most of the on-screen objects used by the PenPoint Notebook User Interface (NUI). By using these objects, your application can conform to the PenPoint NUI, without a great amount of work on your part.

In this manual, you will learn about the PenPoint operating system, the PenPoint development environment, and, of course, how to write applications for the PenPoint operating system. The PenPoint Software Development Kit (SDK) contains several sample applications that you can compile and run. These sample applications are used throughout this manual to demonstrate concepts and programming techniques.

Intended Audience 1.1

This manual is intended for programmers who want to write applications for the PenPoint operating system. It assumes that you are familiar with the C programming language and related development tools, such as make utilties.

You should also be aware of the information in the companion volume, *PenPoint Development Tools*. Pay particular attention to Chapter 2, Roadmap to SDK Documentation, which describes the organization of the PenPoint SDK documentation and recommends a path through the manuals.

Organization of This Manual 1.2

Chapter 1, this chapter, introduces the organization of this manual.

Chapter 2, System Overview, presents an overall look at the PenPoint Operating System environment. This chapter does not focus on writing applications.

Chapter 3, Application Concepts, presents applications from a conceptual point of view. This chapter describes most of the parts of an application that you must

write, along with parts of applications that are provided for you by the PenPoint Application Framework.

Chapter 4, PenPoint Class Manager, describes the parts of the PenPoint Operating System that you use to create classes and to create instances of classes.

Chapter 5, Developing an Application, discusses a number of points that you must consider when creating an application. This chapter presents a checklist that you can use to ensure that your application is complete; the chapter also and discusses GO's coding conventions.

Chapter 6, A Simple Application (Empty App), introduces a minimal application. By experimenting with this application, you can see just how much the PenPoint Application Framework does for you. This chapter also describes how to compile and debug PenPoint applications.

Chapter 7, Creating Objects (Hello World: Toolkit), describes how you create instances of predefined PenPoint classes and use these objects in your application.

Chapter 8, Creating a New Class (Hello World: Custom Window), describes how to create custom windows and presents additional information about using instance data.

Chapter 9, Saving and Restoring Data (Counter App), describes how to save and restore data from your application.

Chapter 10, Handling Input (Tic-Tac-Toe), describes how to handle input, while also describing a much larger application: Tic-Tac-Toe.

Chapter 11, Refining the Application (Tic-Tac-Toe), describes how to add polish to your application.

Chapter 12, Releasing the Application, describes the steps necessary to make your application available to other PenPoint users.

The appendix contains the complete sample code for the tutorial programs referred to in the previous chapters, along with descriptions of other sample programs provided in the SDK.

Other Sources of Information 1.3

For conceptual information about the various classes in PenPoint, see the *PenPoint Architectural Reference.*

For information on running PenPoint on a PC and using the PenPoint development tools and utilities, such as the PenPoint source-level debugger, see *PenPoint Development Tools.* The volume also contains a master index for all SDK volumes, except the *PenPoint API Reference.*

For reference information on the classes, messages, macros, functions, and structures defined by PenPoint, see the *PenPoint API Reference.* The information in the *PenPoint API Reference* is derived directly from the PenPoint header files (in \PENPOINT\SDK\INC).

Chapter 2 / PenPoint System Overview

When GO Corporation undertook to build a mobile, pen-based computer system, we quickly recognized that existing standard operating systems were not adequate for the task. Those systems, designed for the very different needs of keyboard-based desktop computers, would require such extensive rewriting to support this new market that they would no longer run the installed base of applications that made them standard in the first place. We therefore determined that a new, general-purpose operating system would be needed, designed specifically for the unique requirements of pen-based computing. The result is the PenPoint operating system. This document is a brief introduction and overview of its design goals, architecture, and functionality.

Design Considerations 2.1

After extensive research and analysis, GO identified the following key requirements for pen-based system software:

- A direct, natural, intuitive, and flexible graphical user interface
- Strong support for handwriting recognition and gesture based commands
- A richer organizational metaphor than the traditional file-system model
- A high degree of memory conservation through extensive sharing of code, data, and resources
- Ability to run on RAM-only as well as disk-based computers
- Priority-based, preemptive multitasking
- Detachable networking and deferred data transfer
- Hardware independence (ability to move to new processors quickly)

The PenPoint operating system was developed to satisfy these requirements.

User Interface 2.2

PenPoint's most distinctive feature is its innovative user interface. The user interface is the cornerstone on which the entire system is built; all other design considerations follow from it. The user interface, in turn, is based on two main organizing principles:

- The use of a pen as the primary input device
- A notebook metaphor that is natural and easy to use

The consequences of these two basic design features permeate the entire system.

The Pen 2.2.1

The pen naturally combines three distinct system control functions: pointing, data input, and command invocation. Like a mouse, it can point anywhere on the screen to designate an operand, specify a location, draw a picture, drag an object or select from a menu. Through sophisticated handwriting recognition software, it can replace the keyboard as a source of text input. Finally, it can do something neither a mouse nor a keyboard can do: issue commands through graphical gestures.

Gestures 2.2.1.1

Gestures are simple shapes or figures that the user draws directly on the screen to invoke an action or command. For example, a scratch out ✗ gesture is used to delete, a circle ○ to edit, and a caret ∧ to insert. A set of built-in core gestures form the heart of the PenPoint user interface:

Caret ∧	Check ✓
Circle ○	Cross out ✗
Flick left —	Flick right —
Flick up |	Flick down |
Insert space ⌐	Pigtail ʔ
Press !	Tap !
Tap press ·!	Undo ⊗

To exploit the unique properties of the pen, PenPoint provides strong support for gestural command invocation. The same handwriting translation subsystem that recognizes characters for text input also recognizes those shapes that constitute meaningful gestures. The form, location, and context of the gesture then determine the action to be performed and the data objects affected. Because a gesture can be made directly over the target object, it can specify both the operand and the operation in a single act. This gives the pen-based interface a directness and simplicity that cannot be achieved with a mouse.

"PenPoint" Control 2.2.1.2

The pen has one more notable property as a control device. Because it draws directly on the face of the screen (rather than on a physically separate working surface such as a mouse pad or graphics tablet), it eliminates a major source of difficulty among new computer users—the relationship between movement of the and the movement of the cursor on the screen. With a pen, the user's eye is focused exactly where his or her hand is working. Most PenPoint applications can thus dispense with an on-screen cursor for tracking the pen, though it can still be offered as an optional user preference.

Notebook Metaphor 2.2.2

Instead of a traditional file system based on a hierarchy of nested directories and cryptic file names, PenPoint uses a "notebook" metaphor for information storage and retrieval. By using familiar models of working with paper-based documents, the notebook approach provides a rich variety of natural and intuitive techniques for organizing and accessing information:

- A bookshelf upon which multiple user notebooks may reside, as well as system notebooks for help information and stationery, an inbox and outbox, and various tools and accessories. A user can have any number of notebooks open at once; typical use involves one main notebook.
- A table of contents offering an overview of all available documents in the notebook, allowing easy manipulation and navigation at the global level. The table of contents can be organized in natural page number order, or sorted by name, size, type or date.
- Sections and subsections for hierarchical organization
- Page numbers and notebook tabs for direct random access
- Page turning for sequential access

Because the notebook is a familiar, physical, and stable model, a user can employ spatial memory of layout and juxtaposition to help find and organize their information.

Object-Oriented Architecture 2.3

To facilitate code sharing and overall memory conservation, PenPoint uses an object-oriented approach to system architecture. All application programming interfaces (APIs) above the kernel layer are implemented using object-oriented programming techniques of subclass inheritance and message passing. This helps to ensure that PenPoint and its APIs have these characteristics:

- Compact, providing a body of shared code that need not be duplicated by all applications
- Consistent, since all applications share the same implementation of common system and user interface functions
- Flexible, allowing applications to modify PenPoint's behavior by subclassing its built-in classes

The event-driven, object-oriented nature of the system minimizes the need to "reinvent the wheel" with each new application. Programmers can "code by exception," reusing existing code while altering or adding only the specific behavior and functionality that their own applications require. Because the object-oriented architecture is system-wide, these benefits are not restricted to single applications; in fact, applications can share code with each other just as readily as with the system itself.

Architecture and Functionality 2.4

PenPoint's overall software architecture is organized into five layers:

1. The kernel, which provides multitasking process support, memory management, and access to hardware. The kernel works closely with the PenPoint class manager, which makes PenPoint object oriented.
2. The system layer, which provides windowing, graphics, and user interface support in addition to common operating system services such as filing and networking
3. The component layer, which consists of general-purpose subsystems offering significant functionality that can be shared among applications
4. The Application Framework, which serves as a "head start" for building applications
5. The applications themselves

Each of these layers is discussed in detail below.

Kernel Layer 2.5

The kernel is the portion of the PenPoint operating system that interacts directly with the hardware. Besides handling such low-level tasks as process scheduling and synchronization, dynamic memory allocation, and resource management, it also provides these services, which are needed to support the object-oriented software architecture:

- Priority-based, preemptive multitasking
- Processes and threads (lightweight tasks sharing the same address space)
- Interprocess communication and semaphores
- Task-based interrupt handling
- 32-bit flat memory model
- Protected memory management and code execution
- Heap memory allocation with transparent relocation and compaction (no fixed-length buffers)
- Object-oriented message passing and subclass inheritance

All hardware dependencies in the kernel are isolated into a library subset (the MIL or machine interface layer) to facilitate porting to a wide variety of hardware and processor architectures. The kernel runs on both PC and pen-based machines. All of PenPoint's APIs use full 32-bit addresses.

Other parts of the kernel layer support features that keep PenPoint small and efficient, such as:

Loader Unlike a traditional, disk-based operating system, PenPoint's loader does not require multiple copies of system and application code to be present in the machine at the same time. Instead, it maintains a single

instance of all code and resources, which are shared among all clients. When installing a new application, the loader reads in only those components that are not already present in memory.

Power Conservation When running on battery-powered hardware, the kernel reduces power consumption by shutting down the CPU whenever there are no tasks awaiting processor time. Subsequent events such as pen activity or clock-chip alarms generate interrupts that reactivate the CPU. The kernel also monitors the main battery and will refuse to run if power is too low, ensuring reliable protection of user data.

Diskless Reboot Hardware memory protection is used to preserve the integrity of all code, resources, and data. In the event of a crash, the kernel can restart all processes (including its own), with minimal loss of user data and without the need to restart the system from disk drives.

C Runtime The kernel includes runtime support for all WATCOM C/386 runtime functions except those DOS- and PC-specific calls that are not applicable to a pen-based notebook environment.

Class Manager PenPoint's Class Manager works closely with the kernel to support object-oriented programming techniques such as single-inheritance subclassing and message passing. The Class Manager also provides important protection and multitasking services not found in C++ or other object-oriented languages. These services safeguard the operating system against possible corruption arising from the use of object-oriented techniques. For example, instance data for system-defined classes is protected so that the data cannot be altered by any subclasses. Applications thus derive the benefits of subclassing without jeopardizing the integrity of the system.

System Layer 2.6

PenPoint's system layer provides a broader range of support services than a traditional operating system. In addition to the usual system facilities such as filing and networking, it also provides such high-level services as windowing, graphics, printing, and user interface support. This helps keep application code compact and consistent while facilitating application development for the machine.

File System 2.6.1

PenPoint's file system is designed for compatibility with other existing file systems, particularly MS-DOS, and includes full support for reading and writing MS-DOS-formatted disks. It provides many of the standard features of traditional file systems, including hierarchical directories, file handles, paths, and current working directories, as well as such extended features as 32-character file names, memory-mapped files, object-oriented APIs, and general, client-specified attributes for files and directories.

The PenPoint file system is a strict superset of the MS-DOS file system; all PenPoint-specific information is stored as an MS-DOS file within each MS-DOS

directory. This approach is used when mapping to other file systems as well. Additional, installable volume types are also supported. At present, these include RAM volumes and remote volumes (for access to PC, Macintosh, and file servers).

Resource Manager 2.6.2

PenPoint's Resource Manager and the resource files that it controls allow applications to separate data from code in a clean, structured way. The Resource Manager can store and retrieve both standard PenPoint objects and application-defined data, in either a specific file or a list of files. Resources can be created directly by the application or by compiling a separate, text-based resource definition file.

Networking 2.6.3

PenPoint provides native support for smooth connectivity to other computers and networks. Multiple, "autoconfiguring" network protocol stacks can be installed on the fly. AppleTalk protocol is built in, enabling connection to other networks through a variety of AppleTalk-compatible gateways. By purchasing the appropriate TOPS software, users can configure their systems to connect directly to DOS or Macintosh computers.

Through the use of these networking facilities, remote services such as printers are as easily accessible to PenPoint applications as if they were directly connected. Remote file systems on desktop computers and network file servers are also transparently available via a remote-file-system volume. A user can browse PC and file-server directories, for instance, using PenPoint's connections notebook. Several remote volumes can be installed at once: for example, a PenPoint system can hook directly to a Macintosh and a DOS computer at the same time.

A typical user, while on an airplane, might mark up a FAX, fill out an expense report to be electronically mailed to the payables department, draft a business letter to be printed, and edit an existing document to be returned to its "home" on a PC's hard disk. Upon connection to the physical devices, conventional operating systems would require that user to run each application, load each document and dispense with it. PenPoint's In Box and Out Box services allow the user to defer and batch data transfer operations for completion at a later time. Upon return to the office and establishing the physical connection, the documents are automatically faxed, printed, and mailed. These services are extensible and can support a wide variety of transfer operations, including electronic mail, print jobs, fax transmissions, and file transfers.

Windowing 2.6.4

The window system supports nested hierarchies of windows with multiple coordinate systems, clipping, and protection. Windows are integrated with PenPoint's input system, so that incoming pen events are automatically directed to the correct process and window. Windows use little memory and can therefore be used freely by applications to construct their user interface.

Usually windows appear on screen, but they can also be created on other, off-screen image devices, such as printers.

The window system maintains a global, screen-wide display plane called the acetate plane, which is where ink from the pen is normally "dribbled" by the pen-tracking software as the user writes on the screen. The acetate plane greatly improves the system's visual responsiveness, both in displaying and in erasing pen marks on the screen.

Graphics 2.6.5

PenPoint's built-in graphics facility, the ImagePoint™ imaging model, unifies text with other graphics primitives in a single, PostScript-like imaging model. ImagePoint™ graphics can be arbitrarily scaled, rotated, and translated, and can be used for both screen display and printing. ImagePoint's graphics capabilities include these elements:

Polylines	Bezier curves
Rectangles	Ellipses
Rounded rectangles	Arcs
Polygons	Sectors
Sampled images	Chords
Text	

A picture segment facility allows ImagePoint messages to be stored and played back on demand, facilitating a variety of drawing and imaging applications. For improved performance, the imaging system dynamically creates machine code when appropriate for low-level graphics operations such as direct pixel transfer ("bitblt"). The ImagePoint interface also supports the use of color, (specified in conventional RGB values) allowing PenPoint to run on grey-scale and color screens.

To conserve memory, ImagePoint™ uses outline fonts to render text at any point size. (Bitmap fonts are automatically substituted at low resolutions for improved visual clarity.) Fonts are heavily compressed and some character styles are synthesized to minimize memory requirements. If a requested font is not present, ImagePoint will find the closest available match. Text characters can be scaled and rotated in the same way as other graphical entities.

Printing 2.6.6

The ImagePoint imaging model is used for printing as well as screen display, allowing applications to use the same image-rendering code for both purposes, rebinding it to either a screen window or a printer as the occasion demands. PenPoint handles all printer configuration, and automatically controls margins, headers, and footers, relieving the application of these details. (As in most other areas of PenPoint, applications can override the default behavior.)

One key benefit of this approach is that documents to be faxed are rendered specifically for a 200 DPI output device. The resulting output will be of sufficiently high quality that mobile users may not require a portable printer at all, opting instead to use a nearby plain paper FAX machine.

PenPoint supports dot-matrix and HP Laserjet printers. When the printer does not have a requested font, the ImagePoint imaging model will render and download one from its own set of outline fonts, ensuring good WYSIWYG correspondence and shielding the user from the complexities of font management.

User Interface Toolkit 2.6.7

PenPoint's User Interface Toolkit offers a wide variety of on-screen controls:

- Menu bars
- Pulldown menus
- Section tabs
- Window frames
- Title bars
- Scroll bars
- Option sheets
- Dialog boxes
- Progress bar
- Modal alerts
- Trackers
- Nonmodal alerts
- Pushbuttons
- Exclusive choice buttons
- Nonexclusive choice buttons
- Pop-up choice lists
- List boxes
- Editable text fields
- Handwriting pads
- Grabbers
- Busy clock

A major innovation in PenPoint's User Interface Toolkit is automatic layout. Instead of specifying the exact position and size of controls, the application need only supply a set of constraints on their relative positions, and the Toolkit will dynamically calculate their exact horizontal and vertical coordinates. This makes it easy for programmers or users to resize elements of the user interface, change their fonts or other visual characteristics, or switch between portrait and landscape screen orientations, while preserving the correct proportions and positional relationships.

Input and Handwriting Translation 2.6.8

PenPoint's input subsystem translates input events received by the hardware into messages directed to application objects. The low-level pen events include:

- In proximity
- Tip down
- Move down
- Window enter
- Out of proximity
- Tip up
- Move up
- Window exit

These low-level events can be grouped into higher-level aggregates called **scribbles**, which are then translated by the handwriting translation (HWX) subsystem into either text characters or command gestures. These characters or

gestures in turn are dispatched to the appropriate objects via a rich input distribution model that includes filtering, grabbing, inserting, and routing of input up and down the window hierarchy.

The portion of the GOWrite handwriting translation engine that matches and recognizes character shapes is replaceable, allowing PenPoint to improve its HWX techniques as better algorithms become available. There are two parts to the handwriting translation engine: the first part matches shapes, the second part uses context to improve the translation.

The current HWX engine, developed entirely by GO, recognizes hand-printed characters and has the following characteristics:

- Operates in real time (shape matcher operates at 60 characters per second on 33Mhz 80486)
- Runs in a background process
- Handles mixed upper- and lowercase letters, numerals, and punctuation
- Tolerates characters that overlap or touch
- Recognizes characters independently of stroke order, direction, and time order
- Distinguishes non-unique character forms such as the letter "O" and the numeral "0" (in context)
- Tolerates inconsistency by same user (that is, the user may shape the same character in different ways at different times)
- Can accept optional context-sensitive aids (such as word lists, dictionaries, and character templates) provided by an application. Applications are given great control over this process; they may issue constraints that merely influence the result or force a match against a predefined list.

Although PenPoint is designed primarily for pen-based input, it is not limited to the pen. For high volume data entry, PenPoint is designed to accept input from a keyboard.

As an alternative, PenPoint also provides a software "virtual keyboard." Users can display the keyboard on the screen and input text by tapping on the keys with the pen.

Selection Manager and Data Transfer 2.6.9

The Selection Manager subsystem maintains a system-wide selection, which is the target for all editing operations. The Selection Manager also implements a single-level stack for temporarily saving the current selection. Editing is based on a move-and-copy model, rather than a "clipboard" (cut-and-paste) model. The source and destination applications negotiate data transfers from one application to another. The destination application requests a list of available data formats from the source application. PenPoint supports a variety of standard transfer formats, including RTF (Rich Text Format), structured graphics, and TIFF

(Tagged Image File Format); applications can extend this list to include other formats as well.

PenPoint's object-oriented architecture also makes possible the PenPoint EDA™ or embedded document architecture. This is a unique form of "live" data transfer in which the transferred data carries with it an instance of its own source application. Through object-oriented message passing, this embedded application instance can then be used to display, edit, or otherwise manipulate the data from within the destination application. Although more conventional forms of "hot links" and DDE (Dynamic Data Exchange) linking are still possible in PenPoint, such live application embedding obviates the need for most of them.

Component Layer 2.7

Above and beyond the traditional kernel and system facilities, PenPoint adds a rich, powerful, and extensible component layer. Components are general-purpose code units with application-level functionality that can be shared and reused as building blocks by multiple client applications. They speed the development of applications, reduce memory consumption, and provide for more consistent user interfaces and tighter integration across diverse applications.

PenPoint includes several components, such as a multi-font, WYSIWYG text editor and a scribble editing window that can be embedded within any application that needs them. You can include these components in your application without licensing them from GO.

Third-party developers may market components to other developers. Applications may also provide their own general-purpose components to be installed and shared in the PenPoint runtime environment.

Application Framework Layer 2.8

The Application Framework is a set of protocols rigorously defining the structure and common behavior of a PenPoint application. Through the Application Framework, applications inherit a wide variety of standard behavior, including installation and configuration, creation of new documents, application stationery (template documents), on-line help, document properties, spell checking, search and replace, import/export file dialogs, and printing. New code is required only for added functionality or to modify or override specific aspects of the default behavior. Use of the Application Framework thus yields significant savings in programming time and code space.

An application developer creates the application code and any resources needed by the application. When a user installs an application, the PenPoint Application Framework takes care of:

- Copying the application code and all other auxiliary files to the system
- Creating new documents
- Creating and terminating tasks

- Storing and retrieving user data in the file system
- Creating and destroying a main window for the application.

Active documents save their internal state in the file system, but this is invisible to the user: there is no need to save or load the application's state explicitly from one session to the next.

The most innovative aspect of PenPoint's Application Framework is its ability to create true "live compound documents." Users and developers can freely embed a document created by one application inside another document (for example, a business graphics application within a text document). GO refers to this architecture as EDA™ (Embedded Document Architecture).

Application Layer 2.9

Using the "live" recursive embedding available through EDA, PenPoint's notebook metaphor and user interface are implemented as a set of bundled system applications. Although the user simply perceives these collectively as "the notebook," they are in fact distinct applications, providing a cleanly delineated and modular architecture.

The key bundled applications include bookshelf, notebook, and section applications that together constitute the core notebook metaphor.

- The Table of Contents (TOC) application provides a user interface for specialized organization and retrieval at the front of the notebook.
- A bundled text editor provides end users with intuitive, pen-based Rich Text editing.
- A standard Send user interface and an Address List allow for the addressing of all electronic mail, fax, and file transfers.
- A file browser allows the user to point to files and directories and use standard gesture commands to manipulate them.

Multiple instances of the notebook can be created; in fact, the Create, Help, Configuration, and InBox/OutBox applications are all instances of the notebook application. Developers benefit from this code sharing; users benefit from decreased memory requirements as well as greater consistency in the user interface. The Help Notebook, for example, consists of help documents ordered by section (application), and therefore looks just like the standard table of contents. Users already know how to navigate through this notebook and can even create hyperlink references to important sections. Developers can simply write ASCII text to provide on-line documentation. Documents in the help notebook can be any type of PenPoint application documents. Developers can also leverage existing application code to build very powerful help systems that can demonstrate real functionality.

Software Development Environment 2.10

With the exception of some hardware-dependent code, PenPoint and the applications it supports are written entirely in ANSI C, using current versions of leading PC-based development tools. Developers already acquainted with object-oriented concepts, and with the graphical user interfaces and multitasking found in operating systems like Macintosh and OS/2 Presentation Manager, will find the development environment familiar and will quickly be able to do productive work.

Software Development Kit 2.10.1

The PenPoint SDK provides developers with the documentation and tools to develop applications. The kit includes a source-code symbolic debugger, as well as an outline font editor for creating scalable and rotatable application-specific glyphs. Because PenPoint runs on DOS 386 machines, the full application edit-compile-debug cycle can be accomplished solely on a PC, or on a combination of a PC and a computer running PenPoint. In the former configuration, you use a pen-driven digitizer tablet to simulate pen input. In the latter configuration, the PC serves as a debugging monitor, as well as a convenient repository of the development system libraries, header files, on-line documentation, and source code.

Coding Conventions 2.10.2

All PenPoint code is written in accordance with modern software engineering standards, including:

- Consistent naming conventions for modules, functions, and variables
- Carefully designed modularity
- Proper commenting and formatting of source code.

Almost all of the C code is structured using object-oriented programming techniques. Classes are defined and objects are created and sent messages by making calls into a library of C routines called the Class Manager. These techniques are in the mainstream of currently evolving industry practices, but the details are unique to GO and are well documented in the SDK materials.

Extensibility 2.10.3

PenPoint is extensible in a variety of ways, allowing for the addition of new networking protocols, imaging models, font models, and file-system volumes. PenPoint can run on computer architectures ranging from RAM-only, pen-based pocket computers to powerful disk-based workstations with pen-tablet screens.

The operating system is a working whole, with most modules integrated and tested as part of the full system since early 1988. Through techniques such as hardware memory protection, object-oriented programming, rigorous

modularization, and extensive sharing of code, PenPoint has the foundations of a highly reliable operating system.

PenPoint Design Guidelines 2.11

To this point, this chapter has presented concepts that relate to the PenPoint operating system as a whole. The remainder of the chapter describes important points that application developers will have to keep in mind while designing and coding PenPoint applications.

Conserve Memory 2.11.1

Do not squander memory. Your application should use little memory when active. It must be able to further reduce its memory usage when off-screen. An application that is packed with functionality but consumes a lot of memory is less likely to be successful than one which meets the key needs while requiring very little memory.

Think Small 2.11.2

Most PC programs stand alone as large monolithic programs that attempt to do everything. In the cooperative, multitasking PenPoint environment with its Embedded Document Architecture, it makes more sense to provide programs that present a facet of functionality or that orchestrate other applications and components. Use existing classes and components where possible rather than writing your own from scratch.

Use a Modular Design 2.11.3

Consider writing your application as a set of separable components. A **component** is a separately loadable module (a dynamic link library or DLL) that provides software functionality. A component has a well-defined programmatic interface so that other software can reuse it or replace it. With modular design, your application becomes an organizing structure that ties together other components in a useful way. For example, an outliner application might use a drawing component, a charting component, and a table entry component; you could license these components to or from other developers. GO is working to develop a market for third-party components, and itself offers several components, including Text View™ and the TableServer™.

Avoid Duplicating Data 2.11.4

In some PenPoint memory configurations (single-tier, RAM volume), everything is in memory. The RAM volume co-exists with running applications in the same memory. Usually, computers running PenPoint will not be attached to external media. You should be aware of the occasions when data in your application's memory space needlessly duplicates data or code that is also present in the file system. One way to avoid duplication is to use memory-mapped files for your application's data.

When designing for a memory-resident file system, many of the trade-offs appropriate to traditional software design no longer apply. For example, deciding to read a startup file "into memory" makes sense when memory access is several orders of magnitude faster than file access, but in the case of single-tier memory configurations, the file system *is* memory.

Your Application Must Recover 2.11.5

Users may go for weeks or months without backing up their PenPoint computer's file system. If your application goes wrong, the PenPoint operating system will try to halt your application rather than the entire computer, but it is your responsibility to ensure that a new invocation of your application will be able to recover cleanly using whatever information it finds in the file system. This precept sometimes conflicts with avoiding data duplication, because the memory file system is more bullet-proof than the address space of a running application. For this reason, filed state will usually survive a process crash.

Moreover, most users will not have the PenPoint computer boot disks on hand. That means you cannot rely on the user being able to press the reset switch in a jam. PenPoint uses hardware and software protection techniques to secure against applications unintentionally corrupting the kernel and/or file system, but it is not foolproof.

Take Advantage of Object-Oriented Programming 2.11.6

You don't get to vote on using object-oriented techniques. You must write a **class** for your application that **inherits** from **clsApp**. The windows your application displays on the screen must be **instances** of **clsWin** (or instances of a class that inherits from **clsWin**). Of course, there are tremendous payoffs from PenPoint's object-oriented approach in program size reduction, code sharing, application consistency, programmer productivity, and elimination of boilerplate code (those large chunks of setup or housekeeping code that appear unchanged in every application).

Consider Sharing Code and Data 2.11.7

Think about what other parts of PenPoint need to access your classes, what tasks need to run the code in them, and who maintains their data. If your application has a client-server architecture, a separate back-end or a core engine, you'll need to have the picture in mind when choosing local or global memory, dynamic or well-known objects, process or subtask execution, protecting shared data with semaphores and queued access, and so on.

PenPoint is a rich operating system that makes its kernel features available to applications. But a straightforward application may not need to concern itself with any of these decisions. It just interacts with PenPoint subsystems, which make careful use of these features. For example, none of the tutorial programs use any advanced kernel features.

Use Document Orientation 2.11.8

In the PenPoint operating system, the user sees documents, not separate programs and program files. Every document on a page is the conjunction of data and a process running an application. This leads to a document-centered approach to application design in place of a program-oriented approach. By comparison, on a Macintosh or IBM-PC compatible computer, the user tends to fire up a program and work on a succession of files. Under PenPoint, the user turns to a new document (or taps in a floating document) and the system unobtrusively turns control over to the right program for that document.

There are many ramifications of this orientation: applications have no Open... or Save As... commands; the PenPoint operating system, not the user, saves data and quits programs; you deliver application templates and defaults to the user as stationery.

Design for File Format Compatibility 2.11.9

The PenPoint application environment differs from that of other operating systems in that PenPoint saves your application data, along with information about objects in the document. Because of this filing method, your data formats within PenPoint will differ from their PC equivalents.

Most PenPoint users, however, will need to read and write application data in formats that are understood by other non-PenPoint applications. Either your application should be able to read and write data in other formats, or you should create an import or export filter for your PenPoint files. PenPoint provides import and export filters for some common file formats. Because the import-export mechanism is class based, you or other application developers can create import-export filters for other file formats.

Exploit the Pen 2.11.10

Graphical user interfaces built around a mouse or other pointing devices lead to flexible program architectures that respond to the user's actions instead of requiring the user to perform certain steps. The pen-oriented notebook interface of PenPoint is even more free-form. Just as with a mouse, the user can point to and manipulate (click, drag, stretch, wipe) entities on-screen, but in the PenPoint operating system the user can also make gestures and handwrite characters "on" the visual entities. Taking advantage of the pen is a challenge and a tremendous opportunity.

Use the PenPoint User Interface 2.11.11

The Notebook User Interface (NUI) differs from other graphical user interfaces. If you are porting a DOS or Macintosh-based program to PenPoint, rethink your user interface so that it takes advantage of the PenPoint UI Toolkit. Do not create your own interface.

The *PenPoint UI Design Reference* describes the PenPoint User Interface, its rationale, and how and when to use its components. You should have good reason before you deviate from the PenPoint interface. Remember that a consistent user interface allows users to learn your application quickly; a bad or inconsistent user interface will count against your application in product reviews (and acceptance in the marketplace).

The PenPoint UI toolkit contains classes that create almost every on-screen object in the PenPoint NUI. If you use these classes, it is hard to deviate from the standard. Additionally, it is easier to follow the conventions by using these classes than to subclass and change their default behavior.

Chapter 3 / Application Concepts

This chapter gives you the big picture of application development for the PenPoint™ operating system. It introduces the design issues you need to consider when writing an application for a mobile, pen-based computer, how applications work under PenPoint, and how you use the PenPoint classes.

This chapter also presents concepts in general terms to provide the fundamental understanding that puts the balance of this manual in context. You needn't have read any of the other documentation before reading this chapter. However, if you have the SDK software, you might want to read the "Getting Started" document in the *Open Me First* packet for detailed instruction on how to compile and run the tutorial programs.

If you want a basic look at how the PenPoint operating system works, without a focus on writing applications, read Chapter 2, System Overview. If you need an introduction to object-oriented programming, read these industry publications:

- *Principles of Object Oriented Design*, Grady Booch, The Benjamin/Cummins Publishing Co., 1991
- *Object-Oriented Programming for the Macintosh*, Kurt Schmucker, Hayden Book Company, 1986.
- *Object-Oriented Programming: An Evolutionary Approach*, Second Edition, Brad J. Cox and Andrew J. Novobilski, Addison-Wesley Publishing Company, 1991.

However you do it, make sure you come to understand the basics of object-oriented programming, because in PenPoint every application must be class-based.

This chapter points out some of the aspects of the PenPoint Operating System that particularly affect your approach to application design.

As you know, application development takes place at two levels:

- At an architectural level, where you design your application
- At the line-by-line level of program statements.

At the architectural level, this chapter assumes that you have basic familiarity with object-oriented programming. In developing a PenPoint application you'll be designing different kinds of objects and the interactions between them and PenPoint. The section "How Applications Work" introduces the PenPoint **Application Framework**, which influences and supports the structure of *all* PenPoint applications.

At the programming statement level, this chapter assumes that you are well-versed in C programming. You'll be writing C code that makes heavy use of the PenPoint Class Manager. The section "Understanding Classes" introduces the Class Manager and shows you what lines of code in PenPoint look like.

With some understanding of the Application Framework and the Class Manager, you'll have the tools necessary to understand simple programs both architecturally and line-by-line. Later chapters in this manual describe the SDK sample programs in \PENPOINT\SDK\SAMPLE (the installation procedure for the SDK creates the \PENPOINT directory on your hard disk) .

PenPoint Programming is Unique 3.1

Just as a PenPoint™ computer is used in work environments that differ from other computers, PenPoint applications execute in an environment that differs from conventional PC application environments. There are eight key differences:

- Stylus-based user interaction
- Object-oriented programming
- Disk storage not necessary
- Multitasking
- Cooperating, simultaneously active, embedable applications
- Graphics-intensive user interface
- Notebook metaphor
- Document-orientation instead of applications and files orientation.

Dealing with these aspects of PenPoint requires you to observe a number of guidelines, described in the following sections. The good news is that the software architecture of PenPoint shoulders much of the load for you.

The **Class Manager** supports the pervasive use of classes and objects throughout PenPoint; not only in the user interface area, but also in areas such as the file system and the imaging model. These classes provide you with ready-made components which you can use as is or customize in your applications. These objects already conserve memory, exploit the pen interface, cooperate with other processes, and so on. In particular, nearly all of the work your application needs to do to work within the PenPoint Notebook is already implemented by pre-existing classes which comprise the PenPoint Application Framework.

How Applications Work 3.2

In the PenPoint™ operating system, the environment in which your application runs and how it starts up are unlike any other operating system.

MS-DOS accepts a command line, executes a single program at a time, and pretty much gets out of the way while that program is running. The PenPoint Application Framework takes an active role in running your application. The Application Framework is responsible for activating, saving, restoring, and

terminating your application. Additionally, the Application Framework plays a part in installing and deinstalling your application.

Because all PenPoint applications use the Application Framework, all applications behave consistently. Additionally, the Application Framework handles the housekeeping functions that Macintosh or MS-DOS programs must perform from boilerplate code. Meanwhile, the PenPoint Application Framework presents the PenPoint user with multiple small, concurrent *documents* as part of a consistent, rich notebook metaphor.

It's difficult to cleanly define the PenPoint Application Framework, because it is both external to your application and something your application is itself a part of. But here's an attempt:

The PenPoint Application Framework is both the protocol for supporting multiple, embeddable, concurrent applications in PenPoint, and the support code that implements most of an application's default response to the protocol.

To help you understand how an application fits into the PenPoint computing environment, this section walks through some important stages in the life of an application. By its end you should understand a little about the PenPoint Application Framework, some of the classes of objects in PenPoint, and why classes are so important. The next section explains class-based programming in PenPoint.

With an understanding of the PenPoint Application Framework and the Class Manager under your belt, you'll be able to work through the tutorials on PenPoint programming that begin in Chapter 6. The tutorial summarizes other PenPoint subsystems: windows, User Interface (UI) toolkit, filesystem, and handwriting translation. The tutorial incorporates these subsystems into a set of increasingly functional sample programs.

Installing and Starting Applications 3.2.1

After acquiring an application, the user must install the application in the PenPoint computer. Usually an application distribution disk contains the code and data that implement the application's classes, and any other classes required by the application.

We'll first look at how a user installs and starts a program on a traditional PC operating system (MS-DOS). Then we'll compare these operations with installing and running an application on PenPoint.

MS-DOS Installation 3.2.2

In MS-DOS, the user usually installs a program by copying the program from distribution disk to a hard disk. Once on the hard disk, the program does nothing until the user types a command to start the program up.

Some MS-DOS programs require the user to copy the files from to the hard disk; others provide their own installation programs, which copy the files to the hard disk and alter system configuration parameters for their program. Installation varies tremendously from program to program.

When the user types the startup command for a program, MS-DOS loads the program into memory from the hard disk and transfers control to the program. Once the program is running, it controls most of the operations of the CPU until the user leaves the program.

PenPoint Installation 3.2.3

In PenPoint, the user installs a program by opening the Connections or Settings notebook on the Bookshelf and turning to the installable software sheet (or by inserting a disk that contains quick installer information).

From the installable software sheet, the user can choose various categories of installable items, including applications, services, dictionaries, and so on. When the user turns to a page for an installable item, the Installer shows all the available applications that can be installed from the currently open volumes. The user selects an item and taps the Installed? checkbox next to the item. The Installer copies the program to an area of memory set aside for programs (the loader database) and copies other files required by the program (such as help files, application resource files, and stationery files) to the file system.

From this point, running PenPoint applications differs significantly from the MS-DOS model. Once a program is in the loader database, PenPoint can transfer control directly to it; there is no intermediate step of loading the program into memory, because it is there already.

PenPoint transfers control to your program for two different reasons: the user is installing your program, or the user is opening a document that requires your program (we will cover this case in the next section).

Installer Responsibilities 3.2.4

During installation, the Installer calls a standard entry point in your program (called **main**) in such a way that you can tell that your program is being installed. At this time, most programs create their application class and any other classes that they need. Some programs initialize files or common data structures such as dictionaries or stationery.

If your application requires code for other classes (such as a special character-entry class) and resources (such as a special font), the Installer ensures that these classes and resources are present in the computer. If they are not present, the Installer copies and installs them also. In turn, these classes may require additional classes and resources, and so on.

The installer keeps track of all installed applications. When the Installer initializes your application, the application specifies whether it should go in the Tools accessory palette or in the Stationery notebook, or both (or neither). Depending

how your application initializes itself, the user will now see the application in the Accessories window, or in the Stationery notebook and Stationery pop-up menu.

After installation, your code is in a similar state to an MS-DOS .EXE or .COM program that has just been loaded into memory but not yet run. However, when the MS-DOS program terminates, it removes itself from memory. PenPoint programs stay in memory until the user removes the application.

Running a PenPoint Application 3.3

When running an MS-DOS program, the user has to find a file that contains data understood by the program. When the user decides to stop using the program, he or she must save the data to a file and then exit. If the user chooses a file that the program doesn't understand, the program might display garbled information, at best, and at worst the program might crash.

PenPoint takes a fundamentally different approach; the user creates a **document** from a list of available applications and, at some later time, tells PenPoint to activate the document. The user doesn't have to activate the document immediately after creating it and, in fact, can create many, many documents without activating any of them.

Life Cycle of a Document 3.3.1

The standard components of an application include its application code, application object, resource file, instance directory, process, and main window. The full life cycle of a document created by an application includes these operations:

- Document creation (create file)
- Activation (create process)
- Opening (open on screen)
- Closing (remove from screen)
- Termination (terminate process)
- Destruction (delete file)

Active documents save their internal state in the file system, but this is invisible to the user: there is no need to save or load the application's state explicitly from one session to the next.

Activating a Document 3.3.2

When the user activates the document, PenPoint finds out from the document what application it requires and creates a process that "runs" the application (the reason for the quotes is explained below under Application Classes and Instances). When the user deactivates the document, PenPoint saves all information for the document and then destroys the application process.

The important thing in PenPoint is that the document remains in the computer from the time it is created until the time that the user deletes it, but the application process exists only while the document is active.

Not All Active Documents are On-Screen 3.3.3

It's only when the user activates a particular document that the document has a running application process. When the user activates a document, the PenPoint Application Framework creates an application process and calls the standard entry point in your code (**main**) in such a way that your application can tell that it is starting an application process (and not being installed).

However, just because a document is running, doesn't mean that it must be on-screen; conversely, if a document is not on-screen, its process might still be running.

The most common example of this is when the user makes a selection in a document and then turns to another document (perhaps to find a target for a move or copy). The document that owns the selection must remain active until it is told to release the selection.

A second example is the user chooses Accelerated Access Speed from the Access document option sheet (sometimes called hot mode), the application processes will continue running, even when the user has turned to another page.

For a third example, you might want to create a stock-watcher type program that runs in the background most of the time. This type of program will also be active but not on-screen.

Application Classes and Instances 3.3.4

A PenPoint computer contains only one copy of your application code in memory, but a user can simultaneously activate several documents that use your application. PenPoint can do this because your application code is a PenPoint class and an active document is an instance of your application class.

When the user installs your application, your application creates your application class. When the user activates a document that uses your application, the Application Framework creates an instance of your application class.

Accept this as Gospel now. We will spend pages and pages in this and other manuals explaining how this works.

PenPoint Drives Your Application 3.4

Because of all these states that an application can be in, an application can't take control and start drawing on the screen and processing input when its **main** function is called. Nor can your application find out on its own if it is on-screen or should terminate. Instead it *must* be directed what to do by the PenPoint Application Framework. The Application Framework sends messages to documents (and hence to your application code) to initialize data, display on screen, save their state, read their state, shut down, and so on. This is why applications *must* be implemented as classes.

For example, when a document needs to be started up to do some work, the PenPoint Application Framework sends **msgAppActivate** (read this as "message

app activate") to the document. When the user turns to a document's page, the PenPoint Application Framework sends it **msgAppOpen**.

A typical MS-DOS program written in C has a `main` routine that displays a welcome message, parses its command line, creates a user interface, initializes structures, and then waits for user input. By contrast, a PenPoint application's **main** routine usually creates the application object and then immediately goes into a loop waiting for messages to the application object to arrive. Because all applications enter this loop, there is a routine, **AppMain**, which enters the loop for you.

Application Objects 3.5

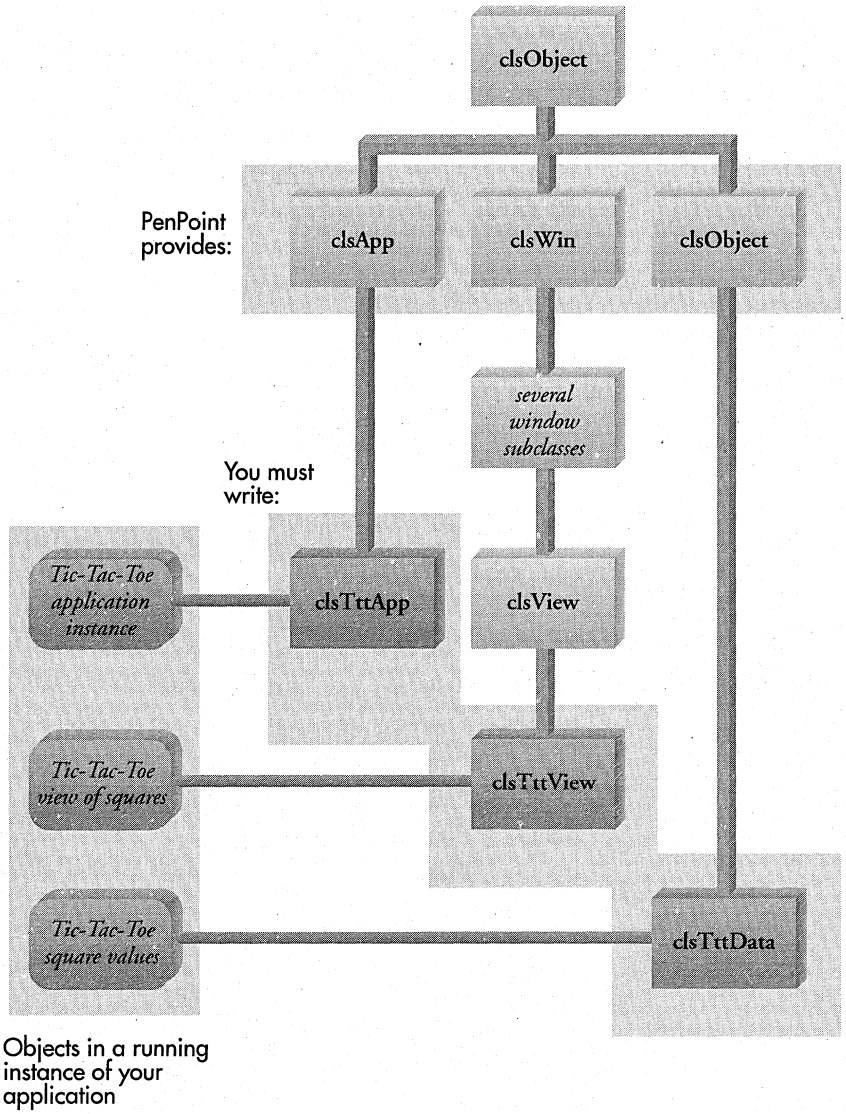
Most PenPoint applications perform three minimum actions:

- Respond to user and system events (including PenPoint Application Framework messages)
- Create one or more windows for user input and to display output
- Create one or more objects to maintain their data.

There are object classes already written in PenPoint for each of these actions: **clsApp**, **clsWin**, and **clsObject**, respectively. These classes do the right kinds of things for applications themselves, windows, and data. They provide a skeleton of correct behavior, although obviously GO's code doesn't create tic-tac-toe applications, tic-tac-toe windows, or 3x3 "board" data objects. To get the behavior you want, you often need to use **descendant** classes that **inherit** from existing classes.

This section overruns with the terminology of classes. **Descendant**, **Inheritance**, and other terms are explained in the next section, "How Classes Interact."

Figure 3-1
Application, View, and Object Classes



A Descendant of clsApp

3.5.1

The EMPTYAPP sample program in the Tutorial does nothing significant in response to any message, yet because it inherits from clsApp you can create EmptyApp documents, copy them, float them, embed them, and so on.

Lots of stuff is done for you!

The PenPoint Application Framework's interactions are sophisticated and complex. You'll learn more about them in the following sections. Applications need to behave in a standard way to work well in the framework. To simplify life for the application developer, your application class inherits most of this standard behavior from the class **clsApp**. **clsApp** handles all the common machinery of application operation, so that many applications do not need to do anything in response to messages like **msgAppActivate** and **msgAppOpen**. Applications rely on **clsApp** to create their main window, display the main window, save state, terminate the application instance, and so on.

You must write a descendant class of **clsApp** and create it during installation. In the example shown here, the descendant is **clsTttApp**. At the appropriate time,

the PenPoint Application Framework sends this class a message to create an instance of the class (*Tic-Tac-Toe application instance* in the figure). However, you must decide when to create your application's other objects (windows and filing objects).

An Instance of clsWin

3.5.2

Frames support only one client window, but you can insert other windows inside the client window.

The PenPoint Application Framework creates a **frame** for your application by default. This is a window with many "decorations": a title bar, a shadow if **the window is floating**, optional resize corners, close box, menu bar, tab bar, command bar, etc. These decorations surround space for a **client window**. It is up to you to create the client window. You can also create windows to go into your frame's menu bar, tab bar, and command bar, and you can create floating windows, additional frames, and so on. Most applications create one or more windows to draw in and allow user input.

All window classes inherit from **clsWin**. This class does not paint anything useful in its window, so you must either create your own window class which draws what you want or use some of the many window descendant classes in PenPoint.

Some Window Classes

3.5.2.1

The Tic-Tac-Toe application, shown in Figure 3-1, for example, creates several kinds of windows based on existing classes in PenPoint:

- A scrolling client window (an instance of **clsScrollWin**), which lets the user scroll its contents
- An option sheet for its options (**clsOption**)
- An option card for the option sheet (**clsOptionTable**)
- Various user interface component windows (**clsButton**, **clsLabel**, **clsIntegerField**) for the option card
- Menus
- A Tic-Tac-Toe view (**clsTttView**) to display the grid and Xs and Os.

Like **clsTttApp**, you have to write the code for **clsTttView** and create the class at installation. Your application must create the various windows at the appropriate times, such as when it receives **msgAppInit** or **msgAppOpen**.

Using clsView

3.5.2.2

Many applications will use **clsView**, a specialized descendant of **clsWin**, for their custom windows. **clsView** associates its window with the data object it is displaying; the data object sends the view a message when its data changes. In the case of Tic-Tac-Toe, **clsTttView** inherits from **clsView**, so the Tic-Tac-Toe window is a view.

In Tic-Tac-Toe, a **clsTttView** instance **observes** the data object (an instance of **clsTttData**). More than one view can be associated with the same data; in theory

two views of the Tic-Tac-Toe board could show their state in different ways. When the data changes, all the views are **notified** and can redraw themselves.

An Instance of clsObject 3.5.3

Instead of managing all of the data involved with an application itself, a PenPoint application typically creates separate objects that maintain and **file** different parts of the data. These objects respond to messages like "Save yourself" and "Restore yourself from a file."

clsObject is actually the ancestor of all classes in PenPoint, including **clsWin** and **clsApp**. There is no class specifically for objects that must be filed. Filing is such a general operation that all objects in the PenPoint operating system are given the opportunity to respond to **msgSave** and **msgRestore** messages. PenPoint supplies various descendant classes, which help in storing structured data, such as a list class (**clsList**), a picture segment (**clsPicSeg**), a block of styled text (**clsText**), and so on.

In Figure 3-1, the data for the Tic-Tac-Toe application (the values of the nine squares) are maintained by a separate object, **Tic-Tac-Toe square values**, an instance of the specialized class **clsTttData**.

Understanding the Application Hierarchy 3.6

You may have wondered how PenPoint keeps track of all the sections, documents, and embedded documents in a notebook if application objects are not immediately up and running when they are created. The answer is that each document and section in a notebook is represented in an **application hierarchy** in the PenPoint file system. The Notebook table of contents displays a portion of this application hierarchy.

The application hierarchy differs from the class hierarchy explained in the next chapter, and from the hierarchy of windows on-screen.

The reason it is called an application *hierarchy* is that the directory structure is the same as the hierarchy of documents in PenPoint (including embedded documents, accessories, and other floating documents not on a page in the Notebook). Each notebook has a directory in the file system. Within the notebook, each document or section has a directory. Within each section, each document or section has a directory. Within each document, all embedded documents have a directory, and so on.

As an example, when the user creates a document in a section of the Notebook, the PenPoint Application Framework creates a new application directory in that section's directory. When the application is told to save its state by the PenPoint Application Framework, the PenPoint Application Framework gives it a file to save to in that application directory.

All PC operating systems have a file system, and in most you can store application data in a similar hierarchy of directories and subdirectories. Some may even provide a folder or section metaphor for their file system. But they do not directly weave applications into this file system. The Notebook's TOC (tap on its

Contents tab to move to it) shows the organization of documents in the Notebook, and *this is* the organization of part of PenPoint's file system.

In PenPoint, the application hierarchy exists in the \PENPOINT\SYS\BOOKSHELF directory on **theSelectedVolume**. You can inspect the application hierarchy yourself. Modify your ENVIRON.INI file so that the DebugSet parameter specifies /DB800. Run PenPoint and go to the connections notebook. Using the directory view, browse through the disk volume. In the \PENPOINT directory, you should see directories called NOTEBOOK, SECTION, and so on. Compare this with the Notebook TOC. The browser shows exactly what the file system looks like, while the Notebook TOC interprets this part of the file system as the application hierarchy.

If your selected volume is your hard drive, you can also inspect this hierarchy from DOS. However, to keep path names short, all of the PenPoint directory names below \PENPOINT use two letter names. For example, the SYS directory is SS in DOS, the BookShelf directory is BF, the Notebook is NK, and so on.

The Notebook's Own Hierarchy 3.6.1

The PenPoint classes and application hierarchy probably seem obscure and confusing at this point. So let's look at how the Notebook itself is written using this metaphor. Each component of the Notebook is itself a document, with its own main window, a parent window, and a directory in the file system's application hierarchy.

The important concept to grasp is that there is a correspondence among: Strange and important!

- PenPoint applications,
- The functionality of the parts of the notebook metaphor,
- The visual presentation of parts of the Notebook,
- The PenPoint file system layout.

Some of these relationships are:

Running documents are instances of application classes.

Functionality of notebooks, sections, and pages is delivered by application classes.

Visual components of a notebook are these applications' windows.

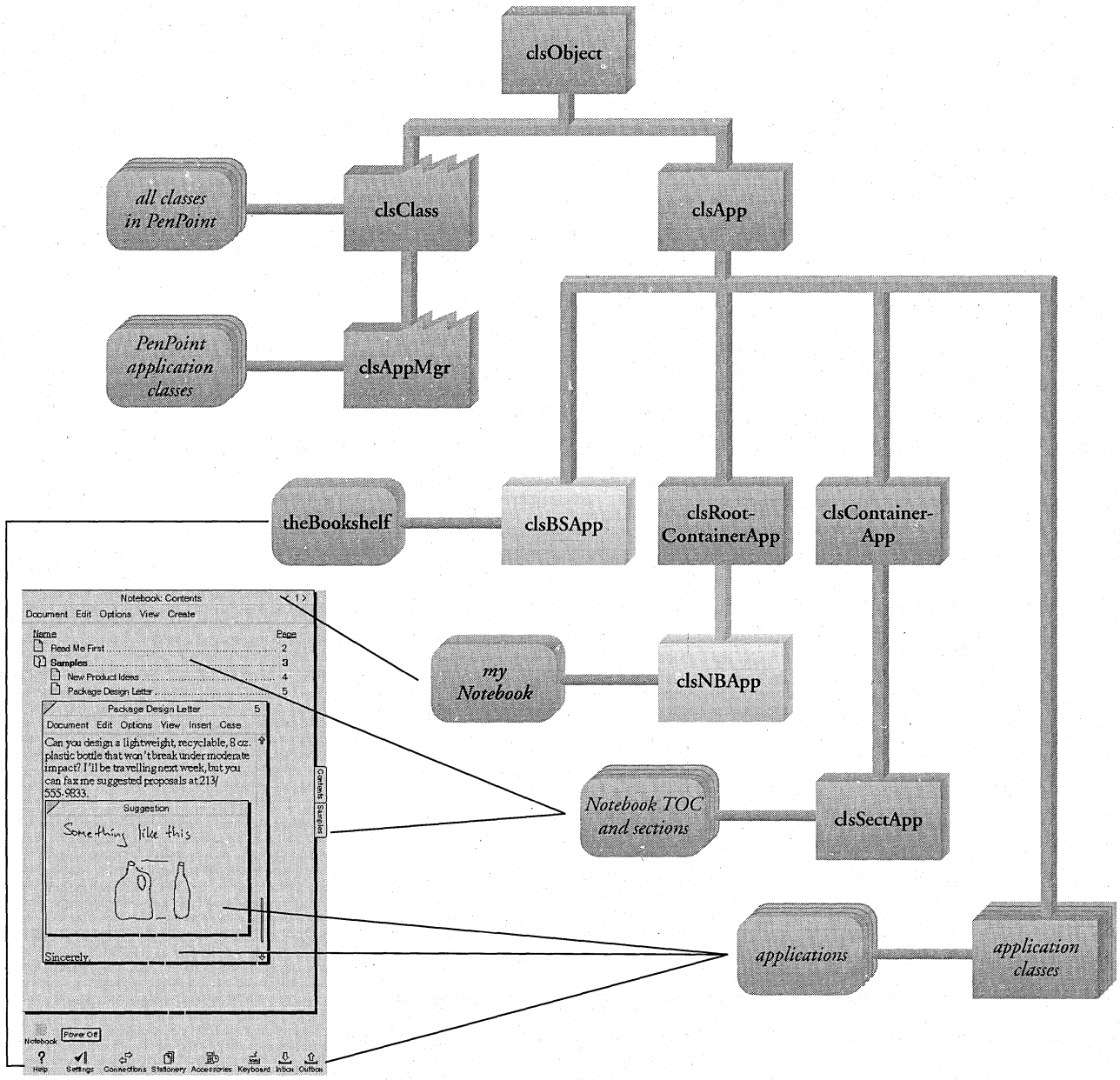
Sections and pages in a notebook are these applications' directories.

Section name and page number location in a notebook combine to form a location in the file system.

This figure shows how a typical mix of applications in a running PenPoint system uses different kinds of classes.

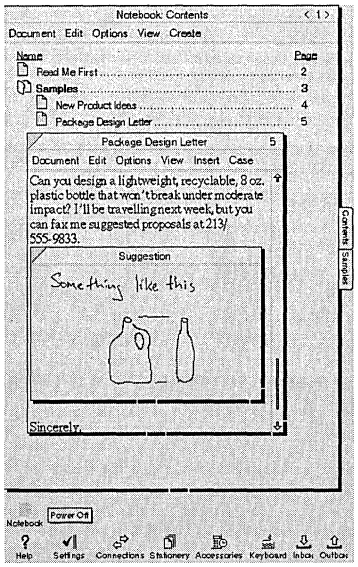
The following figures are explained in more detail in *Part 2: Application Framework* of the *PenPoint Architectural Reference*

Figure 3-2
The PenPoint Application Framework and the Notebook Hierarchy



The next figures indicate how the same visual components exist in the file system, and as processes and objects.

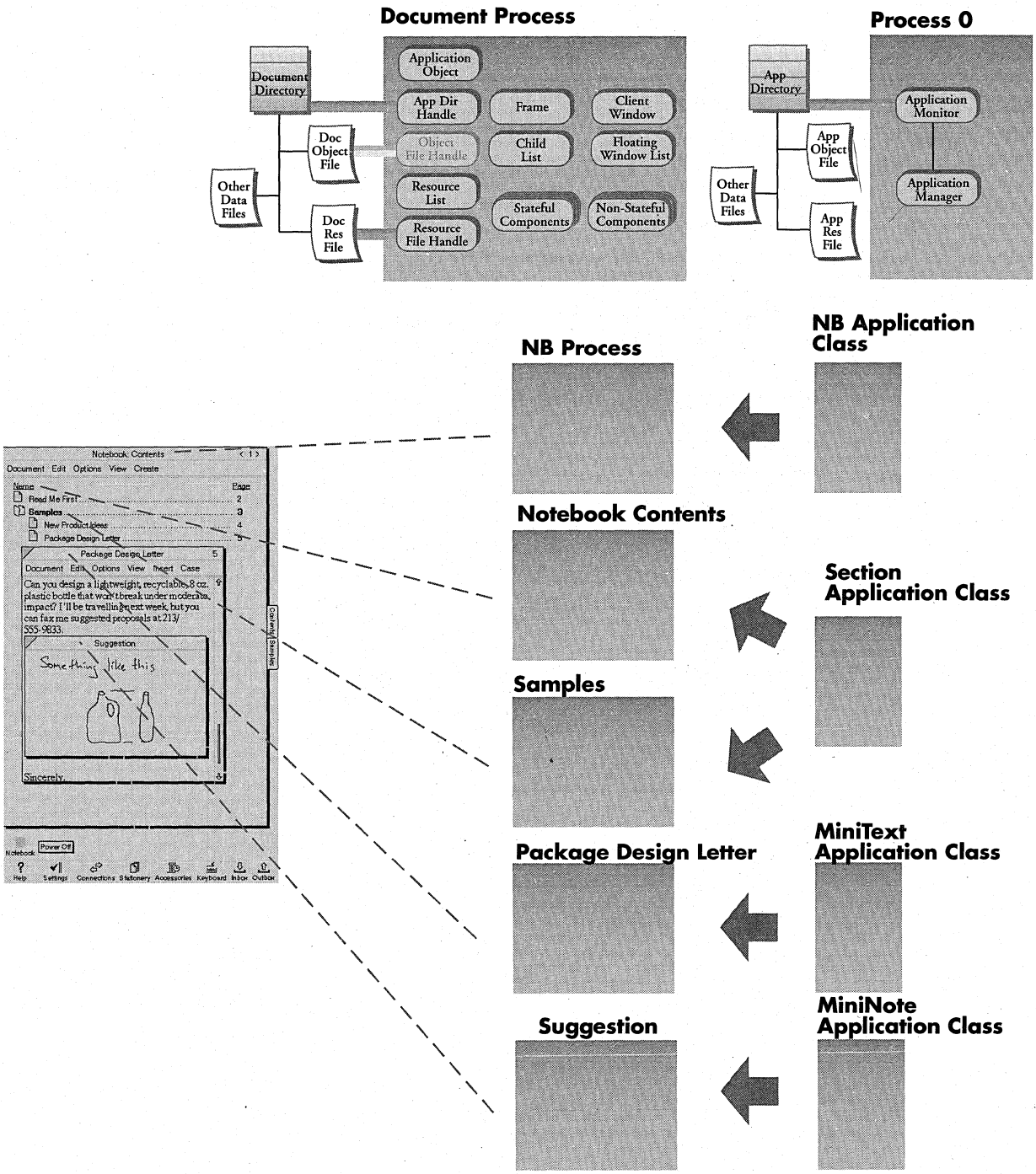
Figure 3-3
The Notebook Hierarchy as Mirrored by the File System



Bookshelf
- doc.res
- docstate.res
- **Notebook**
  - doc.res
  - docstate.res
  - **Contents**
    - doc.res
    - docstate.res
    - browstate
    - **Read Me First**
      - doc.res
      - docstate.res
    - **Samples**
      - doc.res
      - docstate.res
      - browstate
      - **New Product Ideas**
        - etc...
      - **Package Design Letter**
        - doc.res
        - docstate.res
        - **Suggestion**
          - doc.res
          - docstate.res
    - etc...

Here are the same visual components as they exist in the PenPoint file system.

Figure 3-4
The Notebook Hierarchy as Mirrored by Application Processes



You can use the Disk Viewer accessory to explore the relationship between documents and the file system yourself. To view the RAM volume in the Disk Viewer, you need to set the **B** debug flag to hexadecimal 800 in order to view the contents of the RAM file system. The easiest way to do this is to modify the `DebugFlag` line in ENVIRON.INI.

The Desktop

3.6.2

The highest level of the application hierarchy is the Desktop. This is an application, but there is only ever one instance of it—you can't create additional desktops. The Desktop application manages the bookshelfs and floating applications. Its **parent window** is the entire screen of the PenPoint computer. It draws the white background.

The Notebook

3.6.3

Below the Desktop's directory lies the directory of the main Notebook (and other documents on the bookshelf). The Notebook application presents the familiar visual metaphor of a notebook, with pages and tabs. All applications that "live" on a page have subdirectories in the Notebook. There are usually several notebooks on a PenPoint computer: the Main notebook, the Stationery notebook, and the Help notebook. Even the In Box and Out Box are implemented as notebooks.

The Notebook document stores the section tab size, the current page shown in the Notebook, the page numbering scheme, and so on in its directory.

When the user taps to turn a page in the Notebook, the Notebook traverses the application hierarchy to the next document directory and sends a PenPoint Application Framework message to that document's application to start it up.

The Notebook's window covers most of the screen except for the Bookshelf at the bottom.

Page-Level Applications

3.6.4

The subdirectories in the Notebook's directory relate directly to the documents and sections in the Notebook. The name of the subdirectory *is* the name of the document or section. Each of these subdirectories contains the filed state of an instance of a section or document.

Actually, sections are documents that know how to behave in a table of contents.

This table lists some of the items in the Notebook (shown in Figure 3-4, the directory in which each of the items are stored, and the class from which each item is instantiated.

Table 3-1
Notebook Organization and the File System

Document or Section	Stored in Directory	Instance of Class
Samples	Notebook Contents	clsSection
New Product Ideas	Samples	clsMiniText
Package Design Letter	Samples	clsMiniText
Suggestion	Package Design Letter	clsMiniNote

Most applications have a menu bar. The PenPoint Application Framework supplies a set of standard application menus (**SAMs**), to which applications add their own menu items. The PenPoint Application Framework provides support

for the menus (Document, Edit, and Options) and many of the items on the menus.

Applications draw in the window that the Notebook provides for them. A page-level applications's window is the Notebook area; except for the tabs area.

Sections

3.6.5

Sections are similar to other applications: they are instances of an application class (**clsSectApp**), they appear on a Notebook page, they can have tabs. A section application displays a table of contents showing the documents that are in that section: these are simply the application subdirectories in the section's own directory.

One difference between a section and other applications is that a section has a special flag in its directory entry. When the Notebook is traversing the application hierarchy (to display its table of contents, or turn to the next page), if it comes across a section it *descends* into the section. This enables the Notebook to number pages correctly.

Actually, a section has a special directory attribute.

Section data stored in the section's directory entry includes the state of its table of contents view (expanded or compressed).

The Notebook Contents page is an instance of **clsSectApp**, just like other sections. To show the TOC for everything in the Notebook, it must have everything in the Notebook in its directory. This is why the actual file system organization of the application hierarchy is \PENPOINT\SYS\MY_NOTEBOOK\NOTEBOOK APPS\... Where NOTEBOOK APPS is the directory of the "section" for the entire Notebook contents.

Floating Accessories

3.6.6

Most PenPoint applications are part of the Notebook. But some applications, such as the calculator, the disk viewer, and the snapshot tool, don't "live" on a page in the Notebook. These **accessories** "float" on the Desktop when active, appearing over pages in the Notebook. Their parent window is the Desktop, not the Notebook page area. They aren't part of the Notebook's table of contents and you can't turn the page to them. However, a floating application is still part of the same underlying model: it has a directory (it's just not a subdirectory of the Notebook), it is sent messages, and so on.

Embedded Applications

3.6.7

It is possible to **embed** documents in other documents which permit it. For example, an on-line "electronic newspaper" document might install an instance of a crossword puzzle application in itself; the crossword puzzle class might allow the user to embed an instance of a text application in a crossword puzzle document to let the user jot down notes and guesses. The design of PenPoint makes it easy to write applications which can embed, and be embedded in, other applications.

When the user creates a new document in the notebook, PenPoint actually embeds the application in the Notebook application. This document embedded in the notebook is called a page-level application.

Only page-level applications appear in the Notebook's Table of Contents—applications that are embedded in page-level applications do not. It doesn't make sense for a user to turn the page to an application embedded in the current page.

Application embedding is very straightforward. When the user moves or copies an application, the Desktop application sends a **msgAppCreateChild** message to the destination application. If the application permits embedding, the PenPoint Application Framework handles this message by creating a directory for the embedded application within the destination application's directory.

When an application is embedded in another, the embedded application is inserted into two hierarchies: the file system hierarchy and the window system hierarchy. In the file system, the application directory for an embedded application is a subdirectory of the application directory of the application in which it is embedded. In the window system, the parent application supplies a window into which the embedded application can insert its main window.

Thus, in our example, the newspaper application uses an application directory for the newspaper document. Within that directory is an application directory for the crossword document. Within the crossword application directory is a directory for the text editor document. The newspaper document window contains a window which is the main window for the crossword document. The crossword document window contains a window which is the main window for the text editor.

Application Data

3.6.8

A document stores data in its directory so that when its running process is terminated, its state lives on in the file system. The Application Framework can later create a new process for the document and direct the document's application to restore the document from this filed state.

Some information is of interest to this instance only, such as the visible part of the file, the user's current selection, and so on. This would probably be saved by the application itself, that is to say, when the application receives **msgSave** it writes this information out.

The application can also tell the Application Framework to send **msgSave** to other objects to get them to save their data (your application can't send **msgSave** directly to another object). For example the image in a sketching program might be implemented as a separate object; when the application is told to save, it tells the Application Framework to save the image object.

There are many mechanisms that automatically propagate msgSave to related objects. Frames can be set to save child windows, views save their data objects, and so on.

By default **clsApp** saves the information about the document, including its comments, frame window position, mode, and so on, so you only need to save those things created by your application class.

Activating and Terminating Documents 3.7

In the section "Application Classes and Instances," we described how an instance of the application is created. The previous section should help clarify the relationship between the file system and an instance of your application. The location of a document in the file system hierarchy has a one-to-one correspondence with its location in the Notebook, on a page, within a section, and so on. See the figure to get a sense of the relationship.

The main determinant of how and when documents blossom from being directories and data in the file system to being live running processes and objects is the user's action of turning the page.

When the user turns to a page, the documents on that page become visible; if they aren't already running, the Application Framework activates them.

Turning a Page and msgAppClose 3.7.1

When the user turns to another page, the document on the original page no longer needs to appear on screen, so the PenPoint Application Framework sends **msgAppClose** to the application instance, indicating that it can close down its user interface.

When it receives **msgAppClose**, the application might still have some processing to do or it might be talking to another application. The application can finish its work before acting in response to **msgAppClose**.

To respond to **msgAppClose**, the application should save (to the file system) any data about on-screen objects that the user moved or changed. The application should then destroy and remove all windows that it created, thereby reducing memory usage.

An application instance may receive **msgAppTerminate** after **msgAppClose** (if it isn't in "hot mode"). When it receives **msgAppTerminate**, the application must save all data that will be required to restore the document to the screen exactly as it was before, because **msgAppTerminate** kills the document's process.

Restoring Inactive Documents 3.7.2

When the user turns back to the saved document, the Application Framework looks at that document's directory. If the process for the document was terminated, the Application Framework starts a new process, creates a new instance of the application class, and recreates the document based on information in the directory. As part of this re-creation, the Application Framework sends the document **msgRestore**, which tells it to read its state back in from the file system.

The Application Framework then sends **msgAppOpen** to the application, telling it to prepare to draw on the screen. The Application Framework also sends **msgRestore** and **msgAppOpen** to any embedded applications in that document.

Finally the Application Framework inserts the application's windows into the screen, and the windows receive messages telling them to paint.

From this point the user can interact with the document. When the user makes a gesture within the document, the document's application controls the resulting action.

Page Turning instead of Close 3.7.3

As described in "Turning a Page and msgAppClose," most PenPoint applications don't need a Close menu item. Most documents are active until the user turns the page; others may be active even when off-screen (for instance, if they have the selection or are involved in a copy operation). The user doesn't know what a *running* application is: when the user turns to a page, everything on it appears exactly as it was when the page was last "open," and every window responds to the pen. The fact that some of the applications may have been running all the time while others were terminated and restarted should be inconsequential to the user.

Saving State (No Quit) 3.7.4

In an MS-DOS or Macintosh program, the user explicitly **Quit**s the application, and thus doesn't expect the application to reappear in exactly the same state.

Because of PenPoint's notebook and paper paradigm, you must preserve all the visual state of your application so that when it is restarted it appears the same. This has strong implications for the kinds of information your application needs to save when an application receives **msgSave**.

Documents, not Files and Applications 3.8

It's important to understand that the application instance and the file it is editing are conjoined.

The user should rarely, if ever, see "files," instead she or he sees only documents. (The exception to this is when importing data from and exporting data to other computers.) Ordinarily, for every document in the application hierarchy there is an application.

A user can deinstall an application without deleting the application's documents in the file system. If the user tries to turn to one of these documents, there is no code to activate them. Instead, these orphan applications are handled by a "mask" application that tells the user that the application has been deinstalled and prompts the user to reinstall the application.

No New, No Save As... 3.8.1

On a PC, the user usually starts an application, and then chooses what file to open with that application. But in the PenPoint operating system, the user can start an application by:

- Turning to the page that contains a document,
- Floating a document,
- By creating a new embedded document.

The document open on a page (and any floating or embedded documents on that page) *are* all applications with open files. You do not open a file from within an application. Instead, you turn to (or float or embed) another document and PenPoint starts up the correct application for that document.

Thus, it does not make sense to try to open another document from the current application, or to save the current document as another document.

The only time that an application needs to actually open a file from disk is when it is importing data from or exporting data that will be used by a file-oriented program on a file-oriented operating system.

Stationery 3.8.2

Users often want new instances of an application to start off from a particular state. Instead of opening a template from within the application, Penpoint supports application-specific **stationery**. The default piece of stationery is an application instance started from scratch. The user can create additional stationery documents, which are just filed documents kept in a separate notebook.

In the case of Tic-Tac-Toe, each document shows a view of its own board. There is no new command, because the user can always create a new document. There is no save command either—the Tic-Tac-Toe state is saved on every page turn. There's no open command, because the user can either turn to another Tic-Tac-Toe's page to "read it in," or can start from a desired template by accessing documents in the Stationery menu or auxiliary notebook.

Shutting Down and Terminating Applications 3.9

If a document is in the application hierarchy, it always exists as a directory in the file system, whether it has a running process or not, and whether it is visible or not. When the user deletes a document (page-level or embedded), PenPoint deletes its directory from the file system.

The user can also elect to use the installer to deinstall or deactivate an application. This might be necessary when the user needs more room on the computer for a different application, or when the user isn't using an application any more. Deinstallation removes all application code from the loader database, which prevents the user from running it. However, the documents still exist in the application hierarchy, and can spring back to life if and when the user re-installs the application. Deactivation also removes the application code, but PenPoint remembers where the application came from, so that it can prompt the user to insert the appropriate disk if the user chooses to reactivate the application.

While the application is not available, the mask application handles the application's documents.

Conserving Memory 3.9.1

When a document is active, it is obviously consuming memory, but when it is not active, it can still consume memory (if the computer is using a RAM-based file system). The document's saved state is in the application hierarchy, which can be

in the RAM file system; the RAM file system shares RAM with running processes. This emphasizes how important it is to conserve memory.

You should also try to conserve memory when an instance is running but not open (for example, if it has the selection but is off-screen). This is an opportunity to destroy UI controls and other objects which are only needed when your application is on screen.

Avoiding Duplication 3.9.2

Documents receive messages from the Application Framework telling them to save their state to their directory. When a document starts up, its corresponding application often reads all of this state back into memory. This means that there are two copies of the document's state, the one in its address space and the saved copy in the file system. This can be quite wasteful of space. There are several approaches to eliminating this redundancy:

- Don't read state back into memory. Read information in from the file system when needed. This works well for database-type objects. Because the application hierarchy is in memory, file I/O is faster than you might think, but this is still slow. It does prevent the user from reverting to the filed state of the document, since the filed state is always being updated. Your application would have to disable **Revert**, or make its own backup copy of filed state.
- Use memory-mapped files to map filed state into the application's address space. This works well for large data files, but it does interfere with **Revert**.
- Read state back into memory, then delete the information from the file system. This means that if the application instance crashes, there is nothing in the file system to recover.
- Refuse to save state to the file system. This implies that the application process can't be terminated. This also means that the application state can't be recovered.

Hot Mode 3.9.3

The last alternative above is supported by the PenPoint Application Framework. An application class or the user (by choosing Accelerated for **Access Speed** in the application's option sheet) can tell the PenPoint Application Framework that an application instance should not be terminated. This is called "hot mode." It means that the document will appear much faster when the user turns to it, because its process never went away. Ordinarily the Application Framework must start a new process, create a new application object, tell it to restore its state, then put it on-screen.

Components 3.9.4

As we have seen, you can embed applications within other applications. This is the basis for the Application Framework's hierarchy. Applications require a good deal of overhead: each has its own directory, has code in the loader database, and runs as its own process (in addition to the directories and processes used by that application's documents).

You can reduce the size of an application by using components. Components are separate DLL's that provide a well-defined API to their clients. Most **components** can be used as part of an applications, but they don't require much overhead.

Components don't run as a separate process, and don't have a separate directory. Some components, such as Reference buttons, manifest themselves as visible objects and let the user embed, move, and copy them. Others, such as text views, are visible but can be added to applications only programmatically. Others, such as the Address Book, do not even have a UI; that is, they do not display on screen (the address book provides information that other applications then format and display).

Chapter 4 / PenPoint Class Manager

The previous chapter introduced some of the concepts in the PenPoint™ Application Framework. This section quickly covers the PenPoint operating system's object-oriented Class Manager. The Application Framework largely determines the overall structure of your applications; sending messages to objects using the Class Manager makes up 80% of the line-by-line structure of your code. With an understanding of the Application Framework and the Class Manager, you can start the tutorial.

There are three elements to the PenPoint operating system's object-oriented software environment: objects, messages, and classes.

Perhaps the simplest way to introduce the concepts of objects, classes, and messages is by looking at an example. The example discussed in the next three sections outlines what must happen to set the title of an icon. A user sets an icon's title by making the ✓ gesture over it. When its option sheet appears, the user enters a new icon title, makes sure the layout style is one that includes the title, and taps Apply.

If you feel that you understand the concepts of object-oriented, message-passing, class-based systems, you can skip this introduction and go directly to the section titled "Sending a Message."

Objects Instead of Functions and Data 4.1

In a non-object-oriented system, the icon and its title would be stored in a data structure. Any piece of code that gets or sets information pertaining to the icon must know the exact organization of that data structure. To modify the icon title, the program would locate the data structure that represented the icon; for example, it might change the icon's title string by changing a **pTitleString** pointer. This program will break if the internal structure changes or if the string is later implemented by storing a compact resource identifier.

In an object oriented system, anything in the system can be an object. In our example, the icon is represented by an **object**. The object knows about both the data for an icon and the functions that manipulate it. The object hides, or encapsulates, the details of its data structures and implementation from clients. One of the messages understood by the object might be "Set Your Title String," which tells the object to change its title.

Because the object contains the code for the functions that manipulate it, the object locates its own internal data structures that represent the title, and changes the title.

This encapsulation reduces the risk of clients depending, either deliberately or accidentally, on implementation details of a subsystem. If the internal structure changes, only the object's code that manipulates the structures must change. Any client that sends the "Set Your Title String" message can still send that message and will still get the same effect.

Note "Client" here and elsewhere in the SDK documentation means any code making use of a software facility.

Some objected-oriented systems, including PenPoint, use software and hardware protection facilities to prevent clients from accessing or altering the internal structures of objects, whether accidentally or maliciously.

Ideally, in object-oriented operating systems, the objects presented to clients should model concrete ideas in the application. For example, if your application's user interface requires a button, it should create an object for that button; if your application has a counter, it should create an object to maintain that counter value.

Messages Instead of Function Calls

4.2

Modular software systems are sometimes object-oriented without being message-passing. That is to say, they have objects that hide data structures from clients (such as "window"), and you pass these software objects as arguments to functions which act on them. Using the example of setting an icon's string, in such systems you might pass the **icon_window** object to a routine called **Window_Set_String**.

But this approach requires that clients know which function to call, or that the function handle many different kinds of objects. The implementation of icon strings might change so that icons need to be handled specially by a new **Icon_Set_String** function. Again, all clients would have to change their function calls.

Message-passing systems flip this control structure so that the object hides the routines it uses. A client simply sends a message to the object, and the object figures out what to do. This is known as data encapsulation. In the example we're using, clients send the message **msgLabelSetString** to the icon; the only argument for the message is a pointer to the new Title string.

Because icons (or other objects) respond to messages, it doesn't restrict the implementation of icons: if, in the future, icons handle titles differently than other labels, they can still respond to **msgLabelSetString** correctly.

"The object figures out what to do" sounds like black magic, but it is actually not very complicated. You call a C routine to send a message to an object. Inside the Class Manager code, the Class Manager looks up that message in a table (created by the developer of the icon class) that specifies what function to call for different messages. If the message is in the table, the Class Manager then calls the icon's internal function which actually implements the message.

One benefit of using messages instead of function calls is that many different objects can respond to the same message. All objects that come from a common ancestor will usually respond to the messages defined by that ancestor. For instance, you can send **msgLabelSetString** to almost any object. (In some systems this is called "operator overloading.")

An object can respond to any message sent to it; the message does not have to be defined by the object's class or its ancestor class.

You can send any message to any object. Depending on whether it knows how to respond to the message, the object chooses what to do:

- If the object understands the message and can handle it, the object processes the message.
- If the object doesn't understand the message, it gives the message to its ancestor, to see if its ancestor knows how to handle the message (more on ancestors later in this section).
- If the object understands the message, but doesn't want to handle it, the object can ignore the message (by returning a non-error completion status), reject the message (by returning an error completion status), or give the message to its ancestor.

Classes Instead of Code Sharing

4.3

Icons and several other similar objects have titles. Thus, each of those objects that has a modifiable title must handle the "set string" message in some way or other.

In other programming methodologies, programmers take advantage of functional overlap by copying function code, trying to make data structures conform so the same routine can be used, or calling general routines from object-specific routines. However, whether you copy code or link with general routines, the resulting executable file contains a static copy of the shared code. The best you can hope for is shared code implemented by the system, which is rare.

In a class-based system, an object is an **instance** of a specific class. The class defines the data structures that are used by its instances, but doesn't necessarily describe the data in the structures (it is the data stored in these structures that differentiates each instance). The class also contains the functions that manipulate the object's data.

Each instance of a class contains the data for the specific thing being described (such as an icon). Each instance also knows to which class it belongs. Thus, there can be many **instances** of a class (and data for each instance), but the code for that class exists in only one place in the entire system.

If an existing class does almost everything you want, but not quite, you can create a new class that **inherits** its behavior from the existing class. The new class is said to be a **subclass** or **descendant** of its **ancestor** class. The subclass contains unique functionality that was not previously available in its ancestor.

The subclass should not reproduce anything that was defined by its ancestor. The subclass only defines the additional data structures required to describe the new thing and the functions required to handle messages for the new thing.

Of course, subclassing does not stop at one generation. The icon window class, for example, has eight ancestors between it and **clsObject**, which is the fundamental class for all classes in PenPoint.

Take a look at the PenPoint Class Hierarchy in the class hierarchy poster. Find the relationship between **clsIcon** and **clsLabel** (they're near the lower right edge).

Handling Messages

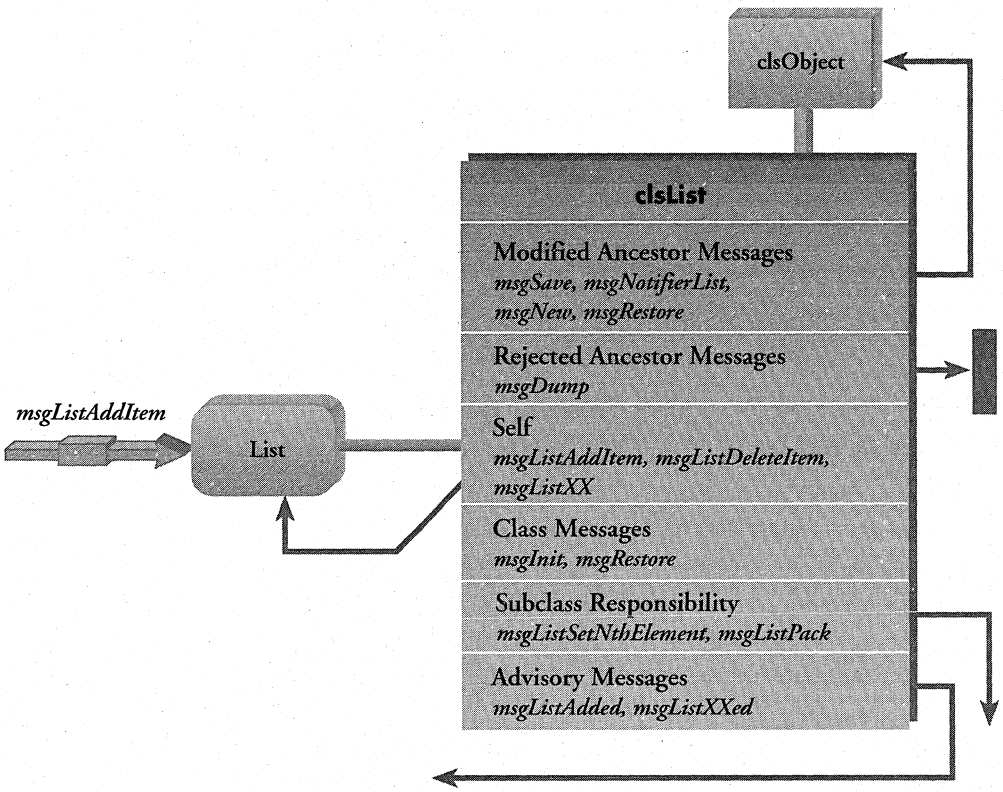
4.3.1

Remember that even when the ancestor handles the message, it uses the data for the object that initially received the message.

An object can send a message to its ancestor class either when it doesn't recognize the message or when it chooses to allow its ancestor class to handle the message.

Because the icon window class inherits from the window class, the icon window automatically responds to all the messages that a window responds to (such as **msgWinDelta** to move it or **msgWinSetVisible** to hide it) in addition to all the messages specific to an icon window (such as **msgIconSetPictureSize**). A class can override or change some of its ancestors' messages; for example, the icon window responds to **msgWinRepaint** by letting its ancestor label paint the string, then it draws its picture.

Figure 4-1
Message Handling by a Class and its Ancestors



By making it very easy to inherit behavior from existing classes, class-based systems encourage programmers to extend existing classes instead of having to write their own software subsystems from scratch. If you create a new kind of window, say an icon with a contrast knob, you can make it a descendant of

another class, and it will inherit all the behavior of that class, or as much behavior as you choose.

You may find it easier to understand class-based programming by viewing code instead of reading abstract explanations. The next few pages give some simple examples of using messages and classes, and even the very simplest program in the tutorial is—has to be!—fully class-based.

Sending a Message 4.4

In PenPoint, you usually send a message to an object using the **ObjectCall** function (if the object is owned by another process you use **ObjectSend**). The differences between **ObjectCall** and **ObjectSend** are detailed in *Part 1: Class Manager*, of the *PenPoint Architectural Reference.*

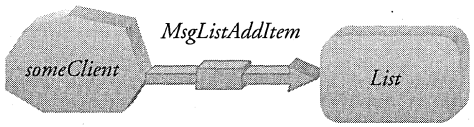
Here's a real-life example of sending a message. PenPoint provides a utility class, **clsList**, which maintains a list object. The messages that **clsList** responds to are documented in *Part 9: Utility Classes*, of the *PenPoint Architectural Reference* and in the **clsList** header file (\PENPOINT\SDK\INC\LIST.H). This is the definition of **msgListAddItemAt** from LIST.H:

```
/************************************************************************
    msgListAddItemAt    takes P_LIST_ENTRY, returns STATUS

    Adds an item to a list by position.
************************************************************************/
#define msgListAddItemAt                MakeMsg(clsList, 10)
```

Don't worry about the details of the definition right now; this just tells us that **msgListAddItemAt** is defined by **clsList**, that the message uses a P_LIST_ENTRY structure to convey its arguments, and that the message returns a value of type STATUS when it completes.

We want to send **msgListAddItemAt** to a list object, telling it to add the value 'G' to itself at position three in the list.

Figure 4-2
Sending msgListAddItemAt to a List



Message Arguments 4.4.1

Now, in order for a list object to respond appropriately to **msgListAddItem**, it's going to need some additional information. In this case the additional information is the item to add to the list ('G'), and where to add it (third postion). Most messages need certain information for objects to respond correctly

to them. The information, called **message arguments**, you pass to the recipient along with the message.

In this case, the header file informs us that **msgListAddItemAt** takes a P_LIST_ENTRY. In PenPoint's C dialect, this means "a pointer to a LIST_ENTRY" structure. Here's the structure:

The use of a weak word like "takes" is deliberate. Although a class usually requires a specific message argument structure, there is no mechanism available to detect when you pass it the wrong structure.

```
typedef struct LIST_ENTRY {

    U16         position;
    LIST_ITEM   item;

} LIST_ENTRY, *P_LIST_ENTRY;
typedef P_UNKNOWN LIST_ITEM, *P_LIST_ITEM;
```

U16 is an unsigned 16-bit number, P_UNKNOWN means a 32-bit pointer to an unknown. (Chapter 5 of this manual describes the rest of PenPoint's ubiquitous `typedefs` and `#defines`.)

When you can deliver the message and its arguments to a list object, you're set. Here's the C code to do it:

```
LIST        list;       // the object
LIST_ENTRY  add;        // structure for message arguments
STATUS      s;          // most functions return a STATUS value

// Add an item to the list:
//  1.  Assemble the message arguments;
add.position    = 3;
add.item        = (P_LIST_ITEM)'G';
//  2.  Now send the message and message arguments to the object.
if ((s = ObjectCall(msgListAddItemAt, list, &add)) != stsOK) {
    Debugf("add item failed: status is: 0x%lX", s);
}
```

ObjectCall Parameters

4.4.2

The code fragment above assumes that the list object (**list**) has already been created; object creation is covered later in this chapter. As you can see, **ObjectCall** takes three parameters:

The term "parameters" is used in function calls; the term "arguments" is used for data required for a specific message.

We use ***bold face*** to indicate items defined by PenPoint and other symbols used in examples.

- The message (**msgListAddItem**). Messages are just 32-bit constants defined by a class in its header file. You can send an object a message defined by any of the classes from which it inherits. (Some objects even respond to messages defined by classes that are not their ancestors.)
- The object (**list**). Objects are referenced by UID's, unique 32-bit ID numbers. UID's are discussed in more detail later.
- The arguments for the message (**add**). Not all messages take arguments (**msgFrameClose**, for example, takes none), but others do (**msgIconSetPictureSize**, for example, takes a width and height). The *PenPoint Architectural Reference* manual and the header files (in this case, \PENPOINT\SDK\INC\LIST.H) document each message's arguments.

ObjectCall has one 32-bit parameter for all the message's arguments; if a message takes more arguments than can fit in 32 bits, you must assemble the arguments in a structure and pass **ObjectCall** a pointer to the structure. In this case, **msgListAddItem** takes a P_LIST_ENTRY, a pointer to a LIST_ENTRY structure. (The PenPoint convention is that a type that begins with P_ is a pointer to a type.) Hence the address of the **add** structure (`&add`) is passed to **ObjectCall**.

Returned Values 4.4.3

The result of sending a message is returned as a status value (type STATUS). **stsOK** ("status OK") is zero. All status values that represent error conditions are less than zero. Note that STATUS is a 32-bit quantity, hence the %lX in the **Debugf** statement to print out a long hexadecimal.

Some messages are designed to return errors that you should test for. For example, the status returned by sending **msgIsA** to an object is **stsOK** if the object inherits from the specified class, and **stsBadAncestor** if the object does not.

Some objects respond to messages by returning a positive value (which is not a status value, but an actual number). Others return more complex information by filling in fields of the message argument structure supplied by the caller (or buffers indicated by pointers in the message argument structure) and passing back the structure.

How Objects Know How to Respond 4.4.4

The **list** object responds to **msgListAddItem** because it is an instance of **clsList**. But what does that mean?

The list object has several attributes. Among them are the class that created the object and the instance data for that object. As described above, when you define a class, you must also create a table of the messages handled by your class.

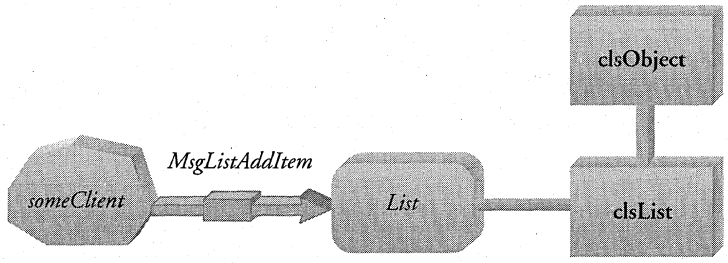
The Class Manager finds out which class created the object and looks for the method table for that class. The method table tells the Class Manager that the class has a function entry point for that message, so the Class Manager calls that function entry point, passing in the message and the message argument structure.

The Class Manager gives the object a pointer to the object's instance data. This is one aspect of PenPoint's data integrity.

Although the object receives the message, its class has the code to handle the message.

If the class decides to give the message to its ancestor, it passes the message and the message arguments to the ancestor (but the instance data is still the instance data for the object that received the message).

Figure 4-3
How Messages to Instances are Processed by Classes



Creating an Object 4.5

Where did the **list** object in the example above come from?

The short answer is that a client asked **clsList** to create an instance of itself by sending **msgNew** to **clsList**. In many ways this is no different than when we sent **msgListAddItem** to the list object in the previous example.

Classes and Instances 4.5.1

The longer answer involves understanding the relationship among classes and instances. In the section "Sending a Message" we discussed the fact that you send messages to objects and those objects respond to the messages. We also discussed how a class describes the data structures and the code used by its instances.

A class responds to **msgNew** by manufacturing an instance of itself. What is an instance? It is merely an identifier and the data structures that represent an object. Thus, the class asks the Class Manager to allocate the data structure and assign an identifier to the structure.

How can a class respond to a message? This is a fundamental concept and one that is hard to understand at first: a class is an object, just like any other PenPoint object. And just like any other PenPoint object, an object is an instance of a class. In the case of classes, all classes are instances of **clsClass**.

In other words, all classes are objects, but not all objects are classes.

You can think of classes as objects that know how to create instances.

When a client sends a message to a class, the class behaves like any other object and allows the class that created it (**clsClass**) to handle the message. **clsClass** contains the code that creates new objects.

When the object created by clsClass is an instance of clsClass, the new object **is** *a class.*

Thus, in answer to our original question about how did the list object come into being: a client sent **msgNew** to the object named **clsList**. **clsList** is an instance of **clsClass**, so the code in **clsClass** created a new object that is an instance of **clsList**.

An Alternative Explanation 4.5.2

At an implementation level, here's what actually happens.

The PenPoint Class Manager maintains a database of data structures; each data structure represents an object. The PenPoint Class Manager locates these objects

by 32-bit values, called UIDs (unique identifiers); UIDs are explained later in this chapter in "Identifying the New Object: UIDs." The data structure for each object contains some consistent information (defined by **clsObject**) that indicates the class to which the object belongs and other attributes for the object. Other information in the data structure varies from object to object, depending on which class created the object.

When a client sends a message to an object, the Class Manager uses the UID to locate the object. The Class Manager then uses the object's data structure to find the class that created the object. The Class Manager finds the class and uses the class's method table to find the entry point for the function that handles the message.

To create an object, the process works the same way. A client sends **msgNew** to a class object. The Class Manager locates the object, finds the class that created the object (**clsClass**), and calls the function in **clsClass** that creates new objects.

The _NEW Structure

4.5.3

You send **msgNew** to nearly every class to create a new instance of that class. In the case of **msgNew**, the message argument value is always a pointer to a structure that defines characteristics for the new object. This structure is commonly called the class's **_NEW structure** because the name of the structure is a variation of the class name, followed by _NEW. For **clsList**, the _NEW structure is LIST_NEW.

The exceptions are pseudo classes and abstract classes (see the Glossary).

For many classes, the _NEW structure is identical to the structure that contains the object's metrics.

The _NEW structure is mainly used to initialize the new instance. For example, when creating a new window you can give it a size and specify its visibility.

The _NEW structure differs depending on the class to which you send it. You can find the specific _NEW structure to use when creating an instance of a class by looking in the *PenPoint API Reference* manual or in the class's header file. For **clsList**, messages and message arguments are defined in \PENPOINT\SDK\INC\LIST.H. The _NEW structure is LIST_NEW. This excerpt comes from the LIST.H file:

```
typedef struct LIST_NEW_ONLY {
    LIST_STYLE      style;
    LIST_FILE_MODE  fileMode;       // Filing mode.
    U32             reserved[4];    // Reserved
} LIST_NEW_ONLY, *P_LIST_NEW_ONLY;

#define listNewFields       \
    objectNewFields         \
    LIST_NEW_ONLY           list;

typedef struct LIST_NEW {
    listNewFields
} LIST_NEW, *P_LIST_NEW;
```

Reading the _NEW Structure Definition 4.5.3.1

To read the _NEW structure definition, you need to perform the work that the compiler does in its preprocessor phase, expanding the macro definitions. The _NEW structures in the *PenPoint API Reference* have all been expanded for your convenience.

Start by looking for the `typedef` for the _NEW struct (`typedef struct LIST_NEW`) at the end of the example. The structure is represented by a `#define` name (in this case **listNewFields**).

Here's where it gets tricky; start thinking about inheritance. The `#define` name (**listNewFields**) has two parts:

- The `#define` name for the NewFields structure of the class's immediate ancestor (in this case, **objectNewFields**, which defines the arguments required by **clsObject**).
- A _NEW_ONLY structure for the class being defined (LIST_NEW_ONLY). The LIST_NEW_ONLY structure contains the actual **msgNew** arguments required for **clsList**.

Each subclass of a class adds its own _NEW_ONLY structure to the NewFields #define used by its immediate ancestor. This is how the _NEW structure for a class contains the arguments required by that class, by its ancestor class, by that class's ancestor, by that class's ancestor, and so on.

In this case, however, there is only one ancestor, **clsObject**. **objectNewFields** is defined in \PENPOINT\SDK\INC\CLSMGR.H:

```
#define objectNewFields     OBJECT_NEW_ONLY object;
```

OBJECT_NEW_ONLY is defined in the same file. It has many fields:

```
typedef struct OBJECT_NEW {

  U32   newStructVersion;              // Out: [msgNewDefaults] Validate msgNew
                                       // In:  [msgNew] Valid version
  OBJ_KEY                  key;        // In:  [msgNew] Lock for the object
  OBJECT                   uid;        // In:  [msgNew] Well-known uid
                                       // Out: [msgNew] Dynamic or Well-known uid
  OBJ_CAPABILITY           cap;        // In:  [msgNew] Initial capabilities
  CLASS                    objClass;   // Out: [msgNewDefaults] Set to self
                                       // In:  [msgObjectNew] Class of instance
                                       // In:  [msg*] Used by toolkit components
  OS_HEAP_ID               heap;       // Out: [msgNewDefaults] Heap to use for
                                       //  additional storage. If capCall then
                                       //  OSProcessSharedHeap else OSProcessHeap
  U32                      spare1;     // Unused (reserved)
  U32                      spare2;     // Unused (reserved)

} OBJECT_NEW_ONLY, OBJECT_NEW, * P_OBJECT_NEW_ONLY, * P_OBJECT_NEW;
```

Most elements in an argument structure are passed **In** to messages—you're specifying what you want the message to do. **Out** indicates that an element is set during message processing and passed back to you. **In:Out** means that you pass in an element *and* the message processing sets the field and passes it back to you.

A _NEW_ONLY for Each Class 4.5.3.2

Why such a complicated set of types? Thanks to class inheritance, when you create an instance of a class, you are also creating an instance of that class's immediate ancestor class, and that ancestor's ancestor class, and so on up the inheritance hierarchy to the root Object class. Each ancestor class typically allows the client to initialize some of its instance data. Many classes allow you to supply the **msgNew** arguments of their ancestor(s) along with their own arguments.

This is true for **clsList**: it inherits from **clsObject** (as do all objects) and part of its **msgNew** argument structure is the OBJECT_NEW argument structure for **clsObject**. **clsList** has three **msgNew** arguments of its own: how it should file the entries in the list, a list style, and a reserved U32.

These large message arguments structures are intimidating, but the good news is that by sending **msgNewDefaults**, you get classes to do the work of filling in appropriate default values. You then only need to change a few fields to get the new object to do what you want.

Identifying _NEW Structure Elements 4.5.4

As a class adds a _NEW_ONLY structure to a _NEW structure, it also gives a name to the _NEW_ONLY structure. From the **clsList** example, we can expand the LIST_NEW definition as:

```
typedef struct LIST_NEW {
    objectNewFields
    LIST_NEW_ONLY           list;
} LIST_NEW, *P_LIST_NEW;
```

The name list identifies the LIST_NEW_ONLY structure within the LIST_NEW structure with the name **list**. We can carry on the expansion to apply the definition of **objectNewFields**:

```
typedef struct LIST_NEW {
    OBJECT_NEW_ONLY         object;
    LIST_NEW_ONLY           list;
} LIST_NEW, *P_LIST_NEW;
```

You can see now, when you create an identifier of type LIST_NEW, you can specify the _NEW_ONLY structures by specifying their names. For example, if your code contains:

```
LIST_NEW myList;
```

You can refer to the LIST_NEW_ONLY structure by **myList.list**, and the OBJECT_NEW_ONLY structure by **myList.object**.

Code to Create an Object 4.5.5

This example code creates the list object to which we sent a message in the first code fragment. Later code will show how the list class is itself created.

The preceding discussion mentioned that the client sends **msgNew** to a class to create an instance of the class. The function parameters used in **ObjectCall** for

msgNew are the same as before (the object to which you send the message, the message, and the message argument value).

As we have seen, the _NEW structure can get quite large (because most subclasses add their own data fields to the _NEW structure). Many classes have default values for fields in the _NEW structure, yet clients must be able to override these defaults, if they want.

To initialize the _NEW structure to its defaults, clients must send **msgNewDefaults** to a class before sending **msgNew**. **msgNewDefaults** tells a class to initialize the defaults in the _NEW structure for that class. After **msgNewDefaults** returns, the client can modify any fields in the _NEW structure and then can call **msgNew**.

```
    LIST    list;           // Object we are creating. */
    LIST_NEW    new;        // Structure for msgNew arguments sent to clsList.
    STATUS  s;

// Initialize _NEW structure (in new).
ObjCallRet(msgNewDefaults, clsList, &new, s);

//  Modify defaults as necessary...
new.list.fileMode = listFileItemsAsData;

//  Now create the object by sending msgNew to the class.
ObjCallRet(msgNew, clsList, &new, s);
// The UID of the new object is passed back in the _NEW structure.
list = new.object.uid;
```

Status values less than stsOK indicate errors.

Because almost every message returns a status value (to say nothing of most function calls), your code tends to become littered with status checking. Hence \PENPOINT\SDK\INC\CLSMGR.H defines several macros to check for bad status values. This fragment uses one of those macros, **ObjCallRet**. **ObjCallRet** does a standard **ObjectCall** with its first three parameters, and assigns the return value to its fourth. If the returned value is less than **stsOK**, **ObjCallRet** prints a warning (when compiled with the DEBUG flag) and returns the value to the caller of the function. There are many other macros of a similar nature; they are documented in *Part 1: Class Manager* of the *PenPoint Architectural Reference.*

Identifying the New Object: UIDs

4.5.6

When you send **msgNew** to a class, the message needs to give you an identifier for the new object (so your code can use it). As mentioned above, messages often pass back values in the structure that contains the message arguments. In this case, **clsObject** passes back the UID of the newly created object in its OBJECT_NEW structure (in **object.uid**).

In our code example, the UID for the new object was passed back in **new.object.uid**. The sample copied the value to the object named **list**, and henceforth uses list when refering to the new list object.

You refer to objects using **UID**s. A UID is a 32-bit number used by the Class Manager to indicate a specific PenPoint object. An object's UID is *not* a C pointer; it consists of information used by the Class Manager to find an object

and information about the object's class and other things. The symbol **list** in this example is the UID of our list object; **clsList** is the UID of the list class.

PenPoint defines many classes that clients can use to create instances for their own use (such as the list class, the window class, and so on). All of these built-in classes are depicted in the class hierarchy poster.

When a client sends **msgNew** to a class to create a new object, the class is identified by a unique value. If an application knows this value and the class is loaded in PenPoint, the application can create an instance of the class. This value is called a **global well-known UID**.

There are other types of UIDs: local well-known UIDs and local private UIDs. There are no global, private UIDs.

The global well-known UIDs of all the public PenPoint classes, including **clsList**, are defined in \PENPOINT\SDK\INC\UID.H. Because all PenPoint programs include this header file when they are compiled, all programs know about these classes.

clsList is defined with this line in UID.H:

```
#define clsList                    MakeWKN(10,1,wknGlobal)
```

MakeWKN (pronounced "Make well-known") is a macro that returns a 32-bit constant. Here the parameters to **MakeWKN** mean "create a well-known UID in global memory for version 1 of administered ID 10." No other well-known UID uses the number 10.

Eventually, when you finalize your application, you will need to define your own well-known UIDs. To get the administered number, contact GO Developer Technical Support; they will assign you a specific administered value.

Until that time, you can use some spare UIDs, defined in \PENPOINT\SDK\INC\UID.H, for this purpose. These UIDs have the values **wknGDTa** through **wknGDTg**.

Creating a Class

4.6

You have seen how to send a message to an object and how to send **msgNew** to a class to create a new object. You use the same procedure to create any object and send it messages, so you can send messages to any instance of any class in PenPoint.

The last step is to create your *own* classes for your application. At the very least you must create a class for your own application; frequently, you will also create special window classes and data objects that draw and store what you want.

Creating a class is similar to creating an instance, because in both cases you send **msgNew** to a class. When you create a class, you send **msgNew** to **clsClass**. This is the class of classes. Remember that a class is just an object that knows how to create instances of itself; in this case **clsClass** knows how to create objects which themselves can create objects.

In short, to create a class, you send **msgNew** to **clsClass**, and it creates your new class object. A routine much like this in the PenPoint source files creates **clsList**; it is executed when the user boots PenPoint (when the SYSUTIL.DLL is loaded).

Some classes, such as clsList are created at boot time; other classes are created later, such as at application installation.

```
/*****************************************************************************
    ClsListInit

    Install clsList
*****************************************************************************/
STATUS ClsListInit (void)
{
    CLASS_NEW   new;
    STATUS      s;


    ObjCallRet(msgNewDefaults, clsClass, &new, s);
    new.object.uid          = clsList;
    new.class.pMsg          = (P_MSG) ListMethodTable;
    new.class.ancestor      = clsObject;
    new.class.size          = SizeOf(P_UNKNOWN);
    new.class.newArgsSize   = SizeOf(LIST_NEW);
    ObjCallRet(msgNew, clsClass, &new, s);

    return stsOK;

}   // ClsListInit
```

New Class Message Arguments

4.6.1

The important thing, as always, is the group of message arguments. Here the message is **msgNew**, just as when we created the list object; because we are sending it to a different class, the message arguments are different. When sent to **clsClass**, **msgNew** takes a pointer to a CLASS_NEW structure. Like LIST_NEW, CLASS_NEW includes the arguments to OBJECT_NEW as part of its message arguments. Briefly, the CLASS_NEW message arguments are:

- The same OBJECT_NEW arguments used by other objects—a lock, capabilities, a heap to use (and a UID field in which the Class Manager returns the UID of the object).
- The **method table** (**new.class.pMsg**) which is where you tell the class which functions handle which messages. You must write the method table. This is the core of a class, and is discussed in great detail in the next section.
- The **ancestor** of this class (**new.class.ancestor**). The Class Manager has to know what the class's ancestor is so that your class can inherit behavior from it, that is, let the ancestor class handle some messages. In this case, **clsList** is an immediate descendant of **clsObject**.
- The size of the data needed by instances of the class (**new.class.size**). The Class Manager needs the information to know how much room to allocate in memory when it creates a new instance of this class.
- The size of the structure that contains information used to create a a new instance of the class (**new.class.newArgsSize**)

For a list, the instance data is just a pointer to the heap where it stores the list information, hence the size is (SIZEOF) sizeof(P_UNKNOWN). For other objects, the instance data may include a lot of things, such as window height and width, title, current font, etc. Note that an object has instance data for each of the classes it is an instance of—not just its immediate class, but that class's ancestor, and that ancestor's ancestor, and so on.

P_UNKNOWN is the typedef used in PenPoint for a pointer to an unknown type.

The instance data size must be a constant! If, say, a title string is associated with each instance of your class, then you need either to have a (small) fixed-size title or to keep the string separate and have a pointer to it in the instance data.

Important! Instance data size must be a constant.

Method Tables

4.6.2

Nearly all classes respond to messages differently than their ancestors do—otherwise, why create a new class? As a class implementer, you have to write **methods** to do whatever it is you want to accomplish (such as maintain a list, draw an icon, and so on) in response to a particular message.

Some classes exist just to define a set of messages; the implementation of those messages is up to its descendants.

In PenPoint, a method is a C function, called a **message handler**. The terms message handler and method are used interchangably.

When a client sends a message to an instance of your class, you want the Class Manager to call the message handler that is appropriate for that message. You tell the Class Manager what to do with each message through a **method table**.

A method table is simply a mapping that says "for message **msgSomeMsg**, call my message handler **MyFunction**." You specify the table as a C array in a file that is separate from your code (you must compile it with the method table compiler, described below). A method table file has the extension .TBL. Each class has its own method table; however, a single method table file can have method tables for several classes. At the end of the file is a class info table that maps a class to the method table for that class. There must be an entry in the class info table for each method table in the file. The file looks something like this:

```
MSG_INFO clsYourClassMethods[] = {
    msgNewDefaults,     "myClassNewDefaults",   objCallAncestorBefore,
    msgSomeMsg,     "MyFunction",       flags,

    0,
};
CLASS_INFO classInfo[] = {
    "clsYourClass", clsYourClassMethods,    0,
    0
};
```

The quotation marks around the messages and classes are required. You can tell the Class Manager to call your ancestor class with the same message before or after calling your function by setting flags in the third field in the method table (the third field in the CLASS_INFO table is not currently used and should always contain 0).

Identifying a Class's Message Table

4.6.2.1

To convert the method table file into a form the Class Manager can use, you compile the table file with the C compiler, then run the resulting object through the Method Table compiler (\PENPOINT\SDK\UTIL\CLSMGR\MT.EXE). This turns it into a .OBJ file which you link into your application.

The most important argument you have to pass to **msgNew** when creating a class is a pointer to this method table (**new.class.pMsg** in the code fragment above). When you create the class, you set **new.class.pMsg** to *clsYourClass*.

When an object is sent a message, the Class Manager looks in its class's method table to see if there is a method for that message. If not, the Class Manager looks in the class's ancestor's method table, and so on. If the Class Manager finds a method for the message, it transfers execution to the function named in the method table.

When the Class Manager calls the function named in the method table, it passes the function several parameters:

- The message sent (**msg**)
- The UID of the object that originally received the message (**self**)
- The message arguments (**pArgs**). The Class Manager assumes that the message arguments are a pointer to a separate message arguments structure).
- Internal context the Class Manager uses to keep track of classes (**ctx**)
- A pointer to the instance data of the instance.

Self

4.6.3

Self is the UID of the object that received the message.

As we discussed before, when an object receives a message, the class manager first sees if the object's class can handle the message, then it passes the message to its ancestor, which passes the message to its ancestor, and so on. However, the data that each of those classes work on is the data in the object that first received the message (which is identified by self). This is fundamental to understanding object-oriented programming in PenPoint: calling ancestor makes more methods available to the data in an object, it doesn't add any new data.

Of course, each ancestor deals with only the parts of the object data that it knows about; an ancestor can't modify a structure defined by its descendant.

A second fundamental concept is that an ancestor may need to make a change to the data in the object. However, rather than making the change immediately by calling a function, the ancestor sends a message to **self** to make the change. Be careful not to get pulled into the semantic pit here, **self** means the object that received the original message, not the ancestor class handling the message. (Remember that the ancestors only make more functions available; not more data.)

Because the message is sent to **self**, **self**'s class can inspect the message and choose whether it wants to override the message or allow its ancestor to handle it. Each ancestor inspects the message and can either override the message or pass it to its ancestor. This continues until the ancestor that sent the original message receives

the message itself and, having given all of its decendents the opportunity to override the message, now handles the message itself (or even passes the message to its ancestor!).

Possible Responses to Messages

4.6.4

Here are some of the flavors of responses you can make to a message in a message handler:

- Do something before and/or after passing it to the ancestor class. This might include modifying the message arguments, sending self some other message, calling some routine, and so on. This means that the class will respond to the message differently than its ancestor.
- Do something with the message, but *don't* pass the message to the ancestor class. This is appropriate if the message is one you defined, because it will be unknown to any ancestor classes. If the message is one defined by an ancestor, this response means that you're blocking inheritance, which is occasionally appropriate.
- Do nothing but return some status value. This blocks inheritance, and means that it's up to descendant classes to implement the message. This is not as rare as it sounds; many classes send out **advisory messages** informing their instances or other objects that something has happened. For example, **clsWindow** sends self the message **msgWinSized** if a window changes size. This is useful for descendant classes which need to know about size changes, but **clsWin** itself doesn't care.

When such a message is new to a class (no ancestor), it is called an **abstract message**.

What messages does your message handler have to respond to? It usually ought to respond to all the messages specific to your class which you define—no other ancestor class will! Ordinarily an instance of each class has its own data, so most classes intercept **msgNew** to execute a special initialization routine; if there are defaults for an instance's data, the class will also respond to **msgNewDefaults**. Most classes should also respond to **msgFree** to clean up when an instance is destroyed.

Here is **clsList**'s method table.

```
//
//  Include files
//
#include <LIST.H>       // where the msgs are defined
MSG_INFO ListMethods [] =
{
    /* clsObject methods */
    msgNewDefaults, "ListNewDefaults", 0,
    msgInit, "ListInit", objCallAncestorBefore,
    msgFree, "ListMFree", 0,
    msgSave, "ListSave", objCallAncestorBefore,
    msgRestore, "ListRestore", objCallAncestorBefore,
```

```
    /* clsList methods */
    msgListFree,        "ListMFree", 0,
    msgListAddItem,     "ListAddItem", 0,
    // Functions for the rest of clsList's messages.
    ...,
    ...,
    0
};
CLASS_INFO classInfo[] =
{
    "ListMethodTable", ListMethods, 0,
    0
};
```

Note that **clsList** responds to most intercepted messages by calling an appropriate function (**ListInit**, **ListMFree**, and so on). The functions that implement the various list messages are not printed here; indeed, external code should never call routines internal to a class. One of the goals of object-oriented programming is to hide the implementation of a class from clients using the class.

Chapter 5 / Developing an Application

Thus far, we have described PenPoint and PenPoint applications from a conceptual point of view. By now you should understand how PenPoint differs from most other operating systems and what the PenPoint Application Framework and class manager do for you.

With this chapter we start to address what you, as a PenPoint application developer, have to do when writing PenPoint applications.

- The first section describes many of the things that you have to think about when designing an application.
- The second section describes some of the things that you have to consider when designing an application for an international market.
- The third section describes the functions and data structures that you will create when you write an application.
- The fourth section describes the cycle of compiling, and linking that you will follow when developing an application.
- The fifth section provides a checklist of things that you must do to ensure that your application is complete.
- The sixth and following sections describe the coding standards and naming conventions used by GO. Included in these sections is a discussion of some of the debugging assistance provided by PenPoint.
- The last section describes the tutorial programs provided with the SDK.

Designing Your Application 5.1

When you design a PenPoint application, there are several separate elements that you need to design:

- The user interface
- The classes
- The messages
- The message handlers
- The program units

This section points out some of the questions you must ask yourself when designing an application. This section does not attempt to answer any of the questions; many answers require a good deal of explanation, and many decisions involve your own needs.

Just read this section and keep these questions in mind as you read the rest of the manual.

Designing the User Interface 5.1.1

The most obvious part of a PenPoint application is the user interface. Almost as soon as you determine what your application will do, you should begin to consider your user interface.

Your user application should be consistent with the PenPoint user interface, which is described in detail in the *PenPoint User Interface Design Guidelines.*

Designing Classes 5.1.2

PenPoint provides a rich set of classes that can do much of the work for your application. Your task is to decide which of these classes will serve you best. The *PenPoint Architectural Reference* describes the PenPoint classes and what they can provide for you.

If the classes provided by PenPoint don't do exactly what you need, you should look for the class that comes closest to your needs, then create your own class that inherits behavior from that class.

Designing Messages 5.1.3

After determining that you need to create your own class, you need to decide what messages you need. Usually you add new messages to those already defined by your class's ancestors.

However, the real trick to subclassing comes when you decide how to handle the messages provided by your class's ancestors. If you do not specify how your class will handle your ancestors' messages, the PenPoint class manager sends the messages to your immediate ancestor, automatically. If you decide to handle an ancestor message, you then need to decide when your ancestors handle the message, if at all. Do you:

- Call the ancestor before you handle the message?
- Call the ancestor after you handle the message?
- Handle the message without passing it to your ancestor at all (thereby overriding ancestor behavior)?

Designing Message Handlers 5.1.4

After determining the messages that you will handle, you then need to design the methods that will do the work for each of the messages. In considering the methods and the information they need, you will probably start to get an idea of the instance data that your class needs to maintain.

Designing Program Units 5.1.5

When you understand the classes that you require, you should consider how to organize your classes and their methods into program units. The common approach used in our sample code is to place the source for each class into a separate file.

You should consider whether a class will be used by a number of different applications or used by a single application. If the class can be used by more than one application (such as a calculator engine), you should compile and link it into a separate DLL (dynamic link library). Each application tells the installer which DLLs it needs at install time. The installer then determines whether the DLL is present or not. If not, it installs the DLL.

Designing for Internationalization and Localization 5.2

PenPoint 2.0 will contain support for applications that are written for more than one language or region. The process of generalizing an application so that it is suitable for use in more than one country is called **internationalization**. Modifying an application so that it is usable in a specific language or region is called **localization**.

PenPoint 1.0 already includes many features that will be used to support internationalization. For example, PenPoint 1.0 uses PenPoint resource files to store its text strings. When localizing to a specific language, a different resource file will be created that contains text strings in that language.

There are two aspects to the changes implied by PenPoint 2.0. The first is making your application port easily to PenPoint 2.0. The second is internationalizing your application.

Preparing for PenPoint 2.0 5.2.1

PenPoint 2.0 will incorporate some major changes that will cause applications compiled for PenPoint 1.0 to be incompatible with PenPoint 2.0. The data created by 1.0 applications should still work under 2.0, and properly writtern 1.0 applications should be portable to PenPoint 2.0 with nothing more than a recompilation.

This section describes how to write your PenPoint 1.0 application so that it will be portable to PenPoint 2.0. Using these guidelines does *not* mean that you will have internationalized your application! Internationalization and Localization are much larger issues, and are dealt with elsewhere. These instructions are intended only to make it easier for you to port your American English application to PenPoint 2.0.

The biggest change is that PenPoint 1.0 uses the ASCII character set, while PenPoint 2.0 uses Unicode. ASCII is an 8-bit character set; Unicode is a 16-bit character set. This affects character types, string routines, quoted strings,

Character Types 5.2.1.1

PenPoint provides three character types: CHAR8, CHAR16, and CHAR. The first two provide eight and sixteen bit characters, respectively. In PenPoint 1.0, the plain CHAR type is 8 bits long; in PenPoint 2.0, CHAR is 16 bits long. You need to convert all of your character data to use the CHAR type, except where you know the size you'll need will be the same under PenPoint 1.0 and PenPoint 2.0 (for example, in the code that saves and restores data).

Any places where you depend on a CHAR having a small value, you should rethink the problem. For example, if you currently translate a character by indexing 256-element array (`CHAR array[sizeof(CHAR)]`), you probably won't want to use the same strategy when `sizeof(CHAR)`, and therefore the size of your array, is 65,536.

Any places where you depend on `sizeof(CHAR)` being one byte, you need to change the value.

String Routines 5.2.1.2

All of the familiar C string routines (`strcmp`, `strcpy`, and so on) will still exist in PenPoint 2.0, and they will still work on 8-bit characters. The INTL.H header file in PenPoint 1.0 defines a new set of string routines (named `Ustrcmp`, `Ustrcpy`, and so on) that perform the equivalent functions on 16-bit Unicode characters.

In PenPoint 1.0, the U... functions are identical to their 8-bit namesakes. In PenPoint 2.0, they will be true 16-bit routines. In other words, the old routines only work on CHAR8 strings, the U... routines in 1.0 work on CHAR8 strings, in 2.0 the U... routines will work on CHAR16 strings. If you use the U... versions and CHAR strings now, you will not have to change anything at 2.0.

You should convert all your string routines to the U... version wherever you are converting to CHAR strings.

Character and String Constants 5.2.1.3

When you use CHAR8, you can use standard C conventions for forming character and string constants. That is:

```
CHAR8 *s = "string";
CHAR8 c = 'c';
```

When you use the CHAR16 type, you must preceed the character or string constant with the letter L, which tells the compiler you are using a 16-bit (or long) character, as in:

```
CHAR16 *s = L"string"
CHAR16 c = L'c'
```

When you use the CHAR type, you must preceed the character or string constant with the identifier "U_L", which means UNICODE, long. In PenPoint 1.0, this tells the compiler to use 8-bit characters; in PenPoint 2.0, this tells the compiler to use 16-bit characters.

```
CHAR *s = U_L"string";
CHAR c = U_L'c';
```

Debugging 5.2.1.4

Debugf() and related routines will continue to take ordinary 8-bit strings. This means that all debugging output will continue to be ASCII based.

Versioning Data 5.2.1.5

Under PenPoint 2.0, you will still want your application to be able to unfile any data it filed under 1.0. That is, although your users will have to install a new version of your product, you don't want them to have to throw away anything they created with it!

When you respond to **msgRestore**, check the filed version. If the version number is less than the defined value **penpointRev2_0**, read it into a structure that uses explicit CHAR8 where required, then copy it into your instance data. If the version number is greater than or equal to **penpointRev2_0**, your saving and restoring code should remain the same (and use **CHAR** types).

Preparing for Internationalization 5.2.2

PenPoint 1.0 does not contain all the messages, functions, and tools that you will need to internationalize your application. However, there are several facilities available in PenPoint 1.0 that you can use right now to reduce the work needed to internationalize. This section lists these facilities.

Move Strings into Resource Files 5.2.2.1

You should move as many of your text strings into resource files as possible.

When text strings are hard-coded into your application, they are very difficult to translate and do not allow users to change language dynamically.

However, if you move your application's text strings into resource files they are easy to translate and allow users to change language simply by substituting one resource file for another.

If you use the StdMsg facility for displaying dialog boxes, error messages, and progress notes, your text strings are already in resource files. The positional parameter facility provided with StdMsg and the compose text string routines do not depend on the order of replaceable values in the function parameters. These functions are unlike `printf`, where the order of the function parameters is directly related to the order of replaceable values in the string. When you use StdMsg or compose text, the function parameters are always in the same order,

but your string can use them in the order dictated by the national language in which you are writing.

Identify and Modularize Code that Varies with Locale 5.2.2.2

When internationalizing an application, moving its text strings to resource files allows users to change the language, but in order to support another language, parts of your application code must be equally replaceable. For example, when sorting characters in another language, you must be prepared to handle different sort sequences.

In 2.0, GO will provide a number of **services** to perform functions that vary by language. Under consideration are routines that provide sorting, number formatting, number scanning, numbers with units, times, and dates (input and output), character comparisons, character conversions, spell checking and so on.

The PenPoint Services Architecture enables you to create functions that users can install and activate whenever they choose. For instance, users can install several different printer drivers, but they only make one driver current at a time. Similarly, users will be able to install several different sort engines and choose one to use with the current language.

Right now, you can start to identify language-dependent routines, such as text manipulation, of your own. You can flag these routines and move them into separate modules.

Part 13: Writing PenPoint Services in the *PenPoint Architectural Reference* describes how to create your own services. If you make your language-dependent functions into services in PenPoint 1.0, the change to 2.0 will be much easier.

New Text Composition Routines 5.2.2.3

The file CMPSTEXT.H contains **ComposeText** routines for assembling a composite string out of other pieces. Use these routines to create strings in your UI — *don't use* `sprintf`! The **ComposeText** routines will also save you effort because you can specify the resId of a format string and the code will read it from the resfile for you. You can, of course, give the format string directly to the routines.

Development Strategy 5.3

Where do you start writing an application?

The PenPoint Application Framework provides so much boilerplate work for you, it is very easy to create applications through incremental implementation. You start with an empty application, that is, one that allows the Application Framework to provide default handling of most messages. Then, one by one, you add new objects and classes to the application, testing and debugging as you go.

As we shall see in Chapter 6, the PenPoint SDK includes sources for an empty application, Empty App. You can copy, compile, install, and run Empty App.

This section describes the fundamental parts of PenPoint applications. These are the parts that you will probably work on first. They are also the parts you will return to many times to modify.

Application Entry Point 5.3.1

All PenPoint applications must have a function named **main**, which is the entry point for an application. When the application is installed, **main** creates the application class and can create any other private classes required by all instances of the application.

Application Instance Data 5.3.2

In PenPoint, objects that are instances of the same class share the same code. For example, if there are two insertion pads visible on the screen, they are both running same copy of the insertion pad class code, but each instance of the insertion pad has different instance data.

As soon as your application has data that can be different for each of its documents, your application needs to maintain instance data.

What do you save in instance data?

The most common use of instance data is to save identifiers for objects created by your application. The PenPoint object-data model suggests that any time you have data, you should use a class to maintain that data.

When your application class has instance data, it must be prepared to respond to **msgInit** by initializing values in the instance data (if needed).

Any class with instance data must respond to **msgInit** in the same way.

Creating Stateful Objects 5.3.3

Stateful objects contain data that must be preserved when a document is not active.

You can do some interesting things with an application that uses only the behavior provided by the Application Framework. However, soon after you start developing an application, you will want the application to be able to save and restore data when the user turns away from and turns back to its documents. To save and restore documents, you need to create, save, and restore stateful objects.

Usually an application's instance data contains some stateful objects and some non-stateful objects.

If your application class has stateful objects, you must be prepared to handle:

msgAppInit by creating and initializing the stateful objects required by a new document. Your application can create additional stateful objects later.

msgSave by saving all stateful objects to a resource file.

msgRestore by restoring all stateful objects from a resource file.

Displaying on Screen 5.3.4

Most applications need to display themselves on screen. The PenPoint Application Framework provides access to the screen by creating a frame object.

When your application receives **msgAppOpen**, it should create the remaining non-stateful objects that it needs to display on screen, and then should display itself in the frame provided by the application framework.

When your application receives **msgAppClose**, it should remove itself from the frame and destroy all of its non-stateful objects.

Creating Component Classes 5.3.5

If you create new component classes that can be shared by a number of different applications (or other components), you usually define the component classes in a DLL file.

As an application executable file must have a function named **main**, a DLL file must have a function named **DLLMain**. **DLLMain** creates the component classes defined in the DLL.

Development Cycles 5.4

The compile, install, test, and debug cycle in PenPoint is similar to the development cycle for most other operating systems. This section briefly describes the steps involved in the development cycle. Later sections cover these steps in greater detail.

Compiling and Linking 5.4.1

There are several types of files used to compile and link PenPoint applications. These files include:

- The WATCOM make file
- The application's method table files
- The application's C source and header files
- The linker command file for the application
- The PenPoint SDK header and library files.

Method Table Files 5.4.1.1

You create a method table file to equate the messages handled by your class to a function defined in your source. You create one method table per class, but one method table file can contain several method tables.

You compile the method table and then compile the resulting intermediate object file with the PenPoint method table compiler, MT. This produces:

- A header file that you use when you compile your C source
- An object file that you use when you link your application.

C Source and Header Files 5.4.1.2

PenPoint applications are written in the C language; the object oriented extensions are provided through standard C function calls. The source for each class (application or component) is maintained in a separate file.

Following normal C programming practice, it is advisable to define your symbols, structures, macros, and external declarations in one or more header (.H) files.

Linker Command Files 5.4.1.3

PenPoint applications compiled for the Intel 80386 processor use the protected mode features of the chip. To produce a protected mode object file, you must link your application with the WATCOM OS/2 linker. The OS/2 linker requires additional commands that specify internal names and memory requirements for the resulting executable (.EXE) or dynamic link library (.DLL) files.

These commands are placed in the command files used by the linker. You can either build the command files separately or you can build the command files dynamically in your application's makefile. The sample applications in \PENPOINT\SDK\SAMPLE all build their linker command files dynamically.

PenPoint SDK Files 5.4.1.4

The PenPoint SDK header and library files are in the directories \PENPOINT\SDK\INC and \PENPOINT\SDK\LIB, respectively.

You should include these directories in your compiler and linker search paths.

Installing the Application 5.4.2

One difference between PenPoint and most other operating systems is that once you have compiled an application, you must install the application into PenPoint before you can use it. There is no "run" command in PenPoint, so you must use the Notebook to transfer control to the application.

Additionally, all application code in PenPoint is shared. PenPoint must know where your application code is installed so that all instances of your application use the same code.

There are two ways to install an application into PenPoint:

- Install when you boot PenPoint
- Install explicitly with the PenPoint application installer

You can install an application when you boot PenPoint by adding your application's PenPoint name to your APP.INI file.

You can explicitly install a PenPoint application by running the PenPoint application installer (found in the Connections and Settings notebooks).

You can use the connections notebook to tell PenPoint to display the installable applications (or any other installable items) whenever a volume becomes available.

Debugging 5.4.3

There are a number of tools available to you to aid in debugging. Among them are:

- Using **Debugf** or **DPrintf** statements to send text to the debugger stream. You can use a second monitor or the system log application to view the debugger stream. You can also save the debugger stream in a log file. The **Debugf** and **DPrintf** statements are described later in this Chapter. The system log application is described in *PenPoint Development Tools.*
- Using the PenPoint source debugger (DB) to debug your application. The debugger is described in *PenPoint Development Tools.*
- Handling **msgDump**. **msgDump** requests an object to format its instance data and send it to the debugger stream. While developing an application, you can send **msgDump** to any object whose state is questionable. From the PenPoint source debugger, you can use the **od** command to send **msgDump** to an object. It is not a good idea to send **msgDump** in production code.

A Developer's Checklist 5.5

When your PenPoint application does what you want it to, you can stop and move on to your next project. However, PenPoint applications are far more useable when they can interact with the PenPoint operating system and other applications. There is such a wealth of interaction that it is easy to omit some behavior from your application.

This section presents two checklists. The first checklist details all the interactions that you should include in your PenPoint application, starting at the fundamental Application Framework interactions. The second checklist lists the interactions that you should consider adding to your application to improve its appearance or usability.

Checklist Of Required Interactions 5.5.1

You should use this checklist to ensure that your application is complete. The items in the checklist point to parts of this manual and the *PenPoint Architectural Reference* where the item is described in detail.

- ☐ Handle application class installation (in **main** when **processCount** equals 0)
  - ☐ create the application class
  - ☐ create any private classes used by the application class
- ☐ Handle application object instantiation (in **main** when **processCount** is greater than 0)
  - ☐ create an instance of your class
  - ☐ create any private classes required by an instance of your application class
  - ☐ create any other objects required at the time

- ☐ Create and display windows
  - ☐ insert yourself into frame on **msgAppOpen**
  - ☐ remove yourself from frame on **msgAppClose**.
- ☐ Handle application termination.
  - ☐ respond to **msgFree** protocol
- ☐ Handle application deactivation or deinstallation (**msgAppTerminate**).
- ☐ Handle **msgDump**
- ☐ Handle **msgSave**
  - ☐ save data
  - ☐ save objects
- ☐ Handle **msgRestore**
  - ☐ restore data
  - ☐ restore objects
  - ☐ observe objects
- ☐ Handle input
  - ☐ handle selection protocol
- ☐ Respond to Printing messages

Checklist of Non-Essential Items 5.5.2

Use this checklist to ensure that you have considered all possible non-essential additions to your application. The items in the checklist point to parts of this manual and the *PenPoint Architectural Reference* where the item is described in detail.

- ☐ Add menus to SAMs
- ☐ Handle Option sheet protocols
  - ☐ Create an option sheet
  - ☐ Create application-specific option cards
- ☐ Allow Application Embedding
- ☐ Respond to move/copy protocol
- ☐ Handle document import and export
- ☐ Handle Undo
- ☐ Respond to traversal protocols
- ☐ Define document icons
- ☐ Create Stationery
- ☐ Create Help notebook files
- ☐ Create Quick Help Resources

GO's Coding Conventions 5.6

At GO, we have developed techniques to make PenPoint code easier to write, understand, debug, and port. Some of our techniques are stylistic conventions, such as how variable and function names should be capitalized. Others fall under the category of extensions to C, including a suite of basic data types that are compiler and architecture independent. This section describes:

- The conventions that GO code follows
- The global types, macros, constants, and constructions provided in PenPoint
- PenPoint's global debugging macros and other functions that we have found useful to diagnose program errors.

While we would be delighted for you to follow all of our conventions, we obviously do not expect every developer to do so. Conventions are a matter of taste, and you should follow a style that is comfortable to you. However, we do recommend that you make use of our extensions. They will help make your code easier to debug and port. Also, by describing our style, we hope to make it easier for you to understand our header files and sample code.

Typedefs 5.6.1

All `typedefs` are CAPITALIZED and use the underscore character to separate words.

```
typedef unsigned short  U16;
typedef U16                 TBL_ROW_COUNT;
```

Pointer types have the prefix `P_`.

```
typedef unsigned short  U16, * P_U16;
typedef TBL_ROW_COUNT       *P_TBL_ROW_COUNT;
```

In structure definitions, the name of the structure type is also the structure tag.

```
typedef struct LIST_ENTRY {

    U16         position;
    LIST_ITEM   item;

} LIST_ENTRY, *P_LIST_ENTRY;
```

The tag name is used by the PenPoint Source-level Debugger.

Variables 5.6.2

Variable names are mixed case, always starting with a lowercase letter, with capitialization used to distinguish words. Variable names do not normally include underscore characters.

```
U16     numButtons;
```

Pointer variables are prefixed the name with a lowercase p. The letter following the p is capitalized.

```
P_U16       pColorMap;
```

Functions 5.6.3

Functions are mixed case, always starting with a capital letter, with capitialization used to distinguish words. Function names do not normally include underscore characters.

Function names often use a Noun-Verb style. The verb is what the function does, the noun is the target of the function's action.

```
TilePopUp();
PenStrokeRetrace();
```

However, the main function is simply **main**.

Defines (Macros and Constants) 5.6.4

Defines follow the same capitalization rules as variables and functions. Macros follow the rules for function names (mixed-case, first letter uppercase) and constants follow the rules for variable names (mixed-case, first letter lowercase).

```
#define OutRange(v,l,h)  ((v)<(l)||(v)>(h))
#define maxNameLength        32
#define nameBufLength        (maxNameLength+1)
```

Class Manager Constants 5.6.5

You use several special kinds of constants when writing Class Manager code:

- Class names
- Well-known objects
- Messages
- Status values.

Class Names 5.6.5.1

Class names start with **cls** followed by the name of the class: **clsList**, **clsScrollBar**, and so on.

Well-Known Objects 5.6.5.2

Pre-existing objects in PenPoint to which you can send messages have the prefix "**the**": **theRootWindow**, **theSystemPreferences**, and so on.

Messages 5.6.5.3

Messages follow the standard style for constants, but have special prefix "**msg**". This is followed by the name of the class that defines the message (possibly abbreviated) and finally by the action requested by the message: **msgListRemoveItem**, **msgAddrBookChanged**, and so on.

The exceptions to this rule are the basic **clsObject** messages, including **msgNew**, **msgSave**, and **msgFree**, which apply to all classes. These basic messages do not identify their class.

Status Values 5.6.5.4

Like messages, status values follow the standard style for constants. However, all status values start with the prefix **sts**. This is followed by the name of the class that defines the status value (possibly abbreviated) and finally by a description of the status: **stsListEmpty** and **stsListFull**.

For more information on the way unique messages and status values are constructed for a class, please refer to *Part 1: Class Manager*.

Exported Names 5.6.6

At GO, we use prefixes to indicate the architectural subsystem or component that defines an exported variable, define, type, or function. Prefixes help lower the possibility of name conflicts across PenPoint. They also help developers find which files contain the relevant source code.

Note that fields within exported structures are not prefixed, and locals within sample code source files are generally not prefixed either.

For example, exported System Service names are all prefaced with OS:

```
#define osNumPriorities         51
#define osDefaultPriority       0
...
...
typedef U16                 OS_INTERRUPT_ID;    // logical interrupt ID
...
...
STATUS EXPORTED0 OSProgramInstall (
    P_CHAR pCommandLine,            // dlc or exe name (and arguments)
    P_CHAR pWorkingDir,             // working dir of the program
    P_OS_PROG_HANDLE pProgHandle,   // Out: program handle
    P_CHAR pBadName,                // Out: If error, dll/exe that was bad
    P_CHAR pBadRef                  // Out: If error, reference that was bad
);
```

The file \PENPOINT\SDK\UTIL\TAGS\TAGS lists most of the exported names in PenPoint. You can scan it to see if a particular prefix is used.

The standard global include file \PENPOINT\SDK\INC\GO.H does *not* prefix its identifiers—if something is common across PenPoint, such as the U16 type, it is not prefixed in any way.

PenPoint File Structure 5.7

At GO, we follow a similar structure for both header files and source code files.

The general structure of a header file is shown below:

```
file header comments
#includes
#defines
typedefs
global variables
function prototypes
message headers
```

Here is the general format of the source code file for a class implementation:

```
file header comment
#includes
#defines
typedefs
global variables
internal functions
exported functions
"methods" implementing messages
class initialization function
main function (in an app class file)
```

File Header Comment 5.7.1

The file header comment contains a brief description of the contents of the file. It also includes the revision number of the header file. If you have a problem using a PenPoint API, the revision level of the software is important information.

Includes 5.7.2

The include directives all follow the file header and are of the form:

```
#include <incfile.h>
```

Note that the filename for the include file does *not* contain any directory information. To locate include files, you specify an include path externally (either in the INCLUDE system variable or as a compiler flag.

Multiple Inclusion 5.7.2.1

PenPoint has many subsystems, each linked to other subsystems. Each element tends to have its own header file(s). Consequently, including the header file for one subsystem leads to it including dozens of other subsystems. Often the same header files are included by other header files. This can slow down compiling and may lead to errors if header files are compiled in more than once.

All PenPoint header files guard against being included multiple times by defining a unique string (*FILENAME*_INCLUDED) and checking to see if this string has been defined:

```
/*****************************************************************
 filename.h
 (C) Copyright 1991, GO Corporation, All Rights Reserved.

 Include file format.

 $Revision$
   $Author$
     $Date$
*****************************************************************/
#ifndef FILENAME_INCLUDED
#define FILENAME_INCLUDED
// defines, types, and so on of header file
#endif  // FILENAME_INCLUDED
```

where FILENAME is the name of the include file itself.

You can speed up compiling by putting the same checks in your files to avoid even reading in a header file:

```
#ifndef LIST_INCLUDED
#include <list.h>
#endif  // LIST_INCLUDED
```

Common Header Files 5.7.2.2

In a class implementation, if you include the header file of your immediate ancestor, this will usually include the header files of all your ancestors.

If you include any header file at all, you will not need to include <GO.H>.

Defines, Types, Globals 5.7.3

This section of a file holds all of the `#defines`, `typedefs`, and global and static declarations used only in this file. By grouping these items in one place, you will be able to find them more easily.

Function Prototypes 5.7.4

Function prototypes in header files indicate the parameters and format of PenPoint functions. Each is preceded by a comment header:

```
/*****************************************************************************
 Function returns TYPE
    Brief description.

 Comments, remarks.
*/
function declaration;
```

For example:

```
/*****************************************************************************
 OSHeapBlockSize    returns STATUS
    Passes back the size of the heap block.

 The size of the heap block is the actual size of the block. This may
 be slightly larger than the requested size.

 See Also
    OSHeapBlockAlloc
    OSHeapBlockResize
*/
STATUS EXPORTED OSHeapBlockSize (
    P_UNKNOWN pHeapBlock,       // pointer to the heap block
    P_SIZEOF pSize              // Out: size of the heap block
);
```

The header file descriptions of functions provide a "reminder" facility, not a tutorial.

Message Headers 5.7.5

Many header files contain message headers, which are where messages are described and where their constants and related data structures are defined. Message headers have the following format:

```
/*****************************************************************
 msgXxxAction   takes STRUC_TURE, returns STATUS
    category: message use
    Brief description.

 Comments, remarks.
*/
#define msgXxxAction      MakeMsg(clsXxxAction, 1)
typedef struct STRUC_TURE {
    ...
} STRUC_TURE, *P_STRUC_TURE;
```

For example:

```
/*****************************************************************************
 msgAddrBookGetMetrics  takes P_ADDR_BOOK_METRICS, returns STATUS.
    Passes back the metrics for the address book.
*/
#define msgAddrBookGetMetrics              MakeMsg(clsAddressBook, 8)

typedef struct ADDR_BOOK_METRICS {
    U16         numEntries;         // Total number of entries
    U16         numServices;        // Number of known services
    U16         numGroups;          // Number of groups in the address book
    U32         spare1;
    U32         spare2;
} ADDR_BOOK_METRICS, *P_ADDR_BOOK_METRICS;
```

We relied on the regular format of message descriptions in header files to generate the datasheets for messages in the *PenPoint API Reference.*

In, Out, and In-Out 5.7.5.1

In a message header, you can assume that all parameters and message arguments are input-only (In) unless otherwise specified (Out or In-Out).

Indentation 5.7.6

Most PenPoint header files use four spaces per tab for indentation. Most programmer's editors allow you to adjust tab spacing; setting it to four will make it easier to read GO files.

Comments 5.7.7

In general, slash-asterisk C comments (/* and */) indicate the start and end of functional areas, and slash C (//) comments are used for in-line comments within functions.

Some Coding Suggestions 5.7.8

Here are some of the other conventions that GO code follows (more or less).

- Always include the default case in your switch statements to explicitly show that you are aware of what happens when the switch fails.
- Don't use load-time initializations, except for constant values. Since PenPoint restarts code without reloading it, your code should explicitly initialize your variables.
- Use `#defines` for constants and put the defines in an include file (if it is used across multiple files) or at the beginning of the source file with a comment to indicate its use.
- When defining an external function, use prototype declarations to describe the parameters and types it requires.
- Make calls to external functions as specified by the include file of the subsystem exporting the function.
- If your files fully declare the types of their functions, this will help them to be independent of any flags that may be set during compilation.
- A source file should compile without warnings. PenPoint code compiles without any warnings at warning level 3 using the Watcom C compiler.
- Structure names must not be used as exported names. Use the type name to export a structure type. Structure names should be used only for self-referencing pointers.
- Code for a single function should not exceed a few pages. Break it up (but don't go overboard!).
- Use GO's Class Manager to support standard object-oriented programming methodologies.
- The most important parameter to a function should be the first parameter, for example, WindowDrag(pWin, newx, newy). This is usually the object on which the function acts.

PenPoint Types and Macros 5.8

In developing PenPoint, we found it useful to establish a "base" environment which goes beyond the structures and macros provided by the C language. This section describes many of these extensions. For a complete list, please look at \PENPOINT\SDK\INC\GO.H, where all of our extensions are defined.

Data Types 5.8.1

To allow for portability between different C compilers and processors, we define six basic data types that directly indicate their size in bits. Three are signed: S8, S16, and S32. The others are unsigned: U8, U16, and U32. We also define corresponding pointers for each, prefixed with P_, and pointers to pointers, which are prefixed with PP_.

To plan for internationalization efforts, we provide the CHAR data type. CHAR is functionally equivalent to char and is defined to be a U8 in PenPoint 1.0. In our 2.0 release, which will include support for international character sets, we will change CHAR to U16. Simply stated, you should use CHAR instead of `char` to ensure an easier transition to PenPoint 2.0.

CHAR has two related data types: P_CHAR, which represents a pointer to a character (`char *`), and PP_CHAR, which is a pointer to a string (`char **`).

P_UNKNOWN is for uninterpreted pointers, that is, pointers that you do not dereference and about which code makes no assumptions.

P_PROC is for pointers to functions. It assumes the Pascal calling convention.

The SIZEOF type is for the sizes of C structures returned by sizeof.

The status values returned by many functions are of type STATUS. This is a signed 32-bit value, although most subsystems encode status values to indicate the class defining the error to avoid status value conflicts. Section 2.8 describes status values in greater detail.

Basic Constants 5.8.2

Use the enumerated type BOOLEAN for logical values **true** and **false**. The BOOLEAN type also defines the values **True**, **False**, TRUE, and FALSE to preempt any discussion about capitalization rules.

Similarly, **null** is the preferred spelling for `null (0)`, but NULL is also defined. **pNull** is a null pointer.

minS8, **maxS8**, **minS16**, **maxS16**, **minS32**, and **maxS32** are the minimum and maximum integer values for the three signed types. **maxU8**, **maxU16**, and **maxU32** are the maximum values of the three unsigned types. Obviously, the minimum unsigned value is zero.

Names in many PenPoint subsystems can be no longer than 32 characters. This limit is defined as **maxNameLength**. Since strings are normally null-terminated, we define **nameBufLength** to be **maxNameLength** + 1.

Legibility 5.8.3

GO.H defines AND, OR, NOT, and MOD to be the corresponding C logical "punctuation;" this avoids confusion with the double-character bit operators `&&` and `||`.

Compiler Isolation 5.8.4

GO.H provides macros and other #defines that you can use to ensure compiler independence.

Function Qualifiers 5.8.4.1

GO.H introduces a layer in between the `pascal`, `cdecl`, and so on keywords of the Watcom C compiler by providing uppercase versions of all these keywords.

Using the uppercase versions allow you to easily remove or redefine these keywords in source code if necessary. This allows you to experiment with changing the segmentation or calling sequences of your code to check for errors or changes.

It's important to explicitly specify calling conventions in your function prototypes so that code can compile with a different set of compiler switches from GO's defaults, yet still observe the protocol requirements.

STATIC, LOCAL, and GLOBAL are compiler #defines that support the appearance (if not the reality) of modular programming.

Instead of using these detailed qualifiers, you can use higher-level constructs. If you want to make a function available to other modules, EXPORTED says that it is a PASCAL function.

There are other qualifiers that are only used by specialized kernel code or drivers. EXPORTED0 means that a function is a call gate. RINGCHELPER identifies functions which can be called by code at lower privilege levels.

Enumerated Values 5.8.4.2

Some compilers base the size of an `enum` value on the fields in that `enum`. This has unfortunate side effects if an enum is saved as instance data; programs compiled under different compilers might read or write different amounts of data, based on the size of the `enum` as they perceive it.

To guarantee that an enum is a fixed size, use the **Enum16** and **Enum32** macros. These macros create enums that are 16 and 32 bits long, respectively. The macros expect a single argument—the name of the `enum` to be defined..

Within an **Enum16** or **Enum32**, use the bit flags (**flag0** through **flag31**, also defined in GO.H) to define enumerated bits.

Most PenPoint header files indicate when bits in an `enum` can be ORed to specify several flags. If a PenPoint header file uses the **flag0**-style bit flags, assume that you can OR these flags.

Data Conversion/Checking 5.8.5

Abs, **Even**, and **Odd** are macros that perform comparisons, returning a boolean. **Max** and **Min** return the larger and lesser of two numbers, respectively.

OutRange and **InRange** check whether a value falls within a specified range. They work with any numeric data type.

Be careful when using the **Abs**, **Min**, **Max**, **OutRange**, and **InRange** macros because their parameters are evaluated multiple times. If a function call is used as an argument, multiple calls to the function will be made to evaluate the macro.

Bit Manipulation 5.8.6

GO.H defines each bit as **flag0** through **flag31**, with **flag0** being the least-significant (rightmost) bit.

LowU16, **HighU16**, **LowU8**, and **HighU8** extract words and bytes by casting and logical shifts. **MakeU16** and **MakeU32** assemble words and 32-bit quantities out of 8-bit and 16-bit quantities.

FlagOn and **FlagOff** check whether a particular flag (bit) is set or reset. **FlagSet** and **FlagClr** set a particular flag. All four can take a combination of flags ORed together. You can use these bit manipulation macros with U8, U16, or U32 data types.

Tags 5.8.7

There are several types of values passed around or otherwise shared among subsystems and applications in PenPoint:

- Class names
- Messages
- Return values
- Window tags.

All of these are 32-bit constants (U32). As you develop code and classes, you will define your own. It is vital that they not conflict, so GO provides a **tag** mechanism to guarantee unique names for them. GO administers a number space in which every developer can reserve a unique set of numbers. A tag is simply a 32-bit constant associated with an adminstered number. With each administered number you can define 256 different tags: because the administered numbers are unique, so will be the tags.

You usually use your classes' administered number to define messages, status values, and window tags, since these are all usually associated with a particular class. See *Part 1: Class Manager* for an explanation of how classes, tags, and administered numbers relate to each other.

Return Values 5.8.8

Most PenPoint code returns error and feedback information by returning special values from functions rather than generating exceptions. PenPoint still uses exceptions for certain types of errors: GP fault, divide by 0, and so on. Otherwise, functions that return success or failure must return a status value. Status values are 32-bit tags, defined in GO.H:

```
typedef S32 STATUS, * P_STATUS;
```

The universal status value defined to mean "All is well" is **stsOK**. By convention, return values less than **stsOK** denote errors, while return values greater than **stsOK** indicate that the function did not fail, but may not have completed in the usual way.

There is a set of GO standard status values that you can use in different situations (described below), but usually each subsystem needs to define its own specific status values. To guarantee uniqueness among status values returned by third-party software, group your status values by class, even if the status does not come from a class-based component. GO administers well-known numbers for classes, as explained above in "Tags."

Defining Status Values 5.8.8.1

GO.H defines a macro, **MakeStatus(wkn,sts)**, to make a 32-bit error status value from a well-known 32-bit identifier and an error number. Usually, the well-known number is the class that defines the error.

To make a status value that does not indicate an error, use **MakeWarning(cls, msg)**, which creates a positive tag.

So, if you want to define status values, all you need is a reserved class. GO can allocate one for you. You can then define up to 256 error status values and 255 success status values, using **MakeStatus** and **MakeWarning** with numbers in the range 0-255. If you need more status values, you can request another class UID.

Pseudoclasses for Status Values 5.8.8.2

Since not everything in the PenPoint API is a message-based interface to an object-oriented class, there are several pseudoclasses defined solely to provide "classes" for status values from some subsystems: **clsGO**, **clsOS**, **clsGoMath**, and so on. You can ask GO for your own pseudoclasses for error codes if necessary.

Testing Returned Status Values 5.8.8.3

To test a STATUS value for the occurrence of an error, just test whether the value is less than **stsOK**. To test for one specific error, compare the value to the full error code from the appropriate header file. There are macros to assist in this, described in Section 3.11, "Error Handling Macros."

There are a small number of system-wide error/status conditions. You can return a generic status value instead of defining your own, *so long as you use it consistently with its definition.* If you need to convey a slightly different sense, define your own context-specific status value.

Here are the generic status values. Their "class" identifier is the pseudo-class **stsGO**.

Table 5-1
Generic Status Values

Status Value	Description
stsOK	Everything's fine.
	Errors
stsBadParam	One or more parameters to a function call or message are invalid.
stsNoMatch	A lookup function or message was unable to locate the desired item.
stsEndOfData	Reached the end of the data.
stsFailed	Generic failure.
stsTimeOut	A time-out occurred before the requested operation completed.
stsRequestNotSupported	As it sounds.
stsReadOnly	The target can't be modified.
stsIncompatibleVersions	The message has a different version than the recipient.
stsNotYetImplemented	The message is not yet fully implemented.
stsOutOfMem	The system has run out of memory.
	Non-Error Status Values
stsRequestDenied	The recipient decided not to perform the operation.
stsRequestForward	The recipient asks the caller to forward the request to some other object.
stsTruncatedData	The request was satisfied, but not all the expected data has been passed back.

The macro **StsOK** returns **true** if the status returned by an expression is greater than or equal to **stsOK**. If you want to check for *any* status other than **stsOK**, use **StsFailed**. See "Error-Handling Macros," below.

Return Status Debugging Function 5.8.9

The function **StsWarn** evaluates any expression that returns a STATUS. If you do not set the DEBUG preprocessor variable during compilation, **StsWarn** is defined to be the expression itself—a no-op. This means that whenever you call a function that returns a status value, you can use **StsWarn**.

If DEBUG *is* defined, and the expression evaluates to an error (less than **stsOK**), then **StsWarn** prints the status value returned by the expression together with the file and line number where **StsWarn** was called (the special compiler keywords `__FILE__` and `__LINE__`).

Human-Readable Status Values 5.8.9.1

You can load tables of symbol names in the Class Manager so that if you have set DEBUG, the above functions will print out a string for status return values, instead of a number. For an example of this, see the S_TTT.C file of the Tic-Tac-Toe sample program (\PENPOINT\SDK\SAMPLE\TTT), explained in the *PenPoint Application Writing Guide.*

Error-Handling Macros

5.8.10

Every PenPoint function or message returns a STATUS which you should check. The following status macros make function checking much easier by handling typical approaches to handling errors.

Table 5-2
Status Checking Macros

Error Handling Approach	Macro
check for an error (no warning)	StsChk
check for an error and warn	StsFailed
return if result is an error	StsRet
jump to an error handler if result is an error	StsJmp
check that the result is not an error	StsOK

The Class Manager defines similar macros for checking the status values returned when sending a message.

Each status value checker works with any expression that evaluates to a STATUS. Each takes the expression and a variable to assign the status to. All of these macros (except **StsChk**) call **StsWarn**, so that they print out a warning message if you set the DEBUG preprocessor variable during compilation.

Since often one function calls another which also returns STATUS, using these macros consistently will give a "stack trace" indicating the site of the error and the nested set of functions which produced the error.

The examples below assume that **MyFunc**() returns STATUS.

StsChk(*se, s*)

Checks for an error.

Description

Sets the STATUS **s** to the result of evaluating **se**. If **s** is less than **stsOK**, returns **true**, otherwise returns **false**. Does not print out a warning message.

Example

```
STATUS  s;
if (StsChk(MyFunc(param1, param2), s)) {
    // MyFunc() failed
}
```

StsFailed(*se, s*)

Checks for an error.

Description

Sets the STATUS **s** to the result of evaluating **se**. If **s** is anything other than **stsOK**, returns **true** and prints an error if DEBUG is set. If *s* is **stsOK**, returns **false**.

Example

```
STATUS  s;

if (StsFailed(MyFunc(param1, param2), s)) {
    // MyFunc() returned other than stsOK, so check status
    switch (Cls(s)) {
    ...
} else {
    // MyFunc() did the expected thing, so continue
}
```

Remarks

This is analogous to **StsOK**, but it reverses the sense of the test in order to be more consistent with other checking macros.

StsJmp(*se, s, label*)

Jump to label on error.

Description

Sets the STATUS **s** to **se**. If **s** is less than **stsOK**, it prints an error if DEBUG is set and does a goto to **label**. This is useful when you have a sequence of operations, any of which can fail, each having its own clean-up code.

Example

```
    STATUS  s;

    pMem1 = allocate some memory;
    StsJmp(MyFunc(param1, param2), s, Error1);
    pMem2 = allocate some more memory;
    StsJmp(MyFunc(param1, param2), s, Error2);
    ...
    return stsOK;

Error2:
    // Handle error 2.
    OSHeapBlockFree(pMem2);
Error1:
    // Handle error 1.
    OSHeapBlockFree(pMem1);

    return s;
```

StsOK(*se, s*)

Checks that things are OK.

Description Sets the STATUS **s** to the result of evaluating **se**. If **s** is greater than **stsOK**, returns **true**. Otherwise, prints an error if DEBUG is set and returns.

Example

```
STATUS  s;

if (StsOK(MyFunc(param1, param2), s)) {
    // MyFunc() succeeded, continue.
} else {
    // MyFunc() failed, check status.
    switch (Cls(s)) {
    ...
}
```

Remarks This is analogous to **StsFailed**, but reverses the sense of the test and returns **true** for any status value that is not an error. In other words, this could return **true** but the status might be some other value than **stsOK**, such as **stsNoMatch**.

StsRet(*se, s*)

Returns status on error.

Description Sets the **STATUS s** to **se**. If **s** is less than **stsOK**, prints an error if DEBUG is set and returns **s**. This is useful if one function calls another and should immediately fail if the second function fails.

Example

```
STATUS  s;
...
// If MyFunc has problems, return.
StsRet(MyFunc(param1, param2), s);
...
```

Debugging Assistance 5.9

GO has developed a set of useful functions and macros to assist in debugging PenPoint applications. They are no substitute for DB, the PenPoint Source-level debugger, or the PenPoint mini-debugger (both these debuggers are documented in *PenPoint Developement Tools*). However, they help you trace the operation of a program without using a debugger. They are an elaboration of the time-honored technique of inserting `printf`s in your code.

Printing Debugging Strings 5.9.1

DPrintf and **Debugf** print text to the debugger stream. They take a formatting string and optional parameters to display, in the same manner as as the standard C function `printf`. The only difference between **DPrintf** and **Debugf** is that **Debugf** supplies a trailing newline (if you want a newline at the end of **DPrintf** output, end it with \n).

```
Debugf("Entering init method for clsApp");
Debugf("main: process count = %d", processCount);
```

Debugger Stream 5.9.1.1

The debugger stream is a pseudo device to which programs (including PenPoint) can write debugging information. There are several ways to view the debugger stream.

- If you have a single screen, you can see the most recent lines written to the debugger stream when you press Pause.
- If you have a second (monochrome) monitor, serial terminal, or PC running communications software, you can constantly watch the debugger stream on this monitor while you run PenPoint on the main (VGA) monitor.
- You can send the debugger stream to a log file, by setting the D debugger flag to the hexadecimal value 8000. Usually you do this in the ENVIRON.INI file, but you can also do it from the PenPoint symbolic debugger, or from the mini-debugger.

```
DebugSet=/DD8000
DebugLog=\\boot\tmp\run3.log
```

- You can use the System Log application to view the debugger stream while running a PenPoint appliction.

None of these destinations are mutually exclusive.

Assertions 5.9.2

Often when working on functions called by other functions, you assume that the software is in a certain state. The ASSERT macro lets you state these assumptions, and if DEBUG is set, it checks to see that they are in fact the case. If they are not satisfied, it will print an error. For example, a square root function might rely on never being called with a negative number:

```
void MySqRoot(int num)
{
    ASSERT(num >= 0, "MySqRoot: input parameter is negative!");

    // Calculate square root...
```

The test is only performed if DEBUG is defined.

Debug Flags 5.9.3

At different times you want to print different debugging information, or you want your program to work a certain way. DEBUG is the common `#define` used by PenPoint to include debugging output; if you set DEBUG when compiling, the status-checking macros print out additional information, the ASSERT macro is enabled, and so on. You can use your own C preprocessor directives to get finer control over program behavior, for example

```
    OBJECT  myDc
#ifdef MYDEBUG1
    // Dump DC state
    ObjectCall(msgDump, myDc, Nil(P_ARGS));
#endif
```

The disadvantage of this technique is that you must recompile your program to enable or disable this code.

Another approach is to check the value of a flag in your code. PenPoint supports 256 global **debugging flag sets**. Each flag set is a 32-bit value, which means that you can assign at least 32-different meanings to each debugging flag set.

Because there are 256 debug flags sets, they can be indexed by an 8-bit character. Commonly, we refer to a specific debugging flag set by the character that indexes that flag. GO has reserved all the uppercase character debug flags sets (A through Z), and has reserved some of the lowercase characters also. To find which debug flags set are available, see the file \PENPOINT\SDK\INC\DEBUG.H.

You can set the value of a flag set, and retrieve it. The typical way you use debugging flag sets is to set the value of a flag set before running a program, and in the program check to see which bits in the flag set are on. The function **DbgFlagGet** returns the state of a flag set ANDed with a mask.

For example, if you were using the flag F in your program and were checking the third bit in it to see whether or not to dump an object, the code above would look like:

```
if (DbgFlagGet('F', 0x0004)) {
    // Dump DC state
    ObjectCall(msgDump, mydc, Nil(P_ARGS));
}
```

You only need to compile your program once, and you can turn on object dumping by changing the F flag set to 0x4 (or 0x8, or 0xF004, and so on). The disadvantage of this is that the flag-testing code is compiled into your program, increasing its size slightly. Often programmers bracket the entire **DbgFlagGet** test inside an #ifdef DEBUG/#endif pair so that the flag-testing code is only compiled while in the testing version of their program.

Setting Debugging Flag Sets 5.9.3.1

There are several ways to set debugging flag sets. Note that there is a single set of these flags shared by all processes.

- In \PENPOINT\BOOT\ENVIRON.INI, set the flag to the desired bit pattern with:

  ```
  DebugSet=/DFnnnn /Dfmmmm ...,
  ```

  where *F* and *f* are letters that identify a particular flag set and *nnnn* and *mmmm* are a hexadecimal values. For example, DebugSet=/DFE004.

- By typing **fs** F nnnn in either the PenPoint source-level debugger or the PenPoint mini debugger.

- By using **DbgFlagSet**() in a program, for example:

  ```
  DbgFlagSet('F',0xE004).
  ```

Suggestions 5.9.4

Isolate Debugging Messages 5.9.4.1

In general, always isolate all debugging code using

```
#ifdef DEBUG
/* Debugging */
#endif
```

DEBUG is the conventional flag for debugging code, used by much of PenPoint.

Use the Status-Checking Macros 5.9.4.2

Using the status-checking macros **StsOK**, **StsJump**, and so on, and their counterparts for sending messages may seem cumbersome, but they provide useful debugging information if DEBUG is defined. Also, since most functions and message sends return the error status if they encounter an error, the "stack" of status prints provides a traceback showing where the error first occurred and who called it.

This status error listing shows the result of sending **msgDrwCtxSetWindow** to **objNull**:

```
C> ObjectCall: sts=stsBadObject                    "tttview.c".@232 task=0x05d8
C>   object=objNull
C>   msg=msgDrwCtxSetWindow, pArgs=26ec0438
>> StatusWarn: sts=stsBadObject                    "tttview.c".@330 task=0x05d8
>> StatusWarn: sts=stsBadObject                    "tttview.c".@743 task=0x05d8
Page fault in task 05D8 at 1B:440CCD52.  Error code = 0004.
 EAX=00000000 EBX=04000002 ECX=E002E5CF EDX=440CCD05 ESI=41BC8EF0 EDI=4401EC38
 EIP=440CCD52 EBP=004329E0 ESP=004329CC FLG=00010246 CR2=0000000C CR3=00077000
 CS=001B DS=002B SS=002B ES=002B FS=0000 GS=0000 TSS=05D8 TNAME=TIC1
```

Use the Debuggers 5.9.4.3

If your code crashes unexpectedly, you can use the PenPoint mini-debugger to get a stack trace at the assembly-language level (type **st** at its > prompt). The linker's .MAP files enable you to translate assembly language addresses to functions and line numbers.

If you suspect that your code is going to crash or behave improperly, run it from the PenPoint Source-level Debugger. This lets you step through your code, query and set values, and evaluate simple C expressions.

Both debuggers are described in *PenPoint Development Tools.*

The Tutorial Programs

5.10

Now that you've read the broad overview of PenPoint and its class-based applications, views, and objects, you are ready to get down to some of the nuts and bolts of writing an application. This section describes the remaining chapters in this book and the sample programs used in those chapters. The programs are:

Empty Application
Hello World (toolkit)
Hello World (custom window)
Counter Application
Tic-Tac-Toe
Template Application

Chapter 6 explains how to compile and run programs using Empty Application. The chapter is quite long because it teaches the general development cycle:

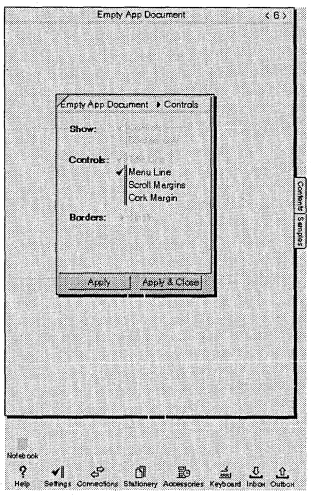
- How to compile an application
- How to install an application on a PC or PenPoint computer
- How to run an application
- Some interesting things to look for when running any application
- How to use some of the PenPoint debugging tools.

The Empty Application is used to illustrate these steps, but the comments are applicable to all the other sample applications.

Empty Application

5.10.1

The tutorial starts off with an extremely simple application, Empty Application. Chapter 6 explains how to build and run it and how the application works. Empty Application has no view, no data, and no application-specific behavior (apart from printing a debugging message). It only responds to one message from the Application Framework. However, it does create an application class (as all PenPoint applications must), and through inheritance from **clsApp**, you can create, open, float, zoom, close, rename, file, embed, and destroy Empty Application documents.



Hello World (Toolkit)

5.10.2

The next application is the traditional "Hello World" application. This prints Hello World! in its window. Rather than creating a window from scratch, this uses the existing User Interface Toolkit components. One of these is **clsLabel**, which displays a string. Hello World (toolkit) uses this existing class instead of creating its own. The components in the UI Toolkit are rich in features; for example, labels can scale their text to fit. If you can use a toolkit class, do so.

Hello World (toolkit) is described in Chapter 7.

Hello World (Custom Window)

5.10.3

Of course it is possible to draw text and graphics yourself. Hello World (custom window) draws the text Hello World in its window, and draws a stylized exclamation mark beside it. To do this, the application must create a separate window class and create a system drawing context to draw in its window, which is substantially harder than using toolkit components.

Hello World (custom window) is described in Chapter 8.

Counter Application

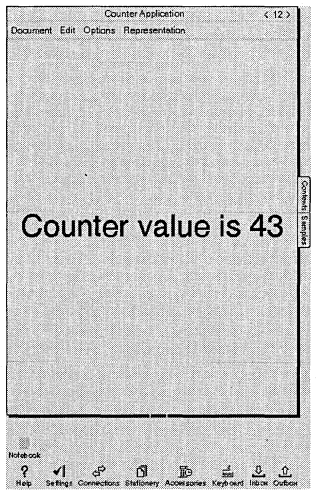
5.10.4

Counter Application displays the value of a counter object in a label. It creates a separate counter class and interacts with it. The application has a menu created from UI Toolkit components which lets the user choose whether to display the counter value in decimal, hexadecimal, or octal.

Both the application and the counter object must file state. The tutorial programs presented before Counter Application are not stateful, that is, they don't have data that the user can change permanently. Realistic applications must allow users to change things, so they must file their state.

The application object uses a memory-mapped file to keep track of its state. Using a memory-mapped file avoids duplicating data in both the memory file system in program memory. By contrast, the counter object writes its value to a file when it is saved.

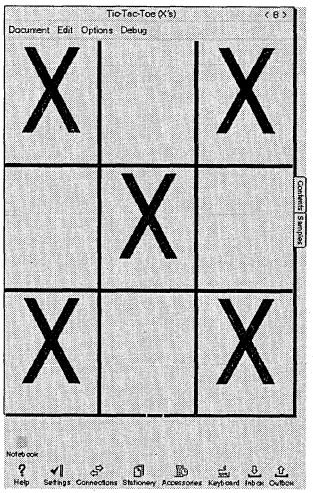
The counter application is described in Chapter 9.



Tic-Tac-Toe

5.10.5

The rest of the tutorial develops a "real" working application, Tic-Tac-Toe. This application is covered in Chapters 10 and 11.



Tic-Tac-Toe presents a tic-tac-toe board and lets the user write Xs and Os on it. It is not a true computerized game—the user does not play tic-tac-toe against the computer. Instead, it assumes that that there are two users who want to play the game against each other.

Although a tic-tac-toe game is not exactly a typical notebook application, Tic-Tac-Toe has many of the characteristics of a full-blown PenPoint application. It has a graphical interface, handwritten input, keyboard input, gesture support, use of the notebook metaphor, versioning of filed data, selection, import/export, option cards, undo support, stationery, help text, and so on.

Template Application

5.10.6

As its name implies, Template Application is a template, "cookie cutter" application. As such, it does not exhibit much functionality. However, it does handle many "typical" application messages. This aspect makes Template Application a good starting point for building a real application.

Other Code Available

5.10.7

Other source code is provided in the SDK in addition to the tutorial code.

All the source to sample programs is on-disk in \PENPOINT\SDK\SAMPLE. Some of the other sample programs are described in Appendix A of this manual. Excerpts from sample programs also appear and are described in those parts of the *Architectural Reference* that cover related subsystems.

Chapter 6 / A Simple Application (Empty App)

Applications written for many operating systems have to perform housekeeping functions by implementing their own boilerplate code, that is, code that is essentially the same from one application to the next. In PenPoint, the PenPoint Application Framework performs most of these housekeeping functions. By using the Application Framework, you can create an application that can be installed, that can create multiple instances of itself, that can handle page turns, floats and zooms, and that can display an option sheet, all without writing an additional line of code.

Empty Application is a very simple application that does all of these things. The only additional code in Empty Application is a method that responds to **msgDestroy** by sending a message to the debug stream (when the program is compiled with the DEBUG preprocessor `#define` name).

The PenPoint Application Framework is responsible for everything else Empty Application does. Because the Application Framework handles so much of an application's interaction with the system, even such an insubstantial application has substantial functionality.

Files Used 6.1

The code for Empty Application is in \PENPOINT\SDK\SAMPLE\EMPTYAPP. There are three files in the directory:

EMPTYAPP.C Contains the application class's code and initialization routine

METHODS.TBL Contains the list of messages that the application class responds to and the associated message handlers to call

MAKEFILE Contains the WATCOM Make file for EMPTYAPP. The make file also defines information required by the OS/2 linker.

Not the Simplest 6.1.1

The name Empty Application is not quite accurate, because it isn't totally empty. You could create an application with no method table at all, that is, one that responds to no messages at all and relies entirely on **clsApp** to do the right thing. Empty Application handles one message by printing a string to the debug stream, so it needs a method table.

Compiling and Linking the Code 6.2

The source code for sample applications is in subdirectories of \PENPOINT\SDK\SAMPLE. The subdirectory contains a "makefile" that tells WATCOM Make how to build the application. Thus to compile and link Empty Application, just change directory to \PENPOINT\SDK\SAMPLE\EMPTYAPP and type **wmake**.

However, you do need to understand what the files are doing, so that you can later modify the makefiles to fit your needs.

These sections describe the actual commands used to compile, link, and stamp EMPTYAPP.

Compiling Method Tables 6.2.1

You compile method tables into an object file by running them through the PenPoint method table compiler (in \PENPOINT\SDK\UTIL\CLSMGR\MT.EXE).

By convention, method tables have the suffix .TBL. The control files used by WATCOM Make have a default rule for compiling a method table.

MT produces an object file and a header file for the method table. You use these files when you compile and link the application.

```
> cd \penpoint\sdk\sample\emptyapp
> mkdir \penpoint\app\emptyapp
> set WCC386=/3s /Oif+ /s /W3 /We /Zc /Zq /fpc /zff /D2 /En /DDEBUG
> \penpoint\sdk\util\clsmgr\mt methods.tbl -Fo=$methods.obj > $methods.err
```

Compiling the Application 6.2.2

To compile the application, use the WATCOM C/386 compiler for protected-mode applications (WCC386P.EXE).

```
> set WCC386=/3s /Oif+ /s /W3 /We /Zc /Zq /fpc /zff /D2 /En /DDEBUG
> wcc386p /Foemptyapp.obj emptyapp.c > emptyapp.err
```

Compiler and Linker Flags 6.2.2.1

The meanings of the various flags are listed in Table 6-1 (for more information, consult the WATCOM C/386 documentation):

Table 6-1
WATCOM Compiler and Linker Flags

Flag	Description
/3s	Generate 80386 instructions, pass arguments using stack.
/D2	Include full symbolic debugging information
/DDEBUG	Create the preprocessor #define name DEBUG
/En	Emit routine name before prolog
/Fpc	Generate calls to floating-point library
/Oif+	Optimization flags
/s	Remove stack overflow checks

continued

Table 6-1 (continued)

/W3	Set warning level to 3
/We	Treat all warnings as errors
/Zc	Place literal strings in code segment
/zff	Makes the FS register floating (required for PenPoint)
/Zq	Suppress compiler informational messages.

Linking the Application 6.2.3

You link the application with the WATCOM OS/2 linker for protected mode applications (WLINKP.EXE).

Because PenPoint applications run in protected mode, you must tell the linker to produce a protected-mode executable file. The Makefile does this by creating a WLINK command file (the Make script deletes this command file before exiting).

Caution This is **only** true for WLINKP! If you have trouble compiling, make sure you are using the correct linker.

The command file specifies what routines your code imports from DLL (dynamic link library) files, the memory model, privilege levels, access attributes, and protection.

If you created the file by hand, it would contain:

```
SYSTEM PenPoint
NAME \386\PENPOINT\app\emptyapp\emptyapp.exe
DEBUG ALL
FILE METHODS.OBJ
FILE EMPTYAPP.OBJ
LIBRARY PENPOINT
LIBRARY APP
OPTION Quiet, Map=emptyapp.mpe, NOD, Verbose, Stack=15000, MODNAME='GO-EMPTYAPP_EXE-V1(0)'
```

To tell the linker that it should get its options from a command file, precede the name of the command file with an at sign (@). In this case, which was drawn directly from WMAKE output, the command file is called EMPTYAPP.ELN.

```
> wlinkp @emptyapp.eln
```

MODNAME is the name that PenPoint attaches to the process. It can be longer than the DOS eight-character file length limitation. In PenPoint, it contains important versioning information. It should be composed of `your company's name` followed by a dash, the project name followed by a dash, and a version string.

The process name is not the application name. The process name only shows up in the debugger; the application name (the name of the directory in \PENPOINT\APP) appears in the Installer.

If your application has a separate DLL, it will need its own command file.

If the compile and link is successful, these commands create an EMPTYAPP.EXE file in \PENPOINT\APP\EMPTYAPP.

Stamping Your Application 6.2.4

The last thing you must do in compiling an application is to give specific PenPoint attributes to the executable. To do this, you use the STAMP command, which creates an entry for your application in the PENPOINT.DIR file in the

directory that contains the your application executable. You use STAMP to specify:

- A long PenPoint file name for your application. PenPoint applications can have longer names than in DOS, just as PenPoint documents can have long names.
- The PenPoint name of the application executable file. The .EXE name must match the directory name. Thus the second STAMP command below gives the file EMPTYAPP.EXE the PenPoint name Empty Application.EXE.
- The version of PenPoint for which your application was compiled
- A text string that describes the file for the Browser (APPLICATION, FONT, SERVICE, and so on)
- A special identifier that tells the Installer that the file is an application (the value 10001A0)

For EMPTYAPP, the stamp commands are:

```
> \penpoint\sdk\util\dos\stamp \penpoint\app\emptyapp\.. /g "Empty Application" /D emptyapp
> \penpoint\sdk\util\dos\stamp \penpoint\app\emptyapp /g "Empty Application.exe" /D emptyapp.exe
> \penpoint\sdk\util\dos\stamp \penpoint\app\emptyapp\.. /g "Empty Application" /a 01E00208 "1.0"
> \penpoint\sdk\util\dos\stamp \penpoint\app\emptyapp\.. /g "Empty Application" /a 0660013a
"Application"
> \penpoint\sdk\util\dos\stamp \penpoint\app\emptyapp\.. /g "Empty Application" /a 0080013a
10001a0
```

Installing and Running Empty Application 6.3

As described in the section "How Applications Work" in Chapter 3, you must install an application in PenPoint before you can run it. To install Empty Application, install it either at boot time or use the Application Installer on a running PenPoint system. The Application Installer is described in *Using PenPoint.*

To install the application at boot time:

- Add a line that says \\BOOT\PENPOINT\SDK\SAMPLE\Empty Application to \PENPOINT\BOOT\APP.INI.
- Boot PenPoint on your PC.
- When the Notebook appears, draw a caret ^ in the TOC to insert an Empty Application document in the Notebook.

When you create an Empty Application document in the Notebook, PenPoint creates a directory for the document in the application hierarchy (that's why it shows up in the table of contents), but it's only when you turn to the document's page that a process for the document is *activated.* Until then the document isn't running and doesn't have a process or a valid **clsEmptyApp** object.

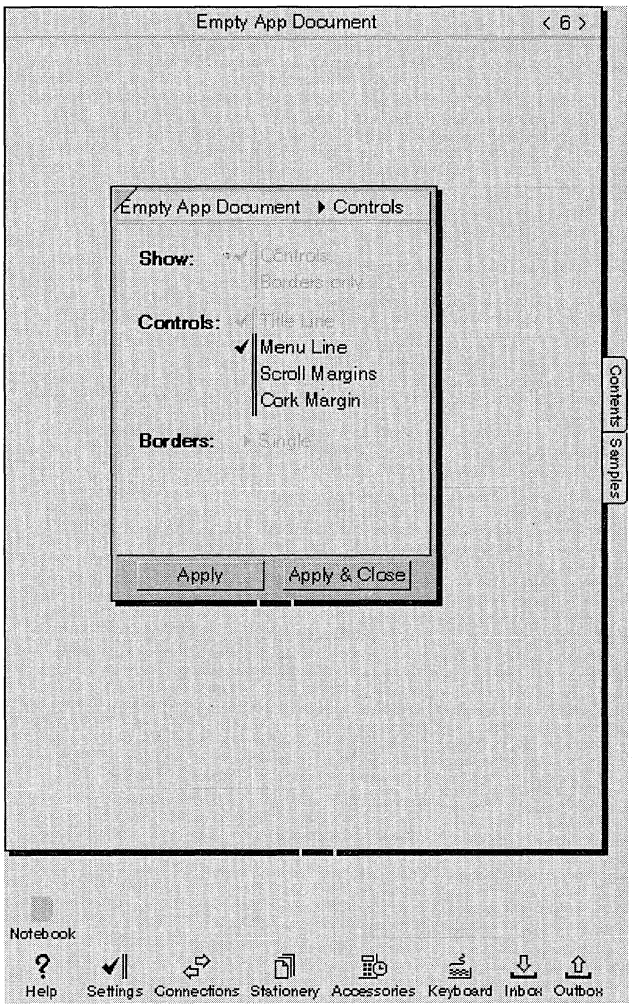
The section "Installation and Activation," below, explains the difference between Installation and Activation, and the relationship between PenPoint processes and application classes.

Interesting Things You Can Do with Empty Application 6.4

Although Empty Application doesn't perform any work, you can learn a lot about the operation of PenPoint by studying it. PenPoint provides a host of features and support to even the simplest application. You can try the following:

- Create multiple instances (*documents*) of it. The PenPoint file system appends a number to each document to guarantee a unique application directory name in the application hierarchy. You create documents by performing one of these actions:
  - Choose Empty Application from the Create menu in the Notebook contents page.
  - Choose Empty Application from the pop-up menu that appears when you draw a caret ^ on the contents page.
  - Use the Stationery notebook to create Empty Application documents in the Notebook
  - Tap and hold on the title bar or name of an Empty Application document in the TOC to make a copy an existing document. Drag the icon that appears to where you want it to go, such as on the icon bookshelf, or elsewhere in the TOC.
  - Tap the Accessories icon in the bookshelf below the Notebook and tap the Empty Application icon in its window.
- Float a Notebook Empty Application document by turning to the Notebook's table of contents and double-tapping on its page number (you must first enable floating in the Float & Zoom section of PenPoint Preferences). Compare the difference between an accessory and a floating document—accessories have no page number.
- Zoom a floating Empty Application by flicking upwards on its title bar (you must first enable zooming in the Float & Zoom section of PenPoint Preferences).
- Display the properties of an Empty Application document by drawing a checkmark ✓ in its title bar. An option sheet for the document appears, with several cards in it for the document's appearance.

Figure 6-1
Empty Application Option Sheet



- In the table of contents, press and hold on a Empty Application title until a dashed line appears around it. You can now move the document around. Try moving it to another place in the Notebook.
- Set the `B` debug flag to 800 hexadecimal in \PENPOINT\BOOT\ENVIRON.INI by adding or modifying a DebugSet line. As documented in \PENPOINT\SDK\INC\DEBUG.H, this flag enables the connections notebook to display the contents of the file system. Boot PenPoint and start up the disk viewer by tapping on the connections notebook icon below the Notebook. Open up the boot volume by double-tapping on it. Select Directory on the View menu. By repeated double-tapping, open the \PENPOINT\SYS\Notebook directory. Its subdirectories should reflect the hierarchy of accessories and the hierarchy of documents and sections in the Notebook.

  Ordinarily, you aren't able to open the active file system from the Disk Browser. This prevents users from accidentally modifying the application hierarchy.
- Give the Empty Application document a tab in the notebook by writing a "T" in its title bar. You can use the tab to navigate to the Empty Application document quickly.

- Give the Empty Application document a corkboard margin by writing a "C" in its title bar. A thick strip appears at the bottom of its window.
- Before you install Empty Application, set the F debug flag to 1. You can either set this in \PENPOINT\BOOT\ENVIRON.INI by adding or modifying a line like DebugSet = /DF1, or by pressing Pause to go to the mini-debugger, entering **fs F 1**, then entering **g** to continue. When PenPoint initializes **clsEmptyApp**, **clsEmptyApp** checks this flag to see if it should trace messages. You can also use the System Log application to do this.
- As you turn the pages, note the sequence of messages sent to each instance of **clsEmptyApp** by the PenPoint Application Framework.
- On a PC, modify the VOLSEL line in ENVIRON.INI to use a hard disk. Run PenPoint, create a document, turn to it, then quit PenPoint. Look at the PenPoint Application Framework directory structure in \PENPOINT\SS\NK.
- Select an Empty Application document in the table of contents, then use the disk viewer to open a directory on your hard disk. Copy the document to the hard disk. Then delete the document by drawing an X over it.
- Set the G debugger flag to 1000 in \PENPOINT\BOOT\ENVIRON.INI (or set the flag with the **fs** mini-debugger command). This turns on debugging info for reading and writing resources in **clsResFile**. This is the class that files objects during **msgAppSave** processing.
- Select an Empty App document in the TOC and move it by pressing and holding on its title. Move it inside another open document. If the other application supports it, the PenPoint Application Framework will embed the Empty Application document inside the other.
- With the F1 debugging flag set, select an Empty App document in the TOC, then turn the page. Note how the document doesn't receive some messages. Now select something else, and see the Empty Application document receive the "missing" messages.

Code Run-Through 6.5

Enough details of running Empty Application; now let's look at its C code. First we'll look at the layout of PenPoint source files.

PenPoint Source Code File Organization 6.5.1

Most source code in PenPoint has a similar structure. Although Empty Application is a very simple application, it has a similar layout to other applications.

Remember that application programs have at least one class (the application class itself), so an application program is composed of at least these two files:

- The **method table** that specifies the messages to which this class responds and the functions that handle those messages

- The C source code for the class. The C source code for applications is usually organized in the following way:
  - `#defines` and `typedefs`
  - Message handlers
  - Class initialization routine
  - `main` entry point.

The method table file always has the suffix .TBL. It looks like C code, but you process it with the method compiler MT before linking it into your program.

Method Table File 6.5.1.1

The method table file lists all the messages that the class handles. The PenPoint Class Manager sends any messages not listed in the method table to the class's ancestor for handling (and possibly to the ancestor's ancestor). Looking at a class's method table gives you a good feel for what the class does.

A single method table file can have method tables for several different classes. The names of the method tables are usually pretty self-explanatory, for example, **clsEmptyAppMethods** is Empty Application's method table.

You can still have one function that handles several different messages, by using the wild-card capabilities of method tables. Method table wild cards match any message within a given set of messages and call the associated method. Method table wild cards are described in *Part 1: Class Manager* of *PenPoint Architectural Reference, Volume I.*

Application C Code File 6.5.1.2

The application's `main` routine is at the end of the source file. The operating system calls the application's `main` routine under two circumstances:

- When installing the application (this happens only once)
- When activating individual documents (this happens each time the user turns to or floats a document that uses the application).

The C files for non-application classes don't have `main` routines, because only applications actually start C processes. The declaration for the main routine is:

```
main(argc, argv, processCount)
```

The **argc** and **argv** parameters are not used in PenPoint. PenPoint uses the **processCount** parameter to pass in the number of processes running this application. When **processCount** is 0, there are no other processes running this application; this indicates that PenPoint is installing the application. Once an application is installed, the process that has a **processCount** of 0 stays in memory until the application is deinstalled.

On installation, **main** initializes the application class, by calling an initialization routine. This routine precedes **main** in the source file. Standard practice is to name this routine using the name of the application class (with an initial capital

letter), followed by "**Init**". For example, the initialization routine for **clsEmptyApp** is **ClsEmptyAppInit**.

When the initialization routine creates the application class, it specifies the method table used by the application class.

In the method table, you establish a relationship between the messages that your class handles and the name of a function in your C code file that handles each message. These functions are called **message handlers** and are similar to the "methods" of other object-oriented systems. Message handlers should be local static routines that return STATUS. If your class does handle a message, the method table also indicates whether the Class Manager should call your class's ancestor before or after (if at all).

Message Handler Parameters 6.5.1.3

Because the Class Manager calls your message handlers, you don't get to choose message handler parameters. The arguments passed to all message handlers are:

msg The message itself.

self The object that received the message.

pArgs The message argument. This 32-bit value can be either a single argument or a pointer to a structure containing a number of arguments.

ctx A context maintained by the Class Manager.

pData The instance data of **self**.

Because the parameters to message handlers are always the same, \PENPOINT\SDK\INC\CLSMGR.H defines several macros to generate standard message handler functions, given only the function name (**MsgHandler**), or given the function name and types to cast its arguments to (**msgHandlerWithTypes**).

At the beginning of an application source file are these items:

- `#include` directives for the header files required by the application
- The internal routines used by your application's methods
- Internal `#defines`, and so on.

Empty Application's Source Code 6.5.2

Here's an abstract of the Empty Application's C code and method table file:

Method Table 6.5.2.1

The method table file, METHODS.TBL, specifies that Empty Application has one message handler; **clsEmptyApp** handles **msgDestroy** in a function called **EmptyAppDestroy**.

```
MSG_INFO clsEmptyAppMethods [] = {
#ifdef DEBUG
    msgDestroy,     "EmptyAppDestroy",  objCallAncestorAfter,
#endif
    0
};
```

The #ifdef and #endif statements cause the message handler to be defined only when you specify /DDEBUG in the compiler options.

C Source Code 6.5.2.2

There are three significant parts of EMPTYAPP.C:

- The main routine, which handles application installation and application startup
- The initialization routine, which is invoked by main at installation time
- The message handler for **msgDestroy**, which was specified in the method table.

This section presents this code without further comment. Subsequent sections in this chapter examine the code in detail.

The main routine for EMPTYAPP.C is:

```
/************************************************************************
    main

    Main application entry point (as a PROCESS -- the app's MsgProc
        is where messages show up once an instance is running).
************************************************************************/
void CDECL
main (
    int         argc,
    char *      argv[],
    U16         processCount)
{
    Dbg(Debugf("main: starting emptyapp.exe[%d]", processCount);)

    if (processCount == 0) {

        // Create application class.
        ClsEmptyAppInit();

        // Invoke app monitor to install this application.
        AppMonitorMain(clsEmptyApp, objNull);

    } else {

        // Create an application instance and dispatch messages.
        AppMain();
    }
    // Suppress compiler's "unused parameter" warnings
    Unused(argc); Unused(argv);
} /* main */
```

The initialization routine invoked by main on installation is:

```
/************************************************************************
  ClsEmptyAppInit

  Install the EmptyApp application class as a well-known UID.
************************************************************************/
STATUS
ClsEmptyAppInit (void)
{
```

```
    APP_MGR_NEW new;
    STATUS      s;

    //
    // Install the Empty App class as a descendant of clsApp.
    //
    ObjCallRet(msgNewDefaults, clsAppMgr, &new, s);
    new.object.uid              = clsEmptyApp;
    new.object.key              = (OBJ_KEY)clsEmptyAppTable;
    new.cls.pMsg                = clsEmptyAppTable;
    new.cls.ancestor            = clsApp;

    //
    // This class has no instance data, so its size is zero.
    //
    new.cls.size                = Nil(SIZEOF);

    //
    // This class has no msgNew arguments of its own.
    //
    new.cls.newArgsSize         = SizeOf(APP_NEW);
    new.appMgr.flags.accessory  = true;
    strcpy(new.appMgr.company, "GO Corporation");
    strcpy(new.appMgr.defaultDocName, "Empty App Document");
    ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);

    //
    // Turn on message tracing if flag is set.
    //
    if (DbgFlagGet('F', 0x1L)) {
        Debugf("Turning on message tracing for clsEmptyApp");
        (void)ObjCallWarn(msgTrace, clsEmptyApp, (P_ARGS) true);
    }

    return stsOK;

Error:
    return s;
} /* ClsEmptyAppInit */
```

Finally, the message handler for **msgDestroy** is:

```
/*****************************************************************************
    EmptyAppDestroy

    Respond to msgDestroy by printing a simple message if in DEBUG mode.
*****************************************************************************/
MsgHandler(EmptyAppDestroy)
{

#ifdef DEBUG
    Debugf("EmptyApp: app instance %p about to die!", self);
#endif

    //
    // The Class Manager will pass the message onto the ancestor
    // if we return a non-error status value.
    //
    return stsOK;
    MsgHandlerParametersNoWarning;  // suppress compiler warnings
} /* EmptyAppDestroy */
```

Libraries and Header Files 6.5.3

You interact with most of PenPoint by sending **messages** to objects. Thus a typical application only uses a few functions and only needs to be linked with APP.LIB and PENPOINT.LIB. However, you need to pick up the definitions of all the messages you send, status values you check, and objects to which you send messages from their respective header files.

Because Empty Application only looks for CLSMGR.H and APP.H messages, it only needs to include a few header files from \PENPOINT\SDK\INC:

Table 6-2
Common Header Files

Header File	Purpose
GO.H	Fundamental constants and utility macros in PenPoint.
OS.H	Operating System constants and macros.
DEBUG.H	Functions and macros to put debugging statements in your code.
APP.H	Messages defined by clsApp.
APPMGR.H	msgNew arguments of clsAppMgr used when an application class is created.
CLSMGR.H	Functions and macros that provide PenPoint's object oriented extensions to C.

Class UID 6.5.4

To write even the simplest application you must create your own application class, so that's primarily what Empty Application does.

Your application needs to have a **well-known UID** (Unique IDentifier, the "handle" on a Class Manager object) so the system can start it. All well-known UIDs contain a value that is administered by GO—this keeps them unique. When you finalize your application, you must contact GO Developer Technical Support for your own administered values. Until you register your application, you can use the predefined well-known UIDs that are set aside for testing. These test UIDs, **wknGDTa** through **wknGDTg**, are defined in \PENPOINT\SDK\INC\UID.H for this purpose. Just define your class to be one of them:

```
#define clsMyClass  wknGDTa
```

This is the approach that Empty Application takes. However, most other sample applications use well-known UIDs assigned to them by GO. Because most applications aren't part of the PenPoint API, these well-known UIDs don't show up in \PENPOINT\SDK\INC\UID.H.

You can use local well-known UIDs instead of global well-known UIDs for classes that your application uses internally. These do not contain an administered value; however, you must ensure that they remain unique within your application. (One bit in the UID indicates whether it is local or global, another indicates whether it is well-known or private—making essentially 2 to the 30th possible local well-known UIDs.)

Be on the lookout for conflicts with other test software when using the well-known testing UIDs (**wknGDTa** through **wknGDTg**). If another application should use the same well-known testing UID for one of its classes, you will have problems installing your application.

Class Creation 6.5.5

The initialization routine **ClsEmptyAppInit** creates the **clsEmptyApp** class. It also should look familiar to you from the discussion of classes in *Chapter 3, Application Concepts*. However, application classes are slightly different from other classes. You create most classes by sending **msgNew** to **clsClass**, whereas you create application classes by sending **msgNew** to **clsAppMgr**.

```
STATUS
ClsEmptyAppInit (void)
{
    APP_MGR_NEW new;
    STATUS      s;

    // Install the Empty App class as a descendant of clsApp.
    //
    ObjCallRet(msgNewDefaults, clsAppMgr, &new, s);
    new.object.uid              = clsEmptyApp;

...

    strcpy(new.appMgr.defaultDocName, "Empty App Document");
    ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);
```

clsAppMgr Explained 6.5.5.1

The PenPoint Application Framework needs to know a lot of things about an application before it can set in motion the machinery to create an instance of the application. It needs to know

- Whether the application supports embedding child applications
- Whether the application saves its data or runs continuously ("hot mode")
- Whether the application's documents appear as stationery or accessories
- The icon to use for the application's documents.
- The default name for the application's documents.

Tip Actually it is better to specify the default document name in the APP.RES file. This makes your application internationalizable.

Instances of the application class can't provide this information because the PenPoint Application Framework needs this information *before* it creates an application instance. To solve this cleanly, application classes are not instances of **clsClass**, but instead are instances of **clsAppMgr**, the application manager class. When an application is installed, its **clsAppMgr** instance is initialized, and this instance can supply the needed information.

```
new.cls.newArgsSize         = SizeOf(APP_NEW);
new.appMgr.flags.accessory  = true;
strcpy(new.appMgr.company, "GO Corporation");
strcpy(new.appMgr.defaultDocName, "Empty App Document");
ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);
```

Application classes should be well known so that other processes can send messages to them. Otherwise, the Notebook would not be able to send messages to your application class to create new documents when the user chooses it from the Create menu. You supply the UID for your application class in the **msgNew** arguments.

```
ObjCallRet(msgNewDefaults, clsAppMgr, &new, s);
new.object.uid              = clsEmptyApp;
new.object.key              = (OBJ_KEY)clsEmptyAppTable;
new.cls.pMsg                = clsEmptyAppTable;
new.cls.ancestor            = clsApp;
```

The key field is a way of protecting your application from accidental deinstallation. Only clients that know the key value that you used in the **msgNew** arguments will be able to deinstall your application.

The **class.pMsg** argument to **msgNew** establishes the connection between the new class and its method table. More on this later.

Documents, Accessories and Stationery 6.5.6

We have been referring to all copies of an application as **documents**. Not all documents in the system live on a page in the Notebook. Tools such as the clock and the personal dictionary float above the Notebook.

If you set **appMgr.flags.accessory** to **true**, **clsAppMgr** will put your application in the Accessories palette. When the user taps on your application's document icon, **clsApp** will insert the new document on screen as a floating document. If you set **appMgr.flags.stationery** to **true**, **clsAppMgr** will put a blank instance of your application in the Stationery notebook (whether or not your application has custom stationery). When the user selects and copies the stationery document from the Stationery palette, **clsApp** will insert the new document in the Notebook.

Tip *For debugging purposes, it's convenient to be able to create documents both as floating accessories and Notebook pages.*

Where Does the Application Class Come From? 6.6

The connection between a process running in PenPoint and an application class is not immediately obvious. You're probably wondering who calls the initialization routine for **clsEmptyApp**, who sends **msgNew** to create a new Empty Application instance, what process corresponds to this application instance, and why the familiar-looking C `main` routine doesn't do very much.

Installation and Activation 6.6.1

The connection between an application class and a PenPoint process is an application's `main` routine. Every executable must have a `main` routine; it is the routine that PenPoint calls when it creates a new process running your application's executable image.

```
void CDECL
main (
    int         argc,
    char *      argv[],
    U16         processCount)
{
    Dbg(Debugf("main: starting emptyapp.exe[%d]", processCount);)
```

The kernel keeps track of the number of processes running a particular program, and passes this to **main** as a parameter (**processCount**). For applications, there are two points at which PenPoint executes does this: **application installation** and **document activation.**

Application installation occurs when the user or APP.INI installs the application, that is, when PenPoint loads the application from disk into memory. No application documents are active at this point, but the code is present on the PenPoint computer.

Document activation occurs every time the user starts up a document that uses the application, typically by turning to its page.

When the user creates a document in the Notebook's TOC, PenPoint does *not* execute the application code, it merely creates a directory for the document in the application hierarchy. Try it: while turned to the TOC, create a new Empty Application document. The **Debugf**() statement in `main` does not print out anything until you turn to the document.

In MS-DOS, loading and executing code are part of the same operation; on a PenPoint computer, installing an application, creating documents for that application, and executing application code are three separate operations.

On MS-DOS, quitting an application is an action under the control of the user. In PenPoint, when the user turns away from a document, PenPoint determines whether it should destroy the application process or not. PenPoint does not keep running processes around for every application on every page, so it destroys processes that aren't active (thereby destroying application objects).

PenPoint starts and destroys application processes without the user's knowledge and, ideally, without any effect apparent to the user.

A Simple Discussion of main 6.6.1.1

When an application is installed, PenPoint creates a process and calls the application's **main** to run in the process. At this time, this is the only copy of the application running on the machine; thus, **processCount** contains the value 0. During installation, you should create your application class and any other classes you need. You then call **AppMonitorMain**, which handles application installation, import, copying stationery and resources, and so on. Empty Application doesn't take explicit advantage of any of these features, but other programs do.

```
    if (processCount == 0) {

        // Create application class.
        ClsEmptyAppInit();

        // Invoke app monitor to install this application.
        AppMonitorMain(clsEmptyApp, objNull);

    } else {
    ...
```

The process that PenPoint created at application installation keeps on running until PenPoint deactivates or deinstalls the application. Therefore, all subsequent processes that run the application's code will have **processCount** values greater than 0.

When a document is activated (typically by the user turning to its page), PenPoint calls `main` (**processCount** is greater than zero). At this point you should call the PenPoint Application Framework routine **AppMain**. This creates an instance of your application class and starts dispatching messages to it (and other objects created by the application) so that the new instance can receive Class Manager messages:

```
if (processCount == 0) {
    ...
    } else {

        // Create an application instance and dispatch messages.
        AppMain();
    }
} /* main */
```

Most applications follow these simple steps and have a `main` routine similar to the one in EMPTYAPP.C.

A Complex Explanation of main 6.6.1.2

The following paragraphs explain the process interactions taking place around `main`. Read on if you really want to understand how application start-up works.

Installation occurs when PenPoint reads \PENPOINT\BOOT\APP.INI (and SYSAPP.INI) and when the user installs applications using the Installer. PenPoint or the Installer calls the System Services routine **OSProgramInstall**, which loads the executable code for your application (EMPTYAPP.EXE) into a special area of PenPoint memory called the loader database. **OSProgramInstall** also creates a new PenPoint process and calls the function `main` with **processCount** equal to 0. At this point your code should initialize any information that all instances will need, such as its application class and any other non-system classes required by your application. The one thing every Empty Application instance needs is **clsEmptyApp** itself, hence when the `main` routine in EMPTYAPP.C is called with **processCount** of 0, it creates **clsEmptyApp**.

Application Installation 6.6.1.3

The process that PenPoint creates when **processCount** equals 0 also manages other application functions that are not specific to an individual document. These functions include copying stationery during installation, de-installation, file import, and so on. Rather than saddle your application with all these responsibilities, the PenPoint Application Framework provides a class, **clsAppMonitor**, which provides the correct default behavior for all these functions. When you call **AppMonitorMain** it creates one of these objects and dispatches messages to it. If your application needs to do more sophisticated installation (shared dictionaries, configuration, and so on), or can support file import, you can subclass **clsAppMonitor** and have a custom application installation manager.

Activation occurs when the user chooses Empty Application from the Tools notebook or the Stationery notebook, but in a roundabout fashion. The Notebook or bookshelf application sends **msgAppCreateChild** to the current selection. When **clsApp** receives this message, it creates a new slot in the application hierarchy for the new document. But a process and an application object aren't created until needed. The document may not be activated until the user turns to the document's page, or otherwise needs to interact with it.

Activating an Application 6.6.1.4

At or before the point where a live application instance is needed, the PenPoint Application Framework sends the application's parent **msgAppActivateChild**. While processing this, **clsApp** calls the System Services routine **OSProgramInstantiate**. **OSProgramInstantiate** creates a new PenPoint process, and in the context of that process it calls the function `main` with `processCount` set to a *non-zero* number.

Finally there is a running process for an Empty Application document! In theory, you could put any code you want in `main`, just like a vanilla C program. However, the *only* way a PenPoint application knows what to do—when to initialize, when it's about to go on-screen, when to file, etc.—is by messages sent to its application object. So, the first and only thing you need to do in `main` when `processCount` is non-zero is to create an instance of your application class and then go into a dispatch loop to receive messages. This is what the **AppMain** call does. From here on until the user turns away from the document and the application instance can be terminated, **AppMain** does not return.

Handling a Message 6.7

clsEmptyApp only responds to one message. That doesn't mean that Empty Application documents don't receive messages—if you turned on tracing while running Empty Application, you'll have seen the dozens of messages that an Empty Application application instance receives during a page turn. It means only that **clsEmptyApp** lets its ancestor take care of all messages, and it turns out that **clsApp** does an excellent job of handling PenPoint Application Framework messages.

A real application or other class has to intercept some messages, otherwise it has the same behavior as its parent class. In the case of an application class, the application needs to respond to PenPoint Application Framework messages that tell documents when to start up, when to restore themselves from the file system, when they are about to go on-screen, and so on. If the application has standard application menus (SAMs), it will receive messages such as **msgAppPrint**, **msgAppPrintSetup**, and **msgAppAbout**, from the buttons in the menus.

Often, the class responds to these messages by creating, destroying, or filing other objects used by the application. EMPTYAPP.C doesn't do any of this; all it does is print a string when it receives one particular message, **msgDestroy**.

Method Table

6.7.1

Objects of your classes (especially application instances) receive lots of messages regardless of whether or not you want your class to deal with those messages. Your class' method table tells the Class Manager which messages your class intercepts.

This code sample is from Empty Application's method table file (METHODS.TBL):

```
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

MSG_INFO clsEmptyAppMethods [] = {
#ifdef DEBUG
    msgDestroy,         "EmptyAppDestroy",  objCallAncestorAfter,
#endif
    0
};

CLASS_INFO classInfo[] = {
    "clsEmptyAppTable",     clsEmptyAppMethods,     0,
    0
};
```

This basically says "If an instance of **clsEmptyApp** receives **msgDestroy**, call **EmptyAppDestroy**, then pass the message to **clsEmptyApp**'s ancestor."

The link between the functions in a method table and a particular class is established by one of the **msgNew** arguments when you create the class (**new.class.pMsg**). This is the name you associate with the class's MSG_INFO array in the CLASS_INFO array; in this example, the **pMsg** is **clsEmptyAppTable**. This code sample is from **ClsEmptyAppInit** in EMPTYAPP.C:

```
// Install the Empty App class as a descendant of clsApp.
//
ObjCallRet(msgNewDefaults, clsAppMgr, &new, s);
new.object.uid              = clsEmptyApp;
new.object.key              = (OBJ_KEY)clsEmptyAppTable;
new.cls.pMsg                = clsEmptyAppTable;
new.cls.ancestor            = clsApp,
...
```

msgDestroy 6.7.2

The names of most messages identify the class that defined them: for example, **msgAppOpen** is defined by **clsApp**. Messages defined by the Class Manager itself are the exception to this convention. **msgDestroy** is defined by the Class Manager in \PENPOINT\SDK\INC\CLSMGR.H; this is why Empty Application's METHODS.TBL `#includes` this header file. The Class Manager responds to **msgDestroy** by destroying the object that received **msgDestroy**.

*The Class Manager actually turns around and sends the object another message, **msgFree**, to free the object.*

Message Handler 6.8

The message handler (also known as **method**) is just a C routine you write that does something in response to the **message**. Empty Application's message handler for **msgDestroy** is **EmptyAppDestroy**, which just prints a string to the debugger stream.

The name you give the message handler must match the name you specified in the method table (**EmptyAppDestroy**).

Parameters 6.8.1

The parameters that the Class Manager passes to a message handler are:

msg The message received by the instance

self The UID of the instance that received the message

pArgs The message arguments passed along with the message by the sender of the message

ctx A context that helps the Class Manager keep track of the class in the instance's hierarchy that is currently processing the message

pData A pointer to the **instance data**, information specific to the instance whose format is defined by the class.

Here's the definition from CLSMGR.H:

```
//  Definition of a pointer to a method.
typedef STATUS (CDECL * P_MSG_HANDLER) (
    MESSAGE     msg,
    OBJECT      self,
    P_ARGS      pArgs,
    CONTEXT     ctx,
    P_IDATA     pData
);
```

You never call your message handlers, the Class Manager does, and always with the same set of parameters. The PenPoint Method Table Compiler generates a header file containing function prototypes for all the message handlers specified in the message table; you can guard against accidentally leaving out a parameter by including these files in your class implementation C files:

```
#include <debug.h>              // for debugging statements.
#include <app.h>                // for application messages (and clsmgr.h)
#include <appmgr.h>             // for AppMgr startup stuff
#include <string.h>             // for strcpy().
#include <method.h>             // method function prototypes generated by MT
```

MsgHandler is a macro that expands into the correct definition of a pointer to a message handler. It saves you typing all these parameters.

```
/************************************************************************
    EmptyAppDestroy

    Respond to msgDestroy by printing a simple message if in DEBUG mode.
************************************************************************/
MsgHandler(EmptyAppDestroy)
{
```

Parameters in EmptyAppDestroy 6.8.2

It turns out that Empty Application's **EmptyAppDestroy** routine doesn't need most of the parameters. The informative string prints out the UID of self (the Empty Application document that received the message) and doesn't use the rest of the parameters.

```
#ifdef DEBUG
    Debugf("EmptyApp: app instance %p about to die!", self);
#endif
```

We aren't interested in the **msg**, since the Class Manager should only call this function with **msgDestroy**. **clsEmptyApp** has no instance data, so we don't need **pData**. (Remember, we specified that **class.size** is 0 when we created **clsEmptyApp**.) Although we don't need these parameters, there is no way to tell the class manager not to send them.

The C compiler will warn about unused parameters in functions. Since many message handlers won't use all their parameters, CLSMGR.H defines a fragment of code, **MsgHandlerParametersNoWarning**, which mentions each parameter. You can stick this in your message handler at any point.

```
    MsgHandlerParametersNoWarning;      // suppress compiler warnings
} /* EmptyAppDestroy */
```

Status Return Value 6.8.3

Message handlers are supposed to return a status value. This is important both to indicate to the sender of the message that the message was handled successfully, and to control how the Class Manager passes the message up the class ancestry chain. Empty Application's method table directed the Class Manager to pass **msgDestroy** to **clsEmptyApp**'s ancestor after calling Empty Application's handler:

```
    msgDestroy,                 "EmptyAppDestroy",  objCallAncestorAfter,
```

If **EmptyAppDestroy** were to return an error status value, the Class Manager would not call the ancestor, and the normal result of sending **msgDestroy** would be pre-empted (the application object would not go away). Sometimes this is what you want, but not in this case, so we return **stsOK**.

```
    // The Class Manager will pass the message onto the ancestor
    // if we return a non-error status value.
    return stsOK;
```

Message Handlers are Private 6.8.4

Although message handlers are just regular C functions, you normally do *not* want other code to call your message handlers. One of the goals of object-oriented programming is to hide the implementation of functionality from clients of that functionality. Clients should communicate with your objects by sending them messages, not by calling your functions. That way you can change the names and implementation of a message handler without affecting clients of your API.

Using Debug Stream Output 6.9

There are two main ways to debug programs in PenPoint:

- Send data to the debugger stream
- Use the PenPoint Source-Level Debugger

Additionally, you can use the PenPoint mini-debugger, which is part of PenPoint, but is most useful when debugging kernel and device-interface code.

Note that you can't use standard DOS-type debugging tools (such as WATCOM Debugger or Microsoft CodeView) because these packages require your executable file to be runnable under DOS.

This section discusses sending data to the debugger stream. For a complete tutorial on how to use the PenPoint Source-Level Debugger (DB) and mini-debugger (mini-DB), see the part on debugging in *PenPoint Development Tools*.

The Debugger Stream 6.10

You can send data to the debugger stream with **Debugf** and **DPrintf** statements in your code. This is much like debugging a DOS application by adding `printf` statements to the code.

EMPTYAPP.C uses the system debugging output function **Debugf** to print strings to the debug stream (Empty Application doesn't use its PenPoint windows to display anything).

Debugf is much like `the standard C function printf`. The `%p` formatting code in the format string means "print this out as a 32-bit hexadecimal pointer." Because UIDs such as **self** are 32 bits, this is a quick and dirty way to print a UID value. The Class Manager defines routines that convert UIDs to more meaningful values which this application could have used instead; the message tracing and status warning debugging facilities use these fancier output formats.

Seeing Debug Output 6.10.1

There are several ways to view the information sent to the debugger stream:

- If you press Pause while running PenPoint, your screen will switch from graphics to text display and you will see strings that have been written to the debugger stream.

- If you have a second monitor and do not set monodebug=off in your MIL.INI file, debugger stream data is displayed on the second monitor.
- If you turn on the 8000 bit in the D debug flag, debugging strings will be copied to the file \PENPOINT.LOG on **theBootVolume** (the volume specified with VOLSEL in ENVIRON.INI).
- You can run the System Log application.

The System Log application is a PenPoint application that allows you to review data sent to the debugger stream. To use it, install it by uncommenting it in SYSAPP.INI or by installing from disk (just as you install any other application in PenPoint). When the System Log application is installed, it adds its icon to the Accessories window. Tap on the icon to open the application.

Debug strings appear in the System Log application. You can scroll up and down to see its contents.

You can also check flags, see available memory, and set flags from the System Log application. To learn more about the System Log application, see the Part on debugging in *PenPoint Developer Tools.*

Chapter 7 / Creating Objects (Hello World: Toolkit)

Although Empty App shows that the Application Framework can do many things for an application, Empty App is still rather boring, in that it doesn't contain anything or show anything on screen. This chapter describes how to create objects. It so happens that these objects also display things on screen.

A standard, simple test program is one that prints "Hello World." In PenPoint, there are two different ways to approach this:

- Use PenPoint's UI Toolkit to create a standard **label** that contains the text.
- Create a window and draw text in it using text and drawing services provided by the ImagePoint imaging model.

These two styles mirror two general classes of program. Programs such as database programs and forms can use standard user interface components to create dialogs with the user. Programs such as presentation packages and graphics editors do a lot of their own drawing. They need to create a special kind of window and draw in it.

This chapter shows the first approach; the application **clsHelloWorld** calls on the UI Toolkit to create a label object. The next chapter describes how to create a window and draw in it (and also discusses how to create a new class).

Even programs that do use custom windows will make heavy use of the UI Toolkit. Every application has a **menu bar** with standard **menu buttons**, a **frame**, and at least one **option sheet**, and most programs will add to these to implement other controls and dialogs with the user.

An application can choose not to use these, but doing so involves extra work and goes against GO's User Interface guidelines.

Moreover, using the UI Toolkit is much simpler than using a window. The toolkit component classes are *all* descendants of **clsWin**, the class which supports overlapping windows on the screen (and printer). But they know how and when to draw themselves and file themselves, so there's very little you need to do besides create them and put them in your application's frame.

HelloTK

7.1

Hello World (toolkit) uses UI Toolkit components to display the words "Hello World!" These components know how to draw themselves and position themselves. Consequently, it's extremely simple to create the application.

The directory \PENPOINT\SDK\SAMPLE\HELLOTK actually contains two different versions of Hello World (toolkit). The first version, HELLOTK1.C, creates a single label in its frame. Usually you want to put several windows in a frame; this is more complex and is handled by HELLOTK2.C.

Compiling and Installing the Application 7.1.1

Both versions of Hello World (toolkit) (HELLOTK1.C and HELLOTK2.C) have a single C file. Consequently, compiling, downloading, and running it are the same as for Empty Application. Because there are multiple versions of the code, copy the version you want to run to HELLOTK.C before running WMAKE:

```
> copy hellotk1.c hellotk.c
> wmake
```

This creates a \PENPOINT\APP\HELLOTK directory and compiles a HELLOTK.EXE file in it. It uses STAMP to give the directory the long name "Hello World (toolkit)" and the .EXE the long name "Hello World (toolkit).exe."

Install Hello World (toolkit) either by adding \\BOOT\PENPOINT\APP\Hello World (toolkit) to \PENPOINT\BOOT\APP.INI before starting PenPoint or by installing the application using the Installer.

Create Hello World (toolkit) application instances from the Stationery notebook, from the stationery quick menu, or from the Accessory palette.

Interesting Things You Can Do with HelloTK 7.1.2

Alas, Hello World (toolkit) doesn't do much more than Empty Application besides display a label. It doesn't do anything *less*, so you can create multiple instances of it as accessories or as pages in the Notebook, you can trace messages to it (by setting the `F` flag to `0x20`), and so on.

The only new thing to do is to notice how the label draws itself. Try zooming or resizing a Hello World (toolkit) document.

Code Run-Through for HELLOTK1.C 7.2

HELLOTK1.C creates a single label in its frame.

Highlights of HELLOTK1 7.2.1

The method table for Hello World (toolkit) only responds to one message, **msgAppInit**.

```
msgAppInit, "HelloAppInit", objCallAncestorBefore,
```

In order to avoid clashing with other Hello World applications, HELLOTK1.C uses a different testing well-known UID.

```
#define clsHelloWorld   wknGDTb // avoids clashing with other HelloWorlds
```

Most of the work is done in the message handler **HelloAppInit**, which responds to **msgAppInit** by creating the client window (a label).

So that it can use the same method table as HELLOTK2.C, HELLOTK1.C responds to **msgAppOpen** and **msgAppClose** as well as **msgAppInit**; however, it does nothing with these messages but return **stsOK**.

The only significant thing that happens in Hello World (toolkit) is that it responds to **msgAppInit** by creating a label. The code to do this is very simple, about 35 lines, but deciding what to do in those few lines introduces several key concepts in PenPoint application development:

- Choosing what classes to use
- Deciding when to create objects.

It also involves some common programming techniques:

- Creating an instance of a class
- Sending messages to self.

Sending Messages 7.2.2

Empty Application receives messages, but does not send messages. Often in responding to a message, your application must send other messages. It might send messages to other objects, or even send itself messages to get its ancestor classes to do things. Hello World (toolkit) shows how to send a few simple messages.

ObjectCall 7.2.2.1

Use **ObjectCall** to pass a message to another object in your process. This works like a function call: the thread of control in your application's process continues in the message handler of the other object's class, and returns to your code when the other object's class returns a status value to your code.

There are other ways to send a message:

- Asynchronously
- Using the input queue
- Between processes.

In a simple application, stick to **ObjectCall**.

Testing Return Values and Debugging 7.2.2.2

Because messages return a status value, you should usually check their return values. This would ordinarily lead to lots and lots of constructs such as the following in your code:

```
if ((s = ObjectCall(msgXxx, someObject, &args) < stsOK) {
    // Print standard warning if DEBUG set
    // Handle error...
}
```

To save typing and code complexity, for every Class Manager function that returns a status value, there are macro versions of the function that jump to an error handler, or return true if there's an error, etc. For **ObjectCall**, these are **ObjCallWarn**, **ObjCallRet**, **ObjCallJmp**, **ObjCallChk**, and **ObjCallOK**.

ObjCallWarn's value is the status value returned by **ObjectCall**. If compiled with the DEBUG flag set, then **ObjCallWarn** prints out an error string if the status value is an error (that is, less than **stsOK**).

The other macros incorporate **ObjCallWarn** into their behavior:

ObjCallRet calls ObjCallWarn and then returns the status value if it is an error

ObjCallJmp calls ObjCallWarn and then jumps to a error label (where you can handle the error) if the status value is an error

ObjCallChk calls ObjCallWarn and then returns the value **true** if the status value is an error

ObjCallOK calls ObjCallWarn and then returns the value **true** if the status value is not an error (that is, greater than or equal to **stsOK**).

Creating Toolkit Components

7.2.3

HELLOTK1.C responds to **msgAppInit** by creating a **label**. Labels are one of the many components provided by the UI Toolkit. But why does it create this particular kind of component?

What Kind of Component?

7.2.3.1

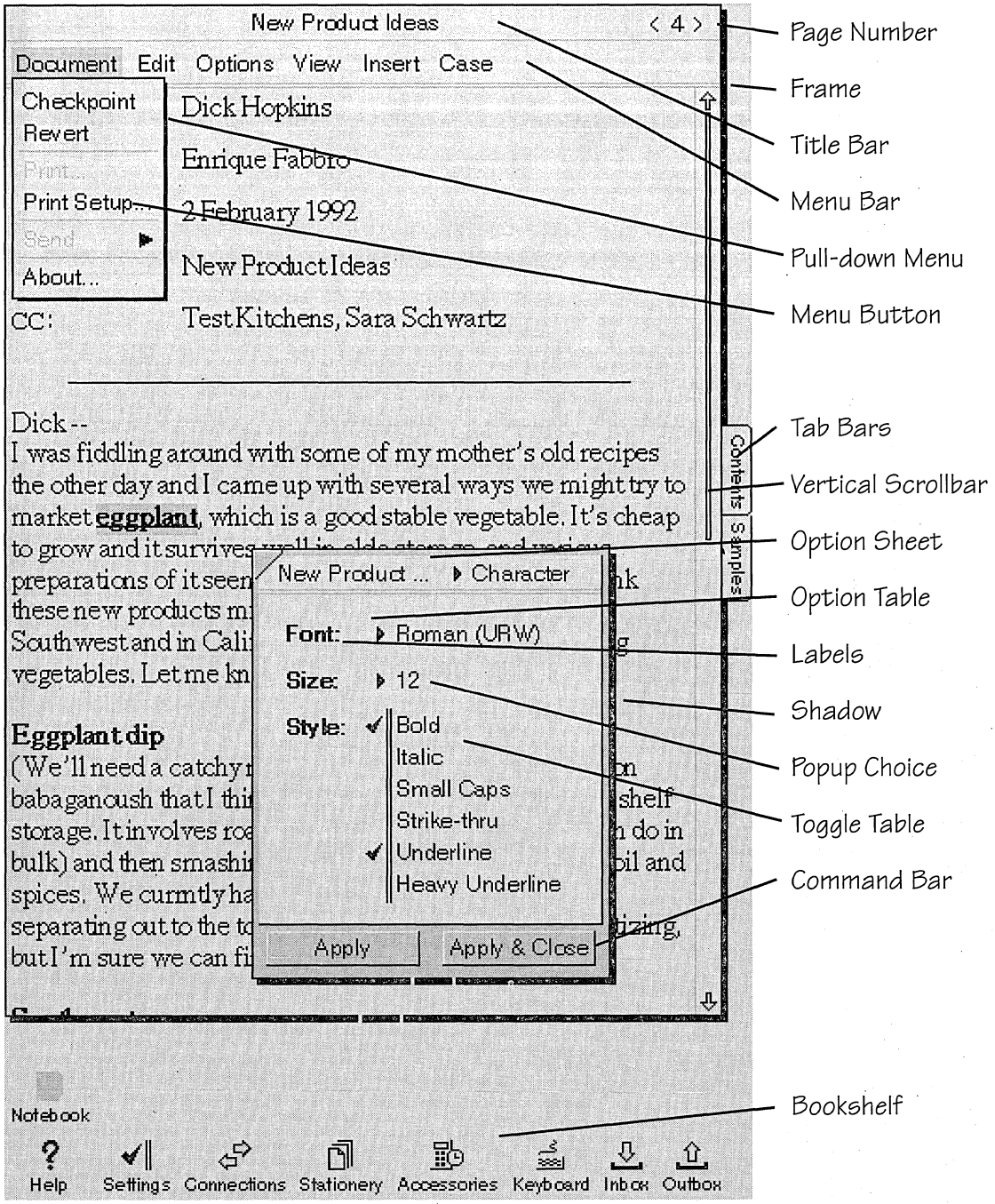
It's worth taking a close look at the class hierarchy poster to see all the toolkit classes.

Most of the UI Toolkit classes are windows. There's only one class in the system that knows how to do windows (multiple overlapping regions on a pixel device), and that's **clsWin**. But it's the descendants of **clsWin** that know how to draw something interesting in themselves. The toolkit components inherit from **clsBorder**, a special kind of window which knows how to draw a border. You'll find the toolkit classes "under" **clsBorder** in the class list, class diagram, and class browser.

Tip *Some of the key decisions you make in any object-oriented programming system are choosing what built-in classes to use and which built-in classes to subclass.*

Part 4: UI Toolkit of the *PenPoint Architectural Reference* explains the UI Toolkit in all its multi-level glory. For a hint of what it can do, here's a screen shot indicating all the different kinds of UI Toolkit components present:

Figure 7-1
UI Toolkit Components



There are many other classes in the UI Toolkit. There are several base classes that provide lower-level functionality. And there are many specialized components classes, such as date handwriting input fields.

For Hello World (toolkit), all we need is a class that can display a string, such as **clsLabel**.

To learn more about a class, you can try to

- Use the class browser to get a brief description of it and all its messages
- Read about it in its subsystem's Part of the *PenPoint Architectural Reference*
- Look up its "datasheets" in the *PenPoint API Reference*
- Look at its header file in \PENPOINT\SDK\INC.

The class browser, the header, and the documentation all give you the information you need to create an instance of the class.

msgNew Arguments for clsLabel 7.2.3.2

As you learned in *Chapter 3, PenPoint Application Concepts*, you create objects by sending **msgNew** to their class. Different classes allow different kinds of initialization, so you pass different arguments to different classes. The documentation states what **message arguments** a given class needs for **msgNew**. In the header file the information is expressed as follows:

```
msgNew  takes P_LABEL_NEW, returns STATUS
```

This says that you should pass in a pointer to a LABEL_NEW structure when you send **msgNew** to **clsLabel**. What you typically do is declare a LABEL_NEW structure in the routine which sends **msgNew**. You can give this any variable name you want; Hello World (toolkit) calls it `ln` (the first letter of each part of the structure name). At the top of HelloAppInit:

```
/************************************************************************
    HelloAppInit

    Respond to msgAppInit by creating the client window (a label).
************************************************************************/
MsgHandler(HelloAppInit)
{
    APP_METRICS             am;
    LABEL_NEW               ln;
    STATUS                  s;
```

Before you send **msgNew** to a class, you must *always* send **msgNewDefaults** to that class. This takes the same message arguments as **msgNew** (a pointer to a LABEL_NEW structure in this case). This gives the class and its ancestors a chance to initialize the structure to the appropriate default values. It saves your code from initializing the dozens of fields in a _NEW structure.

```
    // Create the Hello label window.
    ObjCallWarn(msgNewDefaults, clsLabel, &ln);
```

Now you're ready to give values to those fields in the structure which you care about. Figuring out what's in a _NEW structure is not easy. It contains initialization information for the class you are sending it to, along with initialization information for that class's ancestor, and for its ancestor's ancestor, all the way to initialization arguments for **clsObject**. Sometimes the only initializations you're interested in are the ones for the class you've chosen, but in

Tip If you have a programmable editor, you can use tags to quickly jump to structure definitions. See \PENPOINT\SDK\UTIL\TAGS\TAGS.DOC for more information.

the case of the UI Toolkit, you often have to reach back and initialize fields for several of the ancestor classes as well.

```
ln.label.style.scaleUnits   = bsUnitsFitWindowProper;
ln.label.style.xAlignment   = lsAlignCenter;
ln.label.style.yAlignment   = lsAlignCenter;
ln.label.pString            = "Hello World!";
```

You can look up the hierarchy for a class by looking in the API Reference for that class. The description of the _NEW structure for **msgNew** always gives the _NEW_ONLY structures that make up the _NEW structure. Thus, the hierarchy for **clsLabel** expands to:

```
LABEL_NEW {
    OBJECT_NEW_ONLY         object;
    WIN_NEW_ONLY            win;
    GWIN_NEW_ONLY           gWin;
    EMBEDDED_WIN_NEW_ONLY   embeddedWin;
    BORDER_NEW_ONLY         border;
    CONTROL_NEW_ONLY        control;
    LABEL_NEW_ONLY          label;
}
```

When in doubt, rely on **msgNewDefaults** to set up the appropriate initialization, and modify as little as possible.

All you need do to create a label is pass **clsLabel** a pointer to a string to give the string a label. However, the LABEL_STYLE structure contains various **style fields** which also let you change the way the label looks.

We want the text to fill the entire window, so the **scaleUnits** field looks promising. This is a bit field in LABEL_STYLE, but rather than hard-code numeric values for these in your code, LABEL.H defines the possible values it can take. One of these is **lsScaleFitWindowProper**. This tells **clsLabel** to paint the label so that it fills the window, but keeping the horizontal and vertical scaling the same. Other style fields control the alignment of the text string within the label. In this example, we'd like to center the label.

By the way, one reason that **clsLabel** has so many style settings and other **msgNew** arguments is that many other toolkit components use it to draw their text, either by creating lots of labels or by inheriting from **clsLabel**. Thus **clsLabel** draws the text in tab bars, in fields, in notes, and so on:

```
// Create the Hello label window.
ObjCallRet(msgNewDefaults, clsLabel, &ln, s);
ln.label.style.scaleUnits   = bsUnitsFitWindowProper;
ln.label.style.xAlignment   = lsAlignCenter;
ln.label.style.yAlignment   = lsAlignCenter;
ln.label.pString            = "Hello World!";
ObjCallRet(msgNew, clsLabel, &ln, s);
```

Now the label window object exists. The Class Manager passes back its UID in `ln.object.uid`. But at this point it doesn't have a **parent**, so it won't ever show up on-screen.

Where the Window Goes 7.2.4

Empty Application appeared on-screen even though it didn't create any windows itself. The Application Framework creates a **frame** for a document. Frames are a UI Toolkit component. A frame can include other windows. Empty Application's frame has a title bar, page number, and resize boxes; you've seen other applications whose frames also include **tab bars**, **command bars**, and **menu bars**.

Most importantly, a frame can contain a **client window**, the large central area in a frame. Empty Application didn't supply a client window (hence it looked pretty dull).

Hello World (toolkit) wants the label it creates to be the client window. The message **msgFrameSetClientWin** sets a frame's client window. But the label must have its frame's UID to send a message to its frame. Hello World (toolkit) didn't create the frame, its ancestor **clsApp** did.

clsApp does not define a message to get the main window. Instead, it provides a message to get diverse information about application instances, including the main window of that application. (An application can have a different main window for itself other than a frame.

Information made public about instances of a class is often called **metrics**, and the message to get this information for an application is **msgAppGetMetrics**. **msgAppGetMetrics** takes a pointer to an APP_METRICS structure, one of the fields in the structure is **mainWin**. Here is how HelloAppInit gets its main window:

```
APP_METRICS             am;
...
// Get the app's main window (its frame).
ObjCallJmp(msgAppGetMetrics, self, &am, s, error);
// Insert the label in the frame as its client window.
ObjCallJmp(msgFrameSetClientWin, am.mainWin, \
                    (P_ARGS)ln.object.uid, s, error);
```

Note that the code sends **msgAppGetMetrics** to **self**. We have been talking loosely about Hello World (toolkit) doing this and that, but remember that this code is run as a result of an instance of **clsHelloWorld** receiving a message, and that **clsHelloWorld** is a descendant of **clsApp**. Thus the document is the application object to which we want to send **msgAppGetMetrics**. In the middle of responding to one message (**msgAppInit**), we need to send a message to the same object which received the message. This is actually very common. The Class Manager provides a parameter to methods, **self**, which identifies the object which received the message.

Why msgAppInit? 7.2.5

Earlier you turned on message tracing to Empty Application. What this does is cause the class manager to dump out every message received by instances of **clsEmptyApp**. You should have noticed that each Empty Application document receives dozens of messages during the course of a page turn to or from itself.

These messages are sent to documents (application instances) by the PenPoint Application Framework.

If you want your application to do something, you must figure out when to do it. Your process can't take over the machine and do whatever it wants, it must do what it wants in response to the appropriate messages.

One of the hardest things in PenPoint programming is figuring out when to do things.

So, when should Hello World (toolkit) create its label? Because it inserts the label in its frame (using **msgFrameSetClientWin**), it can't create the label before it has a frame. But it should have a label in its frame before it goes on screen.

It turns out that **clsApp** creates the document's frame in response to **msgAppInit**. Thus Hello World (toolkit) can get its frame and insert the label in its **msgAppInit** handler, but it must do so *after* **clsApp** has responded to the message. This is why its method table tells the Class Manager to first send the message to its ancestor:

```
MSG_INFO clsHelloMethods [] = {
    msgAppInit,                 "HelloAppInit", objCallAncestorBefore,
    msgAppOpen,                 "HelloOpen",    objCallAncestorAfter,
```

Note that doing this relies on knowing what the ancestor class does. You'll spend a lot of time reading *Part 2: Application Framework* of the *PenPoint Architectural Reference* to learn about the PenPoint Application Framework messages and how **clsApp** responds to them.

Why Did the Window Appear?

7.2.6

If you're familiar with other window systems, you may be wondering how the label gets sized, positioned, and made visible on screen. These will be explained during the development of other tutorial programs. But here's a summary.

When the application is about to go on screen it receives **msgAppOpen**. **clsApp** inserts the main window (the frame) in the Notebook's window and tells it to **lay out**. **clsFrame** takes care of sizing and positioning its title bar, page number, move box, and client window (the label). Each of these windows is sent a message by the window system to repaint itself when it is exposed on screen. **clsLabel** responds to the repaint message by painting its label string. Thus all you need to do is put a toolkit window inside your frame, and the system takes care of the rest for you.

Possible Enhancements

7.2.7

You can change the class of the window created in **HelloAppInit** to be some other kind of window class by changing the class to which Hello World (toolkit) sends **msgNewDefaults** and **msgNew**. But different classes take different message arguments when they are created. You need to replace the declaration of a LABEL_NEW structure with the **msgNew** arguments of the new class.

Warning Passing the wrong message arguments with a message is one of the more common errors in PenPoint programming. The C compiler will not catch the error.

If the class handling the message expects different arguments, it will blindly read past the end of the structure you passed it, and if it passes back values, it will overwrite random memory. A given class receiving a given messsage *has to* be given a pointer to the appropriate structure, otherwise unpredictable results will occur: but it can't *enforce* this.

There are many classes which inherit from **clsLabel**, consequently, if you used one of these, you wouldn't even have to change the initialization of the structure. For example, **clsField** inherits from **clsLabel**, and FIELD_NEW includes the same NEW_ONLY structures as LABEL_NEW, so it takes the same border and label specifications.

Highlights of the Second HelloTK 7.3

HELLOTK2.C is much like HELLOTK1.C. The big difference is that it supports more than one window. Most applications have many windows within their frame.

You compile and run it the same way. Just copy HELLOTK2.C to HELLOTK.C and follow the steps outlined above.

Only One Client Window per Frame 7.3.1

Frames only support a single client window. But usually you'll want several windows in your application. You have two alternatives:

- Subclass **clsFrame** (which is very difficult)
- Create a client window that is another window, then insert all the windows you want into that client window (which is quite easy).

The toolkit provides two window classes that help you organize the windows within the client window. These are called layout windows. To understand why they're needed, you need to know a little bit about **layout**.

Layout 7.3.2

When you're using several windows, something is responsible for positioning them on the screen. You can set a window's position and size to some value with **msgWinDelta**. However, if the user changes the system font size, or resizes the frame, or changes from portrait to landscape mode, the numbers you pick are unlikely to still be appropriate. It's more convenient to specify window locations at an abstract level:

- "I want this window below that one, and extending to the edge of that other one."
- "Position these windows in two columns of equal width."

The UI toolkit provides two layout classes which support these styles, **clsCustomLayout** and **clsTableLayout**. Both are packed with features. Both lay out their own child windows according to the constraints (for custom layout) or algorithm (for table layout) which you specify. The general way of using layout windows is to create one, specify the layout you want, and insert the windows in it.

HELLOTK2.C uses a custom layout window and positions a single label in its center using `ClAlign(clCenter, clSameAs, clCenter)`.

```
// Specify how the custom layout window should position the label.
CstmLayoutSpecInit(&(cs.metrics));
cs.child = ln.object.uid;
cs.metrics.x.constraint = ClAlign(clCenterEdge, clSameAs, clCenterEdge);
cs.metrics.y.constraint = ClAlign(clCenterEdge, clSameAs, clCenterEdge);
cs.metrics.w.constraint = clAsIs;
cs.metrics.h.constraint = clAsIs;
ObjCallJmp(msgCstmLayoutSetChildSpec, cn.object.uid, &cs, s, error2);
```

Possible Enhancements 7.3.3

You might consider trying to add one of these to HELLOTK2.C:

Fields 7.3.3.1

Change the label to be an editable field. There are several ways of handling handwriting in PenPoint. One way is to use a UI component which allows editing, **clsField**. Since fields have similar behavior to labels (they display a string, have a length, font, and so on), **clsField** inherits from **clsLabel**. This makes it easy to update the application: replace the LABEL_NEW structure with FIELD_NEW, and **clsLabel** with **clsField**, and recompile. You can now hand-write into the field.

More Components 7.3.3.2

Add some more controls, using different custom layout constraints. You should be able to put together a simple control panel.

General Model of Controls 7.3.3.3

You specify the **metrics** of each control when you create it, then you insert them in your layout window. The controls lay themselves out, repaint themselves, and support user interaction without any intervention on your part. When the user activates a control, the control sends its client (set in the **msgNew** arguments of **clsControl**, or by **msgControlSetClient**) a notification message.

For more information on controls, see *Part 4: UI Toolkit* of the *PenPoint Architectural Reference.*

Chapter 8 / Creating A New Class (Hello World: Custom Window)

This chapter describes how you create a new class. Along the way, the chapter also describes how you display the string "Hello World!" on screen by creating and drawing in custom windows.

Hello World (Custom Window) 8.1

Hello World (custom Window) creates an instance of a custom window and uses the custom window to display some text.

It's still not a very realistic application because it doesn't file any data, but it does use an additional class, a descendant of **clsWin**, to do its drawing. Your application may be able to use only standard UI components from the UI Toolkit and other PenPoint subsystems; but if not, you will be creating your own window class to draw stuff.

So far our example applications have been quite simple and have not needed to define their own classes (apart from creating a subclass of **clsApp**). One of the big advantages in object oriented programming is that when you *do* define a class, other applications can create instances of the class (rather than defining new classes on their own).

So that other applications can use the new class, developers often define each class in a single C file and then compile and link one or more C files into a DLL. The C file that contains the application class (and has `main`) is compiled into an executable file.

To show this coding style, Hello World (custom Window) is implemented as an application and a separate DLL. There are two parts to Hello World (custom Window), **clsHelloWorld** (the application class), and **clsHelloWin** (the window class). HELLO.C implements **clsHelloWorld** and HELLOWIN.C defines **clsHelloWin**. HELTBL.TBL contains the method table for **clsHelloWorld**; HELWTBL.TBL contains the method table for **clsHelloWin**.

Compiling the Code 8.1.1

Compiling and linking the Hello World (custom Window) executable is somewhat similar to compiling Empty Application. However, Hello World (custom Window) is compiled and linked in two parts: an EXE file that contains the application, and a DLL file that contains the window class (**clsHelloWin**).

You can build Hello World (custom Window) by changing directory to \PENPOINT\SDK\SAMPLE\HELLO and typing **wmake**.

Note that because the application class, **clsHelloWorld**, and its window class are in different files, compiling is more efficient if they have separate method table files (HELTBL.TBL and HELWTBL.TBL).

Linking DLLs

8.1.1.1

When you link DLLs (dynamic link libraries), the information you provide the linker is slightly different from the information you provide when linking an executable image. If you use the makefile to build Hello World (custom window), the file builds the command file for you. If you created the command file by hand, it would contain:

```
SYSTEM PenPoint DLL
NAME \386\PENPOINT\app\hello\hello.dll
DEBUG ALL
FILE helwtbl.obj
FILE dllinit.obj
FILE hellowin.obj
EXPORT=dll
LIBRARY penpoint
LIBRARY win
SEGMENT CLASS 'CODE' Share, CLASS 'DATA' Share, CLASS 'BSS' Share
OPTION Modname='GO-HELLO_DLL-V1(0)'
OPTION Map=hello.mpd
OPTION Quiet,NOD,Verbose
```

For contrast, here is the command file for the Hello World (custom window) executable file:

```
SYSTEM PenPoint
NAME \386\PENPOINT\app\hello\hello.exe
DEBUG ALL
FILE heltbl.obj
FILE hello.obj
LIBRARY penpoint
LIBRARY app
OPTION Quiet, Map=hello.mpe, NOD, Verbose, Stack=15000, MODNAME='GO-HELLO_EXE-V1(0)'
```

In addition to this command file, the WATCOM linker also requires a DLL.LBC file. This file lists all the exported functions defined in the DLL being linked. Usually, PenPoint DLLs only have the single entry point **DLLMain**. The lines in the DLL.LBC file have the form:

```
++entry-point.go-lname
```

The case doesn't matter in the DLL.LBC file.

The *go-lname* is used by the PenPoint installer to identify code modules. An **lname** is composed of a company ID, a project name, and a revision number in the form `Vmajor(minor)`. Where ***major*** is the major revision number and minor is the ***minor*** revision number.

Thus, for Hello World (custom window), the DLL.LBC file contains the single line:

```
++DLLMAIN.'GO-HELLO_DLL-V1(0)'
```

DLC Files 8.1.1.2

Because Hello World (custom window) requires that HELLO.DLL be loaded before HELLO.EXE can run, you need to have a HELLO.DLC file in the Hello World (custom window) application directory that expresses the relationship:

```
GO-HELLO_DLL-V1(0)  hello.dll
GO-HELLO_EXE-V1(0)  hello.exe
```

The PenPoint installer uses this information when installing the Hello World (custom window) application. The first line indicates that the Hello World (custom window) application depends on the DLL file HELLO.DLL, version 1(0). Should this DLL already be loaded, PenPoint will not attempt to load it. The second line tells PenPoint to install the executable file HELLO.EXE.

Because the PenPoint name of the application directory is "Hello World," the makefile must STAMP the .DLC file with the name "Hello World" so that the Installer will find it.

Interesting Things You Can Do with Hello 8.1.2

Notice how the font scales in Hello World and how it uses shades of gray.

Highlights of clsHelloWorld 8.1.3

The method table for **clsHelloWorld** handles two significant messages:

```
msgAppOpen,     "HelloOpen",    objCallAncestorAfter,
msgAppClose,    "HelloClose",   objCallAncestorBefore
```

The handler for **msgAppOpen** creates an instance of **clsHelloWin** and inserts it as the frame's client window.

The handler for **msgAppClose** destroys the client window.

When **processCount** is 0, **main** calls **ClsHelloInit**.

Highlights of clsHelloWin 8.1.4

The **DLLMain** for **clsHelloWin** is the only thing defined in DLLINIT.C. The **DLLMain** calls **ClsHelloWinInit**, the initialization routine for **clsHelloWin**.

```
STATUS EXPORTED DLLMain (void)
    StsRet(ClsHelloWinInit(), s);
```

The method table for **clsHelloWin** (in HELWTBL.TBL) handles three significant messages:

```
msgInit,            "HelloWinInit",     objCallAncestorBefore,
msgFree,            "HelloWinFree",     objCallAncestorAfter,
msgWinRepaint       "HelloWinRepaint",  0
```

clsHelloWin is the first sample application that defines its own instance data (in HELLOWIN.C).

```
typedef struct INSTANCE_DATA {
    SYSDC           dc;
} INSTANCE_DATA;
```

clsHelloWin responds to **msgInit** by zeroing the instance data, creating a drawing context, initializing the drawing context, and storing the drawing context in the hello window object's instance data.

The class responds to **msgDestroy** by destroying the drawing context.

clsHelloWin responds to **msgWinRepaint** by calculating the text width and scaling the window so that it fits the text

Graphics Overview 8.2

To draw in a window you need to create a **drawing context** object (often abbreviated DC). You send messages to the drawing context, *not* your window, to draw. The drawing context's class knows how to perform these graphics operations. There could be different kinds of drawing contexts to choose from on PenPoint; for example, there might be one available from a third-party company which understands 3-D graphics, or you could create your own.

System Drawing Context 8.2.1

The standard **system drawing context** supports the ImagePoint imaging model. You can draw lines, polygons, ellipses, Bezier curves, and text by sending messages to an instance of the system drawing context.

Each of these graphic operations is affected by the current graphics state of your DC. The system drawing context strokes lines and the borders of figures with the current line pattern, width, end style, and corner style, all of which you can set and get using SysDC messages. Similarly, it fills figures with the current fill pattern. Most drawing operations involve both stroking and filling a figure, but by adjusting line width and setting patterns to transparent, you can only fill or only stroke a figure.

The pixels of figures on the screen are transformed according to the current **rasterOp**. This is a mathematical description of how the destination pixels on the screen are affected by the pixels in the source figure. To paint over pixels on the screen, you use the default rasterOp, **sysDcRopCopy**; another common rasterOp is **sysDcRopXOR**, which inverts pixels on the screen.

If you want to draw temporarily on the screen, it's better to set the sysDcDrawDynamic mode instead of directly changing the rasterOp.

At this writing there are no PenPoint computers that support color, however, the system drawing context supports a full color model. You can set the background and foreground colors (on a black and white display, the resulting colors on a will always be black, white, or a shade of gray). The line and fill patterns are mixtures of the current foreground and background color, or **sysDcInkTransparent.**

Because the system drawing context is a normal Class Manager object, you create a new instance of it in the usual way, by sending **msgNew** to **clsSysDrwCtx**. Your drawing messages end up on some window on the screen, so at some point you must bind your DC to the desired window using **msgDcSetWindow**.

Coordinates in Drawing Context 8.2.2

Another vital property of the system drawing context is its arbitrary coordinate system. You can choose whether one unit in your drawing (as in "draw a line one unit long" is one point, 0.01 mm, 0.001 inch, 1/20 of a point, one pixel on the final device. You can then scale units in both the X and Y direction; one useful scaling is to scale them relative to the height and width of your window. You can even rotate your coordinate system. What this gives you is the precision of knowing that your drawing will be an exact size. It also gives you the freedom to use any coordinate system and scale that suits your drawing. The default coordinates are one unit is one point (approximately 1/72 of an inch), and the origin is in the lower left corner of your window.

Hello World (custom window) uses the default units, but scales its coordinate system so that its text output remains at a regular aspect ratio.

When to Paint 8.2.3

Windows need to repaint themselves when they first appear on the screen, when they are is resized, and when they are exposed after other windows have covered them. Windows receive **msgWinRepaint** when the window system determines that they need to repaint, and windows must respond to this.

clsHelloWin *only* paints in response to **msgWinRepaint**. The way most windows work is that they repaint dirty areas rather than paint new ones. When a window wants to draw something new, it can dirty itself and hence will receive **msgWinRepaint**. **clsHelloWin** has no need to dirty itself since it doesn't change what it paints.

When to Create Things? 8.3

The need to manage a separate object (a drawing context) introduces two crucial questions you need to consider when designing an application:

- When do I create and destroy an object (or resource)?
- When do I file it, if at all?

An application can create objects at many stages in its life. It can create objects at installation, at initialization (or at restore time), when opening, or when painting its windows. If your application can create an object just before it needs it, the less memory it consumes in earlier stages. But it takes time to create objects, so you must trade off memory savings with speed.

To decide when to create objects, you need to work backwards from when they are needed. In this case, Hello World only needs a drawing context in its window's repaint routine. Creating a DC every time you need to repaint is OK, but it is a fairly expensive operation. Besides, realistic applications often use a DC in input processing as well, to figure out where the user's pen is in convenient coordinates. However, we do know that a DC will never be needed when the view doesn't exist.

clsHello could create the DC and pass it to **clsHelloWin**, but it's usually much more straightforward for the object that needs another object to create that object.

Hello World creates its window when it receives **msgAppOpen** and destroys its window when it receives **msgAppClose**. These are reasonable times for the window to create its DC, so **clsHelloWin** creates a DC when it receives **msgInit** and destroys the DC when it receives **msgFree**.

Instance Data

8.3.1

In our example, **clsHelloWin** creates its DC in advance. This means that it has to store the UID of the DC somewhere so that it can use it during **msgWinRepaint**. In typical DOS C programs, you can declare static variables to hold information. It is possible to do this in PenPoint, but in general you should not do it in object-oriented code.

Instead, you should store the information inside each object, in its **instance data**. Up until now our classes have not had to remember state, so they haven't needed their own instance data. (Even if the class you create does not define instance data for its objects, its ancestors define some instance data, such as the document name and the label of the toolkit field.)

Specifying instance data is easy. You just tell the Class Manager how big it is (in the **class.size** field) when you create your class. You would typically define a `typedef` for the structure of your class's instance data, then give the size of this as the class.size. In the case of **clsHelloWin**, we define a structure called INSTANCE_DATA:

```
typedef struct INSTANCE_DATA {
    SYSDC           dc;
} INSTANCE_DATA, *P_INSTANCE_DATA;
```

and then in **ClsHelloWinInit**:

```
STATUS ClsHelloWinInit (void)
{
    CLASS_NEW       new;
    STATUS          s;

    // Create the class.
    ObjCallWarn(msgNewDefaults, clsClass, &new);
    new.object.uid          = clsHelloWin;
    new.cls.pMsg            = clsHelloWinTable;
    new.cls.ancestor        = clsWin;
    new.cls.size            = SizeOf(INSTANCE_DATA);
    new.cls.newArgsSize     = SizeOf(HELLO_WIN_NEW);
    ObjCallRet(msgNew, clsClass, &new, s);
```

Is It msgNew or msgInit?

8.3.2

As we discussed, **clsHelloWin** creates its DC when it is created. It does this by responding to **msgInit**.

Note that **clsHelloWin** responds to **msgInit**, not **msgNew**. When you create an object, you send its class **msgNew**. No classes intercept this message, so it goes up

the ancestor chain to **clsClass**, which creates the new object. The Class Manager then sends **msgInit** to the newly-created object, so that it can initialize itself.

Window Initialization 8.3.3

Here's the **HelloWinInit** code which creates the Hello Window in response to **msgInit**:

```
MsgHandler(HelloWinInit)
{
    SYSDC_NEW           dn;
    INSTANCE_DATA       data;
    SYSDC_FONT_SPEC     fs;
    SCALE               fontScale;
    STATUS              s;
```

clsHelloWinInit declares an instance data structure. It does this because the pointer to instance data passed to message handlers by the Class Manager (**pData**, unused in this routine) is read-only.

It then initializes the instance data to zero. It's important for instance data to be in a well-known state. This isn't necessary in the case of **clsHelloWin**, since the only instance data is the DC UID that it will fill in, but it is good programming practice.

```
    // Null the instance data.
    memset(&data, 0, sizeof(INSTANCE_DATA));
```

clsHelloWin then creates a DC:

```
    // Create a dc.
    ObjCallWarn(msgNewDefaults, clsSysDrwCtx, &dn);
    ObjCallRet(msgNew, clsSysDrwCtx, &dn, s);
```

When **msgNew** returns, it passes back the UID of the new system drawing context. This is what **clsHelloWin** wants for its instance data:

```
    data.dc = dn.object.uid;
```

clsHelloWin sets the desired DC state (including the line thickness) and binds it to **self** (the instance that has just been created when **HelloWinInit** is called):

```
    // Rounded lines, thickness of zero.
    ObjectCall(msgDcSetLineThickness, data.dc, (P_ARGS)0);
    if (DbgFlagGet('F', 0x40L)) {
        Debugf("Use a non-zero line thickness.");
        ObjectCall(msgDcSetLineThickness, data.dc, (P_ARGS)2);
    }

    // Open a font. Use the "user input" font (whatever the user has
    // chosen for this in System Preferences.
    fs.id               = 0;
    fs.attr.group       = sysDcGroupUserInput;
    fs.attr.weight      = sysDcWeightNormal;
    fs.attr.aspect      = sysDcAspectNormal;
    fs.attr.italic      = 0;
    fs.attr.monospaced  = 0;
    fs.attr.encoding    = sysDcEncodeGoSystem;
    ObjCallJmp(msgDcOpenFont, data.dc, &fs, s, Error);
```

```
//
// Scale the font. The entire DC will be scaled in the repaint
// to pleasingly fill the window.
fontScale.x = fontScale.y = FxMakeFixed(initFontScale,0);
ObjectCall(msgDcScaleFont, data.dc, &fontScale);

// Bind the window to the dc.
ObjectCall(msgDcSetWindow, data.dc, (P_ARGS)self);
```

At this point, **clsHelloWin** has set up its instance data in a local structure. It calls **ObjectWrite** to get the Class Manager to update the instance data stored in the Hello Window instance:

```
// Update the instance data.
ObjectWrite(self, ctx, &data);
return stsOK;
```

Using Instance Data 8.4

Accessing instance data is easy. The Class Manager passes a read-only pointer to instance data into the class's message handlers.

The Class Manager has no idea what the instance data is, so it just declares the pointer as a mystery type (P_DATA, which is defined as P_UNKNOWN). The **MsgHandler** macro names the pointer **pData**.

clsHelloWin needs to access its instance data during **msgWinRepaint** handling so it can use the DC. It knows that the instance data pointed to by **pData** is type INSTANCE_DATA, so it uses the **MsgHandlerWithTypes** macro, which allows it to provide the types (or casts) for the argument and instance data pointers:

```
MsgHandlerWithTypes(HelloWinRepaint, P_ARGS, P_INSTANCE_DATA)
```

You can pass the pData pointer around freely within your code, but whenever you want to change instance data, you must dereference it into a local (writable) variable, modify the local variable, and then call **ObjectWrite**. **clsHelloWin** creates its DC when it is created, and never changes it, so it doesn't have to worry about de-referencing its instance data into local storage. But **clsCntr**, described in Chapter 9, does have to do this.

No Filing Yet 8.4.1

On a page turn, the process and all objects associated with a Hello World (custom window) document are destroyed. Normally this means that objects have to file their state. However, since **clsHelloWin** destroys its DC when it is destroyed and never changes its DC's state, it doesn't have to file its DC.

The application does not file its view—it creates it at **msgAppOpen** to draw, then destroys it at **msgAppClose**, and there's no useful state to remember from the DC. You could imagine an application which *would* want to remember some of the state of its DC. For example, if the user could choose the font in Hello World (custom window), then the program would need to remember what the font was

so that when the user turns back to the application's page the application continues to use the same font.

Drawing in a Window 8.5

Empty Application prints out messages, but it doesn't draw them in its window. Instead it uses the error output routine **Debugf** to generate output. Hello World (custom window) actually draws something in its window. Windows are separate objects from applications, and the window gets told to repaint, not the application. Hence you need to create a window object. The window object will receive **msgWinRepaint** messages whenever it needs to paint its window, either because the application has just appeared on-screen, or because another window was obscuring part of this window.

clsWin responds to **msgWinRepaint** by filling **self** with the background color and outlining the edge of the window. You could put an instance of **clsWin** inside your frame, but we want something more interesting to appear in the window. So **clsHelloWin** intercepts **msgWinRepaint** and draws its own thing. It draws the strings "Hello" and "World" and then draws an exclamation point using graphics commands. The most complex thing about its repaint routine is its scaling. It measures how long the strings "Hello" and "World" will be, then uses this information to scale its coordinate system so that the words and drawing fit in the window nicely.

Possible Enhancements 8.6

Try drawing some other shapes using other **msgDcDraw...** messages. Nest a **clsHelloWin** window in the custom layout window from HELLOTK2.C.

Debugging Hello World (Custom Window) 8.7

If you want to modify Hello World (custom window), you might need to use DB extensively as you make changes. This section explains techniques developers commonly use to speed up debugging with DB.

To save typing commands over and over to DB, you can store them in files and read them into DB using its **<** command, for example:

```
< \\boot\proj\setbreak.txt
```

When it starts, DB looks for a start-up file called DBCUSTOM.DB. It tries to find this in \\BOOT\PENPOINT\APP\DB, but you can specify the path to another file by specifying the path in a DBCustom line in \PENPOINT\BOOT\ENVIRON.INI. You can use DBCUSTOM.DB to set up the **ctx** and **srcdir** for your application's executables and DLLs, and set breakpoints. Here's how a DBCUSTOM.DB for Hello World (custom window) might look:

```
sym "go-hello_exe-v1(0)" \\boot\penpoint\sdk\sample\hello\hello.exe
srcdir "go-hello_exe-v1(0)" \\boot\penpoint\sdk\sample\hello
sym "go-hello_dll-v1(0)" \\boot\penpoint\sdk\sample\hello\hello.dll
srcdir "go-hello_dll-v1(0)" \\boot\penpoint\sdk\sample\hello
bp HelloWinRepaint
g
```

Whenever you start a new instance of Hello World (custom window) (either by choosing from the Accessory palette or by turning a page), DB will halt. At that point you can type **t** to step a line, **g** to continue, and so on.

Chapter 9 / Saving and Restoring Data (Counter App)

The sample programs we have considered so far do not have any information to save. They always do the same thing in response to the same messages. However, real applications must be able to save and restore data.

PenPoint applications also have information about what is on screen, where was the user interacting with the application last, what options were set, what controls were active at the time, and so on. This information together with the application's data is called the application's **state.**

Because PenPoint is an object oriented system, there is no real distinction between data and state information. An application is built from a series of objects. A scribble object might contain the scribbles that the user just drew, while a scroll window object contains the current scrolling position of the window. The former contains "user data" and the latter contains state information, but to PenPoint they are simply objects.

This chapter discusses how applications save and restore their data.

The last part of this chapter describes how to create a menu using **clsTkTable.**

Saving State

9.1

Remember that as the user turns from page to page in the Notebook, the Application Framework is starting up and shutting down instances of **clsApp.** When you turn the page from Empty Application or Hello World, the Application Framework destroys the **clsEmptyApp** or **clsHelloWorld** application object. When you turn back to that page, the Application Framework creates a new application object.

This is fine, because these applications don't need to remember anything. They start from scratch each time they appear. However, if applications do change state, they must preserve this state, so that the user is not aware that an application instance "behind" what is seen onscreen is coming and going.

Note The basic rule for filing state is: If I don't file this state, will users notice that the application is different when they turn back to its page?

Counter Application

9.2

The Counter Application saves data. Each time the application appears on-screen, it increments a counter and displays the counter's value. It also lets the user choose the format in which to display the counter (decimal, octal, or hexadecimal).

Based on the state filing rule, the application has two pieces of state that it should file:

- The value of the counter
- The format in which it was told to display the counter.

clsCntrApp remembers the format in which it displays the counter value. **clsCntrApp** could also remember the value of the counter, but one of the benefits of an object-oriented system is that you can break up your application code into objects which model the natural structure of the system.

It's natural to see the application as displaying the value of a separate object, so that's the way we implement it: **clsCntrApp** creates and interacts with a separate **clsCntr** object. Because the format could be applied to all counter objects in the application, **clsCntrApp** also remembers the format.

Note the difference between CounterApp and the two Hello World sample programs. The Hello World applications *had* to create other objects to get the behavior they needed. An application object is not a window, so they had to create window objects. In the case of CounterApp's counter object, we're not forced to use a separate object—we could have **clsCntrApp** remember the state of the counter, but for design reasons we choose to implement the counter as a separate object.

PenPoint has several classes that store a numeric value:

clsIntegerField A handwriting field that accepts numeric input

clsPageNum The page number in floating frames

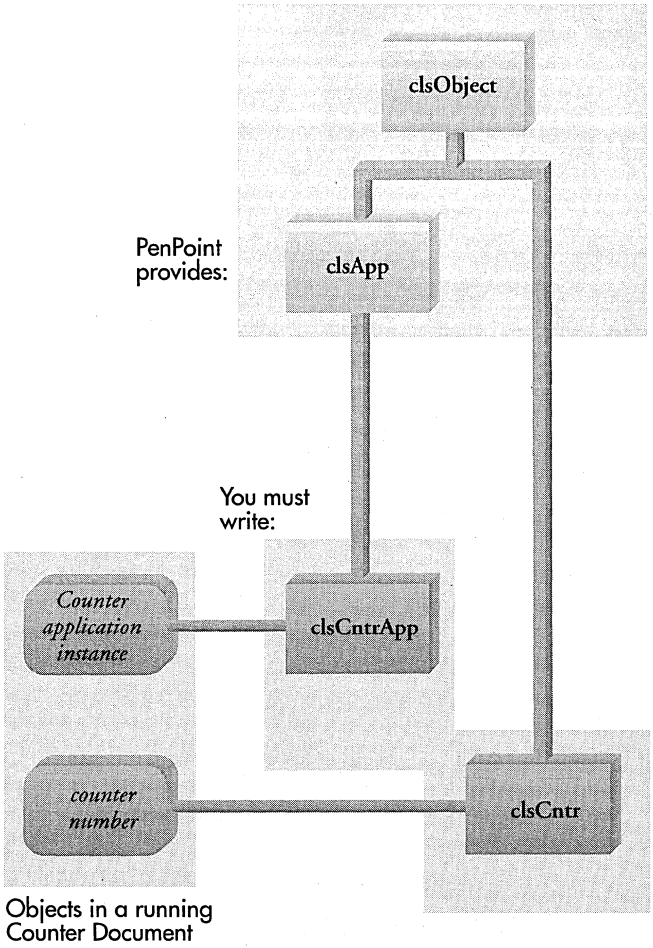
clsCounter The page number with up and down arrows in the Notebook

These are all window classes that display the numeric value. **clsCntrApp** creates a label to display the value of the counter, much like Hello World (toolkit). Hence none of these are quite right for CounterApp, so we create a separate counter class.

Because the UI Toolkit uses the symbol clsCounter already, CounterApp uses the symbol clsCntr for its counter class.

This figure shows the classes defined by CounterApp and their ancestors.

Figure 9-1
CounterApp Objects



Compiling and Installing the Application 9.2.1

To compile CounterApp, change to the \PENPOINT\SDK\SAMPLE\CNTRAPP directory and type **wmake**. This creates a \PENPOINT\APP\CNTRAPP directory and compiles CNTRAPP.EXE in that directory.

Install CounterApp either by adding \\BOOT\PENPOINT\APP\Counter Application to \PENPOINT\BOOT\APP.INI before starting PenPoint or by installing the application using the Installer.

Counter Application Highlights 9.2.2

The method table for **clsCntrApp** handles a number of interesting messages:

```
msgInit,                 "CntrAppInit",          objCallAncestorBefore,
msgSave,                 "CntrAppSave",          objCallAncestorBefore,
msgRestore,              "CntrAppRestore",       objCallAncestorBefore,
msgFree,                 "CntrAppFree",          objCallAncestorAfter,
msgAppInit,              "CntrAppAppInit",       objCallAncestorBefore,
msgAppOpen,              "CntrAppOpen",          objCallAncestorAfter,
msgAppClose,             "CntrAppClose",         objCallAncestorBefore,
msgCntrAppChangeFormat,  "CntrAppChangeFormat",  0,
```

clsCntrApp creates an instance of **clsCntr** at **msgAppInit** time.

method table
#defines, typedefs
message handlers
class initialization
main entry point

clsCntrApp responds to **msgAppOpen** by incrementing the counter, creating a label containing the counter value, making the label the client window, and creating the menu bar.

clsCntrApp responds to **msgAppClose** by destroying the client window.

The class responds to **msgCntrAppChangeFormat**, which is sent by its menu buttons, by changing its stored data format.

When **processCount** is 0, **main** calls **ClsCntrAppInit**.

Counter Class Highlights 9.2.3

The method table for **clsCntr** is also defined in METHODS.TBL and handles these messages:

method table
#defines, typedefs
message handlers
class initialization
main entry point

```
msgNewDefaults,          "CntrNewDefaults",     objCallAncestorBefore,
msgInit,                 "CntrInit",            objCallAncestorBefore,
msgSave,                 "CntrSave",            objCallAncestorBefore,
msgRestore,              "CntrRestore",         objCallAncestorBefore,
msgFree,                 "CntrFree",            objCallAncestorAfter,
msgCntrGetValue,         "CntrGetValue",        0,
msgCntrIncr,             "CntrIncr",            0,
```

Instance Data 9.2.4

The instance data for a **clsCntr** object contains the value of the counter:

```
typedef struct CNTR_INST {

    S32   currentValue;

}  CNTR_INST,
   *P_CNTR_INST;
```

Make sure you notice the difference between CNTR_INST, the counter's instance data, and CNTR_INFO, the structure used for the arguments passed with msgCntrGetValue. In this example, the two structures contain the same data; in a more complex example, the instance data would contain all the stateful information required by an instance of the object, while the message argument structure would only contain the data needed by a particular message.

Because the purpose of **clsCntr** is to maintain a value for its client, **clsCntr** must provide a means for its client to access the value. One common approach lets the client perform these tasks:

Important point.

- Specify an initial value in **msgNew**
- Get the value with a special message
- Set the value with a special message.

clsCntr does all of these except set the value. The _NEW_ONLY information for **clsCntr** contains an initial value. Here is the CNTR_NEW_ONLY structure from CNTR.H,

```
typedef struct CNTR_NEW_ONLY {
S32 initialValue;
} CNTR_NEW_ONLY, *P_CNTR_NEW_ONLY;
```

In case its client doesn't specify an initial value when the client sends **msgNew**, **clsCntr** initializes the **msgNew** argument to a reasonable value (zero) in **msgNewDefaults**:

```
MsgHandlerArgType(CntrNewDefaults, P_CNTR_NEW)
{
    Debugf("Cntr:CntrNewDefaults");
// Set default value in new struct.
pArgs->cntr.initialValue = 0;
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* CntrNewDefaults */
```

In response to **msgInit**, **clsCounter** initializes the instance data to the starting value specified in the **msgNew** arguments:

```
MsgHandlerArgType(CntrInit, P_CNTR_NEW)
{
    CNTR_INST inst;
    Debugf("Cntr:CntrInit");
    // Set starting value.
    inst.currentValue = pArgs->cntr.initialValue;
    // Update instance data.
ObjectWrite(self, ctx, &inst);
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* CntrInit */
```

Getting and Setting Values 9.2.5

clsCntr defines messages to get and set the counter value, **msgCntrGetValue** and **msgCntrInc**. Note how we intentionally limit the API to suit the design of the object: the client can't directly set the counter value, it can only increment it. This makes the counter less general.

The (dubious) advantage of the approach used is that if the design of **clsCntr** changes so that it has more information, CNTR_INFO could change to include more information, and clients of it would only need to recompile.

Getting the Value 9.2.5.1

The handler for **msgCntrGetValue** is straightforward. Note that the client must pass it a pointer to the structure in which **clsCntr** passes back the value.

```
MsgHandlerWithTypes(CntrGetValue, P_CNTR_INFO, P_CNTR_INST)
{
    Debugf("Cntr:CntrGetValue");
    pArgs->value = pData->currentValue;
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* CntrGetValue */
```

In this case, passing a CNTR_INFO structure as the message arguments is not necessary. **msgCntrGetValue** could take a pointer to an S32, instead of a pointer to a structure that contains an S32. However, as soon as you need more than 32 bits to communicate the message arguments, you must define a structure and pass a pointer to the structure.

Incrementing the Value

9.2.5.2

msgCntrIncr increments the value. It doesn't take any arguments.

```
MsgHandler(CntrIncr)
{
    CNTR_INST inst;
    Debugf("Cntr:CntrIncr");
    inst = IDataDeref(pData, CNTR_INST);
    inst.currentValue++;
    ObjectWrite(self, ctx, &inst);
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* CntrIncr */
```

There are a couple of things to note here. The instance data is stored in memory that only the Class Manager can write. When the Class Manager calls the message handler, it passes a pointer to this protected instance data (**pData**). If the code had tried to update **pData->currentValue** directly, PenPoint would have generated a general protection fault immediately. So, the code creates a local structure for the instance data and uses the **IDataDeref** macro to dereference the instance data pointer into local, writable, memory (**inst**).

Note A frequent source of programming errors is trying to modify read-only instance data.

There is nothing mysterious about the **IDataDeref** macro. All it does is provide a short hand way of casting the instance data indicated by **pData**. **IDataDeref** is defined in CLSMGR.H as:

```
#define IDataDeref(pData, type) (*(type*)pData)
```

We could just as easily have written:

```
inst = (CNTR_INST) pData;
```

After incrementing the value in local memory, **clsCntr** calls **ObjectWrite**, which directs the Class Manager to update the instance data stored in the object.

Note Another frequent source of programming errors is failing to update instance data with ObjectWrite after changing instance data.

```
ObjectWrite(self, ctx, &inst);
```

Object Filing

9.3

The way objects preserve state is by filing it at the appropriate time. The Application Framework sends the application instance **msgSave** when the document should save its state and **msgRestore** when the document should recreate itself. The order in which applications receive these and other messages from the Application Framework is explained in *Part 2: Application Framework* of the *PenPoint Architectural Reference* manual.

Objects can also preserve state by refusing to be terminated, although this usually consumes memory.

The message arguments to **msgSave** and **msgRestore** include a handle on a **resource file**. Objects respond by writing out their state to this file and reading it back in. The objects do not care where the resource file is, nor do they care who created it.

The Application Framework creates and manages a resource file for each document. The file handle passed by **msgSave** and **msgRestore** is for this resource file. If you start up the disk viewer with the B debug flag set to 800 hexadecimal, and expand \\RAM\PENPOINT\SYS\Notebook\CONTENTS\Notebook Applications, you should be able to see these files; look for a file called DOC.RES in each document directory.

At the level of **msgSave** and **msgRestore**, classes can just write bytes to a file (the resource file) to save state.

When it receives **msgSave**, **clsCntrApp** could get the value of the counter object (by sending it **msgCntrGetValue**) and just write the number to the file. However, this would introduce dependencies between the two objects, which in object-oriented programming is a bad thing. So, instead **clsCntrApp** tells the counter object to file itself. We'll cover exactly how this happens later, but for now just accept that the **clsCntr** instance receives **msgSave**.

Handling msgSave 9.3.1

The message argument to **msgSave** is a pointer to an OBJ_SAVE structure:

```
MsgHandlerArgType(CntrSave, P_OBJ_SAVE)
```

If you look in \PENPOINT\SDK\INC\CLSMGR.H, you will notice that one of the fields in the OBJ_SAVE structure is the handle of the file to save to. So all **clsCntr** has to do is write that part of its instance data that it needs to save to the file: basically, all of its instance data.

To write to a file, you send **msgStreamWrite** to the file handle. The message takes a pointer to a STREAM_READ_WRITE structure, in which you specify what to file and how many bytes to write.

```
MsgHandlerArgType(CntrSave, P_OBJ_SAVE)
{
    STREAM_READ_WRITE fsWrite;
    STATUS            s;
    Debugf("Cntr:CntrSave");
    //
    // Write instance to the file.
    //
    fsWrite.numBytes= SizeOf(CNTR_INST);
    fsWrite.pBuf= pData;
    ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s);
    return stsOK;
    MsgHandlerParametersNoWarning;

} /* CntrSave */
```

msgStreamWrite passes back information about how many bytes it actually wrote. A real application would check this information to make sure that it successfully filed all its state.

Handling msgRestore 9.3.2

msgRestore is similar to **msgSave**. The Class Manager handles **msgRestore** by creating a new object, so the ancestor must be called first. The message argument to **msgRestore** is a pointer to an OBJ_RESTORE structure:

```
MsgHandlerArgType(CntrRestore, P_OBJ_RESTORE)
```

Again, one of the fields in this structure is the UID of the file handle to restore from. **clsCntr** just has to restore self's instance data from the filed data. This is similar to initializing instance data in **msgInit** handling, except that the information has to be read from a file instead of from **msgNew** arguments. You declare a local instance data structure:

```
MsgHandlerArgType(CntrRestore, P_OBJ_RESTORE)
{
    CNTR_INST     inst;
    STREAM_READ_WRITE fsRead;
    STATUS            s;
```

To read from a file, you send **msgStreamRead** to the file handle, which takes a pointer to the same STREAM_READ_WRITE structure as **msgStreamWrite**. In the structure you specify how many bytes to read and give a pointer to your buffer that will receive the data:

```
    Debugf("Cntr:CntrRestore");
    //
// Read instance data from the file.
    //
fsRead.numBytes= SizeOf(CNTR_INST);
fsRead.pBuf= &inst;
ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s);
```

You call **ObjectWrite** to update the object's instance data.

```
    //
    // Update instance data.
    //
ObjectWrite(self, ctx, &inst);
    return stsOK;
    MsgHandlerParametersNoWarning;

} /* CntrRestore */
```

CounterApp's Instance Data 9.4

clsCntrApp's instance data contains:

- The display format to use for the counter value
- The UID of the counter object
- A memory-mapped file handle (explained below).

When the user turns away from a CounterApp document's page, the Application Framework destroys the counter object (and the application instance). When the user turns back to the CounterApp document, the counter object is restored with a different UID. Hence **clsCntrApp** should *not* file the UID of the counter object, because it will be invalid upon restore. **clsCntrApp** only needs to file the display format and to tell the counter object to save its data.

Tip Saving the UIDs of an object is usually incorrect. Either the object has a well-known UID (there's no reason to file it), or the UID is dynamic (it will be different when restored).

Memory-Mapped File

9.4.1

CounterApp could just write its data to the resource file created by the Application Framework, just as the counter object did. However, a disadvantage of filing data is that there are two copies of the information when a document is open: the instance data in the object maintained by the Class Manager and the filed data in the document resource file maintained by the file system.

CounterApp and the counter object use different filing methods. This is useful when you need to differentiate between instance data and other forms of data.

One way to avoid this duplication of data is to use a **memory-mapped** file. Instead of reading and writing to a file, you can simply map the file into your address space; reading and writing to the file take place transparently as you access that memory.

clsCntrApp stores its data (the current representation) in a memory-mapped file.

Opening and Closing The File

9.4.2

Because you need to open the file both when creating the document for the first time, and when restoring the document after it has been filed, you need to open the file in two different places (**msgAppInit** and **msgRestore**), but you only need close it in one place (**msgFree**).

Why close the file in response to **msgFree**? Why not **msgSave**? Remember that when an application is created, it is sent **msgAppInit** (in response to which it creates and initializes objects) and then is immediately sent **msgSave** (which allows it to save its newly initialized objects before doing anything else). **msgSave** is also sent when the user checkpoints a document. In other words, receiving **msgSave** doesn't necessarily mean that we're about to destroy the application object.

Opening For the First Time

9.4.2.1

When CounterApp receives **msgAppInit**, it creates the counter object:

```
MsgHandler(CntrAppAppInit)
{
    CNTR_NEW                cn;
    FS_NEW                  fsn;
    STREAM_READ_WRITE       fsWrite;
    CNTRAPP_DISPLAY_FORMAT  format;
    CNTRAPP_INST           inst;
    STATUS                  s;

    Debugf("CntrApp:CntrAppAppInit");

    inst = IDataDeref(pData, CNTRAPP_INST);
```

```
    //
    // Create the counter object.
    //
    ObjCallRet(msgNewDefaults, clsCntr, &cn, s);
    cn.cntr.initialValue = 42;
    ObjCallRet(msgNew, clsCntr, &cn, s);

    inst.counter = cn.object.uid;
```

CntrAppAppInit opens the file (called FORMATFILE), initializes the format value, and writes the initial data to the file:

```
    //
    // Create a file, fill it with a default value
    //
    ObjCallRet(msgNewDefaults, clsFileHandle, &fsn, s);
    fsn.fs.locator.pPath = "formatfile";
    fsn.fs.locator.uid = theWorkingDir;
    ObjCallRet(msgNew, clsFileHandle, &fsn, s);
    format = dec;
    fsWrite.numBytes = SizeOf(CNTRAPP_DISPLAY_FORMAT);
    fsWrite.pBuf = &format;
    ObjCallRet(msgStreamWrite, fsn.object.uid, &fsWrite, s);
```

Mapping the file to memory is quite straightfoward. CounterApp sends **msgFSMemoryMap** to the file handle, and passes the pointer to the address pointer (PP_MEM). Finally updates its instance data and returns:

```
    // Map the file to memory
    //
    ObjCallRet(msgFSMemoryMap, fsn.object.uid, &inst.pFormat, s);
    // Update instance data.
    ObjectWrite(self, ctx, &inst);
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* CntrAppAppInit */
```

Opening To Restore

9.4.2.2

CntrAppRestore is essentially similar to CntrAppAppInit, only after creating a handle on FORMATFILE, CntrAppRestore maps the file to memory, which immediately makes the format data available in inst.pFormat.

```
MsgHandlerWithTypes(CntrAppRestore, P_OBJ_RESTORE, P_CNTRAPP_INST)
{
    FS_NEW        fsn;
    CNTRAPP_INST inst;
    STATUS        s;
    Debugf("CntrApp:CntrAppRestore");
    // Get handle for format file, save the handle
    ObjCallRet(msgNewDefaults, clsFileHandle, &fsn, s);
    fsn.fs.locator.pPath = "formatfile";
    fsn.fs.locator.uid = theWorkingDir;
    ObjCallRet(msgNew, clsFileHandle, &fsn, s);
    inst.fileHandle = fsn.object.uid;
    // Map the file to memory
    ObjCallRet(msgFSMemoryMap, fsn.object.uid, &inst.pFormat, s );
```

Closing On msgFree 9.4.2.3

When the application receives **msgFree**, it destroys the counter object, sends **msgFSMemoryMapFree** to unmap the file, and then sends **msgDestroy** to the file handle to close the file.

```
MsgHandlerWithTypes(CntrAppFree, P_ARGS, P_CNTRAPP_INST)
{
    STATUS s;
    Debugf("CntrApp:CntrAppFree");
    ObjCallRet(msgDestroy, pData->counter, Nil(P_ARGS), s);
    // Unmap the file
    ObjCallRet(msgFSMemoryMapFree, pData->fileHandle, Nil(P_ARGS), s);
    // Free the file handle
    ObjCallRet(msgDestroy, pData->fileHandle, Nil(P_ARGS), s );
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* CntrAppFree */
```

Filing the Counter Object 9.4.3

The only thing that is left to do is to tell the counter object when to save and restore its data. For this, you send the resource messages **msgResPutObject** and **msgResGetObject** to the resource file handle created by the Application Framework. These messages are defined in RESFILE.H. You do not send **msgSave** and **msgRestore** directly to the counter object.

The resource file handle is an instance of **clsResFile**. When you send a message to the resource file handle, you tell it which object you want to put or get. In the case of **msgResPutObject**, **clsResFile** writes information about the object to the resource file, then sends **msgSave** to the object. In the case of **msgResGetObject**, **clsResFile** reads information about the object from the file, creates the object, which is essentially empty until **clsResFile** sends **msgRestore** to the object. This is how objects receive **msgSave** and **msgRestore**.

Saving the Counter Object 9.4.3.1

When CounterApp receives **msgSave**, it sends **msgResPutObject** to the file handle passed in with the **msgSave** arguments.

```
MsgHandlerWithTypes(CntrAppSave, P_OBJ_SAVE, P_CNTRAPP_INST)
{
    STATUS s;

    Debugf("CntrApp:CntrAppSave");

    // Save the counter object.
    ObjCallRet(msgResPutObject, pArgs->file, pData->counter, s);

    return stsOK;
    MsgHandlerParametersNoWarning;

} /* CntrAppSave */
```

CounterApp doesn't have to save its own data, because the file FORMATFILE is memory mapped to its data.

Restoring the Counter Object 9.4.3.2

When CounterApp receives **msgRestore**, it sends **msgResPutObject** to the file handle passed in with the **msgRestore** arguments.

```
MsgHandlerWithTypes(CntrAppRestore, P_OBJ_RESTORE, P_CNTRAPP_INST)
{
    FS_NEW        fsn;
    CNTRAPP_INST inst;
    STATUS        s;

(Open the memory mapped file.)

    // Restore the counter object.
    ObjCallJmp(msgResGetObject, pArgs->file, &inst.counter, s, Error);

    // Update instance data.
    ObjectWrite(self, ctx, &inst);

    return stsOK;
    MsgHandlerParametersNoWarning;

Error:

    return s;

} /* CntrAppRestore */
```

Menu Support 9.5

clsCntrApp creates a menu by specifying the contents of the menu statically in a toolkit table. **clsTkTable** is the ancestor of several UI components which display a set of windows, including choices, option tables, and menus. Instead of creating each of the items in a toolkit table by sending **msgNew** over and over to different classes, you can specify in a set of toolkit table entries what should be in the table. When you send **msgNew** to **clsTkTable** (or one of its descendants) it creates its child items based on the information you gave it.

When it receives **msgAppOpen**, **clsCntrApp** appends its menu to the SAMs (standard application menus) by passing its menu as an argument to **msgAppCreateMenuBar**.

```
MsgHandlerWithTypes(CntrAppOpen, P_ARGS, P_CNTRAPP_INST)
{
    APP_METRICS am;
    MENU_NEW    mn;
    LABEL_NEW   ln;
    STATUS      s;
    char        buf[30];

    ...

    // Get app metrics.
    ObjCallJmp(msgAppGetMetrics, self, &am, s, Error);

    // Set the label as the clientWin.
    ObjCallJmp(msgFrameSetClientWin, am.mainWin, ln.object.uid, s, Error);

    // Create and add menu bar.
```

```
ObjCallJmp(msgNewDefaults, clsMenu, &mn, s, Error);
mn.tkTable.client = self;
mn.tkTable.pEntries = CntrAppMenuBar;
ObjCallJmp(msgNew, clsMenu, &mn, s, Error);

ObjCallJmp(msgAppCreateMenuBar, self, &mn.object.uid, s, Error);
ObjCallJmp(msgFrameSetMenuBar, am.mainWin, mn.object.uid, s, Error);
```

Buttons 9.5.1

The items in the menu are a set of buttons. When you create a button in toolkit table entry, you specify:

- The button's string
- The notification message the button should send when the user activates it
- A value for the button.

There are other fields in a TK_TABLE_ENTRY, but you can rely on their defaults of 0.

The menu bar used in CounterApp is described by the TK_TABLE_ENTRY structure named **CntrAppMenuBar** in CNTRAPP.C.

```
typedef enum CNTRAPP_DISPLAY_FORMAT {

    dec, oct, hex

}  CNTRAPP_DISPLAY_FORMAT,
   *P_CNTRAPP_DISPLAY_FORMAT;
typedef struct CNTRAPP_INST {

    P_CNTRAPP_DISPLAY_FORMAT  pFormat;
    OBJECT                    fileHandle;
    OBJECT                    counter;

}  CNTRAPP_INST,
   *P_CNTRAPP_INST;

static TK_TABLE_ENTRY CntrAppMenuBar[] = {
    {"Representation", 0, 0, 0, tkMenuPullDown, clsMenuButton},
        {"Dec", msgCntrAppChangeFormat, dec},
        {"Oct", msgCntrAppChangeFormat, oct},
        {"Hex", msgCntrAppChangeFormat, hex},
        {pNull},
    {pNull}
};
```

When the user taps one of the buttons, it sends **msgCntrAppChangeFormat** to its client, which by default is the application. The message argument is the value of the button (dec, oct, or hex). **clsCntrApp**'s message handler for **msgCntrAppChangeFormat** looks at the message argument to determine which button the user tapped.

```
MsgHandlerWithTypes(CntrAppChangeFormat, P_ARGS, P_CNTRAPP_INST)
{
    APP_METRICS  am;
    WIN          thelabel;
    STATUS       s;
    char         buf[30];
```

```
    Debugf("CntrApp:CntrAppChangeFormat");

    //
    // Update mmap data
    //
    *(pData->pFormat) = (CNTRAPP_DISPLAY_FORMAT)(U32)pArgs;

    // Build the string for the label.
    StsRet(BuildString(buf, pData), s);

    // Get app metrics.
    ObjCallRet(msgAppGetMetrics, self, &am, s);

    // Get the clientWin.
    ObjCallRet(msgFrameGetClientWin, am.mainWin, &thelabel, s);

    // Set the label string.
    ObjCallRet(msgLabelSetString, thelabel, buf, s);

    return stsOK;
    MsgHandlerParametersNoWarning;
} /* CntrAppChangeFormat */
```

Chapter 10 / Handling Input (Tic-Tac-Toe)

Tic-Tac-Toe is a large, robust application which demonstrates how to "play along" with many of the PenPoint protocols affecting applications:

- SAMs (standard application menus)
- Selections
- Move/copy protocol
- Keyboard input focus
- Stationery
- Help
- Option sheets.

The rest of this chapter details the architecture of Tic-Tac-Toe, its files, classes, objects, etc, and describes some enhanced application features implemented in Tic-Tac-Toe.

Tic-Tac-Toe Objects 10.1

No tutorial of this size can give you a course in object-oriented program design. It is an art, not a science. The books mentioned in Chapter 3 will be helpful. No matter what your experience level, you will find that you redesign your object hierarchy at least once. (Here at GO, we redesigned our class hierarchy countless times in the first two years—now they're quite stable.) But there are some generally-accepted techniques for breaking up an application into manageable components, and this tutorial will lead you through them.

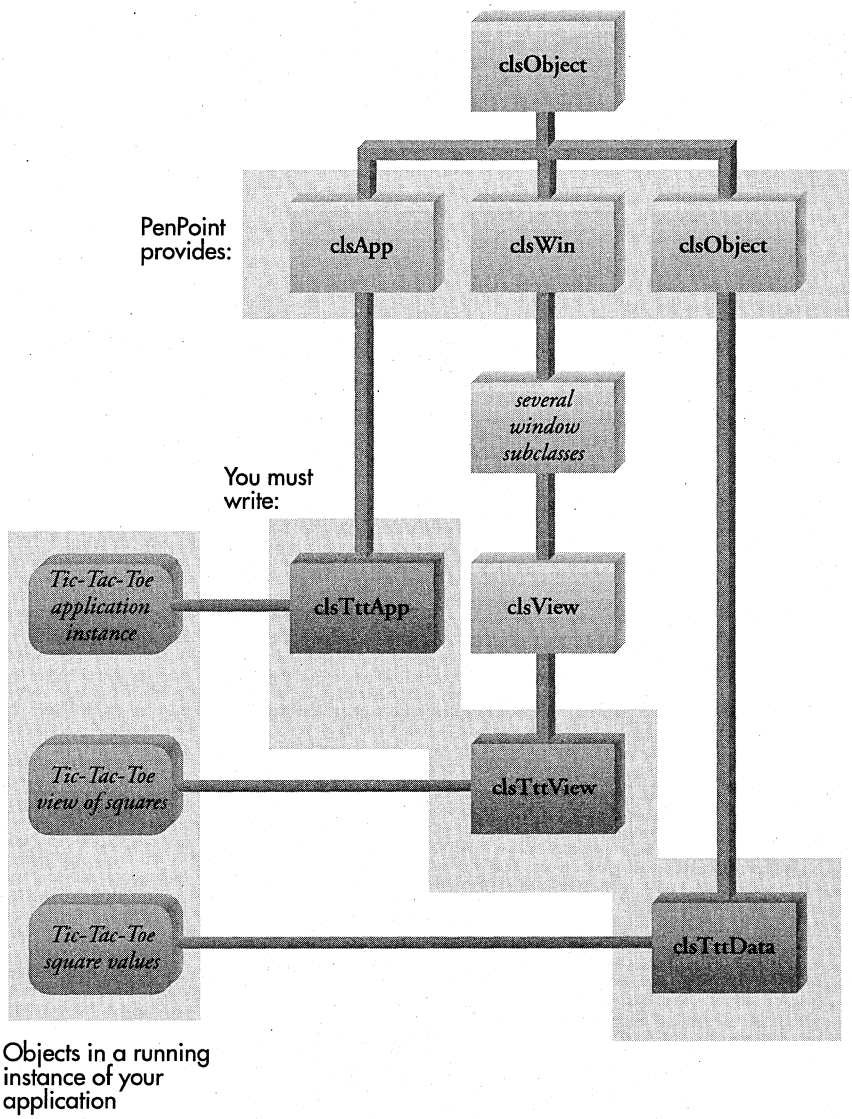
Each section from now on will discuss the various design choices made.

Application Components 10.1.1

A typical functional application does something in its application window, then saves data in the document working directory.

Tic-Tac-Toe does this: it displays a tic-tac-toe board in its window, then stores the state of the board. Its application class is **clsTttApp**. The application creates its own class to display the board, **clsTttView**. It also creates a separate object just to store the state of the board, **clsTttData**.

Figure 10-1
Tic-Tac-Toe Classes and Instances



Separate Stateful Data Objects 10.1.2

The tic-tac-toe data object's set of Xs and Os are the main part of its *state*, which it must preserve.

The application and view also maintain some state, the application files its version, and the view remembers how thick to draw the lines on the tic-tac-toe board.

Tic-Tac-Toe Structure 10.2

This table lists the files in Tic-Tac-Toe:

Table 10-1
Tic-Tac-Toe Files

File Name	Purpose
METHODS.TBL	Message tables for clsTttApp, clsTttView, and clsTttData.
TTTPRIV.H	TttDbgHelper support macro and debug flags, TTT_VERSION typedef, function definitions for routines in TTTUTIL.C and debugging routines in TTTDBG.C, and class UID definitions.
TTTAPP.C	Implements the main routine and most of clsTttApp's message handlers.
TTTVIEW.C	Implements most of clsTttView, handling repaint and input.
TTTDATA.C	Implements clsTttData.
TTTUTIL.C	Utility routines to create scrollwin, create and adjust menu sections, read and write filed data and version numbers, get application components, handle selection. Also application-specific routines to manipulate Tic-Tac-Toe square values.
TTTVOPT.C	Message handlers for the option sheet protocol.
TTTVXFER.C	Message handlers for the move/copy selection transfer protocol.
TTDBG.C	Miscellaneous routines supporting the Debug menu choices (dump, trace, force repaint, etc.).
TTTMBAR.C	Defines the TK_TABLE_ENTRY arrays for Tic-Tac-Toe's menu bar.
TTTAPP.H	Defines clsTttApp messages.
TTTVIEW.H	Defines clsTttView messages and their message argument structures, and defines tags used in the view's option sheet.
TTTDATA.H	Defines possible square values, various Tic-Tac-Toe data structures, and clsTttView messages and their message argument structures.
S_TTT.C	Sets up UID-to-string translation tables which the Class Manager uses to provide more informative debugging output.

Tic-Tac-Toe Window 10.3

There is no pre-existing class which draws letters in a rectangular grid. So, some work is needed here. The PenPoint UI Toolkit provides labels which can have borders, along with **clsTableLayout** which lets you position windows in a regular grid. So, you could create the tic-tac-toe board by creating nine of one-character labels in a table layout window. However, there are some problems with this:

- Labels don't (ordinarily) scale to fit the space available.
- Each label is a window. Windows are relatively "cheap" in PenPoint, but if we were to change to a 16 x 16 board, it would use 256 single-character label windows.

So, to start with, we'll create a single window and draw the squares in this.

Coordinate System 10.3.1

The obvious coordinate system is one unit is one square. However, this system makes it difficult to position characters within a square, since you specify coordinates for drawing operations in S32 coordinates.

The tic-tac-toe view uses local window coordinates for its drawing.

Advanced Repainting Strategy 10.3.2

As explained in Hello World, the window system tells windows to repaint. When a window receives **msgWinRepaint**, it *always* sends self **msgWinBeginRepaint**. This sets up the **update region** of the window—the part of the window where pixels can be altered—to the part of the window that needs repainting. After sending **msgWinBeginRepaint**, a window can only affect its pixels which the window system thinks need repainting, no matter where the window tries to paint.

Because the window system must calculate this dirty area, it makes the area available to advanced clients by passing it back in the message argument structure of **msgWinBeginRepaint**. In a fit of probable overkill, the Tic-Tac-Toe view is such an advanced client. Tic-Tac-Toe looks at the **RECT32** structure passed back and figures out what parts of the grid lines and which squares it needs to repaint. It wants to do this in its own coordinate system, so it sends **msgWinBeginRepaint** to its DC.

View and Data Interaction 10.4

The Tic-Tac-Toe view displays what's in the data object, so it needs access to the data maintained by the data object. There are various ways that a view can get to this state. It could share memory pointers with the data object, or it could use the specialized function **ObjectPeek** to look directly at the data object's instance data.memory. However, both of these methods compromise the separation of view and data into two objects. A purer approach is to have the view object send the data object a message when it needs to know the data object's state, but you still have to decide whether the data object should pass the view its internal data structures or a well-defined public data structure.

These are the classic problems of encapsulation and abstraction faced in object-oriented program design.

Data Object Design 10.4.1

Tic-Tac-Toe's data object class, **clsTttData**, is similar to CounterApp's **clsCntr**. It lets its client perform these tasks:

- Specify an initial board layout in **msgNew**
- Get the value of all the squares (**msgTttDataGetMetrics**)
- Set the value of all the squares (**msgTttDataSetMetrics**)
- Set the value of a particular square (**msgTttDataSetSquare**).

clsTttData gets and sets the square values as part of getting and setting all of its metrics. The theory is that any client that wants to set and get this probably wants all the information about the data object. (In fact, **clsTttData's** instance metrics comprise only its square values.)

Design decision.

Instance Data vs. Instance Info 10.4.2

The instance data for **clsTttApp**, **clsTttView**, and **clsTttData** is a pointer in each case. The instance data points to a data structure outside the instance where the class stores the instance "information." There are some advantages to this:

- You don't have to use **ObjectWrite** to update instance data every time state changes, since the pointer never changes
- A class can have variable-sized instance information.

It does mean that the class has to allocate space for the instance information. The Tic-Tac-Toe classes do this using **OSHeapBlockAlloc** in **msgInit** processing.

Saving a Data Object 10.4.3

clsTttApp tells its view to file, and an instance of **clsView** automatically files its data object.

Handling Failures During msgInit and msgRestore 10.4.4

msgInit and **msgRestore** both create objects. It is vital that the handlers for these messages guarantee that the object is initialized to some well-known state, *even if* your handler or some ancestor failed in some way, because after a failed creation, the object will in fact receive **msgDestroy**.

Note how **clsTttData** writes appropriate data into its instance data even in the case of an error.

```
MsgHandlerWithTypes(TttDataInit, P_TTT_DATA_NEW, PP_TTT_DATA_INST)
{
    P_TTT_DATA_INST pInst;
    STATUS          s;
    DbgTttDataInit(("TttDataInit"))
    //
    // Initialize for error recovery.
    //
    pInst = pNull;
```

The Selection and Keyboard Input 10.5

When a computer permits multiple windows on-screen, it must decide which window receives keyboard input. PenPoint uses a selection model, meaning that it sends keyboard input to the object holding the selection (along with all the other move/copy/options/delete messages which the selected object may receive). So, to allow typing, the Tic-Tac-Toe view must "be selectable."

How Selection Works

10.5.1

Try tapping in a text field or text view.

There can only be one primary **selection** in the Notebook UI at a time. The user usually selects something on-screen by tapping on it. In response to holding the selection, the selected thing is highlighted. Depending on what is selected, the user can then operate on the selection by deleting it, copying it, asking for its option sheet, and so on.

PenPoint's Selection Manager keeps track of which object has the selection. However, it is up to the class implementor to support selections, to highlight the selection, and to implement whatever operations on that selection make sense. Text fields and text views support selections, but **clsWin** and **clsObject** do not.

Which Object?

10.5.1.1

Because the Tic-Tac-Toe view is the object which draws the tic-tac-toe board, it makes sense for it to track selections. The selection does not change the board contents, so the Tic-Tac-Toe data object need not care.

When the user selects in the Tic-Tac-Toe view, the action selects the entire view. A more realistic class would figure out which of its squares the user selected, but the principles used by **clsTttView** are the same.

clsTttView responds to selection messages sent by the Selection Manager. It asks **theSelectionManager** if it holds the selection, and if so, repaints differently to indicate this fact.

What Event Causes Selections?

10.5.1.2

The application developer must decide what input event causes a selection in the crossword view: the usual is a pen-up event or a pen-hold timeout. On receiving this input event, the object wishing to acquire the selection should send **msgSelSetOwner** to the special Selection Manager object. Since the Tic-Tac-Toe view also supports keyboard input, it also calls the routine **InputSetTarget** to acquire the keyboard focus. From this point on, the view receives keyboard input events, and may receive other messages intended for the selection, such as options, move, and copy.

The object which has acquired the selection should highlight the selected "thing" on-screen; **clsTttView** draws the board in gray when it has the selection.

This code is from **TttViewRepaint**:

```
// Fill the dirty rect with the appropriate background. If we hold the
// selection, the appropriate background is grey, otherwise it is white.
//
ObjCallJmp(msgSelOwner, theSelectionManager, &owner, s, Error);
if (owner == self) {
    DbgTttViewRepaint(("owner is self"))
    ObjectCall(msgDcSetBackgroundRGB, (*pData)->dc, \
            (P_ARGS)sysDcRGBGray33);
```

```
} else {
    DbgTttViewRepaint(("owner is not self"))
    ObjectCall(msgDcSetBackgroundRGB, (*pData)->dc, \
            (P_ARGS)sysDcRGBWhite);
}
```

Supporting Selections. 10.5.1.3

When a Tic-Tac-Toe view receives a **msgPenHoldTimeout** input event, **clsTttView** sends self **msgTttViewTakeSel** telling it to acquire the selection, and sends self **msgWinUpdate**, which forces it to repaint the entire board. (If the view supported square-by-square selection, it would convert the input event X-Y coordinates to a square location on the board).

Move/Copy Protocol 10.5.1.4

The selection holder receives a variety of messages, including **msgSelnYield** and the move/copy protocol messages. Because **clsTttView** inherits from **clsEmbeddedWin**, it can rely on **clsEmbeddedWin**'s default handling of many selection messages.

More on View and Data Interaction 10.6

Thus far the data maintained by the data object has been static; now the user can change the data. But the user interacts with the view, not the data object. It's the view that knows what characters the user entered. The view must tell the data object about the change as well as draw the new data.

The natural way to do this might seem to be for the view to draw the new letter in the square, and then tell the data object about the new letter. However, this is not the view-data model. Instead, the view tells the data object about the changed letter by sending it **msgTttDataSetSquare**. When the data object receives this message, it updates its state, then broadcasts **msgTttDataChanged** to all its observers. When the view receives **msgTttDataChanged**, it knows it needs to repaint the board. The advantages of this model are that the data object can remain in control of its data: it could reject the update message from the view, and the view would not display bad data. Also, it allows for several views to display the same data object, since if any of them updates the data object, they all are told about the change.

To actually draw the new square, **clsTttView** dirties the rectangle of the square that changed. This also may seem odd—why not paint the square immediately with the new value when notified by the data object of the new value? But the Tic-Tac-Toe view already knows how to *repaint* itself, it's nice to take advantage of the batching provided by the window system's repaint algorithm.

The Text subsystem is a more compelling argument for the view-data model used by Tic-Tac-Toe. Using the same kind of message flow to update text views and text data objects, Text does indeed allow several views of the same underlying object, and it has a very intelligent window repainting routine.

Handwriting and Gestures 10.7

Views inherit from **clsGWin**, so there is little extra work required to make **clsTttView** respond to input events and gestures.

Input Event Handling 10.7.1

There is one input message in PenPoint, **msgInputEvent**. Within the message arguments of this is a device code that indicates the type of input event. Device codes all begin with **msgKey** or **msgPen**, which is slightly confusing because objects never receive these messages, they always receive **msgInputEvent**. **clsTttView**'s **msgInputEvent** handler calls separate routines to handle pen events and keyboard events. The pen event handler, **TttViewPenInput**, determines whether or not a pen-hold timeout has occurred.

If a timeout occurred, the handler takes the selection by sending **msgTttViewTakeSel** to self, updates its window to show selection, and then determines whether the hold followed a tap or not. If the hold followed a tap (pPen->taps > 0), the handler starts a copy operation; if it didn't follow a tap, the handler starts a move operation.

If no timeout occurred, the handler allows **clsTttView**'s ancestor to handle the message.

```
if (MsgNum(pArgs->devCode) == MsgNum(msgPenHoldTimeout)) {
    P_PEN_DATA pPen = (P_PEN_DATA)(pArgs->eventData);
    ObjCallJmp(msgTttViewTakeSel, self, pNull, s, Error);
    ObjCallJmp(msgWinUpdate, self, pNull, s, Error);
    ObjCallJmp((pPen->taps == 0) ? msgSelBeginMove : msgSelBeginCopy,
            self, &pArgs->xy, s, Error);
    s = stsInputTerminate;
} else {
    s = ObjectCallAncestorCtx(ctx);
}
```

Gesture Handling 10.7.2

If the pen event handler passes a pen event to **clsTttView**'s ancestor, the event ends up being handled by **clsGWin**. If **clsGWin** recognizes the pen event as a gesture, it sends **msgGWinGesture** to **self**. In other words, when the user draws a gesture on the Tic-Tac-Toe view, the view receives **msgGWinGesture**.

The arguments for **msgGWinGesture** include the gesture in the form of a message identifier. The class for the message is **clsXGesture**. The number of the message encodes the actual gesture detected by **clsGWin**.

The handler for **msgGWinGesture** first checks that the class of the gesture is **clsXGesture** (in a `switch` statement). If not, it lets its ancestor handle the message. If the class is **clsXGesture**, the handler uses a switch statement to determine the message number and acts accordingly.

The ClsNum macro extracts the class number from a message; the MsgNum macro extracts the message number from a message.

```
MsgHandlerWithTypes(TttViewGesture, P_GWIN_GESTURE, PP_TTT_VIEW_INST)
{
    STATUS      s;
    OBJECT      owner;
#ifdef DEBUG
    {
        P_CLS_SYMBUF    mb;
        DbgTttViewGesture(("self=0x%lx msg=0x%lx %s", self, pArgs->msg,
                ClsMsgToString(pArgs->msg,mb)))
    }
#endif // DEBUG
    switch(ClsNum(pArgs->msg)) {
        case ClsNum(clsXGesture):
            switch(MsgNum(pArgs->msg)) {
                case MsgNum(xgs1Tap):
                    ObjCallJmp(msgTttViewToggleSel, self, pNull, \
                            s, Error);
                    break;
                case MsgNum(xgsCross):
                    StsJmp(TttViewGestureSetSquare(self, pArgs, tttX), \
                            s, Error);
                    break;
                case MsgNum(xgsCircle):
                    StsJmp(TttViewGestureSetSquare(self, pArgs, tttO), \
                            s, Error);
                    break;
                case MsgNum(xgsPigtailVert):
                case MsgNum(xgsPigtailHorz):
                    StsJmp(TttViewGestureSetSquare(self, pArgs, tttBlank), \
                            s, Error);
                    break;
                case MsgNum(xgsCheck):
                    // Make sure there is a selection.
                    ObjCallJmp(msgSelOwner, theSelectionManager, \
                            &owner, s, Error);
                    if (owner != self) {
                        ObjCallJmp(msgTttViewTakeSel, self, pNull, s, Error);
                        ObjCallJmp(msgWinUpdate, self, pNull, s, Error);
                    }
                    // Then call the ancestor.
                    ObjCallAncestorCtxJmp(ctx, s, Error);
                    break;
                default:
                    DbgTttViewGesture(("Letting ancestor handle gesture"))
                    return ObjCallAncestorCtxWarn(ctx);
            }
            break;
        default:
            DbgTttViewGesture(("Letting ancestor handle gesture"))
            return ObjCallAncestorCtxWarn(ctx);
    }
    DbgTttViewGesture(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewGesture(("Error; return 0x%lx",s))
    return s;
} /* TttViewGesture */
```

Keyboard Handling 10.7.3

clsTttView's keyboard input routine handles multi-key input, for example, when the user presses two keys at once or in rapid succession. The device code for this is **msgKeyMulti**, and the input event data includes the number of keystrokes and an array of their values. The **keyCode** of a key value is a simple ASCII number.

clsTttView handles X or O or Space on the keyboard.

Chapter 11 / Refining the Application (Tic-Tac-Toe)

Tic-Tac-Toe has many of the niceties expected of a real application. Many of these enhancements are independent of the program, and could be added to Empty Application as easily as to Tic-Tac-Toe.

Debugging 11.1

You can use DB, the PenPoint Source-level debugger, to step through code. In an object-oriented system, your objects receive many messages from outside sources, many of which you may not expect. It's useful to be able to easily track the flow of messages through your routines, and to turn this on and off while your program is running. As you've noticed if you've looked at the code, Tic-Tac-Toe has extensive support for debugging. The facilities it uses are:

- Tracing messages using **msgTrace**
- Printing debug messages using **Debugf**
- Dumping objects' state using **msgDump**
- Giving the Class Manager symbolic names for its objects, messages, and status values using **ClsSymbolsInit.**

The complexity in Tic-Tac-Toe arises because it lets you turn features on and off while the program is running.

Tracing 11.1.1

It's very useful to have a log of what messages are coming in. You can get a message log by turning on **message tracing**; you can either turn it on for a class or for a single instance of that class.

In DEBUG mode, TTTMBAR.C defines a debug menu which can turn tracing on or off for the various classes. All the menu items send **msgTttAppChangeTracing** to the application. The message argument encodes the target object to trace and whether to trace it or not:

```
static TK_TABLE_ENTRY traceMenu[] = {
  {"Trace App On",      msgTttAppChangeTracing, MakeU32(0,1)},
  {"Trace App Off",     msgTttAppChangeTracing, MakeU32(0,0)},
  {"Trace View On",     msgTttAppChangeTracing, MakeU32(1,1)},
  {"Trace View Off",    msgTttAppChangeTracing, MakeU32(1,0)},
  {"Trace Data On",     msgTttAppChangeTracing, MakeU32(2,1)},
  {"Trace Data Off",    msgTttAppChangeTracing, MakeU32(2,0)},
  {pNull}
};
```

The **TttDbgChangeTracing** routine is implemented in TTTDBG.C. It simply sends **msgTrace** to the target object, with an argument of true or false.

Debugf Statements and Debug Flags 11.1.2

Going beyond message tracing, it's useful to print out what your application is doing at various stages. One approach is to add simple **Debugf** statements as you debug various sections. However, in a large program you quickly get overwhelmed by debugging statements you're not interested in. Tic-Tac-Toe leaves all the **Debugf** statements in the code, and controls which statements show up by examining a debugging flag set. It uses **DbgFlagGet** to check whether a flag is set, the same as Empty App and the other simpler applications. What Tic-Tac-Toe provides is an easy way to print out a string identifying the routine, followed by whatever `printf`-style parameters you want to use. Thus this code:

```
if (s == stsFSNodeNotFound) {
DbgTttAppCheckStationery(("file not found; s=0x%lx",s))
    goto NormalExit;
}
```

will print out

```
TttAppCheckStationery: file not found; s=0xnum
```

but only if the appropriate debugging flag is set.

So, how is it implemented? A definition of its debug routine precedes each function for which you want to print debugging information, for example, **DbgTttAppCheckStationery**.

```
#define DbgTttAppCheckStationery(x) \
    TttDbgHelper("TttAppCheckStationery",tttAppDbgSet,0x1,x)
```

Call this macro anywhere that you might want to display a debugging string. The parameter to the macro (x) is the printf-style format string and any arguments (("file not found; s=0x%lx",s)). In order to treat multiple parameters as one, they must be enclosed in a second set of parentheses.

The **TttDbgHelper** routine checks if the specified flag (`0x0001`) is set in the specified debugging flag set (**tttAppDbgSet**), and if so prints the identifying string ("TttAppCheckStationery") together with any printf-style format string passed in (x).

There are 256 debugging flag sets, each of which has a 32-bit value. GO uses some of them for its applications—see \PENPOINT\SDK\INC\DEBUG.H for a full list. TTTPRIV.H defines the debugging flag sets used in Tic-Tac-Toe, such as **tttAppDbgSet**:

```
//
// Debug flag sets
//
#define tttAppDbgSet        0xC0
#define tttDataDbgSet       0xC1
#define tttUtilDbgSet       0xC2
#define tttViewDbgSet       0xC3
#define tttViewOptsDbgSet   0xC4
#define tttViewXferDbgSet   0xC5
```

Other routines use other flags.

In case you care, here's the definition of **TttDbgHelper**:

```
#define TttDbgHelper(str,set,flag,x) \
    Dbg(if (DbgFlagGet((set),(U32)(flag))) {DPrintf("%s: ",str); Debugf x;})
```

Dprintf is the same as Debugf, except that Dprintf doesn't insert an automatic new-line at the end of the function.

Dumping Objects 11.1.3

One of the messages defined by the Class Manager is **msgDump**. A class should respond to it by calling its ancestor, then printing out information about self's state. Most classes only implement **msgDump** in the DEBUG version of their code.

Tic-Tac-Toe lets you dump its various objects from its Debug menu. In TTTMBAR.C, it defines the menu:

```
static TK_TABLE_ENTRY debugMenu[] = {
  {"Dump View",         msgTttAppDumpView,          0},
  {"Dump Data",         msgTttAppDumpDataObject,    0},
  {"Dump App",          msgDump,                    0},
  {"Dump Window Tree",  (U32)dumpTreeMenu,          0, 0, tkMenuPullRight},
  {"Trace",             (U32)traceMenu,             0, 0, tkMenuPullRight | tkBorderEdgeTop},
  {"Force Repaint",     msgTttAppForceRepaint,      0, 0, tkBorderEdgeTop},
  {pNull}
};
```

The client of the menu is the application, so to dump the application all the menu item needs to do is send **msgDump**. For the view and data object, you would either have to change the clients of the menu items, or have the application class respond to special **msgTttAppDumpView** or **msgTttAppDumpData** messages by sending **msgDump** to the appropriate target. Tic-Tac-Toe does the latter; the handlers for these messages are in TTTDBG.C.

Dumping Any Object 11.1.3.1

Another approach is to have a generic dump-object function in the DEBUG version of your code which sends **msgDump** to its argument. When running DB, you can call this routine directly, passing it the UID of the object you want dumped.

Symbol Names 11.1.4

All the Class Manager's macros (**ObjCallRet**, **ObjCallWarn**, **ObjCallAncestorChk**, and so on) print a string giving the message, object, and status value if they fail (in DEBUG mode). You can also ask DB to print out messages, objects, and status values. Ordinarily the most the Class Manager and DB can do is print the various fields in the UID, such as the administrated field and message number. However, if you supply the Class Manager a mapping from symbol names to English names, it and DB will use the English names in their debugging output.

The Class Manager routine you use is **ClsSymbolsInit**. The routine takes three arrays, one for objects, one for messages, and one for status values. Each array is composed of symbol-string pairs. Tic-Tac-Toe sets up these arrays in the file S_TTT.C:

```
const CLS_SYM_STS tttStsSymbols[] = {
  0, 0};
const CLS_SYM_MSG tttMsgSymbols[] = {
  msgTttAppChangeDebugFlag, "msgTttAppChangeDebugFlag",
  msgTttAppChangeDebugSet,  "msgTttAppChangeDebugSet",
  ...
  ...
  msgTttViewTakeSel,    "msgTttViewTakeSel",
  0, 0};
const CLS_SYM_OBJ tttObjSymbols[] = {
  clsTttApp,    "clsTttApp",
  clsTttData,   "clsTttData",
  clsTttView,   "clsTttView",
  0, 0};
```

(Tic-Tac-Toe doesn't define any STATUS values.) **ClsSymbolsInit** also takes a fourth parameter, a unique string identifying this group of symbolic names. Here's the routine in S_TTT.C that calls **ClsSymbolsInit**:

```
{
    return ClsSymbolsInit(
            "ttt",
            tttObjSymbols,
            tttMsgSymbols,
            tttStsSymbols);
}
```

At installation (from process instance 0), TTT.EXE calls **TttSymbolsInit** to load these arrays. To save space, all of this code is excluded with an `#ifdef` if DEBUG is not set.

Generating Symbols Automatically 11.1.4.1

It's cumbersome to type in and update the arrays of UID-string pairs. At GO we have developed scripts that automatically generate files like S_TTT.C. These scripts require the MKS toolkit and other third-party utilities, so they are not part of the SDK release and are not supported. If you're interested in these scripts, contact GO Developer Support.

Printing Symbol Names Yourself 11.1.4.2

Tic-Tac-Toe just prints UIDs as long integers when it needs to print them out. You can also print them in hexadecimal format using the %p format code. If you want to print out the long names within your own code, the Class Manager defines several functions to convert objects, messages, and status values to strings, such as **ClsObjectToString**.

Installation Features 11.2

During installation, PenPoint automatically creates several application enhancements based on the contents of the application's installation directory:

- Stationery
- Help notebook documents
- Quick-help for the application's windows
- Application icons.

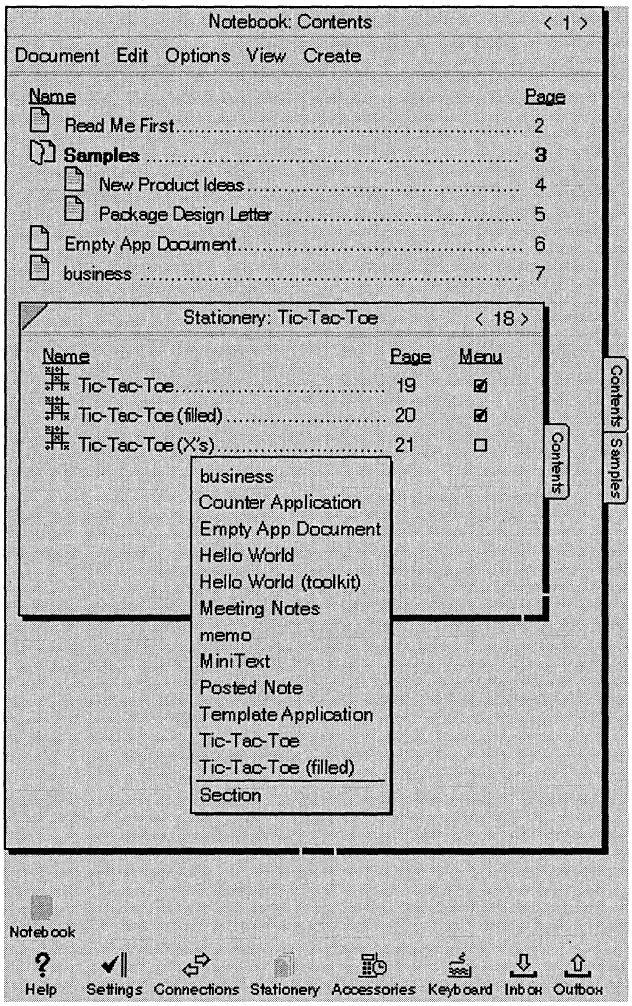
The nice thing about these enhancements is that you can create and modify them separate from writing and compiling the application. In fact all of these features could have been added to Empty Application, the very simplest application.

General details on application installation is covered in detail in *Part 12: Installation API* of the *PenPoint Architectural Reference.* This section only covers what Tic-Tac-Toe does.

Stationery 11.3

The user can pick a tic-tac-toe board to start with from a list of stationery. The user can draw a caret ^ over the table of contents to pop up a stationery menu, or can open the Stationery auxiliary notebook.

Figure 11-1
Stationery Notebook & Stationery Menu



Creating Stationery

11.3.1

The Installer looks for stationery in a subdirectory called STATNRY. Each stationery document should be in a separate directory in STATNRY. You can stamp the directories with long PenPoint names. You can also stamp the directories with attributes indicating whether the stationery should appear in the stationery menu and whether it should appear in the Stationery notebook.

Here's what the Tic-Tac-Toe makefile does:

```
stamp \penpoint\app\ttt\statnry /g "Tic-Tac-Toe (filled)" /d tttstat1 /a 00800274 1
stamp \penpoint\app\ttt\statnry /g "Tic-Tac-Toe (X's)" /d tttstat2
```

This gives each stationery document a different name, and marks one of them for inclusion in the stationery menu.

How Tic-Tac-Toe Handles Stationery 11.3.2

Stationery directories can contain a filed document—a regular instance of the application. To build such stationery you copy a document from the Notebook to the installation volume. One disadvantage of this is that it could make the stationery take up more space, since it's an entire filed document.

Instead, **clsTttApp** always checks for a file called TTTSTUFF.TXT in the document's directory when a document is first run (during **msgAppInit**). The routine is **TttAppCheckStationery** in TTTAPP.C. If it finds a TTTSTUFF.TXT file, **clsTttApp** opens it and sends **msgTttDataRead** to its data object. This tells the data object to set its state from the file.

clsTttData simply reads the first nine bytes of the file and sets its value from those; for example, the TTTSTUFF.TXT file for "Tic-Tac-Toe (filled)" (in \PENPOINT\APP\TTT\STATNRY\STAT1) is simply

```
xoxoxoxox stationery for tttapp
```

This saves a lot of space over a filed Tic-Tac-Toe document; however, note that this form of stationery doesn't include things like the thickness of the grid in the view. The user can always make stationery that is a full document by moving or copying a Tic-Tac-Toe document to the Stationery notebook.

The makefile for Tic-Tac-Toe creates the STATNRY directory in \PENPOINT\APP\TTT, and then creates the two directories STAT1 and STAT2. The makefile copies the file FILLED.TXT to STAT1 and names it TTTSTUFF.TXT; it then copies the file XSONLY.TXT to STAT2 and also names it TTTSTUFF.TXT

Help Notebook 11.4

Tic-Tac-Toe has its own Help information, which the user can view in the Help auxiliary notebook. Each page in the Help Notebook is a separate document.

Tic-Tac-Toe doesn't have to do anything to support this.

Creating Help Documents 11.4.1

During installation, if there is anything in the HELP subdirectory of the application home, the Installer creates a subsection for the application in Applications section of the Help notebook. The Installer automatically installs help documents in this section of the Help notebook. Like stationery, you put help documents in subdirectories of a special subdirectory in the Tic-Tac-Toe installation directory, called HELP. You can stamp the directories with long PenPoint names, these are the names of the pages in the Help notebook.

The Tic-Tac-Toe makefile creates a HELP directory in \PENPOINT\APP\TTT and creates HELP1 and HELP2 directories in HELP. The makefile copies STRAT.TXT to HELP1 and names it HELP.TXT; it then copies RULES.TXT to HELP2 and names it also HELP.TXT.

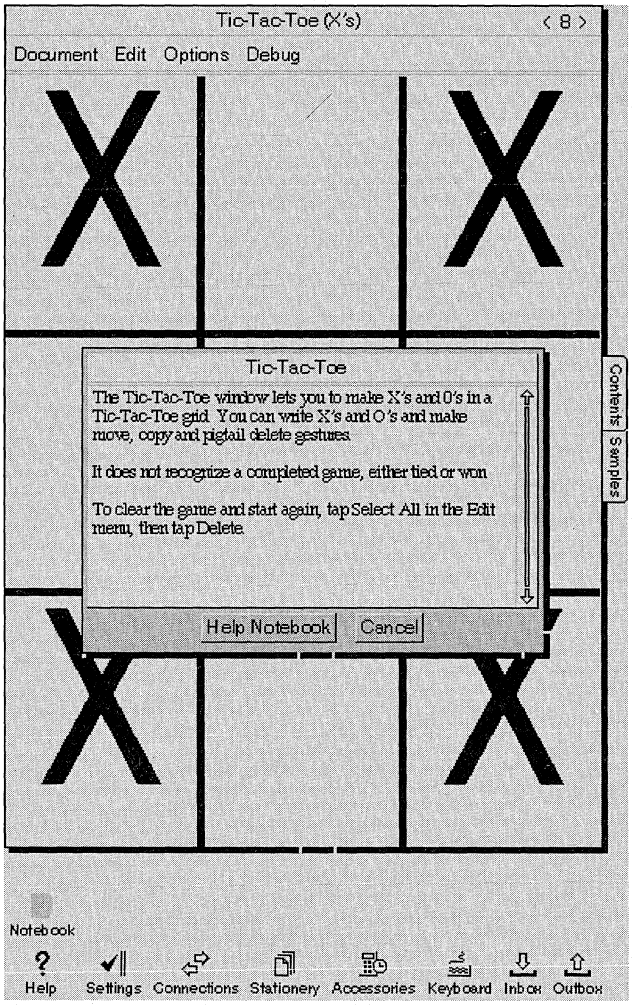
Help documents can either be complete instances of filed documents (of any type, such as Drawing Paper, TextEdit, even a help version of your application), or a

simple text file. If the directory contains a simple text file, the Help notebook will run a version of MiniText on that page, displaying the contents of the file. This is the approach Tic-Tac-Toe uses.

Quick Help 11.5

Quick Help is the other form of help in PenPoint. The Quick Help window appears when the user makes the help gesture ? in a window, or taps on a window when Quick Help is up.

Figure 11-2
Quick Help



clsGWin, the gesture window class, automatically handles the help gesture. It will invoke the Quick Help window, if it knows what to display. Instead of specifying to **clsGWin** what strings to display, you create your strings in a separate resource, and just give **clsGWin** an ID which it uses to locate the strings. In the **msgNewDefaults** handling of **clsTttView**:

```
MsgHandlerWithTypes(TttViewNewDefaults, P_TTT_VIEW_NEW, PP_TTT_VIEW_INST)
{
    DbgTttViewNewDefaults(("self=0x%lx",self))

    pArgs->win.flags.input |= inputHoldTimeout;
    pArgs->gWin.helpId = tagTttQHelpForView;
    ...
    pArgs->view.createDataObject = true;
    ...
```

This is the only thing **clsTttView** must do to handle quick help.

Creating Quick Help Resources 11.5.1

One way to create resources is to tell a resource file to file an object, using say **msgResPutObject**. This is what happens when an application is told to save a document.

However, one goal of resources is to separate the definition of a resource from the application that uses it. So you can also compile resources under DOS, putting them in a resource file, and read them from within PenPoint applications. These resources aren't objects, they are basically predefined data structures.

In the case of Quick Help, a quick help resource consists of three parts:

- The strings that contain the quick help text.
- A tagged string array resource (type RC_TAGGED_STRING) that associates each text string with a tag. The tags are used by the GWin helpIds to associate a GWin with its quick help text.
- An RC_INPUT structure containing:
  - A list resource ID created from the administered portion of the quick help ID (in this cas **clsTttView**) and the quick help group (usually **resGrpQhelp**)
  - A pointer to the tagged string array resource for the class
  - A length field (updated by the resource compiler)
  - The identifer for the string array resource agent (**resTaggedStringArrayResAgent**).

Each quick help string has two parts, which are separated by two vertical line characters (||). The first part is the title for the quick help card; the second part is the quick help text. The vertical line characters are not printed when quick help displays.

This is the quick help string for the Tic-Tac-Toe view, defined in TTTQHELP.RC:

```
//
// Quick Help string for the view.
//
static CHAR tttViewString[] = {
    // Title for the quick help window
    "Tic-Tac-Toe||"
    // Quick help text
    "The Tic-Tac-Toe window lets you to make X's and 0's in a Tic-Tac-Toe "
```

```
    "grid. You can write X's and O's and make move, copy "
    "and pigtail delete gestures.\n\n"
    "It does not recognize a completed game, either tied or won.\n\n"
    "To clear the game and start again, tap Select All in the Edit menu, "
    "then tap Delete."
};
```

This is the RC_TAGGED_STRING resource for tttView and its option card:

```
// Define the quick help resource for the view.
static P_RC_TAGGED_STRING   tttViewQHelpStrings[] = {
    tagCardLineThickness,   tttOptionString,
    tagTttQHelpForLineCtrl, tttLineThicknessString,
    tagTttQHelpForView,     tttViewString,
    pNull
};
```

This is the RC_INPUT structure for the tttView quick help:

```
static RC_INPUT tttViewHelp = {
    MakeListResId(clsTttView, resGrpQhelp, 0),
    tttViewQHelpStrings,            // Name of the string array
    0,
    resTaggedStringArrayResAgent    // Use string array resource agent
};
```

See *Part 11: Resources*, in the *PenPoint Architectural Reference*, for more information on resource compiling and the specifics of quick help resources.

To compile resource definitions into a resource file, you use the PenPoint Resource Compiler (\PENPOINT\SDK\UTIL\DOS\RC).

The Installer copies the application resource file (APP.RES) during installation. Hence the makefile tells the resource compiler to append the quick help resources to APP.RES.

Standard Message Facility 11.6

The PenPoint standard message facility (StdMsg) provides a standard way for your application to display modal dialog boxes, error messages, and progress notes without requiring it to create UI objects. StdMsg uses **clsNote** (see NOTE.H) to display its messages. Notes have a title, a message body, and zero or more command buttons at the bottom.

Message text and command button definitions are stored in resource files. StdMsg supports parameter substitution for the message text and button labels (see CMPSTEXT.H). A 32-bit value (a tag in the case of dialog boxes and a status code in the case of errors) is used to select the appropriate resource.

StdMsg provides the following routines for when the programmer knows exactly which message is to be displayed:

- System and application dialog boxes use StdMsg(tag, ...)
- Application errors use StdError(status, ...)
- System errors use StdSystemError(status, ...)
- Progress notes use StdProgressUp(tag, &token, ...)

StdMsg(), **StdError()**, **StdSystemError()**, **StdProgressUp()** are **varArgs** functions. Any parameter substitutions are supplied with the argument list, much like `printf`. Like `printf`, there is no error checking regarding the number and type of the substitution parameters. The first three functions return an integer, which indicates the command button that the user tapped. Progress notes, which use **StdProgressUp()**, don't have a command bar.

StdMsg also provides support for the situation where an unknown error status is encountered: **StdUnknownError()**. This function does not provide parameter substitution or multiple command buttons, it always displays a single "OK" command button. **StdUnknownError()** replaces any parameter substition specifications in the text with "???".

Using StdMsg Facilities 11.6.1

To use StdMsg, you first define the message text strings. These strings are held in string array resources, like quick help. A single resource holds all the strings for a given class. There is a separate string array for dialog boxes and error messages. You should store the application message resources in the application's APP.RES file.

Here's an example of a typical resource file definition:

```
static P_STRING dialogClsFoo[] = {
"This is the first dialog message.",
@CP = "[Go] [Stop] This is the second dialog message. str: ^1s",
};
static P_STRING errorClsFoo[] = {
"This is the first error message.",
@CP = "[Retry] [Cancel] This is the second error message. count: ^1d",
};
static RC_INPUT dialogTabClsFoo = {
resForStdMsgDialog(clsFoo), dialogClsFoo, 0,
@CP = resStringArrayResAgent
};
static RC_INPUT errorTabClsFoo = {
resForStdMsgError(clsFoo),  errorClsFoo, 0,
@CP = resStringArrayResAgent
};
P_RC_INPUT  resInput [] = {
&dialogTabClsFoo, &errorTabClsFoo,
@CP = pNull
};
```

You must define a tag or error status for each string. The string's position in the string array determines its tag or status index (starting from 0). Here are the definitions for the example above:

```
// #define tagFooDialog1                MakeDialogTag(clsFoo, 0) // @CP = #define tagFooDialog2
MakeDialogTag(clsFoo, 1)
// #define  stsFooError1                MakeStatus(clsFoo, 0) // @CP = #define  stsFooError2
MakeStatus(clsFoo, 1)
```

To create a note from the items defined above, simply call **StdMsg()** or **StdError()**. For example:

```
buttonHit = StdMsg(tagFooDialog2, "String"); s = ObjectCall(...);
if (s  stsOK) {
if (s == stsFooError1) {StdError(stsFooError1);
} else {StdUnknownError(s);
}
}
```

Progress notes are slightly different from the message functions. Your application displays a progress notes when it begins a lengthy operation, and takes the note down when the operation completes. PenPoint 1.0 does not support cancellation of the operation. Here's an example of progress note usage:

```
SP_TOKEN    token;
StdProgressUp(tagFooProgress1, &token, param1, param2);
... Lengthy operation ...
StdProgressDown(&token);
```

Substituting Text and Defining Buttons 11.6.2

The message strings can contain substituted text and definitions for buttons. String substitution follows the rules defined by the compose text function (defined in CMPSTXT.H). A button definition is a substring enclosed in square brackets at the beginning of the message string. You can define any number of buttons, but you must define all buttons at the beginning of the string. The button substrings can contain text substitution. If the string doesn't define any buttons, StdMsg creates a single "OK" button.

StdMsg(), **StdError**(), and **StdSystemError**() return the button number that the user tapped when dismissing the note. Button numbers start with 0. For example, this string definition would result in a return value of 1 if the user tapped Button1:

```
"[Button0] [Button1] [Button2] Here's your message!"
```

Be aware that these functions might also return a negative error status, which indicates that a problem occurred inside the function.

You can break your message up into paragraphs by putting two newline characters at the paragraph breaks. For example:

```
"Here's the first paragraph.\n\nHere's the second one."
```

StdMsg and Resource Files or Lists 11.6.3

There are variations of **StdMsg**() and **StdError**() that allow you to specify the resource file handle or resource list to use. These are most useful for PenPoint Services, where there is no default resource list available. These messages are:

- **StdMsgRes(resFile, tag, ...)**
- **StdErrorRes(resFile, status, ...)**

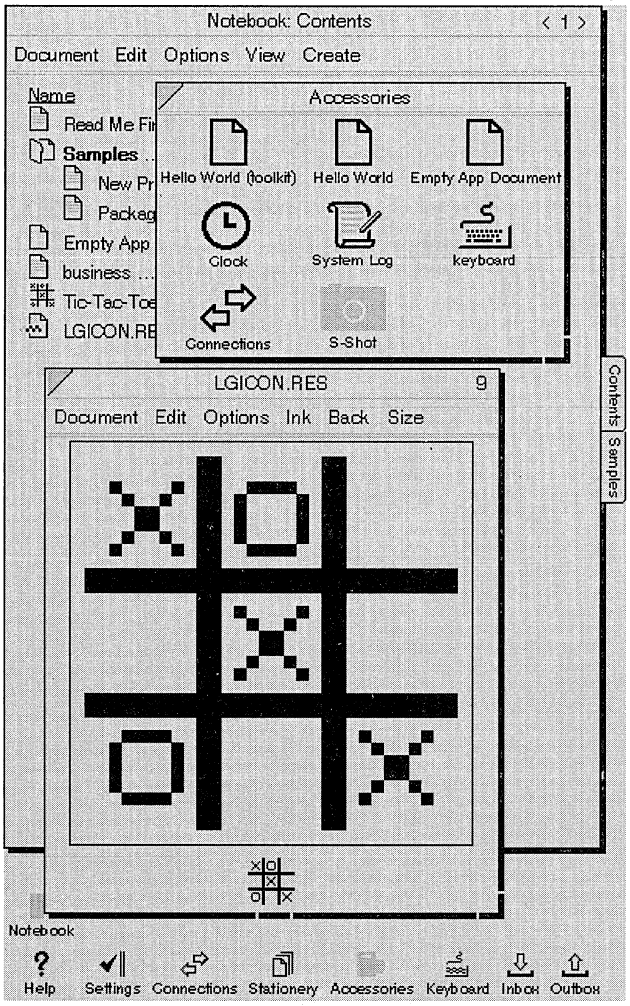
StdMsg Customization Function 11.6.4

The function **StdMsgCustom**() allows you to customize a StdMsg note. The function returns the UID of the note object (created by **clsNote**), without displaying it. You can modify this object as you wish and then display it yourself using the messages defined by **clsNote**.

Bitmaps (Icons) 11.7

PenPoint uses icons to represent applications in the table of contents and in browsers. You can also use icons in your own applications. In PenPoint terminology, the icon includes optional text as well as a bitmap picture. There are default bitmaps for applications and documents, but you can create your own using the bitmap editor.

Figure 11-3
Application and Document Icons



When it needs a bitmap, the Application Framework searches for it by resource ID in your application's resource list. To begin with, your application's resource file (APP.RES) doesn't have the resource, so the search gets the default bitmap in

the system resource file. However, if you put a different icon in your application's resource file, it will be used instead. You don't need to make any changes to your application to support this.

Creating Icons 11.7.1

The bitmap editor application is available in the PC \PENPOINT\APP\BITMAP directory. It is also available on the PenPoint Goodies disk.

The bitmap editor needs to generate a bitmap as a resource and put it in a resource file. However, it conforms to the PenPoint document model, so it has no Save command. Instead, you use the About... menu item in the Document menu to bring up the Export option card to specify the type of bitmap resource, and then use the Export... command to actually generate the bitmap resource. You generally export four bitmaps, two each for the application and document in 16x16 and 32x32 sizes. Although you can export them to the APP.RES file your application's installation directory, it is often preferable to create separate SMICON.RES and LGICON.RES files for the large and small icons.

For more information on using the bitmap editor, see *Part 3: Tools* in *PenPoint Development Tools.*

Chapter 12 / Releasing the Application

You're almost done, but not quite. Before you make your application available to the larger world of PenPoint users, you must complete these tasks:

- Register your classes with GO
- Document the application
- Prepare your distribution disks.

You should also consider making your classes available to other developers. If you do so, you need to document the API for those classes.

Registering Your Classes 12.1

While developing an application, you can identify your classes with the well-known UIDs **wknGDTa** through **wknGDTg**. Of course, if you use these well-known UIDs in a published application, they will conflict with other developers who use your application and attempt to use these well-known UIDs to test their own applications.

When you are fairly sure that you will publish your application, you must contact GO Developer Technical Support (preferably through electronic mail) for an administered object value for each of your *public* classes. Remember that a UID consists of an administered object value, a version number, and a scope (global or local, well-known or private).

Documenting the Application 12.2

The need for quality documentation cannot be over-emphasized. There are three ways in which you should document your application:

- Manuals or other form of separate documentation
- Pages in the Help notebook
- Quick Help text.

Writing Manuals 12.2.1

For more information on documenting your application, contact GO Developer Technical Support and ask for Tech Note #8, Documenting PenPoint Applications. This technical note, written by GO's end-user documentation group, provides information about how GO writes and produces its end-user documentation. The tech note also give print specifications, if you want your documentation to appear similar to GO's.

Screen Shots 12.2.2

The S-Shot utility enables you to capture TIFF images of PenPoint computer screens. You can then incorporate your images into your documentation. S-Shot is on the SDK Goodies disk.

Gesture Font 12.2.3

For developer and end-user documentation, GO created an Adobe Type 1 font that depicts the PenPoint gesture set. If you use a PostScript printer, you can incorporate this font into your documentation and on-line help.

Registered developers may request a copy of the PenPoint Gesture font from GO by contacting PenPoint Developer Tech Support.

On-Disk Structure 12.3

When developing your application, the PenPoint file organization requires you to place your files in certain specific directories under the \PENPOINT directory. This is described in detail in Chapter 3 of *Part 12: Installation API* in the *PenPoint Architectural Reference.*

Before distributing your application, you should ensure that all your auxiliary files, such as Help notebook pages, stationery, resource files, and so on are in their correct directories.

Sharing Your Classes 12.4

If you have created a component class that might be useful to other PenPoint developers, you should consider licensing the class.

Appendix Sample Code

This appendix lists the sample code referred to in the preceeding chapters of this book. The subdirectories in \PENPOINT\SDK\SAMPLE contain the complete source files for these and other sample programs.

The sample code for these applications is listed in this appendix:

Empty App The most simple application you can create.

Hello World (Toolkit) A simple Hello World application that uses the UI Toolkit.

Hello World (Custom Window) A simple Hello World application that uses the Image Point graphics subsystem.

Counter Application A simple application that saves and restores its data.

Tic-Tac-Toe A fully featured PenPoint application.

Template Application A template for a fully-featured PenPoint application.

This appendix describes, but does not list, the sample code for:

Adder A simple pen-centric calculator, limited to addition and subtraction

Calculator A floating, button-operated calculator.

Clock A digital alarm clock accessory.

Notepaper App A notetaking application that uses the NotePaper DLL.

Paint A simple raster painting program.

Toolkit Demo Shows how to use many of the classes in the UI toolkit.

Input Application Demonstrates pen-based input event handling.

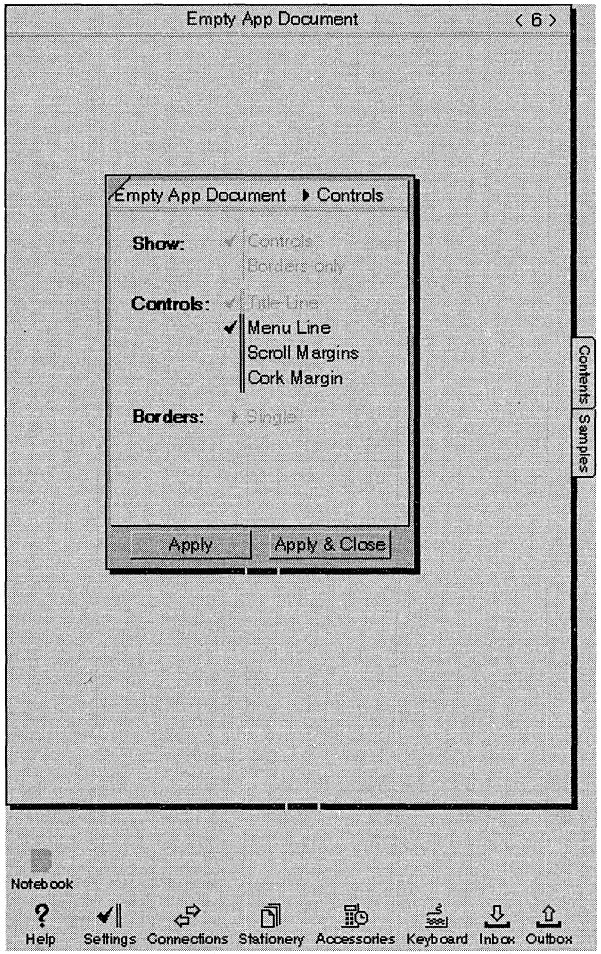
Writer Application Demonstrates handwriting translation.

Basic Service Contains the absolute minimum code required for a service.

Test Service Provides a starter kit for most service writers.

MIL Service Provides a starter kit for device driver writers.

Empty Application



Empty Application is the simplest sample application distributed with the PenPoint Software Developer's Kit. It does not have a view or any data. The only behavior it adds to the default PenPoint application is to print out a debugging message when the application is destroyed.

To provide this behavior, EmptyApp defines **clsEmptyApp**, which inherits from **clsApp**. In its handler for **msgDestroy**, **clsEmptyApp** prints out a simple debugging message.

clsEmptyApp inherits a rich set of default functionality from **clsApp**. When using EmptyApp, you can create, open, float, zoom, close, rename, embed, and destroy EmptyApp documents.

Objectives

EmptyApp is used in the Application Writing Guide to show how to compile, install, and run applications.

This sample application also shows how to:

- Use Debugf and #ifdef DEBUG/#endif pairs
- Turn on message tracing for a class
- Let the PenPoint Application Framework provide default behavior.

Class Overview

Empty Application defines one class: **clsEmptyApp**. It makes use of the following classes:

clsApp

clsAppMgr

Compiling

To compile EmptyApp, just

```
cd \penpoint\sdk\sample\emptyapp
wmake
```

This compiles the application and creates EMPTYAPP.EXE in \PENPOINT\APP\EMPTYAPP.

Running

After compiling EmptyApp, you can run it by

1. Adding \\boot\penpoint\app\Empty Application to \PENPOINT\BOOT\APP.INI
2. Booting PenPoint
3. Creating a new Empty Application document, and turning to it.

Alternatively, you can boot PenPoint and then install Empty Application through the Connections Notebook.

Files Used

The code for Empty Application is in \PENPOINT\SDK\SAMPLE\EMPTYAPP. The files are:

EMPTYAPP.C the application class's code and initialization.

METHODS.TBL the list of messages that the application class responds to, and the associated message handlers to call

METHODS.TBL

```
/*****************************************************************************
 File: methods.tbl
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.0  $
     $Date:   07 Jan 1992 16:37:36  $
 classes.tbl contains the method table for clsEmptyApp.
*****************************************************************************/
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif
MSG_INFO clsEmptyAppMethods [] = {
#ifdef DEBUG
     msgDestroy,          "EmptyAppDestroy",  objCallAncestorAfter,
#endif
     0
};
CLASS_INFO classInfo[] = {
     "clsEmptyAppTable",      clsEmptyAppMethods,      0,
     0
};
```

EMPTYAPP.C

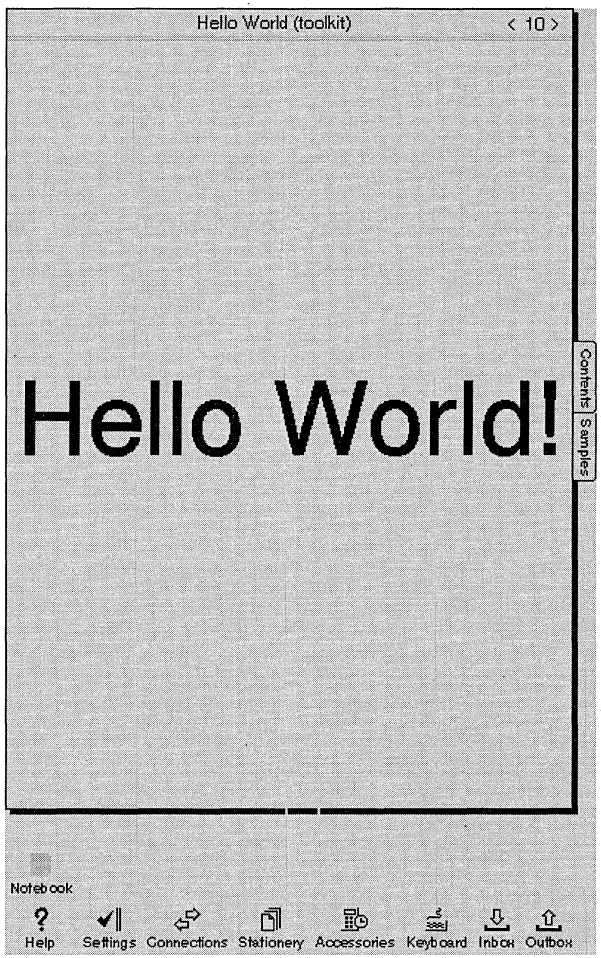
```
/*****************************************************************************
 File: emptyapp.c
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.2  $
     $Date:   07 Jan 1992 16:37:22  $
 This file contains just about the simplest possible application.
 It does not have a window.
 It does not have any state it needs to save.
 This class does respond to a single message, so it has a separate method
 table and a method to handle that message.
 All the method does is print out a debugging string.
 If you turn on the "F1" debugging flag (e.g. by putting
     DEBUGSET=/DF0001
 in \penpoint\boot\environ.ini), then messages to clsEmptyApp will be
 traced.
*****************************************************************************/
#ifndef APP_INCLUDED
#include <app.h>              // for application messages (and clsmgr.h)
#endif
#ifndef DEBUG_INCLUDED
#include <debug.h>                 // for debugging statements.
#endif
#ifndef APPMGR_INCLUDED
#include <appmgr.h>                // for AppMgr startup stuff
#endif
#include <methods.h>               // method function prototypes generated by MT
#include <string.h>                // for strcpy().

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Defines, Types, Globals, Etc     *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
STATUS EXPORTED EmptyAppInit (void);
#define  clsEmptyApp     wknGDTa
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Utility Routines                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Message Handlers                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*****************************************************************************
     EmptyAppDestroy
     Respond to msgDestroy by printing a simple message if in DEBUG mode.
*****************************************************************************/
MsgHandler(EmptyAppDestroy)
{
```

```
#ifdef DEBUG
     Debugf("EmptyApp: app instance %p about to die!", self);
#endif
     //
     // The Class Manager will pass the message onto the ancestor
     // if we return a non-error status value.
     //
     return stsOK;
     MsgHandlerParametersNoWarning;  // suppress compiler warnings
} /* EmptyAppDestroy */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Installation                                     *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*****************************************************************************
  ClsEmptyAppInit
  Install the EmptyApp application class as a well-known UID.
*****************************************************************************/
STATUS
ClsEmptyAppInit (void)
{
     APP_MGR_NEW new;
     STATUS      s;
     //
     // Install the Empty App class as a descendant of clsApp.
     //
     ObjCallRet(msgNewDefaults, clsAppMgr, &new, s);
     new.object.uid              = clsEmptyApp;
     new.object.key              = (OBJ_KEY)clsEmptyAppTable;
     new.cls.pMsg                = clsEmptyAppTable;
     new.cls.ancestor            = clsApp;
     //
     // This class has no instance data, so its size is zero.
     //
     new.cls.size                = Nil(SIZEOF);
     //
     // This class has no msgNew arguments of its own.
     //
     new.cls.newArgsSize         = SizeOf(APP_NEW);
     new.appMgr.flags.accessory  = true;
     strcpy(new.appMgr.company, "GO Corporation");
     strcpy(new.appMgr.defaultDocName, "Empty App Document");
     ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);
     //
     // Turn on message tracing if flag is set.
     //
     if (DbgFlagGet('F', 0x1L)) {
         Debugf("Turning on message tracing for clsEmptyApp");
         (void)ObjCallWarn(msgTrace, clsEmptyApp, (P_ARGS) true);
     }
     return stsOK;
Error:
     return s;
} /* ClsEmptyAppInit */
/*****************************************************************************
     main

     Main application entry point (as a PROCESS -- the app's MsgProc
         is where messages show up once an instance is running).
*****************************************************************************/
void CDECL
main (
     int         argc,
     char *      argv[],
     U16         processCount)
{
     Dbg(Debugf("main: starting emptyapp.exe[%d]", processCount);)
     if (processCount == 0) {

         // Create application class.
         ClsEmptyAppInit();
         // Invoke app monitor to install this application.
         AppMonitorMain(clsEmptyApp, objNull);
     } else {
         // Create an application instance and dispatch messages.
         AppMain();
     }
     // Suppress compiler's "unused parameter" warnings
     Unused(argc); Unused(argv);
} /* main */
```

MAKEFILE

```
##############################################
#
#  WMake Makefile for EmptyApp
#
# Copyright 1990-1992 GO Corporation. All Rights Reserved.
#
# You may use this Sample Code any way you please provided you
# do not resell the code and that this notice (including the above
# copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
# IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
# EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
# LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
# PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
# FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
# THE USE OR INABILITY TO USE THIS SAMPLE CODE.
#
# $Revision:   1.3  $
```

```
#     $Date:   07 Jan 1992 16:37:30  $
#
###############################################
# Set PENPOINT_PATH to your environment variable, if it exists.
# Otherwise, set it to \penpoint
!ifdef %PENPOINT_PATH
PENPOINT_PATH = $(%PENPOINT_PATH)
!else
PENPOINT_PATH = \penpoint
!endif
#  The DOS name of your project directory
PROJ = emptyapp
#  Standard defines for sample code
!INCLUDE $(PENPOINT_PATH)\sdk\sample\sdefines.mif
#  The PenPoint name of your application
EXE_NAME = Empty Application
#  The linker name for your executable : company-name-V<major>(<minor>)
EXE_LNAME= GO-EMPTYAPP_EXE-V1.(0)
#  Object files needed to build your app
EXE_OBJS = methods.obj emptyapp.obj
#  Libs needed to build your app
EXE_LIBS = penpoint app
# Targets
all: $(APP_DIR)\$(PROJ).exe   .SYMBOLIC
#  The clean rule must be :: because it is also defined in srules
clean :: .SYMBOLIC
 -del methods.h
#  Dependencies
emptyapp.obj: emptyapp.c methods.h
#  Standard rules for sample code
!INCLUDE $(PENPOINT_PATH)\sdk\sample\srules.mif
```

One of the simplest applications in any programming environment is one that prints the string "Hello World."

Because PenPoint provides both an API to the ImagePoint imaging model and a rich collection of classes built on top of ImagePoint, there are two different approaches to building a "Hello World" application. They are:

- Create a window and draw text in it using ImagePoint calls
- Use PenPoint's UI Toolkit classes to create a label object.

Each of these approaches is worth demonstrating in a sample application. The first is a good example for programs that need to do a lot of their own drawing, such as freeform graphics editors. The second approach shows how easy it is to use the toolkit classes, and serves as an example for programs that need to draw forms or other structured collections of information.

Therefore, there are two "Hello World" sample applications: Hello World (custom window) and Hello World (toolkit). The rest of this document describes Hello World (toolkit).

Hello World (toolkit) uses **clsLabel** to display "Hello World" in a window. The simplest way of doing this is to make a single label, which also serves as the window for the application. The code for doing so is in HELLOTK1.C. Since developers will typically want to display more than one toolkit class in a window, we created a second file, HELLOTK2.C. This file shows how to create a layout object (a window with knowledge of how to layout toolkit objects) and a label which is inserted into the layout object.

To change between these two source code files, simply copy the version you want to run to HELLOTK.C before compiling the application. (See "Compiling" below for more detailed instructions.)

Objectives

This sample application shows how to:

- Use **clsLabel**
- Create a custom layout window.

Class Overview

Hello World (toolkit) defines one class: **clsHelloWorld**. It makes use of the following classes:

clsApp
clsAppMgr
clsCustomLayout
clsLabel

Compiling

To compile Hello World (toolkit), just

```
cd \penpoint\sdk\sample\hellotk
```

Next, make the version of the application that you want to test:

```
copy HELLOTK1.C HELLOTK.C
wmake
```

or

```
copy HELLOTK2.C HELLOTK.C
wmake
```

This compiles the application and creates HELLOTK.EXE in \PENPOINT\APP\HELLOTK.

Running

After compiling Hello World (toolkit), you can run it by

1. Adding \\boot\penpoint\app\Hello World (toolkit) to \PENPOINT\BOOT\APP.INI
2. Booting PenPoint
3. Creating a new Hello World (toolkit) document, and turning to it.

Alternatively, you can boot PenPoint and then install Hello World (toolkit) through the Connections Notebook.

Testing

Zoom the document, or resize a floating document.

Files Used

The code for Hello World (toolkit) is in \PENPOINT\SDK\SAMPLE\HELLOTK. The files are:

HELLOTK.C source code (actually a copy of either hellotk1.c or hellotk2.c) which is compiled by the makefile

HELLOTK1.C source code for making a single label, which also serves as the window for the application

HELLOTK2.C source code for making a layout object and inserting a label in it

METHODS.TBL the method table for **clsHelloWorld**.

METHODS.TBL

```
/************************************************************************
 File: methods.tbl
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.0  $
     $Date:   07 Jan 1992 16:47:30  $
 Methods.tbl contains the method table for clsHelloWorld (toolkit version).
************************************************************************/
//
//    Include files
//
#include <app.h>
MSG_INFO clsHelloMethods [] = {
     msgAppInit,                "HelloAppInit", objCallAncestorBefore,
     msgAppOpen,                "HelloOpen",    objCallAncestorAfter,
     msgAppClose,               "HelloClose",   objCallAncestorBefore,

     0
};
CLASS_INFO classInfo[] = {
     "clsHelloTable",        clsHelloMethods,        0,
     0
};
```

HELLOTK1.C

```
/************************************************************************
 File: hellotk1.c
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.5  $  (from hello.c 1.1)
     $Date:   08 Jan 1992 11:41:14  $
 This file contains the application class for a "Hello World" application
 using toolkit components.
 This uses the PenPoint UI Toolkit to draw in its window -- thus it
 does not create a window class.
 It creates a label as its client window in response to msgAppInit.
 It has dummy message handlers for msgAppClose and msgAppOpen so it can
 share the same methods.tbl with hellotk2.c.
 It does not have any state it needs to save.
 It does not have any instance data.
 Most applications have more than one window in their frame.  hellotk2.c
 is an alternative version of hellotk.c which creates a label inside a
 custom layout window.
 If you turn on the "F20" debugging flag (e.g. by putting
     DEBUGSET=/F0020
 in \penpoint\boot\environ.ini), then messages to clsHelloWorld will be
 traced.
************************************************************************/
#ifndef DEBUG_INCLUDED
#include <debug.h>                // for debugging statements.
#endif
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>               // for O-OP support.
#endif
#ifndef APP_INCLUDED
#include <app.h>              // for application messages (and clsmgr.h)
#endif
#ifndef APPMGR_INCLUDED
#include <appmgr.h>               // for AppMgr startup stuff
#endif
#ifndef _STRING_H_INCLUDED
#include <string.h>               // for strcpy().
#endif
#ifndef LABEL_INCLUDED
#include <label.h>                // for label.
#endif
#ifndef FRAME_INCLUDED
#include <frame.h>                // for frame metrics.
#endif
#include <methods.h>          // method function prototypes generated by MT
#define clsHelloWorldwknGDTb // avoids clashing with other HelloWorlds
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                              Methods                                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/************************************************************************
     HelloAppInit

     Respond to msgAppInit by creating the client window (a label).
************************************************************************/
```

```
MsgHandler(HelloAppInit)
{
    APP_METRICS             am;
    LABEL_NEW               ln;
    STATUS                  s;

    Dbg(Debugf("HelloTK: Create the client Win");)

    // Create the Hello label window.
    ObjCallWarn(msgNewDefaults, clsLabel, &ln);
    ln.label.style.scaleUnits   = bsUnitsFitWindowProper;
    ln.label.style.xAlignment   = lsAlignCenter;
    ln.label.style.yAlignment   = lsAlignCenter;
    ln.label.pString            = "Hello World!";
    ObjCallRet(msgNew, clsLabel, &ln, s);

    // Get the app's main window (its frame).
    ObjCallJmp(msgAppGetMetrics, self, &am, s, error);

    // Insert the label in the frame as its client window.
    ObjCallJmp(msgFrameSetClientWin, am.mainWin, \
                        (P_ARGS)ln.object.uid, s, error);
    return stsOK;
    MsgHandlerParametersNoWarning;
error:
    ObjCallWarn(msgDestroy, ln.object.uid, Nil(OBJ_KEY));
    return s;
} /* HelloAppInit */
/******************************************************************************
    HelloOpen

    Respond to msgAppOpen by creating UI objects that aren't filed.
    But I create my user interface in msgAppInit, so it's filed and
    restored for me, so do nothing.
******************************************************************************/
MsgHandler(HelloOpen)
{
    Dbg(Debugf("HelloTK: msgAppOpen");)

    // When the message gets to clsApp the app will go on-screen.
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* HelloOpen */

/******************************************************************************
    HelloClose

    Respond to msgAppClose by destroying UI objects that aren't filed.
    But I create my user interface in msgAppInit, so it's filed and
    restored for me, so do nothing.
******************************************************************************/
MsgHandler(HelloClose)
{
    Dbg(Debugf("HelloTK: msgAppClose");)

    // When the message gets to its ancestor the frame will be taken
    // off-screen.
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* HelloClose */

/******************************************************************************
    ClsHelloInit

    Install the Hello application.
******************************************************************************/
STATUS ClsHelloInit (void)
{
    APP_MGR_NEW new;
    STATUS      s;

    // Install the class.
    ObjCallWarn(msgNewDefaults, clsAppMgr, &new);
    new.object.uid              = clsHelloWorld;
    new.object.key              = (OBJ_KEY)clsHelloTable;
    new.cls.pMsg                = clsHelloTable;
    new.cls.ancestor            = clsApp;
    // This class has no instance data, so its size is zero.
    new.cls.size                = Nil(SIZEOF);
    // This class has no msgNew arguments of its own.
    new.cls.newArgsSize         = SizeOf(APP_NEW);
    new.appMgr.flags.stationery     = true;
    new.appMgr.flags.accessory      = true;
    new.appMgr.flags.allowEmbedding = false;
    new.appMgr.flags.hotMode        = false;
    strcpy(new.appMgr.company, "GO Corporation");
    ObjCallRet(msgNew, clsAppMgr, &new, s);
    if (DbgFlagGet('F', 0x20L)) {
        Debugf("Turning on message tracing for clsHelloWorld (toolkit)");
        (void)ObjCallWarn(msgTrace, clsHelloWorld, (P_ARGS) true);
    }
    return stsOK;
} /* ClsHelloInit */
/******************************************************************************
    main

    Main application entry point.
******************************************************************************/
void CDECL main (
    int         argc,
    char *      argv[],
    U16         processCount)
{
    Dbg(Debugf("main: starting HelloTK1.exe[%d]", processCount);)
    if (processCount == 0) {
        // Initialize self.
```

```
        ClsHelloInit();
        // Invoke app monitor to install this application.
        AppMonitorMain(clsHelloWorld, objNull);
    } else {
        // Start the application.
        AppMain();
    }
    Unused(argc);  Unused(argv);    // Suppress compiler warnings
} /* main */
```

HELLOTK2.C

```
/*****************************************************************************
 File: hellotk2.c

 Copyright 1990-1992 GO Corporation. All Rights Reserved.

 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.

 $Revision:   1.6  $  (from hello.c 1.1)
     $Date:   08 Jan 1992 11:41:18  $

 This file contains the application class for a "Hello World" application
 using toolkit components.
 This uses the PenPoint UI Toolkit to draw in its window -- thus it
 does not create a window class.

 Instead, it creates a custom layout window in its frame and inserts a
 label within the layout window.

 Other versions of hellotk.c don't use custom layout, or
 create several toolkit windows.

 It does not have any state it needs to save.
 It does not have any instance data.

 If you turn on the "F20" debugging flag (e.g. by putting
     DEBUGSET=/F0020
 in \penpoint\boot\environ.ini), then messages to clsHelloWorld will be
 traced.
 If you turn on the "F40" debugging flag then the custom layout window
 will be visible (gray background, rounded border).
*****************************************************************************/
#ifndef DEBUG_INCLUDED
#include <debug.h>                  // for debugging statements.
#endif
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>                 // for O-OP support.
#endif
#ifndef APP_INCLUDED
#include <app.h>                    // for application messages
#endif
#ifndef APPMGR_INCLUDED
#include <appmgr.h>                 // for AppMgr startup stuff
#endif
#ifndef _STRING_H_INCLUDED
#include <string.h>                 // for strcpy().
#endif
#ifndef LABEL_INCLUDED
#include <label.h>                  // for label.
#endif
#ifndef FRAME_INCLUDED
#include <frame.h>                  // for frame metrics (and clayout.h)
#endif
#include <methods.h>                // method function prototypes generated by MT
#define clsHelloWorld wknGDTc // avoids clashing with other HelloWorlds
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                                 Methods                                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*****************************************************************************
     HelloAppInit

     Respond to msgAppInit by creating long-lived objects (filed state).
     But I create and destroy my user interface in msgAppOpen/msgAppClose,
     so do nothing.
*****************************************************************************/
MsgHandler(HelloAppInit)
{
     Dbg(Debugf("HelloTK: msgAppInit");)
     // When the message gets to clsApp the frame will be created.
     return stsOK;
     MsgHandlerParametersNoWarning;
} /* HelloAppInit */
/*****************************************************************************
     HelloOpen

     Respond to msgAppOpen by creating the U.I.:
         o   a custom layout window
         o   and a label within it.
*****************************************************************************/
MsgHandler(HelloOpen)
{
     APP_METRICS             am;
     WIN_METRICS             wm;
     CSTM_LAYOUT_NEW         cn;
     CSTM_LAYOUT_CHILD_SPEC  cs;
     LABEL_NEW               ln;
     STATUS                  s;
     Dbg(Debugf("HelloTK: Create the client Win");)
```

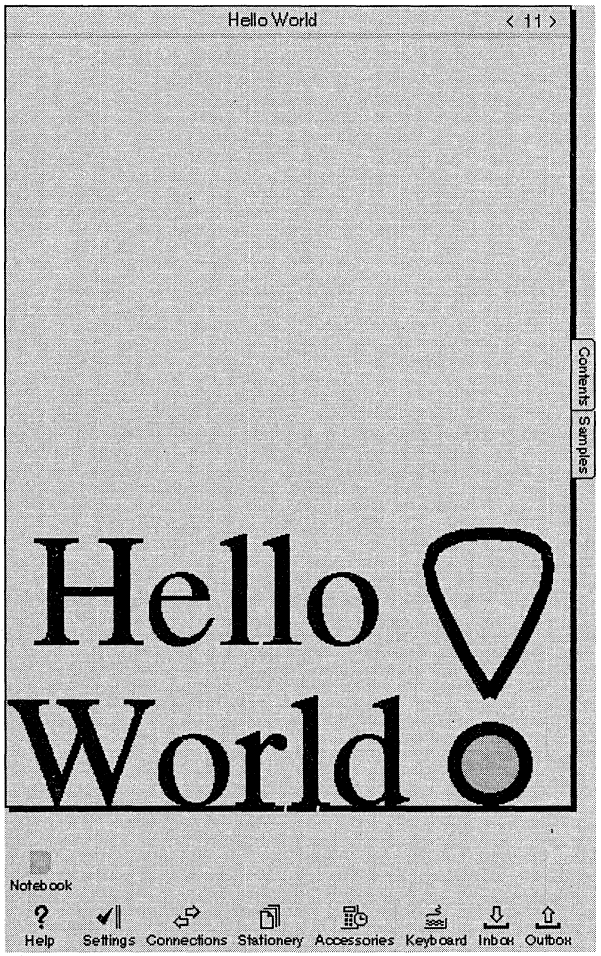
```
    ObjCallWarn(msgNewDefaults, clsCustomLayout, &cn);
    // ?? Needed??  cn.win.parent = frame;
    // If the frame is floating, this will make it wrap neatly
    // around the label.
    cn.border.style.leftMargin = cn.border.style.rightMargin = bsMarginSmall;
    cn.win.flags.style |= wsShrinkWrapHeight;

    if (DbgFlagGet('F', 0x40L)) {
        cn.border.style.join                = bsJoinRound;
        cn.border.style.edge                = bsEdgeAll;
        cn.border.style.backgroundInk   = bsInkGray33;
    }
    ObjCallRet(msgNew, clsCustomLayout, &cn, s);
    // Create the Hello label window.
    ObjCallWarn(msgNewDefaults, clsLabel, &ln);
    ln.label.pString = "Hello World!";
    ObjCallJmp(msgNew, clsLabel, &ln, s, error1);
    // Insert the Hello win in the custom layout window.
    wm.parent = cn.object.uid;
    ObjCallJmp(msgWinInsert, ln.object.uid, &wm, s, error2);
    // Specify how the custom layout window should position the label.
    CstmLayoutSpecInit(&(cs.metrics));
    cs.child = ln.object.uid;
    cs.metrics.x.constraint = ClAlign(clCenterEdge, clSameAs, clCenterEdge);
    cs.metrics.y.constraint = ClAlign(clCenterEdge, clSameAs, clCenterEdge);
    cs.metrics.w.constraint = clAsIs;
    cs.metrics.h.constraint = clAsIs;
    ObjCallJmp(msgCstmLayoutSetChildSpec, cn.object.uid, &cs, s, error2);

    // Get the app's main window (its frame).
    ObjCallJmp(msgAppGetMetrics, self, &am, s, error2);
    // Insert the custom layout window in the frame.
    ObjCallJmp(msgFrameSetClientWin, am.mainWin, \
                        (P_ARGS)cn.object.uid, s, error2);
    // When the message gets to its ancestor this will all go on-screen.
    return stsOK;
    MsgHandlerParametersNoWarning;

error2:
    ObjCallWarn(msgDestroy, ln.object.uid, Nil(OBJ_KEY));
error1:
    ObjCallWarn(msgDestroy, cn.object.uid, Nil(OBJ_KEY));
    return s;
} /* HelloOpen */
/*****************************************************************************
    HelloClose

    Respond to msgAppClose by destroying the client window.
    The ancestor has already taken us off-screen.
*****************************************************************************/
MsgHandler(HelloClose)
{
    APP_METRICS     am;
    WIN             win;
    OBJ_KEY         key = objWKNKey;
    STATUS          s;
    // Get the client window.
    ObjCallRet(msgAppGetMetrics, self, &am, s);
    ObjCallRet(msgFrameGetClientWin, am.mainWin, (P_ARGS)&win, s);
    // Destroy it.
    ObjCallRet(msgDestroy, win, &key, s);
    Dbg(Debugf("HelloTK: back from freeing client Win");)
    // Tell the app that it no longer has a client window.
    ObjCallRet(msgFrameSetClientWin, am.mainWin, (P_ARGS)objNull, s);
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* HelloClose */

/*****************************************************************************
    ClsHelloInit

    Install the Hello application.
*****************************************************************************/
STATUS ClsHelloInit (void)
{
    APP_MGR_NEW new;
    STATUS      s;

    // Install the class.
    ObjCallWarn(msgNewDefaults, clsAppMgr, &new);
    new.object.uid              = clsHelloWorld;
    new.object.key              = (OBJ_KEY)clsHelloTable;
    new.cls.pMsg                = clsHelloTable;
    new.cls.ancestor            = clsApp;
    // This class has no instance data, so its size is zero.
    new.cls.size                = Nil(SIZEOF);
    // This class has no msgNew arguments of its own.
    new.cls.newArgsSize         = SizeOf(APP_NEW);
    new.appMgr.flags.stationery     = true;
    new.appMgr.flags.accessory      = true;
    new.appMgr.flags.allowEmbedding = false;
    new.appMgr.flags.hotMode        = false;
    strcpy(new.appMgr.company, "GO Corporation");
    ObjCallRet(msgNew, clsAppMgr, &new, s);
    if (DbgFlagGet('F', 0x20L)) {
        Debugf("Turning on message tracing for clsHelloWorld (toolkit)");
        (void)ObjCallWarn(msgTrace, clsHelloWorld, (P_ARGS) true);
    }
```

```
    return stsOK;
} /* ClsHelloInit */
/*****************************************************************************
    main

    Main application entry point.
*****************************************************************************/
void CDECL main (
    int         argc,
    char *      argv[],
    U16         processCount)
{
    Dbg(Debugf("main: starting HelloTK2.exe[%d]", processCount);)
    if (processCount == 0) {
        // Initialize self.
        ClsHelloInit();
        // Invoke app monitor to install this application.
        AppMonitorMain(clsHelloWorld, objNull);
    } else {
        // Start the application.
        AppMain();
    }
    Unused(argc);  Unused(argv);    // Suppress compiler warnings
} /* main */
```

MAKEFILE

```
###############################################
#
#  WMake Makefile for HelloTK
#
# Copyright 1990-1992 GO Corporation. All Rights Reserved.
#
# You may use this Sample Code any way you please provided you
# do not resell the code and that this notice (including the above
# copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
# IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
# EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
# LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
# PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
# FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
# THE USE OR INABILITY TO USE THIS SAMPLE CODE.
#
# $Revision:   1.6  $
#     $Date:   07 Jan 1992 16:47:16  $
#
###############################################
# Set PENPOINT_PATH to your environment variable, if it exists.
# Otherwise, set it to \penpoint
!ifdef %PENPOINT_PATH
PENPOINT_PATH = $(%PENPOINT_PATH)
!else
PENPOINT_PATH = \penpoint
!endif
#  The DOS name of your project directory.
PROJ = hellotk
#  Standard defines for sample code (needs the PROJ) definition
!INCLUDE $(PENPOINT_PATH)\sdk\sample\sdefines.mif
#  The PenPoint name of your application
EXE_NAME = Hello World (toolkit)
#  The linker name for your executable : company-name-V<major>(<minor>)
EXE_LNAME= GO-HELLOTK-V1(0)
#  Object files needed to build your app
EXE_OBJS = methods.obj hellotk.obj
#  Libs needed to build your app
EXE_LIBS = penpoint app
# Targets
all: $(APP_DIR)\$(PROJ).exe .SYMBOLIC
#  The clean rule must be :: because it is also defined in srules
clean:: .SYMBOLIC
 -del methods.h
 -del methods.tc
#  Dependencies
hellotk.obj: hellotk.c methods.h
#  Standard rules for sample code
!INCLUDE $(PENPOINT_PATH)\sdk\sample\srules.mif
```

Hello World (Custom Window)



Hello World (custom window) demonstrates how to draw the string "Hello World" by directly using ImagePoint calls. To do so, it defines a descendant of **clsWin**.

In its **msgWinRepaint** handler, the window determines the size of the string "Hello World" and then calls **msgDcDrawText** to actually paint the text. It also paints a large exclamation point after it, using ImagePoint's ability to draw bezier curves.

For demonstration purposes, this application's window is compiled as a separate DLL.

Objectives

This sample application shows how to:

- Create a window, and a drawing context (DC) to draw on
- Draw text and bezier curves
- Separate out part of an application into a reuseable dynamic link library

Class Overview

Hello World (custom window) defines two classes: **clsHelloWorld** and **clsHelloWin**. It makes use of the following classes:

clsApp

clsAppMgr

clsClass

clsSysDrwCtx

clsWin

Compiling

To compile Hello World (custom window), just

```
cd \penpoint\sdk\sample\hello
wmake
```

This compiles the dynamic link library and the application, and creates HELLO.DLL and HELLO.EXE in \PENPOINT\APP\HELLO. It also copies HELLO.DLC to \PENPOINT\APP\HELLO.

Running

After compiling Hello World (custom window), you can run it by

1. Adding \\boot\penpoint\app\Hello World to \PENPOINT\BOOT\APP.INI
2. Booting PenPoint
3. Creating a new Hello World document, and turning to it.

Alternatively, you can boot PenPoint and then install Hello World through the Connections Notebook.

Testing

Zoom the document, or resize a floating document.

Files Used

The code for Hello World (custom window) is in \PENPOINT\SDK\SAMPLE\HELLO. The files are:

DLL.LBC list of exported functions for the Watcom linker

DLLINIT.C the routine to initialize the dll

HELLO.C the source code for the application

HELLO.DLC indicates the dependency of the application upon the window dll

HELLOWIN.C the source code for the window class

HELLOWIN.H the header file for the window class

HELTBL.TBL the method table for the application class

HELWTBL.TBl the method table for the window class.

HELTBL.TBL

```
/*****************************************************************************
 File: heltbl.tbl
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.2  $
     $Date:   13 Nov -1991 18:04:54  $
 heltbl.tbl contains the method table for clsHelloWorld.
*****************************************************************************/

//
//    Include files
//
#include <clsmgr.h>
#include <app.h>
MSG_INFO clsHelloWorldMethods [] = {
     msgAppOpen,                "HelloOpen",        objCallAncestorAfter,
     msgAppClose,               "HelloClose",       objCallAncestorBefore,
     0
};
CLASS_INFO classInfo[] = {
     "clsHelloWorldTable",      clsHelloWorldMethods,   0,
     0
};
```

HELWTBL.TBL

```
/*****************************************************************************
 File: helwtbl.tbl
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.2  $
     $Date:   13 Nov 1991 18:05:00  $
 helwtbl.tbl contains the method table for clsHelloWin.
*****************************************************************************/

//
//    Include files
//
#include <clsmgr.h>
#include <win.h>
MSG_INFO clsHelloWinMethods [] = {
     msgInit,                   "HelloWinInit",     objCallAncestorBefore,
     msgFree,                   "HelloWinFree",     objCallAncestorAfter,
     msgWinRepaint,             "HelloWinRepaint",  0,
     0
};
CLASS_INFO classInfo[] = {
     "clsHelloWinTable",        clsHelloWinMethods,     0,
     0
};
```

HELLO.C

```
/*****************************************************************************
 File: hello.c
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
```

```
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.4  $
     $Date:   13 Nov 1991 18:04:56  $
 This file contains the application class for a simple "Hello World"
 application.
 It creates an instance of clsHelloWin and inserts it in its frame.
 It does not have any state it needs to save.
 It does not have any instance data.
 If you turn on the "F10" debugging flag (e.g. by putting
     DEBUGSET=/DF0010
 in \penpoint\boot\environ.ini), then messages to clsHelloWorld will be
 traced.
*****************************************************************************/
#ifndef DEBUG_INCLUDED
#include <debug.h>                  // for debugging statements.
#endif
#ifndef APP_INCLUDED
#include <app.h>                 // for application messages.
#endif
#ifndef APPMGR_INCLUDED
#include <appmgr.h>                 // for AppMgr startup stuff
#endif
#ifndef FRAME_INCLUDED
#include <frame.h>                  // for frame metrics.
#endif
#include <string.h>                 // for strcpy().
#include <heltbl.h>                 // method definitions
#include <hellowin.h>            // clsHelloWin's UID and msgNew args.
#define clsHelloWorld              MakeWKN(2164,1,wknGlobal)
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Methods                                        *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*****************************************************************************
     HelloOpen

     Respond to msgAppOpen by creating a clsHelloWin instance and inserting
     it as the frame's client window.
*****************************************************************************/
MsgHandler(HelloOpen)
{
     HELLO_WIN_NEW   hwn;
     APP_METRICS     am;
     STATUS          s;

     // Get the app's main window (its frame).
     ObjCallRet(msgAppGetMetrics, self, &am, s);
     // Create the Hello window.
     ObjCallWarn(msgNewDefaults, clsHelloWin, &hwn);
     ObjCallRet(msgNew, clsHelloWin, &hwn, s);
     // Insert the Hello win in the frame.
     ObjCallJmp(msgFrameSetClientWin, am.mainWin, (P_ARGS)hwn.object.uid,
               s, exit);
     // Ancestor will put it all on the screen.
     return stsOK;
     MsgHandlerParametersNoWarning;  // suppress compiler warnings about unused
parameters
exit:
     ObjCallWarn(msgDestroy, hwn.object.uid, pNull);
     return s;
} /* HelloOpen */
/*****************************************************************************
     HelloClose

     Respond to msgAppClose by destroying the client window.
*****************************************************************************/
MsgHandler(HelloClose)
{
     APP_METRICS     am;
     WIN             clientWin;
     STATUS          s;
     // Ancestor has taken the main window (frame) off the screen.
     // Get the client window.
     ObjCallRet(msgAppGetMetrics, self, &am, s);
     ObjCallRet(msgFrameGetClientWin, am.mainWin, &clientWin, s);
     // Destroy it.
     ObjCallRet(msgDestroy, clientWin, objWKNKey, s);
     Dbg(Debugf("Hello: back from freeing HelloWin");)
     // Update the frame since the client window is gone.
     ObjCallRet(msgFrameSetClientWin, am.mainWin, (P_ARGS)objNull, s);
     return stsOK;
     MsgHandlerParametersNoWarning;  // suppress compiler warnings about unused
parameters
} /* HelloClose */

/*****************************************************************************
     ClsHelloInit

     Install the Hello application.
*****************************************************************************/
STATUS ClsHelloInit (void)
{
     APP_MGR_NEW new;
     STATUS      s;

     // Install the application class.
```

```
    ObjCallWarn(msgNewDefaults, clsAppMgr, &new);
    new.object.uid              = clsHelloWorld;
    new.object.key              = (OBJ_KEY)clsHelloWorldTable;
    new.cls.pMsg                = clsHelloWorldTable;
    new.cls.ancestor            = clsApp;
    // This class has no instance data, so its size is zero.
    new.cls.size                = Nil(SIZEOF);
    // This class has no msgNew arguments of its own.
    new.cls.newArgsSize         = SizeOf(APP_NEW);
    new.appMgr.flags.stationery     = true;
    new.appMgr.flags.accessory      = true;
    strcpy(new.appMgr.company, "GO Corporation");
    ObjCallRet(msgNew, clsAppMgr, &new, s);
    if (DbgFlagGet('F', 0x10L)) {
        Debugf("Turning on message tracing for clsHelloWorld");
        (void)ObjCallWarn(msgTrace, clsHelloWorld, (P_ARGS) true);
    }
    return stsOK;
} /* ClsHelloInit */
/*****************************************************************************
    main

    Main application entry point.
*****************************************************************************/
void CDECL main (
    int         argc,
    char *      argv[],
    U16         processCount)
{
    Dbg(Debugf("main: starting Hello.exe[%d]", processCount);)
    if (processCount == 0) {
        //
        // Initialize self.
        //
        // Note that the loader calls DLLMain in the Hello World DLL,
        // which creates clsHelloWin.
        //
        ClsHelloInit();
        // Invoke app monitor to install this application.
        AppMonitorMain(clsHelloWorld, objNull);
    } else {
        // Start the application.
        AppMain();
    }
    Unused(argc);  Unused(argv);    // Suppress compiler warnings
} /* main */
```

HELLOWIN.H

```
/*****************************************************************************
 File: hellowin.h
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.3  $
    $Date:   13 Nov 1991 18:04:58  $
 This file contains the API definition for clsHelloWin.
 clsHelloWin inherits from clsWin.
 It has no messages or msgNew arguments.
*****************************************************************************/
#ifndef HELLO_WIN_INCLUDED
#define HELLO_WIN_INCLUDED
#include <clsmgr.h>
#include <win.h>
#define clsHelloWin             MakeWKN(2165,1,wknGlobal)
/*****************************************************************************
 msgNew              takes P_HELLO_WIN_NEW, returns STATUS
     category: class message
     Creates a new Hello Window.
*/
#define helloWinNewFields \
     winNewFields

typedef struct {
     helloWinNewFields
} HELLO_WIN_NEW, *P_HELLO_WIN_NEW;
/*****************************************************************************
 msgNewDefaults      takes P_HELLO_WIN_NEW, returns STATUS
     category: class message
     Initializes HELLO_WIN_NEW structure to default values.
*/
#endif
```

HELLOWIN.C

```
/*****************************************************************************
 File: hellowin.c
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
```

```
IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
THE USE OR INABILITY TO USE THIS SAMPLE CODE.

$Revision:   1.5  $
    $Date:   13 Nov 1991 18:05:04  $

This file contains a simple "Hello World" window subclass.

It creates a drawing context to paint a welcome message in self.
Since clsHelloWin doesn't use the DC anywhere else but msgWinRepaint,
it could create it on the fly during msgWinRepaint processing, but
instead clsHelloWin saves the DC in its instance data.
Since clsHelloWorld frees the hello window upon receiving msgAppClose,
the DC doesn't take up space when the application is "closed down."

The repainting routine jumps through some geometry/drawing context hoops
to ensure that the drawing fits in the window yet remains proportionately
sized.

If you turn on the "F40" debugging flag (e.g. by putting
    DEBUGSET=/DF0040
in \penpoint\boot\environ.ini), then drawing takes places with thick lines
so that drawing operations are more visible.
If you turn on the "F20" debugging flag, messages to clsHelloWin will be
traced.
*****************************************************************************/
#include <debug.h>
#include <win.h>
#include <sysgraf.h>
#include <sysfont.h>
#include <string.h>              // for memset().
#include <gomath.h>              // for scale calc. in fixed point.
#include <helwtbl.h>          // method definitions
#include <hellowin.h>         // clsHelloWin's UID and msgNew args.

typedef struct INSTANCE_DATA {
    SYSDC           dc;
} INSTANCE_DATA, *P_INSTANCE_DATA;

// Scale font to 100 units to begin with.
#define initFontScale 100
// Line thickness a twelfth of the font scale.
#define lineThickness 8

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                              Methods                                  *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*****************************************************************************
    HelloWinInit

    Create a new window object.
*****************************************************************************/
MsgHandler(HelloWinInit)
{
    SYSDC_NEW           dn;
    INSTANCE_DATA       data;
    SYSDC_FONT_SPEC     fs;
    SCALE               fontScale;
    STATUS              s;

    // Null the instance data.
    memset(&data, 0, sizeof(INSTANCE_DATA));
    // Create a dc.
    ObjCallWarn(msgNewDefaults, clsSysDrwCtx, &dn);
    ObjCallRet(msgNew, clsSysDrwCtx, &dn, s);
    data.dc = dn.object.uid;
    // Rounded lines, thickness of zero.
    ObjectCall(msgDcSetLineThickness, data.dc, (P_ARGS)0);
    if (DbgFlagGet('F', 0x40L)) {
        Debugf("Use a non-zero line thickness.");
        ObjectCall(msgDcSetLineThickness, data.dc, (P_ARGS)2);
    }
    // Open a font.  Use the "user input" font (whatever the user has
    // chosen for this in System Preferences.
    fs.id               = 0;
    fs.attr.group       = sysDcGroupUserInput;
    fs.attr.weight      = sysDcWeightNormal;
    fs.attr.aspect      = sysDcAspectNormal;
    fs.attr.italic      = 0;
    fs.attr.monospaced  = 0;
    fs.attr.encoding    = sysDcEncodeGoSystem;
    ObjCallJmp(msgDcOpenFont, data.dc, &fs, s, Error);
    //
    // Scale the font.  The entire DC will be scaled in the repaint
    // to pleasingly fill the window.
    fontScale.x = fontScale.y = FxMakeFixed(initFontScale,0);
    ObjectCall(msgDcScaleFont, data.dc, &fontScale);
    // Bind the window to the dc.
    ObjectCall(msgDcSetWindow, data.dc, (P_ARGS)self);
    // Update the instance data.
    ObjectWrite(self, ctx, &data);
    return stsOK;
    MsgHandlerParametersNoWarning;  // suppress compiler warnings about unused
parameters
Error:
    ObjCallWarn(msgDestroy, data.dc, Nil(OBJ_KEY));
    return s;
}  /* HelloWinInit */
```

```
/*****************************************************************************
    HelloWinFree

    Free self.
*****************************************************************************/
MsgHandlerWithTypes(HelloWinFree, P_ARGS, P_INSTANCE_DATA)
{
    // Destroy the dc.  (Assumes that this will not fail.)
    // Note that pData is now invalid.
    ObjCallWarn(msgDestroy, pData->dc, Nil(P_ARGS));
    // Ancestor will eventually free self.
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* HelloWinFree */

/*****************************************************************************
    HelloWinRepaint

    Repaint the window.  This is the only paint routine needed; clsHelloWin
    relies on the window system to tell it when it needs (re)painting.
*****************************************************************************/
MsgHandlerWithTypes(HelloWinRepaint, P_ARGS, P_INSTANCE_DATA)
{
    SYSDC_TEXT_OUTPUT   tx;
    S32                 textWidth;
    S32                 helloAdjust, worldAdjust;
    SYSDC_FONT_METRICS  fm;
    SIZE32              drawingSize;
    WIN_METRICS         wm;
    FIXED               drawingAspect, winAspect;
    SCALE               scale;
    RECT32              dotRect;
    XY32                bezier[4];
    STATUS              s;
    //
    // Determine size of drawing in 100 units to a point coord. system.
    // The words "Hello" and "World" have no descenders (in most fonts!!).
    // Height is font height (initFontScale) * 2 - the descender size.
    // Width is max of the two text widths plus em.width (width of
    // the exclamation point.
    //
    // Figure out the widths of the two text strings.

    // Init tx.
    memset(&tx, 0, sizeof(SYSDC_TEXT_OUTPUT));
    tx.underline    = 0;
    tx.alignChr     = sysDcAlignChrBaseline;
    tx.stop         = maxS32;
    tx.spaceChar    = 32;
    // Set the overall text width to whichever text string is wider.
    tx.cp.x         = 0;
    tx.cp.y         = 0;
    tx.pText        = "World";
    tx.lenText      = strlen(tx.pText);
    ObjectCall(msgDcMeasureText, pData->dc, &tx);
    textWidth = tx.cp.x;
    tx.cp.x         = 0;
    tx.cp.y         = 0;
    tx.pText        = "Hello";
    tx.lenText      = strlen(tx.pText);
    ObjectCall(msgDcMeasureText, pData->dc, &tx);
    if (tx.cp.x > textWidth) {
        // "Hello" is wider
        helloAdjust = 0;
        worldAdjust = (tx.cp.x - textWidth) / 2;
        textWidth = tx.cp.x;
    } else {
        // "World" was wider
        worldAdjust = 0;
        helloAdjust = (textWidth - tx.cp.x) / 2;
    }
    // Get font metrics.
    ObjectCall(msgDcGetFontMetrics, pData->dc, &fm);
    drawingSize.w = textWidth + fm.em.w;
    // Remember, descenderPos is negative.
    drawingSize.h = (2 * initFontScale) + fm.descenderPos;
    //
    // Must bracket all repainting with msgWinBegin/EndRepaint.
    // The window system figures out which part of the window needs
    // repainting, and restricts all painting operations to that update
    // area.
    //
    ObjCallRet(msgWinBeginRepaint, pData->dc, pNull, s);
    // Fill the background with white to start.
    ObjectCall(msgDcFillWindow, pData->dc, pNull);
    //
    // We have determined the size of the drawing in points.
    // But if the window is much smaller than this the drawing will
    // be cropped.  So, we must scale it to fit the window.
    // You can scale a DC to match the width and height of a window using
    // dcUnitsWorld, but then the text would be stretched strangely.
    //
    // Instead, we'll compute a consistent scaling factor for the drawing.
    //
    //
    // We need to first determine the size of the window.
    // We send the message to the DC to get the size in DC units.
    //
    ObjCallJmp(msgWinGetMetrics, pData->dc, &wm, s, exit);
```

```
// Now decide whether to scale by the x or y coordinate.
// Have to hassle with Fixed Point!
drawingAspect = FxDivIntsSC(drawingSize.h, drawingSize.w);
winAspect = FxDivIntsSC(wm.bounds.size.h, wm.bounds.size.w);
if (winAspect > drawingAspect ) {
    //
    // The window is "taller" than the drawing.  Scale so the
    // drawing fills the window horizontally.
    //
    Dbg(Debugf("Window is taller than drawing!  Still must calculate
        vertical offset!");)
    scale.x = scale.y = FxDivIntsSC(wm.bounds.size.w, drawingSize.w);
} else {
    //
    // The window is "wider" than the drawing.  Scale so the
    // drawing fills the window vertically.
    //
    Dbg(Debugf("Window is wider than drawing!  Still must calculate
        horizontal offset!");)
    scale.x = scale.y = FxDivIntsSC(wm.bounds.size.h, drawingSize.h);
}
ObjectCall(msgDcScale, pData->dc, &scale);


//
// At this point a more sophisticated program would figure out
// which parts need redrawing based on the boundaries of the
// dirty area.
//
// Display the text.
// Display "Hello". tx was set to do this from before, but need to
// reset tx.lenText because msgDcMeasureText passes back in it the
// offset of the last character that would be drawn in it.
tx.cp.x         = helloAdjust;
tx.cp.y         = initFontScale;
tx.lenText      = strlen(tx.pText);
ObjectCall(msgDcDrawText, pData->dc, &tx);
// Display "World".
tx.cp.x         = worldAdjust;
tx.cp.y         = 0;
tx.pText        = "World";
tx.lenText      = strlen(tx.pText);
ObjectCall(msgDcDrawText, pData->dc, &tx);
// Paint the exclamation point.
ObjectCall(msgDcSetForegroundRGB, pData->dc, (P_ARGS)sysDcRGBGray66);
// Want Foreground color of Gray for edges of Exclamation Point.
ObjectCall(msgDcSetBackgroundRGB, pData->dc, (P_ARGS)sysDcRGBGray33);
ObjectCall(msgDcSetLineThickness, pData->dc, (P_ARGS)lineThickness);
// Paint the teardrop.
// First the left half...
    bezier[0].x = textWidth + (fm.em.w / 2);
    bezier[0].y = fm.ascenderPos;
    bezier[1].x = textWidth;
    bezier[1].y = initFontScale * 3 / 2;
    bezier[2].x = bezier[1].x;
    bezier[2].y = initFontScale + fm.ascenderPos;
    bezier[3].x = bezier[0].x;
    bezier[3].y = bezier[2].y;
    ObjectCall(msgDcDrawBezier, pData->dc, bezier);

    // Then the right half...
    bezier[1].x = textWidth + fm.em.w;
    bezier[2].x = bezier[1].x;
    ObjectCall(msgDcDrawBezier, pData->dc, bezier);
    // Paint the dot.
    dotRect.origin.x = textWidth + (fm.em.w - fm.xPos) / 2;
    dotRect.origin.y = lineThickness / 2;
    dotRect.size.w = dotRect.size.h = fm.xPos;
    ObjectCall(msgDcDrawEllipse, pData->dc, &dotRect);

    // Fall through to return.
    s = stsOK;

exit:
    ObjCallWarn(msgWinEndRepaint, self, Nil(P_ARGS));
    // Need to restore state if no errors, so might as well do it always.
    ObjectCall(msgDcSetForegroundRGB, pData->dc, (P_ARGS)sysDcRGBBlack);
    ObjectCall(msgDcSetBackgroundRGB, pData->dc, (P_ARGS)sysDcRGBWhite);
    ObjectCall(msgDcSetLineThickness, pData->dc, (P_ARGS)0);
    if (DbgFlagGet('F', 0x40)) {
        Dbg(Debugf("Use a non-zero line thickness.");)
        ObjectCall(msgDcSetLineThickness, pData->dc, (P_ARGS)2);
    }
    return s;
    MsgHandlerParametersNoWarning;
} /* HelloWinRepaint */

/*****************************************************************************
    ClsHelloWinInit

    Install the class.
*****************************************************************************/
STATUS ClsHelloWinInit (void)
{
    CLASS_NEW       new;
    STATUS          s;
    // Create the class.
    ObjCallWarn(msgNewDefaults, clsClass, &new);
    new.object.uid          = clsHelloWin;
    new.cls.pMsg            = clsHelloWinTable;
```

```
    new.cls.ancestor        = clsWin;
    new.cls.size            = SizeOf(INSTANCE_DATA);
    new.cls.newArgsSize     = SizeOf(HELLO_WIN_NEW);
    ObjCallRet(msgNew, clsClass, &new, s);
    if (DbgFlagGet('F', 0x20)) {
        Debugf("Turning on message tracing for clsHelloWin");
        (void)ObjCallWarn(msgTrace, clsHelloWin, (P_ARGS) true);
    }
    return stsOK;
} /* ClsHelloWinInit */
```

DLLINIT.C

```
/******************************************************************************
 File: dllinit.c
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.3  $
     $Date:   13 Nov 1991 18:05:02  $
 This file contains the initialization routine for the Hello World dll.
******************************************************************************/
#include <clsmgr.h>
#include <debug.h>
// The creation routines for each class in this dll.
STATUS ClsHelloWinInit    (void);
/******************************************************************************
     DLLMain

     Initialize DLL
******************************************************************************/
STATUS EXPORTED DLLMain (void)
{
     STATUS  s;
     Dbg(Debugf("Beginning hello.dll initialization.");)
     StsRet(ClsHelloWinInit(), s);
     Dbg(Debugf("Completed hello.dll initialization");)
     return stsOK;
} /* DLLMain */
```

DLL.LBC

```
++DLLMAIN.'GO-HELLO_DLL-V1(0)'
```

MAKEFILE

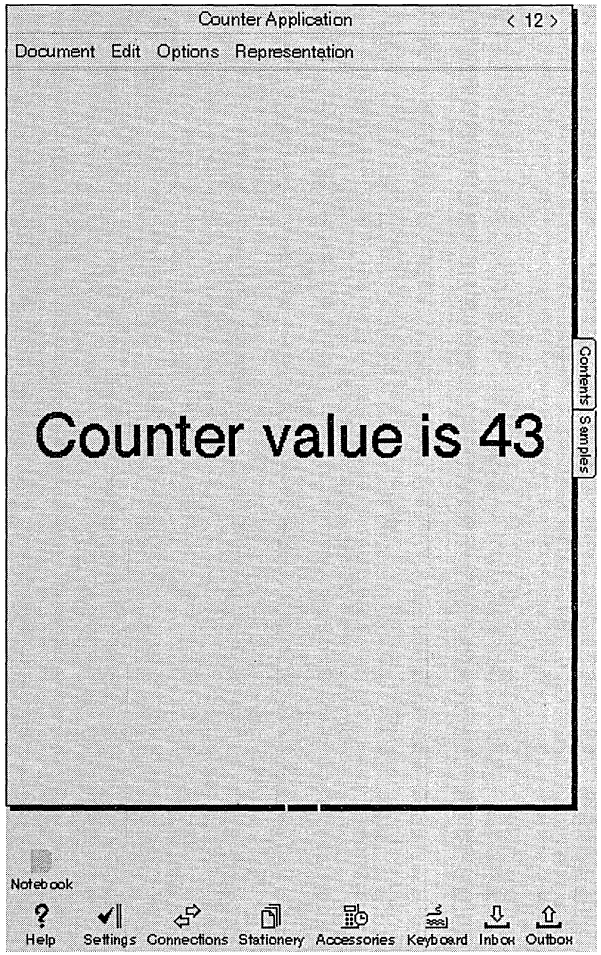
```
###############################################
#
#  WMake Makefile for Hello World
#
# Copyright 1990-1992 GO Corporation. All Rights Reserved.
#
# You may use this Sample Code any way you please provided you
# do not resell the code and that this notice (including the above
# copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
# IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
# EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
# LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
# PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
# FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
# THE USE OR INABILITY TO USE THIS SAMPLE CODE.
#
# $Revision:   1.5  $
#     $Date:   13 Nov 1991 18:05:08  $
#
###############################################
# Set PENPOINT_PATH to your environment variable, if it exists.
# Otherwise, set it to \penpoint
!ifdef %PENPOINT_PATH
PENPOINT_PATH = $(%PENPOINT_PATH)
!else
PENPOINT_PATH = \penpoint
!endif
#  The DOS name of your project directory
PROJ = hello
#  Standard defines for sample code
!INCLUDE $(PENPOINT_PATH)\sdk\sample\sdefines.mif
#  The PenPoint name of your application
EXE_NAME = Hello World
#  The linker name for your executable : company-name-V<major>(<minor>)
EXE_LNAME = GO-HELLO_EXE-V1(0)
#  Object files needed to build your app
EXE_OBJS = heltbl.obj hello.obj
#  Libs needed to build your app
EXE_LIBS = $(DLL_NAME) penpoint app

#  The linker name for the optional DLL : company-name-V<major>(<minor>)
DLL_LNAME = GO-HELLO_DLL-V1(0)
#  Files for the optional DLL
DLL_OBJS = helwtbl.obj dllinit.obj hellowin.obj
```

```
#  Libs needed to build the optional DLL
DLL_LIBS = penpoint win
#  Targets
all: $(APP_DIR)\$(PROJ).exe $(APP_DIR)\$(PROJ).dll .SYMBOLIC
#  The clean rule must be :: because it is also defined in srules
clean:: .SYMBOLIC
 -del heltbl.h
 -del heltbl.tc
 -del helwtbl.h
 -del helwtbl.tc
 -del hello.lib
#  Dependencies
hello.obj: hello.c heltbl.h hellowin.h
hellowin.obj: hellowin.c heltbl.h hellowin.h
#  Standard rules for sample code
!INCLUDE $(PENPOINT_PATH)\sdk\sample\srules.mif
```

HELLO.DLC

```
GO-HELLO_DLL-V1(0)    hello.dll
GO-HELLO_EXE-V1(0)    hello.exe
```


Counter Application



Counter Application displays a number in on the screen. Every time you turn to its page, Counter Application increments the number. It also lets you choose the format in which to display the number (decimal, octal, or hexadecimal).

Objectives

Counter Application is used in the ADC labs. This sample application also shows how to:

- Save and restore application state
- Memorymap state data.

Class Overview

Counter Application defines two classes: **clsCntr** and **clsCntrApp**. It makes use of the following classes:

- clsApp
- clsAppMgr
- clsClass
- clsFileHandle
- clsMenu
- clsMenuButton
- clsObject
- clsLabel

Compiling

To compile Counter Application, just

```
cd \penpoint\sdk\sample\cntrapp
wmake
```

This compiles the application and creates CNTRAPP.EXE in \PENPOINT\APP\CNTRAPP.

Running

After compiling Counter Application, you can run it by

1. Adding \\boot\penpoint\app\Counter Application to \PENPOINT\BOOT\APP.INI
2. Booting PenPoint
3. Creating a new Counter Application document, and turning to it.

Alternatively, you can boot PenPoint and then install Counter Application through the Connections Notebook.

Files Used

The code for Counter Application is in \PENPOINT\SDK\SAMPLE\CNTRAPP. The files are:

CNTR.C clsCntr's code and initialization

CNTR.H header file for **clsCntr**

CNTRAPP.C clsCntrApp's code and initialization

CNTRAPP.H header file for **clsCntrApp**

METHODS.TBL method tables for Counter Application.

METHODS.TBL

```
/****************************************************************************
 File: methods.tbl

 Copyright 1990-1992 GO Corporation. All Rights Reserved.

 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.

 $Revision:   1.1  $
     $Date:   07 Jan 1992 16:32:54  $

 This file contains the method tables for the classes in CntrApp.
****************************************************************************/

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef CNTR_INCLUDED
#include <cntr.h>
#endif

#ifndef CNTRAPP_INCLUDED
#include <cntrapp.h>
#endif

MSG_INFO clsCntrMethods[] = {
     msgNewDefaults,            "CntrNewDefaults",     objCallAncestorBefore,
     msgInit,                   "CntrInit",            objCallAncestorBefore,
     msgSave,                   "CntrSave",            objCallAncestorBefore,
     msgRestore,                "CntrRestore",         objCallAncestorBefore,
     msgFree,                   "CntrFree",            objCallAncestorAfter,
     msgCntrGetValue,           "CntrGetValue",        0,
     msgCntrIncr,               "CntrIncr",            0,
0
};

MSG_INFO clsCntrAppMethods[] = {
     msgInit,                   "CntrAppInit",         objCallAncestorBefore,
     msgSave,                   "CntrAppSave",         objCallAncestorBefore,
     msgRestore,                "CntrAppRestore",      objCallAncestorBefore,
```

```
    msgFree,                    "CntrAppFree",          objCallAncestorAfter,
    msgAppInit,                 "CntrAppAppInit",       objCallAncestorBefore,
    msgAppOpen,                 "CntrAppOpen",          objCallAncestorAfter,
    msgAppClose,                "CntrAppClose",         objCallAncestorBefore,
    msgCntrAppChangeFormat,     "CntrAppChangeFormat", 0,
    0
};

CLASS_INFO classInfo[] = {
    "clsCntrTable",    clsCntrMethods,    0,
    "clsCntrAppTable", clsCntrAppMethods, 0,
    0
};
```

CNTR.H

```
/*****************************************************************************
 File: cntr.h
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.1  $
     $Date:   13 Nov 1991 17:55:24  $
 This file contains the API definition for clsCntr.
*****************************************************************************/
#ifndef CNTR_INCLUDED
#define CNTR_INCLUDED
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif
#define clsCntr MakeWKN( 1, 1, wknPrivate)
#define stsCntrMaxReached MakeStatus(clsCntr, 1)

STATUS GLOBAL ClsCntrInit (void);

/*****************************************************************************
 msgNew takes P_CNTR_NEW, returns STATUS
     Creates a new counter object.
*****************************************************************************/
typedef struct CNTR_NEW_ONLY {
    S32 initialValue;
} CNTR_NEW_ONLY, *P_CNTR_NEW_ONLY;
#define  cntrNewFields  \
    objectNewFields \
    CNTR_NEW_ONLY  cntr;
typedef struct CNTR_NEW {
    cntrNewFields
} CNTR_NEW, *P_CNTR_NEW;
/*****************************************************************************
 msgCntrIncr takes void, returns STATUS
     Bumps counter value by one.
*****************************************************************************/
#define msgCntrIncr MakeMsg(clsCntr, 1)
/*****************************************************************************
 msgCntrGetValue takes P_CNTR_INFO, returns STATUS
     Passes back counter value.
*****************************************************************************/
#define msgCntrGetValue MakeMsg(clsCntr, 2)
typedef struct CNTR_INFO {
    S32 value;
} CNTR_INFO, *P_CNTR_INFO;
#endif // CNTR_INCLUDED
```

CNTR.C

```
/*****************************************************************************
 File: cntr.c
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.2  $
     $Date:   07 Jan 1992 16:31:56  $
 This file contains the class definition and methods for clsCntr.
*****************************************************************************/
#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif
#ifndef FS_INCLUDED
#include <fs.h>
#endif
```

```
#ifndef CNTR_INCLUDED
#include <cntr.h>
#endif
#include <methods.h>
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Defines, Types, Globals, Etc                     *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

typedef struct CNTR_INST {
     S32   currentValue;
}  CNTR_INST,
   *P_CNTR_INST;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Message Handlers                                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************
     CntrNewDefaults
     Respond to msgNewDefaults.
*****************************************************************************/
MsgHandlerArgType(CntrNewDefaults, P_CNTR_NEW)
{
     Debugf("Cntr:CntrNewDefaults");
     // Set default value in new struct.
     pArgs->cntr.initialValue = 0;
     return stsOK;
     MsgHandlerParametersNoWarning;
} /* CntrNewDefaults */

/*****************************************************************************
     CntrInit
     Respond to msgInit.
*****************************************************************************/
MsgHandlerArgType(CntrInit, P_CNTR_NEW)
{
     CNTR_INST inst;
     Debugf("Cntr:CntrInit");
     // Set starting value.
     inst.currentValue = pArgs->cntr.initialValue;
     // Update instance data.
     ObjectWrite(self, ctx, &inst);
     return stsOK;
     MsgHandlerParametersNoWarning;
} /* CntrInit */
```

```
/*****************************************************************************
     CntrSave
     Respond to msgSave.
*****************************************************************************/
MsgHandlerArgType(CntrSave, P_OBJ_SAVE)
{
     STREAM_READ_WRITE fsWrite;
     STATUS            s;
     Debugf("Cntr:CntrSave");
     //
     // Write instance to the file.
     //
     fsWrite.numBytes= SizeOf(CNTR_INST);
     fsWrite.pBuf= pData;
     ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s);
     return stsOK;
     MsgHandlerParametersNoWarning;
} /* CntrSave */

/*****************************************************************************
     CntrRestore
     Respond to msgRestore.
*****************************************************************************/
MsgHandlerArgType(CntrRestore, P_OBJ_RESTORE)
{
     CNTR_INST     inst;
     STREAM_READ_WRITE fsRead;
     STATUS            s;
     Debugf("Cntr:CntrRestore");
     //
     // Read instance data from the file.
     //
     fsRead.numBytes= SizeOf(CNTR_INST);
     fsRead.pBuf= &inst;
     ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s);
     //
     // Update instance data.
     //
     ObjectWrite(self, ctx, &inst);
     return stsOK;
     MsgHandlerParametersNoWarning;
} /* CntrRestore */

/*****************************************************************************
     CntrFree
     Respond to msgFree.
*****************************************************************************/
```

```
MsgHandler(CntrFree)
{
     Debugf("Cntr:CntrFree");
     return stsOK;
     MsgHandlerParametersNoWarning;
} /* CntrFree */
/*****************************************************************************
     CntrGetValue
     Respond to msgCntrGetValue.
*****************************************************************************/
MsgHandlerWithTypes(CntrGetValue, P_CNTR_INFO, P_CNTR_INST)
{
     Debugf("Cntr:CntrGetValue");
     pArgs->value = pData->currentValue;
     return stsOK;
     MsgHandlerParametersNoWarning;
} /* CntrGetValue */
/*****************************************************************************
     CntrIncr
     Respond to msgCntrIncr.
*****************************************************************************/
MsgHandler(CntrIncr)
{
     CNTR_INST inst;
     Debugf("Cntr:CntrIncr");
     inst = IDataDeref(pData, CNTR_INST);
     inst.currentValue++;
     ObjectWrite(self, ctx, &inst);
     return stsOK;
     MsgHandlerParametersNoWarning;
} /* CntrIncr */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Installation                                     *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*****************************************************************************
     ClsCntrInit
     Create the class.
*****************************************************************************/
STATUS GLOBAL
ClsCntrInit (void)
{
     CLASS_NEW   new;
     STATUS      s;
     ObjCallJmp(msgNewDefaults, clsClass, &new, s, Error);
     new.object.uid         = clsCntr;
     new.cls.pMsg           = clsCntrTable;
     new.cls.ancestor       = clsObject;
     new.cls.size           = SizeOf(CNTR_INST);
     new.cls.newArgsSize    = SizeOf(CNTR_NEW);
     ObjCallJmp(msgNew, clsClass, &new, s, Error);
     return stsOK;
Error:
     return s;
} /* ClsCntrInit */
```

CNTRAPP.H

```
/*****************************************************************************
 File: cntrapp.h
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.1  $
     $Date:   13 Nov 1991 17:55:26  $
 This file contains definitions for clsCntrApp.
*****************************************************************************/
#ifndef CNTRAPP_INCLUDED
#define CNTRAPP_INCLUDED
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif
// Define a well known UID for the app
#define clsCntrApp MakeWKN(555, 1, wknGlobal)
// Define a message
#define msgCntrAppChangeFormat MakeMsg(clsCntrApp,1)
#endif // CNTRAPP_INCLUDED
```

CNTRAPP.C

```
/*****************************************************************************
 File: cntrapp.c
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
```

```
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.2  $
     $Date:   07 Jan 1992 16:32:02  $
 This file contains the implementation of the counter application class.
*****************************************************************************/
#ifndef APP_INCLUDED
#include <app.h>
#endif
#ifndef APPMGR_INCLUDED
#include <appmgr.h>
#endif
#ifndef OS_INCLUDED
#include <os.h>
#endif
#ifndef RESFILE_INCLUDED
#include <resfile.h>
#endif
#ifndef FRAME_INCLUDED
#include <frame.h>
#endif
#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif
#ifndef TKTABLE_INCLUDED
#include <tktable.h>
#endif
#ifndef MENU_INCLUDED
#include <menu.h>
#endif
#ifndef CNTR_INCLUDED
#include <cntr.h>
#endif
#ifndef CNTRAPP_INCLUDED
#include <cntrapp.h>
#endif
#include <methods.h>
#include <string.h>
#include <stdio.h>
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Defines, Types, Globals, Etc                     *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
typedef enum CNTRAPP_DISPLAY_FORMAT {
    dec, oct, hex
}  CNTRAPP_DISPLAY_FORMAT,
   *P_CNTRAPP_DISPLAY_FORMAT;
typedef struct CNTRAPP_INST {
    P_CNTRAPP_DISPLAY_FORMAT  pFormat;
    OBJECT                    fileHandle;
    OBJECT                    counter;
}  CNTRAPP_INST,
   *P_CNTRAPP_INST;
static TK_TABLE_ENTRY CntrAppMenuBar[] = {
    {"Representation", 0, 0, 0, tkMenuPullDown, clsMenuButton},
        {"Dec", msgCntrAppChangeFormat, dec},
        {"Oct", msgCntrAppChangeFormat, oct},
        {"Hex", msgCntrAppChangeFormat, hex},
        {pNull},
    {pNull}
};

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Local Functions
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*****************************************************************************
 BuildString
 Local function to build a label string
*****************************************************************************/
STATUS LOCAL BuildString(
    P_STRING        p,
    P_CNTRAPP_INST pData)
{
    CNTR_INFO ci;
    STATUS    s;
    ObjCallRet(msgCntrGetValue, pData->counter, &ci ,s);
    switch (*(pData->pFormat)) {
        case dec:
            sprintf(p, " Counter value is %d ", ci.value);
            break;
        case oct:
            sprintf(p, " Counter value is %o ", ci.value);
            break;
        case hex:
            sprintf(p, " Counter value is %x ", ci.value);
            break;
        default:
            sprintf(p, " Unknown Representation Requested ");
            break;
    }
    return stsOK;
} /* BuildString */
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Message Handlers                               *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*****************************************************************************
     CntrAppInit
     Respond to msgInit.
*****************************************************************************/
MsgHandler(CntrAppInit)
{
     CNTRAPP_INST inst;
     Debugf("CntrApp:CntrAppInit");
     inst.counter = pNull;
     inst.fileHandle = pNull;
     inst.pFormat = pNull;
     // Update instance data.
     ObjectWrite(self, ctx, &inst);
     return stsOK;
     MsgHandlerParametersNoWarning;
} /* CntrAppInit */
/*****************************************************************************
     CntrAppSave
     Respond to msgSave.
     Save the counter object using ResPutObject. The counter will
     receive msgSave and save any data it needs.
     The application doesn't need to save any data, the format data is
     memory mapped.
*****************************************************************************/
MsgHandlerWithTypes(CntrAppSave, P_OBJ_SAVE, P_CNTRAPP_INST)
{
     STATUS s;
     Debugf("CntrApp:CntrAppSave");
     // Save the counter object.
     ObjCallRet(msgResPutObject, pArgs->file, pData->counter, s);
     return stsOK;
     MsgHandlerParametersNoWarning;
} /* CntrAppSave */
/*****************************************************************************
     CntrAppRestore
     Respond to msgRestore.
     Open the file holding the application data, memory map the file.
     Restore the counter object by sending msgResGetObject -- the counter
     will receive msgRestore.
*****************************************************************************/
MsgHandlerWithTypes(CntrAppRestore, P_OBJ_RESTORE, P_CNTRAPP_INST)
{
     FS_NEW        fsn;
     CNTRAPP_INST inst;
     STATUS        s;
     Debugf("CntrApp:CntrAppRestore");
     // Get handle for format file, save the handle
     ObjCallRet(msgNewDefaults, clsFileHandle, &fsn, s);
     fsn.fs.locator.pPath = "formatfile";
     fsn.fs.locator.uid = theWorkingDir;
     ObjCallRet(msgNew, clsFileHandle, &fsn, s);
     inst.fileHandle = fsn.object.uid;
     // Map the file to memory
     ObjCallRet(msgFSMemoryMap, fsn.object.uid, &inst.pFormat, s );
     // Restore the counter object.
     ObjCallJmp(msgResGetObject, pArgs->file, &inst.counter, s, Error);
     // Update instance data.
     ObjectWrite(self, ctx, &inst);
     return stsOK;
     MsgHandlerParametersNoWarning;
Error:
     return s;
} /* CntrAppRestore */

/*****************************************************************************
     CntrAppFree
     Respond to msgFree.
*****************************************************************************/
MsgHandlerWithTypes(CntrAppFree, P_ARGS, P_CNTRAPP_INST)
{
     STATUS s;
     Debugf("CntrApp:CntrAppFree");
     ObjCallRet(msgDestroy, pData->counter, Nil(P_ARGS), s);
     // Unmap the file
     ObjCallRet(msgFSMemoryMapFree, pData->fileHandle, Nil(P_ARGS), s);
     // Free the file handle
     ObjCallRet(msgDestroy, pData->fileHandle, Nil(P_ARGS), s );
     return stsOK;
     MsgHandlerParametersNoWarning;
} /* CntrAppFree */

/*****************************************************************************
     CntrAppAppInit
     Respond to msgAppInit.
     Create the file to hold the memory mapped data.
*****************************************************************************/
MsgHandler(CntrAppAppInit)
{
```

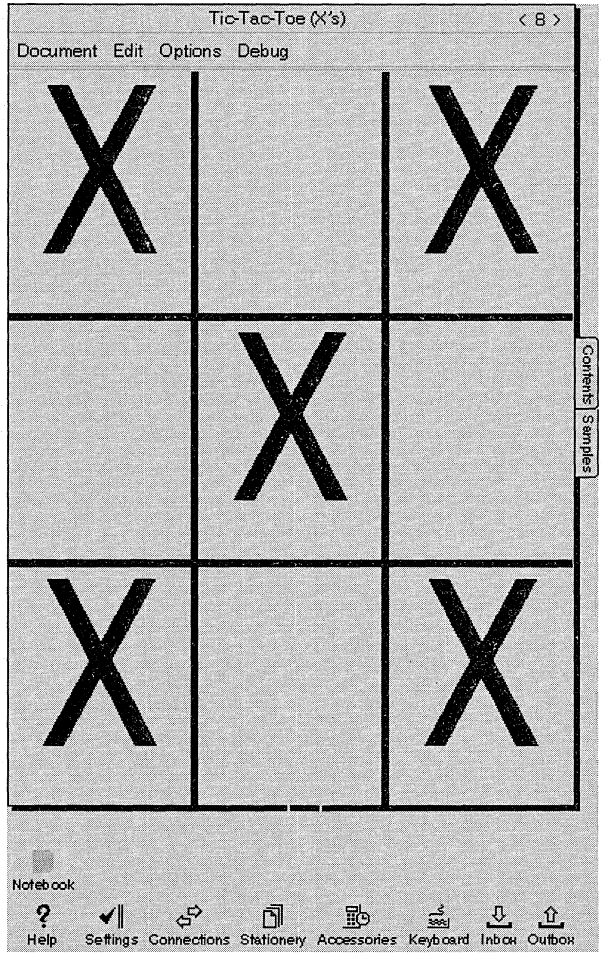
```
    CNTR_NEW                cn;
    FS_NEW                  fsn;
    STREAM_READ_WRITE       fsWrite;
    CNTRAPP_DISPLAY_FORMAT  format;
    CNTRAPP_INST            inst;
    STATUS                  s;

    Debugf("CntrApp:CntrAppAppInit");

    inst = IDataDeref(pData, CNTRAPP_INST);

    //
    // Create the counter object.
    //
    ObjCallRet(msgNewDefaults, clsCntr, &cn, s);
    cn.cntr.initialValue = 42;
    ObjCallRet(msgNew, clsCntr, &cn, s);

    inst.counter = cn.object.uid;

    //
    // Create a file, fill it with a default value
    //
    ObjCallRet(msgNewDefaults, clsFileHandle, &fsn, s);
    fsn.fs.locator.pPath = "formatfile";
    fsn.fs.locator.uid = theWorkingDir;
    ObjCallRet(msgNew, clsFileHandle, &fsn, s);

    format = dec;
    fsWrite.numBytes = SizeOf(CNTRAPP_DISPLAY_FORMAT);
    fsWrite.pBuf = &format;
    ObjCallRet(msgStreamWrite, fsn.object.uid, &fsWrite, s);

    inst.fileHandle = fsn.object.uid;

    //
    // Map the file to memory
    //
    ObjCallRet(msgFSMemoryMap, fsn.object.uid, &inst.pFormat, s);

    // Update instance data.
    ObjectWrite(self, ctx, &inst);

    return stsOK;
    MsgHandlerParametersNoWarning;
} /* CntrAppAppInit */

/*****************************************************************************
    CntrAppOpen

    Respond to msgAppOpen.

    It's important that the ancestor be called AFTER all the frame
    manipulations in this routine because the ancestor takes care of any
    layout that is necessary.
*****************************************************************************/
MsgHandlerWithTypes(CntrAppOpen, P_ARGS, P_CNTRAPP_INST)
{
    APP_METRICS am;
    MENU_NEW    mn;
    LABEL_NEW   ln;
    STATUS      s;
    char        buf[30];

    Debugf("CntrApp:CntrAppOpen");

    // Increment the counter.
    ObjCallRet(msgCntrIncr, pData->counter, Nil(P_ARGS), s);

    // Build the string for the label.
    StsRet(BuildString(buf, pData), s);

    // Create the label.
    ObjCallRet(msgNewDefaults, clsLabel, &ln, s);
    ln.label.pString = buf;
    ln.label.style.scaleUnits = bsUnitsFitWindowProper;
    ln.label.style.xAlignment = lsAlignCenter;
    ln.label.style.yAlignment = lsAlignCenter;
    ObjCallRet(msgNew, clsLabel, &ln, s);

    // Get app metrics.
    ObjCallJmp(msgAppGetMetrics, self, &am, s, Error);

    // Set the label as the clientWin.
    ObjCallJmp(msgFrameSetClientWin, am.mainWin, ln.object.uid, s, Error);

    // Create and add menu bar.
    ObjCallJmp(msgNewDefaults, clsMenu, &mn, s, Error);
    mn.tkTable.client = self;
    mn.tkTable.pEntries = CntrAppMenuBar;
    ObjCallJmp(msgNew, clsMenu, &mn, s, Error);

    ObjCallJmp(msgAppCreateMenuBar, self, &mn.object.uid, s, Error);
    ObjCallJmp(msgFrameSetMenuBar, am.mainWin, mn.object.uid, s, Error);

    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    return s;
} /* CntrAppOpen */

/*****************************************************************************
    CntrAppClose

    Respond to msgAppClose.
    Be sure that the ancestor is called FIRST.  The ancestor extracts the
    frame, and we want the frame extracted before performing surgery on it.
*****************************************************************************/
MsgHandler(CntrAppClose)
{
    APP_METRICS am;
    STATUS      s;

    Debugf("CntrApp:CntrAppClose");

    // Free the menu bar.
    ObjCallJmp(msgAppGetMetrics, self, &am, s, Error);
    ObjCallJmp(msgFrameDestroyMenuBar, am.mainWin, pNull, s, Error);

    return stsOK;
```

```
    MsgHandlerParametersNoWarning;
Error:
    return s;
} /* CntrAppClose */

/******************************************************************************
    CntrAppChangeFormat

    Respond to msgCntrAppChangeFormat.
    Update the memory mapped data.
******************************************************************************/
MsgHandlerWithTypes(CntrAppChangeFormat, P_ARGS, P_CNTRAPP_INST)
{
    APP_METRICS  am;
    WIN          thelabel;
    STATUS       s;
    char         buf[30];

    Debugf("CntrApp:CntrAppChangeFormat");

    //
    // Update mmap data
    //
    *(pData->pFormat) = (CNTRAPP_DISPLAY_FORMAT)(U32)pArgs;

    // Build the string for the label.
    StsRet(BuildString(buf, pData), s);

    // Get app metrics.
    ObjCallRet(msgAppGetMetrics, self, &am, s);

    // Get the clientWin.
    ObjCallRet(msgFrameGetClientWin, am.mainWin, &thelabel, s);

    // Set the label string.
    ObjCallRet(msgLabelSetString, thelabel, buf, s);

    return stsOK;
    MsgHandlerParametersNoWarning;
} /* CntrAppChangeFormat */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Installation                                       *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/******************************************************************************
    ClsCntrAppInit

    Create the application class.
******************************************************************************/

STATUS GLOBAL
ClsCntrAppInit (void)
{
    APP_MGR_NEW new;
    STATUS      s;

    ObjCallJmp(msgNewDefaults, clsAppMgr, &new, s, Error);

    new.object.uid       = clsCntrApp;
    new.cls.pMsg         = clsCntrAppTable;
    new.cls.ancestor     = clsApp;
    new.cls.size         = SizeOf(CNTRAPP_INST);
    new.cls.newArgsSize  = SizeOf(APP_NEW);

    strcpy(new.appMgr.company, "GO Corporation");
    strcpy(new.appMgr.defaultDocName, "Counter Application");

    ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);

    return stsOK;
Error:
    return s;
} /* ClsCntrAppInit */

/******************************************************************************
    main

    Main application entry point.
******************************************************************************/
void CDECL
main(
    int         argc,
    char *      argv[],
    U16         processCount)
{
    if (processCount == 0) {
        StsWarn(ClsCntrAppInit());
        AppMonitorMain(clsCntrApp, objNull);
    } else {
        StsWarn(ClsCntrInit());
        AppMain();
    }
    Unused(argc); Unused(argv); // Suppress compiler's "unused parameter"
        warnings
} /* main */
```

MAKEFILE

```
###############################################
#
#  WMake Makefile for CounterApp
#
# Copyright 1990-1992 GO Corporation. All Rights Reserved.
#
# You may use this Sample Code any way you please provided you
# do not resell the code and that this notice (including the above
# copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
# IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
# EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
# LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
# PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
```

```
# FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
# THE USE OR INABILITY TO USE THIS SAMPLE CODE.
#
# $Revision:   1.4  $
#     $Date:   07 Jan 1992 16:31:50  $
#
##########################################
# Set PENPOINT_PATH to your environment variable, if it exists.
# Otherwise, set it to \penpoint
!ifdef %PENPOINT_PATH
PENPOINT_PATH = $(%PENPOINT_PATH)
!else
PENPOINT_PATH = \penpoint
!endif
#  The DOS name of your project directory.
PROJ = cntrapp
#  Standard defines for sample code (needs the PROJ) definition
!INCLUDE $(PENPOINT_PATH)\sdk\sample\sdefines.mif
#  The PenPoint name of your application
EXE_NAME = Counter Application
#  The linker name for your executable : company-name-V<major>(<minor>)
EXE_LNAME= GO-CNTRAPP-V1(0)
#  Object files needed to build your app
EXE_OBJS = methods.obj cntr.obj cntrapp.obj
#  Libs needed to build your app
EXE_LIBS = penpoint app
# Targets
all: $(APP_DIR)\$(PROJ).exe .SYMBOLIC
#  The clean rule must be :: because it is also defined in srules
clean ::
 -del methods.h
 -del methods.tc
#  Dependencies
cntrapp.obj: cntrapp.c methods.h cntrapp.h cntr.h
cntr.obj: cntr.c methods.h cntr.h
methods.obj: methods.tbl cntr.h cntrapp.h
#  Standard rules for sample code
!INCLUDE $(PENPOINT_PATH)\sdk\sample\srules.mif
```

Tic-Tac-Toe presents a tictactoe board and lets the user enter Xs and Os on it. It is not a true computerized game—the user does not play tic-tac-toe against the computer. Instead, it assumes that that there are two users who want to play the game against each other.

Although a tic-tac-toe game is not exactly a typical notebook application, TicTacToe has many of the characteristics of a fullblown PenPoint application. It has a graphical interface, handwritten input, keyboard input, gesture support, use of the notebook metaphor, versioning of filed data, selection, import/export, option cards, undo support, stationery, help text, and so on.

Objectives

This sample application shows how to:

- Store data in a separate data object
- Display data in a view
- Accept handwritten and keyboard input
- Implement gesture handling
- Support most of the standard application menus (e.g., move, copy, delete, and undo)
- Add applicationspecific menus
- Add applicationspecific option cards
- Provide help
- Provide quick help (using tags in the resource list) for the view, an option card, and the controls in the option card
- Provide stationery documents
- Have both large and small applicationspecific document icons
- Provide customized undo strings
- Use ClsSymbolsInit()
- Specify an application version number.

Class Overview

TicTacToe defines three classes: **clsTttApp**, **clsTttView**, and **clsTttData**. It makes use of the following classes:

clsApp
clsAppMgr
clsClass
clsFileHandle
clsIntegerField
clsIP
clsKey
clsMenu
clsNote
clsObject
clsOptionTable
clsPen
clsScrollWin
clsSysDrwCtx
clsView
clsXferList
clsXGesture
clsXText

Compiling

To compile TicTacToe, just

```
cd \penpoint\sdk\sample\ttt
wmake
```

This compiles the application and creates TTT.EXE and APP.RES in \PENPOINT\APP\TTT. Next, you need to make the stationery and help:

```
wmake stationery
wmake help
```

These two steps make \STATIONRY and \HELP directories in \PENPOINT\APP\TTT, and copy the stationery and help files there.

Running

After compiling TicTacToe, you can run it by

1. Adding \\boot\penpoint\app\TicTacToe to \PENPOINT\BOOT\APP.INI
2. Booting PenPoint
3. Creating a new TicTacToe document, and turning to it.

Alternatively, you can boot PenPoint and then install TicTacToe through the Connections Notebook.

Files Used

The code for TicTacToe is in \PENPOINT\SDK\SAMPLE\TTT. The files are:

FILLED.TXT stationery file (filled with Xs and Os)
LGICON.RES large document icon resource file (compiled)

METHODS.TBL the method tables for all of the TicTacToe classes

RULES.TXT help file (containing the rules for the game)

S_TTT.C symbol name definitions and call to ClsSymbolsInit()

SMICON.RES small document icon resource file (compiled)

STRAT.TXT help file (containing a strategy for playing the game)

TTTAPP.C clsTttApp's code and initialization

TTTAPP.H header file for the application class

TTTDATA.C clsTttData's code and initialization

TTTDATA.H header file for the data class

TTTDBG.C debuggingrelated message handlers

TTTIPAD.C insertion padrelated message handlers

TTTMBAR.C menu barrelated message handlers

TTTPRIV.H private include file for TicTacToe

TTTQHELP.RC quick help resource file (uncompiled)

TTTUTIL.C utility functions

TTTVIEW.C clsTttView's code and initialization

TTTVIEW.H header file for the view class

TTTVOPT.C clsTttView's option cardrelated message handlers

TTTVXFER.C clsTttView's data transferrelated message handlers

XSONLY.TXT stationery file (partially filled, with Xs only).

METHODS.TBL

```
/****************************************************************************
 File: methods.tbl
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.0  $
 $Date:   07 Jan 1992 17:19:06  $
 This file contains the method tables for the classes in TttApp.
****************************************************************************/

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef TTTPRIV_INCLUDED
#include <tttpriv.h>
#endif

#ifndef TTTDATA_INCLUDED
#include <tttdata.h>
#endif

#ifndef TTTVIEW_INCLUDED
#include <tttview.h>
#endif

#ifndef TTTAPP_INCLUDED
#include <tttapp.h>
#endif

#ifndef WIN_INCLUDED
#include <win.h>
#endif

#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef SEL_INCLUDED
#include <sel.h>
#endif

#ifndef GWIN_INCLUDED
#include <gwin.h>
#endif

#ifndef OPTION_INCLUDED
#include <option.h>
#endif

#ifndef UNDO_INCLUDED
#include <undo.h>
#endif

#ifndef XFER_INCLUDED
#include <xfer.h>
#endif

MSG_INFO clsTttViewMethods[] = {
     msgNewDefaults,          "TttViewNewDefaults",    objCallAncestorBefore,
     msgInit,                 "TttViewInit",           0,
     msgFree,                 "TttViewFree",           objCallAncestorAfter,
     msgSave,                 "TttViewSave",           objCallAncestorBefore,
     msgRestore,              "TttViewRestore",        objCallAncestorBefore,
#ifdef DEBUG
     msgDump,                 "TttViewDump",           objCallAncestorBefore,
#endif
     msgTttDataChanged,           "TttViewDataChanged",                    0,
     msgWinRepaint,               "TttViewRepaint",                        0,
```

```
    msgWinGetDesiredSize,       "TttViewGetDesiredSize",                0,
    msgGWinGesture,             "TttViewGesture",                       0,
    msgTttViewGetMetrics,       "TttViewGetMetrics",                    0,
    msgTttViewSetMetrics,       "TttViewSetMetrics",                    0,
    msgTttViewToggleSel,        "TttViewToggleSel",                     0,
    msgTttViewTakeSel,          "TttViewTakeSel",                       0,
    msgInputEvent,              "TttViewInputEvent",                    0,
    msgXferGet,                 "TttViewXferGet",                       0,
    msgXferList,                "TttViewXferList",                      0,
    msgOptionApplyCard,         "TttViewOptionApplyCard",               0,
    msgOptionRefreshCard,       "TttViewOptionRefreshCard",             0,
    msgOptionProvideCardWin,    "TttViewOptionProvideCard",             0,
    msgOptionAddCards,          "TttViewOptionAddCards",                0,
    msgOptionApplicableCard,    "TttViewOptionApplicableCard",          0,
    msgSelYield,                "TttViewSelYield",                      0,
    msgSelBeginMove,            "TttViewSelBeginMoveAndCopy",           0,
    msgSelBeginCopy,            "TttViewSelBeginMoveAndCopy",           0,
    msgSelMoveSelection,        "TttViewSelMoveAndSelCopy",             0,
    msgSelCopySelection,        "TttViewSelMoveAndSelCopy",             0,
    msgSelDelete,               "TttViewSelDelete",                     0,
    msgControlProvideEnable,    "TttViewProvideEnable",     0,
    0
};

MSG_INFO clsTttDataMethods[] = {
    msgNewDefaults,             "TttDataNewDefaults",   objCallAncestorBefore,
    msgInit,                    "TttDataInit",          objCallAncestorBefore,
    msgFree,                    "TttDataFree",          objCallAncestorAfter,
    msgSave,                    "TttDataSave",          objCallAncestorBefore,
    msgRestore,                 "TttDataRestore",       objCallAncestorBefore,
#ifdef DEBUG
    msgDump,                    "TttDataDump",          objCallAncestorBefore,
#endif
    msgTttDataGetMetrics,       "TttDataGetMetrics",    0,
    msgTttDataSetMetrics,       "TttDataSetMetrics",    0,
    msgTttDataSetSquare,        "TttDataSetSquare",     0,
    msgTttDataRead,             "TttDataRead",          0,
    msgUndoItem,                "TttDataUndoItem",      0,
    0
};

MSG_INFO clsTttAppMethods[] = {
    msgInit,                    "TttAppInit",           objCallAncestorBefore,
    msgFree,                    "TttAppFree",           objCallAncestorAfter,
    msgSave,                    "TttAppSave",           objCallAncestorBefore,
    msgRestore,                 "TttAppRestore",        objCallAncestorBefore,
#ifdef DEBUG
    msgDump,                    "TttAppDump",           objCallAncestorBefore,
    msgTttAppDumpView,          "TttDbgDumpView",       0,
    msgTttAppDumpDataObject,    "TttDbgDumpDataObject", 0,
    msgTttAppDumpWindowTree,    "TttDbgDumpWindowTree", 0,
    msgTttAppChangeTracing,     "TttDbgChangeTracing",  0,
    msgTttAppForceRepaint,      "TttDbgForceRepaint",   0,
#endif
    msgAppInit,                 "TttAppAppInit",        objCallAncestorBefore,
    msgAppOpen,                 "TttAppOpen",           objCallAncestorAfter,
    msgAppClose,                "TttAppClose",          objCallAncestorBefore,
    msgAppSelectAll,            "TttAppSelectAll",      0,
    msgAppExport,               "TttAppExport",         0,
    msgControlProvideEnable,    "TttAppProvideEnable",  0,
    0
};

CLASS_INFO classInfo[] = {
    "clsTttViewTable",  clsTttViewMethods,  0,
    "clsTttDataTable",  clsTttDataMethods,  0,
    "clsTttAppTable",   clsTttAppMethods,   0,
    0
};
```

TTTPRIV.H

```
/*****************************************************************************
 File: tttpriv.h
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.1  $
 $Date:   13 Nov 1991 18:40:22  $
 This file contains things shared by various pieces of TttApp.
*****************************************************************************/
#ifndef TTTPRIV_INCLUDED
#define TTTPRIV_INCLUDED
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif
#ifndef GEO_INCLUDED
#include <geo.h>
#endif
#ifndef SYSFONT_INCLUDED
#include <sysfont.h>
#endif
```

```
#ifndef TTTDATA_INCLUDED
#include <tttdata.h>
#endif
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Useful Macros                                  *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

#define TttDbgHelper(str,set,flag,x) \
     Dbg(if (DbgFlagGet((set),(U32)(flag))) {DPrintf("%s: ",str); Debugf x;})

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        CommonDefines                                  *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        CommonTypedefs                                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
typedef U8
TTT_VERSION, * P_TTT_VERSION;
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Utility Routines                               *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
//
// Function definitions for routines in tttutil.c.  Arguably these
// definitions should be in a separate header, but it's simpler just
// to keep them here.
//
STATUS PASCAL
TttUtilCreateScrollWin(
     OBJECT          clientWin,
     P_OBJECT        pScrollWin);

STATUS PASCAL
TttUtilCreateMenu(
     OBJECT          parent,
     OBJECT          client,
     P_UNKNOWN       pEntries,   // really  P_TK_TABLE_ENTRY pEntries
     P_OBJECT            pMenu);

STATUS PASCAL
TttUtilAdjustMenu(
     OBJECT          menu);

STATUS PASCAL
TttUtilWrite(
     OBJECT          file,
     U32             numBytes,
     P_UNKNOWN       pBuf);

STATUS PASCAL
TttUtilWriteVersion(
     OBJECT          file,
     TTT_VERSION     version);

STATUS PASCAL
TttUtilRead(
     OBJECT          file,
     U32             numBytes,
     P_UNKNOWN       pBuf);

STATUS PASCAL
TttUtilReadVersion(
     OBJECT          file,
     TTT_VERSION     minVersion,
     TTT_VERSION     maxVersion,
     P_TTT_VERSION   pVersion);

STATUS PASCAL
TttUtilGetComponents(
     OBJECT          app,
     U16             getFlags,
     P_OBJECT            pScrollWin,
     P_OBJECT            pView,
     P_OBJECT            pDataObject);
//
// Values for the getFlags parameter to TttUtilGetComponents.
//
#define tttGetScrollWin      flag0
#define tttGetView       flag1
#define tttGetDataObject flag2

STATUS PASCAL
TttUtilPostNotImplementedYet(
     P_STRING        featureStr);

STATUS PASCAL
TttUtilGiveUpSel(
     OBJECT  object);

STATUS PASCAL
TttUtilTakeSel(
     OBJECT  object);

void PASCAL
TttUtilInitTextOutput(
     P_SYSDC_TEXT_OUTPUT p,
     U16                     align,
     P_U8                        buf);
```

```
P_STRING PASCAL
TttUtilStrForSquareValue(
     TTT_SQUARE_VALUE    v);

TTT_SQUARE_VALUE PASCAL
TttUtilSquareValueForChar(
     char    ch);

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Debugging Routines                               *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
//
// Function definitions for routines in tttdbg.c Arguably these
// definitions should be in a separate header, but it's simpler just to
// keep them here.
//
#ifdef DEBUG
MsgHandler(TttDbgDumpWindowTree);
MsgHandler(TttDbgDumpView);
MsgHandler(TttDbgDumpDataObject);
MsgHandler(TttDbgDumpDebugFlags);
MsgHandler(TttDbgChangeDebugSet);
MsgHandler(TttDbgChangeDebugFlag);
#endif // DEBUG

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                   Global Variables and Defines                        *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
//
// A Note on global well-knows versus private well-knowns.
//
// All of the UIDs used in this application are global well-knowns.
// Strictly speaking, only the application class UID needs to be global.
// If none of the other classes are referenced outside of this
// application, then they can (and should) be private.
//
// The UIDs here are global because that's the most general and least
// error-prone, and this application's role as template and tutorial
// make generality and stability very important.  However, any production
// application built constructed from this should consider making
// some of these UIDs local.
//
// And, of course, any other application (production or not) constructed
// from this template should get their own allocation of uids.
//
// Also allocated to tic-tac-toe, but not yet used:
//       2225
//
```



```
#define clsTttApp         MakeGlobalWKN(2222,1)
#define clsTttData        MakeGlobalWKN(2223,1)
#define clsTttView        MakeGlobalWKN(2224,1)
//
// Debug flag sets
//
#define tttAppDbgSet          0xC0
#define tttDataDbgSet     0xC1
#define tttUtilDbgSet     0xC2
#define tttViewDbgSet     0xC3
#define tttViewOptsDbgSet 0xC4
#define tttViewXferDbgSet 0xC5

#endif   // TTTPRIV_INCLUDED
```

TTTAPP.H

```
/*****************************************************************************
 File: tttapp.h
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.1  $
 $Date:   13 Nov 1991 18:40:16  $
 This file contains the API definition for clsTttApp.
 clsTttApp inherits from clsApp.
*****************************************************************************/
#ifndef TTTAPP_INCLUDED
#define TTTAPP_INCLUDED
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Defines                                      *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                       Common Typedefs                                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Exported Functions                                   *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*****************************************************************************
 ClsTttAppInit    returns STATUS
     Initializes / installs clsTttApp.
     This routine is only called during installation of the class.
*/
STATUS PASCAL
ClsTttAppInit (void);

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Messages for clsTttApp                               *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
//
// Debug messages aren't in standard header form so that they won't
// show up in automatically-generated datasheets.
//
#ifdef DEBUG
#define dbgMsgStart 0x40
#define  msgTttAppChangeDebugFlag    MakeMsg(clsTttApp, dbgMsgStart + 0)
#define  msgTttAppChangeDebugSet     MakeMsg(clsTttApp, dbgMsgStart + 1)
#define  msgTttAppDumpWindowTree     MakeMsg(clsTttApp, dbgMsgStart + 2)
#define  msgTttAppDumpDebugFlags     MakeMsg(clsTttApp, dbgMsgStart + 3)
#define  msgTttAppDumpView           MakeMsg(clsTttApp, dbgMsgStart + 4)
#define  msgTttAppDumpDataObject     MakeMsg(clsTttApp, dbgMsgStart + 5)
#define  msgTttAppChangeTracing      MakeMsg(clsTttApp, dbgMsgStart + 6)
#define  msgTttAppForceRepaint       MakeMsg(clsTttApp, dbgMsgStart + 7)
#endif
#endif   // TTTAPP_INCLUDED
```

TTTAPP.C

```
/*****************************************************************************
 File: tttapp.h
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.1  $
 $Date:   13 Nov 1991 18:40:16  $
 This file contains the API definition for clsTttApp.
 clsTttApp inherits from clsApp.
*****************************************************************************/
#ifndef TTTAPP_INCLUDED
#define TTTAPP_INCLUDED
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Defines                                              *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Common Typedefs                                      *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Exported Functions                                   *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*****************************************************************************
 ClsTttAppInit    returns STATUS
     Initializes / installs clsTttApp.
     This routine is only called during installation of the class.
*/
STATUS PASCAL
ClsTttAppInit (void);

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Messages for clsTttApp                               *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
//
// Debug messages aren't in standard header form so that they won't
// show up in automatically-generated datasheets.
//
#ifdef DEBUG
#define dbgMsgStart 0x40
#define  msgTttAppChangeDebugFlag    MakeMsg(clsTttApp, dbgMsgStart + 0)
#define  msgTttAppChangeDebugSet     MakeMsg(clsTttApp, dbgMsgStart + 1)
#define  msgTttAppDumpWindowTree     MakeMsg(clsTttApp, dbgMsgStart + 2)
#define  msgTttAppDumpDebugFlags     MakeMsg(clsTttApp, dbgMsgStart + 3)
#define  msgTttAppDumpView           MakeMsg(clsTttApp, dbgMsgStart + 4)
#define  msgTttAppDumpDataObject     MakeMsg(clsTttApp, dbgMsgStart + 5)
#define  msgTttAppChangeTracing      MakeMsg(clsTttApp, dbgMsgStart + 6)
#define  msgTttAppForceRepaint       MakeMsg(clsTttApp, dbgMsgStart + 7)
#endif
#endif   // TTTAPP_INCLUDED
```

TTTDATA.H

```
/****************************************************************************
 File: tttdata.h
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.2  $
 $Date:   02 Dec 1991 19:07:50  $
 This file contains the API definition for clsTttData.
 clsTttData inherits from clsObject.
****************************************************************************/
#ifndef TTTDATA_INCLUDED
#define TTTDATA_INCLUDED
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Defines                                          *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
//
// AKN
//
#define  numberOfRows        5
#define  numberOfColumns     5
#define  tagTttDataUndoDelete        MakeTag(clsTttData, 0)
#define  tagTttDataUndoMoveCopy      MakeTag(clsTttData, 1)
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Common Typedefs                                  *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
typedef OBJECT
TTT_DATA, * P_TTT_DATA;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Exported Functions                               *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/****************************************************************************
 ClsTttDataInit  returns STATUS
     Initializes / installs clsTttData.
     This routine is only called during installation of the class.
*/
STATUS PASCAL
ClsTttDataInit (void);

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Messages for clsTttData                          *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/****************************************************************************
 msgNew      takes P_TTT_DATA_NEW, returns STATUS
     category: class message
     Creates an instance of clsTttData.
*/
typedef char
TTT_SQUARE_VALUE;
#define tttBlank ((TTT_SQUARE_VALUE)' ')
#define tttX   ((TTT_SQUARE_VALUE)'X')
#define tttO   ((TTT_SQUARE_VALUE)'O')
typedef TTT_SQUARE_VALUE
TTT_SQUARES [numberOfRows] [numberOfColumns];
typedef struct TTT_DATA_METRICS {
     TTT_SQUARES squares;
     U32         undoTag;
} TTT_DATA_METRICS, * P_TTT_DATA_METRICS;

typedef struct TTT_DATA_NEW_ONLY {
     TTT_DATA_METRICS    metrics;
} TTT_DATA_NEW_ONLY, * P_TTT_DATA_NEW_ONLY;

#define  tttDataNewFields        \
     objectNewFields         \
     TTT_DATA_NEW_ONLY  tttData;

typedef struct TTT_DATA_NEW {
     tttDataNewFields
} TTT_DATA_NEW, * P_TTT_DATA_NEW;

/****************************************************************************
 msgNewDefaults      takes P_TTT_DATA_NEW, returns STATUS
     category: class message
     Initializes the TTT_DATA_NEW structure to default values.
*/
/****************************************************************************
 msgTttDataGetMetrics takes P_TTT_DATA_METRICS, returns STATUS
     Gets TTT_DATA metrics.
*/
#define msgTttDataGetMetrics            MakeMsg(clsTttData, 1)
```

```
/******************************************************************************
 msgTttDataSetMetrics takes P_TTT_DATA_METRICS, returns STATUS
     Sets the TTT_DATA metrics.
*/
#define msgTttDataSetMetrics              MakeMsg(clsTttData, 2)

/******************************************************************************
 msgTttDataSetSquare takes P_TTT_DATA_SET_SQUARE, returns STATUS
     Sets the value of a single square.
*/
#define msgTttDataSetSquare               MakeMsg(clsTttData, 3)
typedef struct {
     U16                row;
     U16                col;
     TTT_SQUARE_VALUE   value;
} TTT_DATA_SET_SQUARE, * P_TTT_DATA_SET_SQUARE;

/******************************************************************************
 msgTttDataRead  takes P_TTT_DATA_READ, returns STATUS
     Causes data object to try to read stationery from fileHandle.

     Returns stsOK if no errors occur, otherwise returns the error
     status.  Returns stsOK even if values cannot be read from the
     file.  The "successful" field indicates whether a value was
     successfully read.  If successful, the data object's state
     is changed and notifications are set.
*/
#define msgTttDataRead                    MakeMsg(clsTttData, 4)
typedef struct TTT_DATA_READ {
     OBJECT  fileHandle;
     BOOLEAN successful;
} TTT_DATA_READ, * P_TTT_DATA_READ;

/******************************************************************************
 msgTttDataChanged   takes P_TTT_DATA_CHANGED or nothing, returns nothing
     category: observer notification
     Sent to observers when the value changes.
     If pArgs is pNull, receiver should assume that everything has changed.
*/
#define msgTttDataChanged                 MakeMsg(clsTttData, 5)
typedef struct {
     U16 row;
     U16 col;
} TTT_DATA_CHANGED, * P_TTT_DATA_CHANGED;

#endif   // TTTDATA_INCLUDED
```

TTTDATA.C

```
/******************************************************************************
 File: tttdata.c
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.4  $
 $Date:   07 Jan 1992 17:18:50  $
 This file contains the implementation of clsTttData.
******************************************************************************/
#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif
#ifndef RESFILE_INCLUDED
#include <resfile.h>
#endif
#ifndef TTTDATA_INCLUDED
#include <tttdata.h>
#endif
#ifndef TTTPRIV_INCLUDED
#include <tttpriv.h>
#endif
#ifndef OSHEAP_INCLUDED
#include <osheap.h>
#endif
#ifndef UNDO_INCLUDED
#include <undo.h>
#endif
#include <stdlib.h>
#include <string.h>
#include <methods.h>

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Defines, Types, Globals, Etc                     *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

typedef struct TTT_DATA_INST {
     TTT_DATA_METRICS    metrics;
} TTT_DATA_INST,
  * P_TTT_DATA_INST,
```

```
  * * PP_TTT_DATA_INST;

//
// CURRENT_VERSION is the file format version written by this implementation.
// MIN_VERSION is the minimum file format version readable by this
// implementation.  MAX_VERSION is the maximum file format version readable
// by this implementation.
//
#define CURRENT_VERSION 0
#define MIN_VERSION     0
#define MAX_VERSION     0

typedef struct TTT_DATA_FILED_0 {
    TTT_SQUARES squares;
} TTT_DATA_FILED_0, * P_TTT_DATA_FILED_0;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Utility Routines                               *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************
    TttDataFiledData0FromInstData

    Computes filed data from instance data.
*****************************************************************************/
STATIC void PASCAL
TttDataFiledData0FromInstData(
    P_TTT_DATA_INST     pInst,
    P_TTT_DATA_FILED_0  pFiled)
{
    memcpy(pFiled->squares, pInst->metrics.squares,
            sizeof(pFiled->squares));
} /* TttDataFiledData0FromInstData */

/*****************************************************************************
    TttDataInstDataFromFiledData0

    Computes instance data from filed data.
*****************************************************************************/
STATIC void PASCAL
TttDataInstDataFromFiledData0(
    P_TTT_DATA_FILED_0  pFiled,
    P_TTT_DATA_INST     pInst)
{
    memcpy(pInst->metrics.squares, pFiled->squares,
            sizeof(pInst->metrics.squares));
} /* TttDataInstDataFromFiledData0 */

/*****************************************************************************
    TttDataNotifyObservers

    Sends notifications.
*****************************************************************************/
#define DbgTttDataNotifyObservers(x) \
    TttDbgHelper("TttDataNotifyObservers",tttDataDbgSet,0x1,x)
STATIC STATUS PASCAL
TttDataNotifyObservers(
    OBJECT                  self,
    P_ARGS                  pArgs)
{
    OBJ_NOTIFY_OBSERVERS    nobs;
    STATUS                  s;

    DbgTttDataNotifyObservers(("("))

    nobs.msg = msgTttDataChanged;
    nobs.pArgs = pArgs;
    nobs.lenSend = SizeOf(TTT_DATA_CHANGED);
    ObjCallJmp(msgNotifyObservers, self, &nobs, s, Error);

    DbgTttDataNotifyObservers(("return stsOK"))
    return stsOK;
Error:
    DbgTttDataNotifyObservers(("Error; return 0x%lx",s))
    return s;
} /* TttDataNotifyObservers */


/*****************************************************************************
    TttDataRecordStateForUndo

    Records current state with undo manager.
    Assumes that a transaction is already open.
*****************************************************************************/
#define DbgTttDataRecordStateForUndo(x) \
    TttDbgHelper("TttDataRecordStateForUndo",tttDataDbgSet,0x2,x)
STATIC STATUS PASCAL
TttDataRecordStateForUndo(
    OBJECT              self,
    TAG                 undoTag,
    PP_TTT_DATA_INST    pData)
{
    UNDO_ITEM           item;
    STATUS              s;

    DbgTttDataRecordStateForUndo(("("))

    ObjCallJmp(msgUndoBegin, theUndoManager, (P_ARGS)undoTag, s, Error);
    item.object = self;
    item.subclass = clsTttData;
    item.flags = 0;
    item.pData = &((*pData)->metrics);
    item.dataSize = SizeOf(((*pData)->metrics));
    ObjCallJmp(msgUndoAddItem, theUndoManager, &item, s, Error);
    ObjCallJmp(msgUndoEnd, theUndoManager, pNull, s, Error);
```

```
    DbgTttDataRecordStateForUndo(("return stsOK"))
    return stsOK;
Error:
    DbgTttDataRecordStateForUndo(("Error; return 0x%lx",s))
    return s;
} /* TttDataRecordStateForUndo */

/*****************************************************************************
    TttDataPrivSetMetrics

    Sets metrics and (optionally) records information
    needed to undo the set.  Assumes an undo transaction is open.
*****************************************************************************/
#define DbgTttDataPrivSetMetrics(x) \
    TttDbgHelper("TttDataPrivSetMetrics",tttDataDbgSet,0x4,x)
STATIC STATUS PASCAL
TttDataPrivSetMetrics(
    OBJECT              self,
    P_TTT_DATA_METRICS  pArgs,
    PP_TTT_DATA_INST    pData,
    BOOLEAN             recordUndo)
{
    STATUS              s;
    DbgTttDataPrivSetMetrics(("("))
    //
    // Perhaps record undo information
    //
    if (recordUndo) {
        StsJmp(TttDataRecordStateForUndo(self, pArgs->undoTag, pData),
                s, Error);
    }
    //
    // Change data.
    //
    (*pData)->metrics = *pArgs;
    DbgTttDataPrivSetMetrics(("returns stsOK"))
    return stsOK;
Error:
    DbgTttDataPrivSetMetrics(("Error; return 0x%lx",s))
    return s;
} /* TttDataPrivSetMetrics */

/*****************************************************************************
    TttDataPrivSetSquare

    Sets a square and (optionally) records
    information needed to undo the set.  Assumes an undo transaction
    is open.
*****************************************************************************/
#define DbgTttDataPrivSetSquare(x) \
    TttDbgHelper("TttDataPrivSetSquare",tttDataDbgSet,0x8,x)
STATIC STATUS PASCAL
TttDataPrivSetSquare(
    OBJECT                  self,
    P_TTT_DATA_SET_SQUARE   pArgs,
    PP_TTT_DATA_INST        pData,
    BOOLEAN                 recordUndo)
{
    STATUS                  s;
    DbgTttDataPrivSetSquare(("("))
    //
    // Perhaps record undo information
    //
    if (recordUndo) {
        StsJmp(TttDataRecordStateForUndo(self, pNull, pData), s, Error);
    }
    //
    // Change data.
    //
    (*pData)->metrics.squares[pArgs->row][pArgs->col] = pArgs->value;
    DbgTttDataPrivSetSquare(("returns stsOK"))
    return stsOK;
Error:
    DbgTttDataPrivSetSquare(("Error; return 0x%lx",s))
    return s;
} /* TttDataPrivSetSquare */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Message Handlers                                   *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************
    TttDataNewDefaults

    Respond to msgNewDefaults.
*****************************************************************************/
#define DbgTttDataNewDefaults(x) \
    TttDbgHelper("TttDataNewDefaults",tttDataDbgSet,0x10,x)
MsgHandlerWithTypes(TttDataNewDefaults, P_TTT_DATA_NEW, PP_TTT_DATA_INST)
{
    U16 row;
    U16 col;

    DbgTttDataNewDefaults(("("))
    for (row=0; row<3; row++) {
        for (col=0; col<3; col++) {
            pArgs->tttData.metrics.squares[row][col] = tttBlank;
        }
    }
    pArgs->tttData.metrics.undoTag = 0;
```

```
    DbgTttDataNewDefaults(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* TttDataNewDefaults */

/*****************************************************************************
    TttDataInit

    Initialize instance data of new object.
    Note: clsmgr has already initialized instance data to zeros.
*****************************************************************************/
#define DbgTttDataInit(x) \
    TttDbgHelper("TttDataInit",tttDataDbgSet,0x20,x)
MsgHandlerWithTypes(TttDataInit, P_TTT_DATA_NEW, PP_TTT_DATA_INST)
{
    P_TTT_DATA_INST pInst;
    STATUS          s;
    DbgTttDataInit(""))
    //
    // Initialize for error recovery.
    //
    pInst = pNull;
    //
    // Allocate, initialize, and record instance data.
    //
    StsJmp(OSHeapBlockAlloc(osProcessHeapId, SizeOf(*pInst), &pInst), \
           s, Error);
    pInst->metrics = pArgs->tttData.metrics;
    ObjectWrite(self, ctx, &pInst);
    DbgTttDataInit(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    if (pInst) {
        OSHeapBlockFree(pInst);
    }
    DbgTttDataInit(("Error; returns 0x%lx",s))
    return s;
} /* TttDataInit */

/*****************************************************************************
    TttDataFree

    Respond to msgFree.
    Note:  Always return stsOK, even if a problem occurs.  This is
    (1) because there's nothing useful to do if a problem occurs anyhow
    and (2) because the ancestor is called after this function if and
    only if stsOK is returned, and it's important that the ancestor
    get called.
*****************************************************************************/
#define DbgTttDataFree(x) \
    TttDbgHelper("TttDataFree",tttDataDbgSet,0x40,x)
MsgHandlerWithTypes(TttDataFree, P_ARGS, PP_TTT_DATA_INST)
{
    STATUS  s;
    DbgTttDataFree(""))
    StsJmp(OSHeapBlockFree(*pData), s, Error);
    DbgTttDataFree(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttDataFree(("Error; return 0x%lx",s))
    return s;
} /* TttDataFree */

/*****************************************************************************
    TttDataSave

    Save self to a file.
*****************************************************************************/
#define DbgTttDataSave(x) \
    TttDbgHelper("TttDataSave",tttDataDbgSet,0x80,x)
MsgHandlerWithTypes(TttDataSave, P_OBJ_SAVE, PP_TTT_DATA_INST)
{
    TTT_DATA_FILED_0    filed;
    STATUS              s;
    DbgTttDataSave(""))
    StsJmp(TttUtilWriteVersion(pArgs->file, CURRENT_VERSION), s, Error);
    TttDataFiledData0FromInstData(*pData, &filed);
    StsJmp(TttUtilWrite(pArgs->file, SizeOf(filed), &filed), s, Error);
    DbgTttDataSave(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttDataSave(("Error; return 0x%lx",s))
    return s;
} /* TttDataSave */

/*****************************************************************************
    TttDataRestore

    Restore self from a file.
    Note: clsmgr has already initialized instance data to zeros.
*****************************************************************************/
#define DbgTttDataRestore(x) \
    TttDbgHelper("TttDataRestore",tttDataDbgSet,0x100,x)
MsgHandlerWithTypes(TttDataRestore, P_OBJ_RESTORE, PP_TTT_DATA_INST)
```

```
{
    P_TTT_DATA_INST     pInst;
    TTT_DATA_FILED_0    filed;
    TTT_VERSION         version;
    STATUS              s;
    DbgTttDataRestore(("("))
    //
    // Initialize for error recovery.
    //
    pInst = pNull;
    //
    // Read version, then read filed data.  (Currently there's only
    // only one legitimate file format, so no checking of the version
    // need be done.)
    //
    // The allocate instance data and convert filed data.
    //
    StsJmp(TttUtilReadVersion(pArgs->file, MIN_VERSION, MAX_VERSION, \
            &version), s, Error);
    StsJmp(TttUtilRead(pArgs->file, SizeOf(filed), &filed), s, Error);
    StsJmp(OSHeapBlockAlloc(osProcessHeapId, SizeOf(*pInst), &pInst), \
            s, Error);
    TttDataInstDataFromFiledData0(&filed, pInst);
    ObjectWrite(self, ctx, &pInst);
    DbgTttDataRestore(("returns stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    if (pInst) {
        OSHeapBlockFree(pInst);
    }
    DbgTttDataRestore(("Error; returns 0x%lx",s))
    return s;
} /* TttDataRestore */

/*****************************************************************************
    TttDataDump

    Respond to msgDump.
*****************************************************************************/
#ifdef DEBUG
MsgHandlerWithTypes(TttDataDump, P_ARGS, PP_TTT_DATA_INST)
{
    Debugf("TttDataDump: [%s %s %s] [%s %s %s] [%s %s %s]",
            TttUtilStrForSquareValue((*pData)->metrics.squares[0][0]),
            TttUtilStrForSquareValue((*pData)->metrics.squares[0][1]),
            TttUtilStrForSquareValue((*pData)->metrics.squares[0][2]),
            TttUtilStrForSquareValue((*pData)->metrics.squares[1][0]),
            TttUtilStrForSquareValue((*pData)->metrics.squares[1][1]),
            TttUtilStrForSquareValue((*pData)->metrics.squares[1][2]),
            TttUtilStrForSquareValue((*pData)->metrics.squares[2][0]),
            TttUtilStrForSquareValue((*pData)->metrics.squares[2][1]),
            TttUtilStrForSquareValue((*pData)->metrics.squares[2][2]));
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* TttDataDump */
#endif // DEBUG

/*****************************************************************************
    TttDataGetMetrics
*****************************************************************************/
#define DbgTttDataGetMetrics(x) \
    TttDbgHelper("TttDataGetMetrics",tttDataDbgSet,0x200,x)
MsgHandlerWithTypes(TttDataGetMetrics, P_TTT_DATA_METRICS, PP_TTT_DATA_INST)
{
    DbgTttDataGetMetrics(("("))
    *pArgs = (*pData)->metrics;
    DbgTttDataGetMetrics(("returns stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* TttDataGetMetrics */

/*****************************************************************************
    TttDataSetMetrics
*****************************************************************************/
#define DbgTttDataSetMetrics(x) \
    TttDbgHelper("TttDataSetMetrics",tttDataDbgSet,0x400,x)
MsgHandlerWithTypes(TttDataSetMetrics, P_TTT_DATA_METRICS, PP_TTT_DATA_INST)
{
    BOOLEAN transactionOpen;
    STATUS  s;
    DbgTttDataSetMetrics(("("))
    // Steps:
    //      * Initialize for error recovery.
    //      * Begin the undo transaction.
    //      * Change the data and record undo information.
    //      * End the undo transaction.
    //      * Notify observers.
    //
    transactionOpen = false;
    ObjCallJmp(msgUndoBegin, theUndoManager, (P_ARGS)pArgs->undoTag, s, Error);
    transactionOpen = true;
    StsJmp(TttDataPrivSetMetrics(self, pArgs, pData, true), s, Error);
    ObjCallJmp(msgUndoEnd, theUndoManager, pNull, s, Error);
    transactionOpen = false;
    StsJmp(TttDataNotifyObservers(self, pNull), s, Error);
    DbgTttDataSetMetrics(("returns stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
```

```
Error:
    if (transactionOpen) {
        ObjCallJmp(msgUndoEnd, theUndoManager, pNull, s, Error);
                //
                // FIXME: This should abort, not end, the transaction.
                // The abort functionality should be available in M4.8.
                //
    }
    DbgTttDataSetMetrics(("Error; return 0x%lx",s))
    return s;
} /* TttDataSetMetrics */

/*****************************************************************************
    TttDataSetSquare

    Handles both msgTttDataSetMetrics and msgTttDataSetSquare
*****************************************************************************/
#define DbgTttDataSetSquare(x) \
    TttDbgHelper("TttDataSetSquare",tttDataDbgSet,0x800,x)
MsgHandlerWithTypes(TttDataSetSquare, P_TTT_DATA_SET_SQUARE, PP_TTT_DATA_INST)
{
    TTT_DATA_CHANGED    changed;
    BOOLEAN             transactionOpen;
    STATUS              s;
    DbgTttDataSetSquare(("row=%ld col=%ld value=%s", \
            (U32)(pArgs->row), (U32)(pArgs->col), \
            TttUtilStrForSquareValue(pArgs->value)))
    // Steps:
    //      * Initialize for error recovery.
    //      * Begin the undo transaction.
    //      * Change the data and record undo information.
    //      * End the undo transaction.
    //      * Notify observers.
    //
    transactionOpen = false;
    ObjCallJmp(msgUndoBegin, theUndoManager, pNull, s, Error);
    transactionOpen = true;
    StsJmp(TttDataPrivSetSquare(self, pArgs, pData, true), s, Error);
    ObjCallJmp(msgUndoEnd, theUndoManager, pNull, s, Error);
    transactionOpen = false;
    changed.row = pArgs->row;
    changed.col = pArgs->col;
    StsJmp(TttDataNotifyObservers(self, &changed), s, Error);
    DbgTttDataSetSquare(("returns stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    if (transactionOpen) {
        ObjCallJmp(msgUndoEnd, theUndoManager, pNull, s, Error);
                //
                // FIXME: This should abort, not end, the transaction.
                // The abort functionality should be available in M4.8.
                //
    }
    DbgTttDataSetSquare(("Error; return 0x%lx",s))
    return s;
} /* TttDataSetSquare */

/*****************************************************************************
    TttDataRead

    Handles msgTttDataRead
*****************************************************************************/
#define DbgTttDataRead(x) \
    TttDbgHelper("TttDataRead",tttDataDbgSet,0x1000,x)
#define N_BYTES 9
MsgHandlerWithTypes(TttDataRead, P_TTT_DATA_READ, PP_TTT_DATA_INST)
{
    STREAM_READ_WRITE   read;
    U8                  buf[N_BYTES+1];
    U16                 row;
    U16                 col;
    STATUS              s;
    DbgTttDataRead(("",))
    //
    // Read in the 9 bytes that must be present.  If there are fewer
    // than 9 bytes, treat the attempt to read as a failure.
    //
    read.numBytes = SizeOf(buf) - 1;
    read.pBuf = buf;
    ObjCallJmp(msgStreamRead, pArgs->fileHandle, &read, s, Error);
    buf[read.count] = '\0';
    DbgTttDataRead(("read.count=%ld buf=<%s>", (U32)read.count,buf))
    //
    // Now convert the buffer contents to reasonable square values.
    //
    for (row=0; row<3; row++) {
        for (col=0; col<3; col++) {
            int ch;
            ch = buf[(row*3)+col];
            if ((ch == 'X') OR (ch == 'x')) {
                (*pData)->metrics.squares[row][col] = tttX;
            } else if ((ch == 'O') OR (ch == 'o')) {
                (*pData)->metrics.squares[row][col] = tttO;
            } else {
                (*pData)->metrics.squares[row][col] = tttBlank;
            }
        }
    }
    StsJmp(TttDataNotifyObservers(self, pNull), s, Error);
    pArgs->successful = true;
```

```
    DbgTttDataRead(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttDataRead(("Error; return 0x%lx",s))
    return s;
} /* TttDataRead */

/*****************************************************************************
    TttDataUndoItem
*****************************************************************************/
#define DbgTttDataUndoItem(x) \
    TttDbgHelper("TttDataUndoItem",tttDataDbgSet,0x2000,x)
MsgHandlerWithTypes(TttDataUndoItem, P_UNDO_ITEM, PP_TTT_DATA_INST)
{
    STATUS  s;
    DbgTttDataUndoItem(("",))
    if (pArgs->subclass != clsTttData) {
        DbgTttDataUndoItem(("not clsTttData; give to ancestor"))
        return ObjectCallAncestorCtx(ctx);
    }
    StsJmp(TttDataPrivSetMetrics(self, pArgs->pData, pData, false),
           s, Error);
    StsJmp(TttDataNotifyObservers(self, pNull), s, Error);
    DbgTttDataUndoItem(("returns stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttDataUndoItem(("Error; return 0x%lx",s))
    return s;
} /* TttDataUndoItem */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Installation                                       *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************
    ClsTttDataInit

    Install the class.
*****************************************************************************/
STATUS PASCAL
ClsTttDataInit (void)
{
    CLASS_NEW       new;
    STATUS          s;
    ObjCallJmp(msgNewDefaults, clsClass, &new, s, Error);
    new.object.uid          = clsTttData;
    new.object.key          = 0;
    new.cls.pMsg            = clsTttDataTable;
    new.cls.ancestor        = clsObject;
    new.cls.size            = SizeOf(P_TTT_DATA_INST);
    new.cls.newArgsSize = SizeOf(TTT_DATA_NEW);
    ObjCallJmp(msgNew, clsClass, &new, s, Error);
    return stsOK;
Error:
    return s;
} /* ClsTttDataInit */
```

TTTDBG.C

```
/*****************************************************************************
 File: tttdbg.c
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.1  $
 $Date:   13 Nov 1991 18:38:00  $
 This file contains the implementation of miscellaneous debugging routines
 used by TttApp.  The interfaces to these routines are in tttpriv.h.
*****************************************************************************/
#ifndef WIN_INCLUDED
#include <win.h>
#endif
#ifndef VIEW_INCLUDED
#include <view.h>
#endif
#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif
#ifndef OS_INCLUDED
#include <os.h>
#endif
#ifndef APP_INCLUDED
#include <app.h>
#endif
#ifndef FRAME_INCLUDED
#include <frame.h>
#endif
#ifndef CLSMGR_INCLUDED
```

```
#include <clsmgr.h>
#endif
#ifndef TTTPRIV_INCLUDED
#include <tttpriv.h>
#endif
#ifndef TTTDATA_INCLUDED
#include <tttdata.h>
#endif
#include <string.h>
#include <ctype.h>

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Defines, Types, Globals, Etc                     *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Utility Routines                                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Message Handlers                                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************
    TttDbgDumpWindowTree
    Invoked from debugging menu item to display window tree.

    Self must be app.  If the user picked the first command
    in the submenu, pArgs will be 0, and this handler will
    dump from the app's mainWin.  Otherwise, it will dump from the app's
    mainWin's clientWin.
*****************************************************************************/
#ifdef DEBUG
MsgHandler(TttDbgDumpWindowTree)
{
    OBJECT      root;
    APP_METRICS am;
    STATUS      s;
    if (((U32)(pArgs)) == 0) {
        ObjCallJmp(msgAppGetMetrics, self, &am, s, Error);
        root = am.mainWin;
    } else {
        StsJmp(TttUtilGetComponents(self, tttGetView, objNull, &root, \
                objNull), s, Error);
    }
    ObjCallJmp(msgWinDumpTree, root, pNull, s, Error);
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    return s;
} /* TttDbgDumpWindowTree */
#endif // DEBUG

/*****************************************************************************
    TttDbgDumpView

    Self must be app.
*****************************************************************************/
#ifdef DEBUG
MsgHandler(TttDbgDumpView)
{
    VIEW    view;
    STATUS  s;
    StsJmp(TttUtilGetComponents(self, tttGetView, objNull, &view, objNull), \
            s, Error);
    ObjCallJmp(msgDump, view, pNull, s, Error);
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    return s;
} /* TttDbgDumpView */
#endif // DEBUG

/*****************************************************************************
    TttDbgDumpDataObject

    Self must be app.
*****************************************************************************/
#ifdef DEBUG
MsgHandler(TttDbgDumpDataObject)
{
    OBJECT      dataObject;
    STATUS      s;
    StsJmp(TttUtilGetComponents(self, tttGetDataObject, objNull, objNull, \
            &dataObject), s, Error);
    ObjCallJmp(msgDump, dataObject, pNull, s, Error);
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    return s;
} /* TttDbgDumpDataObject */
#endif // DEBUG
```

```
/*****************************************************************************
    TttDbgChangeTracing
*****************************************************************************/
#ifdef DEBUG
MsgHandler(TttDbgChangeTracing)
{
    OBJECT      view;
    OBJECT      dataObject;
    OBJECT      target;
    U32         args = (U32)pArgs;
    U16         targetArg = LowU16(args);
    BOOLEAN     turnTraceOn;
    STATUS      s;
    StsJmp(TttUtilGetComponents(self, tttGetView | tttGetDataObject, objNull, \
            &view, &dataObject), s, Error);
    turnTraceOn = (HighU16(args) == 1);
    if (targetArg == 0) {
        Debugf("Setting tracing of app %s", turnTraceOn ? "On" : "Off");
        target = self;
    } else if (targetArg == 1) {
        Debugf("Setting tracing of view %s", turnTraceOn ? "On" : "Off");
        target = view;
    } else {
        Debugf("Setting tracing of dataObject %s", turnTraceOn ? "On" : "Off");
        target = dataObject;
    }
    ObjCallWarn(msgTrace, target, (P_ARGS)turnTraceOn);
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    return s;
} /* TttDbgChangeTracing */
#endif // DEBUG

/*****************************************************************************
    TttDbgForceRepaint
*****************************************************************************/
#ifdef DEBUG
MsgHandler(TttDbgForceRepaint)
{
    OBJECT      view;
    STATUS      s;
    StsJmp(TttUtilGetComponents(self, tttGetView, objNull, &view, objNull), \
            s, Error);
    ObjCallJmp(msgWinDirtyRect, view, pNull, s, Error);
    ObjCallJmp(msgWinUpdate, view, pNull, s, Error);
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    return s;
} /* TttDbgForceRepaint */
#endif // DEBUG
```

TTTIPAD.C

```
/*****************************************************************************
 File: tttipad.c
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.1  $
 $Date:   13 Nov 1991 18:38:04  $
*****************************************************************************/
#ifndef TTTVIEW_INCLUDED
#include <tttview.h>
#endif
//   AKN - Function to create an insertion pad
STATUS
TttViewCreateIPad(
    P_GWIN_GESTURE  pGesture,
    P_TTT_VIEW_INST pInst,
    VIEW            self)
{
    WIN_METRICS         wm;
    WIN_METRICS         myMetrics;
    STATUS              s;
    IP_NEW              ipNew;
    P_XTEMPLATE         pNewTemplate;
    XLATE_NEW           xNewTrans;
    U16                 xlateFlags;
    U16                 i;
    OBJECT              dataObject;
    TTT_DATA_METRICS    dm;
    //
    // Get my data object and metrics to get existing string
    //
    ObjCallJmp(msgViewGetDataObject, self, &dataObject, s, Error);
    ObjCallJmp(msgTttDataGetMetrics, dataObject, &dm, s, Error);
    //
    // compute and set the hit row and col
    //
```

```
    ObjCallJmp(msgWinGetMetrics, self, &myMetrics, s, Error);
    for(i = 1; i <= numberOfColumns; i++) {
        if(pGesture->hotPoint.x <
                (i * (myMetrics.bounds.size.w / numberOfColumns))) {
            pInst->currentCell.col = i - 1;
            break;
        }
    }
    for(i = 1; i <= numberOfRows; i++) {
        if(pGesture->hotPoint.y <
                (i * (myMetrics.bounds.size.h / numberOfRows))) {
            pInst->currentCell.row = i - 1;
            break;
        }
    }
    //
    //  Create the insertion pad window and insert it as a child of my view
    //
    ObjectCall(msgNewDefaults, clsIP, &ipNew);
    ipNew.border.style.resize    = true;
    ipNew.ip.style.buttonType    = ipsBottomButtons;
    ipNew.ip.style.displayType   = ipsCharBoxButtons;
    ipNew.ip.style.embeddedLook  = false;
    ipNew.ip.style.freeAfter     = true;
    ipNew.ip.style.modal         = ipsNoMode;
    ipNew.ip.style.takeGrab      = true;
    ipNew.ip.style.takeFocus     = true;
    ipNew.ip.style.delayed       = false;
    ipNew.ip.rows = 1;
    ipNew.ip.cols = 10;
    ipNew.ip.maxLen = 10;
    ipNew.win.bounds.origin.x = pGesture->hotPoint.x - 135;
    ipNew.win.bounds.origin.y = pGesture->hotPoint.y - 45;
    ipNew.win.bounds.size.w = 270;
    ipNew.win.bounds.size.h = 90;
    // set the listener field for notifications
    ipNew.ip.client = self;
    // set the initial string from the model
    ipNew.ip.pString = "          ";
    ipNew.ip.pString[0] =
dm.squares[pInst->currentCell.row][pInst->currentCell.col];
    // Create a translator for the insertion pad
    ObjectCall(msgNewDefaults, clsXText, &xNewTrans);
    // Create a template for the new translator
    if (s = XTemplateCompile(xtmTypeCharList,
     0,
 "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz01234567890+-x*/.()^=",
 osProcessHeapId,
 &pNewTemplate) < stsOK ) {
return s;
    }
    xNewTrans.xlate.pTemplate = pNewTemplate;
    ObjCallRet(msgNew, clsXText, &xNewTrans, s);
    ipNew.ip.xlate.translator = xNewTrans.object.uid;
    //
    //  give char list veto power
    //
    ObjCallRet(msgXlateGetFlags, xNewTrans.object.uid, &xlateFlags, s);
    xlateFlags |= xTemplateVeto | spaceDisable;
    ObjCallRet(msgXlateSetFlags, xNewTrans.object.uid, (P_ARGS)xlateFlags, s);
    ObjCallRet(msgNew, clsIP, &ipNew, s);   // create the pad
    //
    //  Insert pad into the window
    //
    wm.parent    = self;
    wm.options   = wsPosTop;
    ObjCallRet(msgWinInsert, ipNew.object.uid, &wm, s);
    ObjectCall(msgWinSetLayoutDirty, ipNew.object.uid, &wm);
    ObjCallRet(msgWinLayout, ipNew.object.uid, &wm, s);
    return stsOK;
Error:
        if (ipNew.object.uid) {
            ObjCallWarn(msgDestroy, ipNew.object.uid, pNull);
        }
        return s;
}   /* CreateInsertWindow */

/*****************************************************************************
    TttViewGestureSetCell

    Handles all gestures that set the value of a single square.
*****************************************************************************/
#define DbgTttViewGestureSetCell(x) \
    TttDbgHelper("TttViewGestureSetSquare",tttViewDbgSet,0x2,x)
STATIC STATUS LOCAL
TttViewGestureSetCell(VIEW self, P_TTT_VIEW_INST pInst)
{
    OBJECT      dataObject;
    STATUS      s;
    ObjCallJmp(msgViewGetDataObject, self, &dataObject, s, Error);
    ObjCallJmp(msgTttDataSetSquare, dataObject, &pInst->currentCell, s, Error);
    DbgTttViewGestureSetCell(("return stsOK"))
    return stsOK;
Error:
    DbgTttViewGestureSetCell(("Error; returns 0x%lx",s))
    return s;
} /* TttViewGestureSetCell */
/* AKN - Method Support for insertion pad
```

```
/*****************************************************************************
     TttViewGetIPadData

     Method for processing the translated data reported by an IP.
     Used for msgIPDataAvailable and msgFieldModified
*****************************************************************************/
MsgHandlerWithTypes(TttViewGetIPadData, OBJECT, PP_TTT_VIEW_INST)
{
     IP_STYLE        ipStyle;
     IP_STRING       ipStr;
     P_STRING        pStr;
     P_STRING        pCopy;
     STATUS          s;
     P_TTT_VIEW_INST pInst;

     pInst = *pData;
     ipStr.len = 20;
     ipStr.pString = "                    ";

     ObjCallRet(msgIPGetXlateString, pArgs, &ipStr, s);
     ObjCallRet(msgIPSetString, pArgs, ipStr.pString, s);
     pInst->currentCell.value = (TTT_SQUARE_VALUE) ipStr.pString[0];
     StsJmp(TttViewGestureSetCell(self, pInst), s, Error);

     return (stsOK);

Error:
     return (s);

//   MsgHandlerParameterNoWarning;
}    // TttViewGetIPadData
```

TTTMBAR.C

```
/*****************************************************************************
 File: tttmbar.c

 Copyright 1990-1992 GO Corporation. All Rights Reserved.

 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.

 $Revision:   1.1  $
 $Date:   13 Nov 1991 18:38:06  $

 This file contains some of the implementation of TttApp's menu bar.
*****************************************************************************/
#ifndef TKTABLE_INCLUDED
#include <tktable.h>
#endif
#ifndef TTTPRIV_INCLUDED
#include <tttpriv.h>
#endif
#ifndef TTTAPP_INCLUDED
#include <tttapp.h>
#endif

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Defines, Types, Globals, Etc                     *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#ifdef DEBUG
static TK_TABLE_ENTRY dumpTreeMenu[] = {
  {"From Frame",  msgTttAppDumpWindowTree, 0},
  {"From View",       msgTttAppDumpWindowTree, 1},
  {pNull}
};

//
// Arguably this could be improved by showing making three exclusive
// choices, and displaying the current state in the menu.  But there's
// no way to get the trace state from the clsmgr, and even if there was,
// it's not terribly important to be that careful with debugging code.
//
static TK_TABLE_ENTRY traceMenu[] = {
  {"Trace App On",        msgTttAppChangeTracing, MakeU32(0,1)},
  {"Trace App Off",       msgTttAppChangeTracing, MakeU32(0,0)},
  {"Trace View On",       msgTttAppChangeTracing, MakeU32(1,1)},
  {"Trace View Off",      msgTttAppChangeTracing, MakeU32(1,0)},
  {"Trace Data On",       msgTttAppChangeTracing, MakeU32(2,1)},
  {"Trace Data Off",      msgTttAppChangeTracing, MakeU32(2,0)},
  {pNull}
};
static TK_TABLE_ENTRY debugMenu[] = {
  {"Dump View",           msgTttAppDumpView,          0},
  {"Dump Data",           msgTttAppDumpDataObject,    0},
  {"Dump App",            msgDump,                    0},
  {"Dump Window Tree",    (U32)dumpTreeMenu,          0, 0, tkMenuPullRight},
  {"Trace",               (U32)traceMenu,   0, 0, tkMenuPullRight |
     tkBorderEdgeTop},
  {"Force Repaint",       msgTttAppForceRepaint,      0, 0, tkBorderEdgeTop},
  {pNull}
};
#endif // DEBUG

TK_TABLE_ENTRY    tttRealMenuBar[] = {
#ifdef DEBUG
  {"Debug",   (U32)debugMenu, 0, 0, tkMenuPullDown},
#endif
  {pNull}
};
```

```
//
// Why two variables, one "real" and one not real?  The reason is that
// including including tktable.h (which is where TK_TABLE_ENTRY is
// defined) defines do many symbols that the compiler is overwhelmed.
// This bit of trickery allows clients to refer to tttMenuBar without
// including tktable.h
//
P_UNKNOWN
tttMenuBar = (P_UNKNOWN)tttRealMenuBar;
```

S_TTT.C

```
//
// WARNING:  File generated by GO utility: Gensyms.
// Do not edit.  Instead, run gensyms again.
//
#ifdef DEBUG
#include <clsmgr.h>
#include <tttpriv.h>
#include <tttapp.h>
#include <tttdata.h>
#include <tttview.h>
const CLS_SYM_STS tttStsSymbols[] = {
  0, 0};
const CLS_SYM_MSG tttMsgSymbols[] = {
  msgTttAppChangeDebugFlag,   "msgTttAppChangeDebugFlag",
  msgTttAppChangeDebugSet,    "msgTttAppChangeDebugSet",
  msgTttAppDumpWindowTree,    "msgTttAppDumpWindowTree",
  msgTttAppDumpDebugFlags,    "msgTttAppDumpDebugFlags",
  msgTttAppDumpView, "msgTttAppDumpView",
  msgTttAppDumpDataObject,    "msgTttAppDumpDataObject",
  msgTttAppChangeTracing, "msgTttAppChangeTracing",
  msgTttAppForceRepaint, "msgTttAppForceRepaint",
  tagTttDataUndoDelete,   "tagTttDataUndoDelete",
  tagTttDataUndoMoveCopy, "tagTttDataUndoMoveCopy",
  msgTttDataGetMetrics,   "msgTttDataGetMetrics",
  msgTttDataSetMetrics,   "msgTttDataSetMetrics",
  msgTttDataSetSquare,    "msgTttDataSetSquare",
  msgTttDataRead, "msgTttDataRead",
  msgTttDataChanged,  "msgTttDataChanged",
  tagTttViewOptionSheet,  "tagTttViewOptionSheet",
  tagTttViewCard, "tagTttViewCard",
  tagCardLineThickness,   "tagCardLineThickness",
  tagTttQHelpForView, "tagTttQHelpForView",
  tagTttQHelpForLineCtrl, "tagTttQHelpForLineCtrl",
  msgTttViewGetMetrics,   "msgTttViewGetMetrics",
  msgTttViewSetMetrics,   "msgTttViewSetMetrics",
  msgTttViewToggleSel,    "msgTttViewToggleSel",
  msgTttViewTakeSel,  "msgTttViewTakeSel",
  0, 0};
const CLS_SYM_OBJ tttObjSymbols[] = {
  clsTttApp,  "clsTttApp",
  clsTttData, "clsTttData",
  clsTttView, "clsTttView",
  0, 0};
STATUS EXPORTED TttSymbolsInit(void)
{
     return ClsSymbolsInit(
             "ttt",
             tttObjSymbols,
             tttMsgSymbols,
             tttStsSymbols);
}
#else
#include <go.h>
STATUS EXPORTED TttSymbolsInit(void)
{
     return stsOK;
}
#endif
```

TTTUTIL.C

```
/*****************************************************************************
 File: tttutil.c
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.1  $
 $Date:   13 Nov 1991 18:38:08  $
 This file contains the implementation of miscellaneous utility routines
 used by TttApp.  The interfaces to these routines are in tttpriv.h.
*****************************************************************************/
#ifndef GO_INCLUDED
#include <go.h>
#endif
#ifndef FS_INCLUDED
#include <fs.h>
#endif
#ifndef SWIN_INCLUDED
#include <swin.h>
#endif
#ifndef MENU_INCLUDED
```

```
#include <menu.h>
#endif
#ifndef APP_INCLUDED
#include <app.h>
#endif
#ifndef FRAME_INCLUDED
#include <frame.h>
#endif
#ifndef TTTPRIV_INCLUDED
#include <tttpriv.h>
#endif
#ifndef SWIN_INCLUDED
#include <swin.h>
#endif
#ifndef VIEW_INCLUDED
#include <view.h>
#endif
#ifndef NOTE_INCLUDED
#include <note.h>
#endif
#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif
#ifndef SEL_INCLUDED
#include <sel.h>
#endif
#ifndef APPTAG_INCLUDED
#include <apptag.h>
#endif
#include <string.h>
#include <stdio.h>
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Utility Routines                             *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************
     TttUtilStrForSquareValue
*****************************************************************************/
//
// Output is nicer if all strings have the same length.  Don't worry
// about the "Unknown" string -- it should never be output anyhow.
//
static P_STRING valueStrings[] = {
     "_", "X", "O", "Unknown"};
P_STRING PASCAL
TttUtilStrForSquareValue(
     TTT_SQUARE_VALUE    v)
{
     if (v == tttBlank) {
         return valueStrings[0];
     } else if (v == tttX) {
         return valueStrings[1];
     } else if (v == tttO) {
         return valueStrings[2];
     } else {
         return valueStrings[3];
     }
} /* TttUtilStrForSquareValue */

/*****************************************************************************
     TttUtilSquareValueForChar
*****************************************************************************/
TTT_SQUARE_VALUE PASCAL
TttUtilSquareValueForChar(
     char    ch)
{
     return ((TTT_SQUARE_VALUE) ch);
} /* TttUtilSquareValueForChar */

/*****************************************************************************
     TttUtilInsertUnique
     Utility routine that inserts and element into an array if
     it's not already in the array.  Assumes enough space in the array.
*****************************************************************************/
#define DbgTttUtilInsertUnique(x) \
     TttDbgHelper("TttUtilInsertUnique",tttUtilDbgSet,0x1,x)
STATIC void PASCAL
TttUtilInsertUnique(
     P_U32       pValues,
     P_U16       pCount,     // In: number of elements in pValues
                             // Out: new number of elements in pValues
     U32         new)
{
     U16         i;
     DbgTttUtilInsertUnique(("*pCount=%ld new=%ld=0x%lx", (U32)(*pCount), \
             (U32)new, (U32)new))
     for (i=0; i<*pCount; i++) {
         if (pValues[i] == new) {
             DbgTttUtilInsertUnique(("found it at %ld; returning",(U32)i))
             return;
         }
     }
     pValues[*pCount] = new;
     *pCount = *pCount + 1;
     DbgTttUtilInsertUnique(("didn't find it; new count=%ld",(U32)(*pCount)))
```

```
} /* TttUtilInsertUnique */

/*****************************************************************************
    TttUtilCreateScrollWin

    This is in a utility routine rather than in the caller because
    including swin.h brings in too many symbols.

    The example of scrolling used here is slightly artificial.
    This tells the scroll window that it can expand the Tic-Tac-Toe view
    beyond its desired size, but should not contract it.  Thus if the
    scroll win shrinks, the TttView will be scrollable.

    When an application does scrolling, the view should get the
    messages.  So we set the client window appropriately now.
*****************************************************************************/
STATUS PASCAL
TttUtilCreateScrollWin(
    OBJECT      clientWin,
    P_OBJECT    pScrollWin)
{
    SCROLL_WIN_NEW  scrollWinNew;
    STATUS          s;

    ObjCallJmp(msgNewDefaults, clsScrollWin, &scrollWinNew, s, Error);
    scrollWinNew.scrollWin.clientWin                    = clientWin;
    scrollWinNew.scrollWin.style.expandChildWidth       = true;
    scrollWinNew.scrollWin.style.expandChildHeight      = true;
    ObjCallJmp(msgNew, clsScrollWin, &scrollWinNew, s, Error);
    *pScrollWin = scrollWinNew.object.uid;
    return stsOK;
Error:
    return s;
} /* TttUtilCreateScrollWin */

/*****************************************************************************
    TttUtilCreateMenu

    This is in a utility routine rather than in the caller because
    including menu.h brings in too many symbols.
*****************************************************************************/
STATUS PASCAL
TttUtilCreateMenu(
    OBJECT          parent,
    OBJECT          client,
    P_UNKNOWN       pEntries,   // really  P_TK_TABLE_ENTRY pEntries
    P_OBJECT        pMenu)
{
    MENU_NEW        mn;
    STATUS          s;

    ObjCallJmp(msgNewDefaults, clsMenu, &mn, s, Error);
    mn.win.parent = parent;
    mn.tkTable.client = client;
    mn.tkTable.pEntries = (P_TK_TABLE_ENTRY)pEntries;
    ObjCallJmp(msgNew, clsMenu, &mn, s, Error);
    *pMenu = mn.object.uid;
    return stsOK;
Error:
    return s;
} /* TttUtilCreateMenu */

/*****************************************************************************
    TttUtilAdjustMenu

    "Adjusts" the menu by removing the items that this app
    does not support.

    Each menu that has an item removed from it must be layed out
    and "break adjusted."  The latter involves adjusting the border edges
    and margins for a menu with lines dividing it into sections.
    But because the Standard Application Menus can change, we don't want
    to compile in knowledge of which menus these are.  (The menu item's
    containing menu is  easy to find -- it's just the menu item's
    parent window.)

    The obvious approach is to simply lay out and break adjust
    the item's menu after removing the item.  Unfortunately
    this potentially results in laying out and break
    adjusting the same menu several times, since several of
    the items could be removed from the same menu.

    Our solution is to keep an array of all unique parents seen.
    The array is known to be no longer than than the array of
    disableTags, so space allocation is easy. TttUtilUniqueSort
    is used to do the unique insertion.  Finally, we run over
    the unique array and do the necessary operations on the
    menus.
*****************************************************************************/
#define DbgTttUtilAdjustMenu(x) \
    TttDbgHelper("TttUtilAdjustMenu",tttUtilDbgSet,0x2,x)
//
// Tags which correspond to the menu items that this application
// will not implement.
//
static TAG disableTags[] = {
    tagAppMenuSearch,
    tagAppMenuSpell,
    tagAppMenuExport};
#define N_DISABLE_TAGS (SizeOf(disableTags) / SizeOf(disableTags[0]))

STATUS PASCAL
TttUtilAdjustMenu(
    OBJECT      menuBar)
{
    WIN         parentMenus [N_DISABLE_TAGS];   // There are at most
                                                // one menu per tag.
    U16         parentMenuCount;
```

```
    WIN_METRICS wm;
    U16         i;
    OBJECT      o;
    STATUS      s;

    DbgTttUtilAdjustMenu(("{"))

    memset(parentMenus, 0, sizeof(parentMenus));
    parentMenuCount = 0;

    for (i=0; i<N_DISABLE_TAGS; i++) {
        DbgTttUtilAdjustMenu(("i=%ld t=0x%lx",(U32)i,(U32)(disableTags[i])))
        if ((o = (OBJECT)ObjCallWarn(msgWinFindTag, menuBar,
                (P_ARGS)(disableTags[i]))) != objNull) {
            ObjCallJmp(msgWinGetMetrics, o, &wm, s, Error);
            TttUtilInsertUnique((P_U32)parentMenus, &parentMenuCount,
                    (U32)(wm.parent));
            ObjCallWarn(msgDestroy, o, pNull);
            DbgTttUtilAdjustMenu(("destroyed it; parent=0x%lx",wm.parent))
        } else {
            DbgTttUtilAdjustMenu(("didn't find tag!"))
        }
    }
    //
    // Adjust the breaks and re-layout each affected menu
    //
    for (i=0; i<parentMenuCount; i++) {
        DbgTttUtilAdjustMenu(("i=%ld parent=0x%lx",(U32)i, parentMenus[i]))
        // pArgs of true tells menu to layout self.
        ObjCallJmp(msgMenuAdjustSections, parentMenus[i], (P_ARGS)true, \
                s, Error);
    }
    DbgTttUtilAdjustMenu(("returns stsOK"))
    return stsOK;
Error:
    DbgTttUtilAdjustMenu(("Error; returns 0x%lx",s))
    return s;
} /* TttUtilAdjustMenu */

/*****************************************************************************
    TttUtillWrite
*****************************************************************************/
STATUS PASCAL
TttUtilWrite(
    OBJECT              file,
    U32                 numBytes,
    P_UNKNOWN           pBuf)
{
    STREAM_READ_WRITE   write;

    write.numBytes = numBytes;
    write.pBuf = pBuf;
    return ObjCallWarn(msgStreamWrite, file, &write);
} /* TttUtilWrite */

/*****************************************************************************
    TttUtilWriteVersion

    Optimization Note: This could put in-line or converted to a macro.
*****************************************************************************/
STATUS PASCAL
TttUtilWriteVersion(
    OBJECT      file,
    TTT_VERSION version)
{
    return TttUtilWrite(file, SizeOf(version), &version);
} /* TttUtilWriteVersion */

/*****************************************************************************
    TttUtilRead
*****************************************************************************/
STATUS PASCAL
TttUtilRead(
    OBJECT      file,
    U32         numBytes,
    P_UNKNOWN   pBuf)
{
    STREAM_READ_WRITE   read;

    read.numBytes = numBytes;
    read.pBuf = pBuf;
    return ObjCallWarn(msgStreamRead, file, &read);
} /* TttUtilRead */

/*****************************************************************************
    TttUtilReadVersion
*****************************************************************************/
#define DbgTttUtilReadVersion(x) \
    TttDbgHelper("TttUtilReadVersion",tttUtilDbgSet,0x4,x)
STATUS PASCAL
TttUtilReadVersion(
    OBJECT              file,
    TTT_VERSION         minVersion,
    TTT_VERSION         maxVersion,
    P_TTT_VERSION       pVersion)
{
    STATUS              s;

    DbgTttUtilReadVersion(("min=%ld max=%ld",(U32)minVersion, (U32)maxVersion))
    StsJmp(TttUtilRead(file, SizeOf(*pVersion), pVersion), s, Error);
    if ((*pVersion < minVersion) OR (*pVersion > maxVersion)) {
        DbgTttUtilReadVersion(("version mismatch; v=%ld",(U32)(*pVersion)))
        s = stsIncompatibleVersion;
        goto Error;
    }
    DbgTttUtilReadVersion(("version=%ld; return stsOK",(U32)(*pVersion)))
    return stsOK;
```

```
Error:
    DbgTttUtilReadVersion(("Error; return 0x%lx",s))
    return s;
} /* TttUtilReadVersion */

/*****************************************************************************
    TttUtilGetComponents
    Note: this is an internal utility routine.  Therefore it does not
    check carefully for null values.
*****************************************************************************/
STATUS PASCAL
TttUtilGetComponents(
    OBJECT          app,
    U16             getFlags,
    P_OBJECT        pScrollWin,
    P_OBJECT        pView,
    P_OBJECT        pDataObject)
{
    OBJECT          view;
    APP_METRICS     am;
    OBJECT          clientWin;
    STATUS          s;
    //
    // Get the scrollWin regardless of the getFlags because we need
    // the scrollWin to get anything else and we assume the getFlags
    // aren't empty.
    //
    ObjCallJmp(msgAppGetMetrics, app, &am, s, Error);
    ObjCallJmp(msgFrameGetClientWin, am.mainWin, &clientWin, s, Error);
    if (FlagOn(tttGetScrollWin, getFlags)) {
        *pScrollWin = clientWin;
    }
    //
    // Do we need anything else?
    //
    if (FlagOn(tttGetView, getFlags) OR FlagOn(tttGetDataObject, getFlags)) {
        //
        // Get the view regardless of the getFlags because we need
        // the view to get either the view or the dataObject.
        //
        ObjCallJmp(msgScrollWinGetClientWin, clientWin, &view, s, Error);
        if (FlagOn(tttGetView, getFlags)) {
            *pView = view;
        }
        if (FlagOn(tttGetDataObject, getFlags)) {
            ObjCallJmp(msgViewGetDataObject, view, pDataObject, s, Error);
        }
    }
    return stsOK;
Error:
    return s;
} /* TttUtilGetComponents */

/*****************************************************************************
    TttUtilPostNotImplementedYet
    FIXME: In final product, this should not be needed!  And if for some
    reason it is needed, the string(s) in here, and all the strings this is
    called with, should be in some centralized spot!
*****************************************************************************/
#define DbgTttUtilPostNIY(x) \
    TttDbgHelper("TttUtilPostNotImplementedYet",tttUtilDbgSet,0x8,x)
static U8 formatStr[] = "Sorry.  %s is not implemented yet.";
#define MAX_FEATURE_STR_LEN 50
#define BUF_SIZE (SizeOf(formatStr) + MAX_FEATURE_STR_LEN + 5)
STATUS PASCAL
TttUtilPostNotImplementedYet(
    P_STRING        featureStr)
{
    NOTE_NEW        noteNew;
    TK_TABLE_ENTRY  tkEntry[2];
    U8              buf[BUF_SIZE];
    STATUS          s;
    DbgTttUtilPostNIY(("featureStr=<%s>",featureStr))
    //
    // Initialize for error recovery.
    //
    noteNew.object.uid = objNull;
    //
    // Set up the note string.  Check for overflow first.
    //
    if (strlen(featureStr) > MAX_FEATURE_STR_LEN) {
        s = stsBadParam;
        goto Error;
    }
    sprintf(buf,formatStr,featureStr);
    DbgTttUtilPostNIY(("buf=<%s>",buf))
    //
    // Set up tk table.  Tk tables are null terminated, which is why
    // we have two entries with the second one null.
    //
    memset(&tkEntry[0], 0, sizeof(TK_TABLE_ENTRY));
    memset(&tkEntry[1], 0, sizeof(TK_TABLE_ENTRY));
    tkEntry[0].arg1 = buf;
    //
    // Create and show the note.  Because the nfAutoDestroy flag is set,
    // we don't have to do anything about freeing the resources later.
    //
    ObjCallJmp(msgNewDefaults, clsNote, &noteNew, s, Error);
    noteNew.note.metrics.flags = nfAutoDestroy | nfDefaultAppFlags;
```

```
    noteNew.note.pContentEntries = tkEntry;
    ObjCallJmp(msgNew, clsNote, &noteNew, s, Error);
    ObjCallJmp(msgNoteShow, noteNew.object.uid, pNull, s, Error);
    return stsOK;
Error:
    if (noteNew.object.uid) {
        ObjCallWarn(msgDestroy, noteNew.object.uid, pNull);
    }
    return s;
} /* TttUtilPostNotImplementedYet */

/*****************************************************************************
    TttUtilGiveUpSel
    This utility routine encapsulates the steps need to give up the selection.
    Whenever this app gives up the selection, it also gives up the focus.
*****************************************************************************/
#define DbgTttUtilGiveUpSel(x) \
    TttDbgHelper("TttUtilGiveUpSel",tttUtilDbgSet,0x10,x)
STATUS PASCAL
TttUtilGiveUpSel(
    OBJECT      object)
{
    STATUS      s;

    DbgTttUtilGiveUpSel(("",))
    ObjCallJmp(msgSelSetOwner, theSelectionManager, pNull, \
            s, Error);
    if (object == InputGetTarget()) {
        StsJmp(InputSetTarget(objNull, 0L), s, Error);
                // FIXME: Put correct flags in here.
    }
    DbgTttUtilGiveUpSel(("return stsOK"))
    return stsOK;
Error:
    DbgTttUtilGiveUpSel(("Error; returns 0x%lx",s))
    return s;
} /* TttUtilGiveUpSel */


/*****************************************************************************
    TttUtilTakeSel
    This utility routine encapsulates the steps need to take the selection.
    Whenever this app takes the selection, it also takes the focus.
*****************************************************************************/
#define DbgTttUtilTakeSel(x) \
    TttDbgHelper("TttUtilTakeSel",tttUtilDbgSet,0x20,x)
STATUS PASCAL
TttUtilTakeSel(
    OBJECT      object)
{
    STATUS      s;
    DbgTttUtilTakeSel(("",))
    ObjCallJmp(msgSelSetOwner, theSelectionManager, (P_ARGS)object, \
            s, Error);
    StsJmp(InputSetTarget(object, 0L), s, Error);
                // FIXME:  These are almost certainly not the right flags.
    DbgTttUtilTakeSel(("return stsOK"))
    return stsOK;
Error:
    DbgTttUtilTakeSel(("Error; returns 0x%lx",s))
    return s;
} /* TttUtilTakeSel */


/*****************************************************************************
    TttUtilInitTextOutput
*****************************************************************************/
void PASCAL
TttUtilInitTextOutput(
    P_SYSDC_TEXT_OUTPUT p,
    U16                 align,
    P_U8                buf)
{
    memset(p, 0, sizeof(*p));
    p->spaceChar = (U16)' ';
    p->stop = maxS32;
    p->alignChr = align;
    p->pText = buf;
    if (buf) {
        p->lenText = strlen(buf);
    }
} /* TttUtilInitTextOutput */
```

TTTVIEW.H

```
/*****************************************************************************
 File: tttview.h
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.3  $
 $Date:   04 Dec 1991 13:39:14  $
 This file contains the API definition for clsTttView.
 clsTttView inherits from clsView.
```

```
 clsTttView displays a representation of clsTttData as a grid of Xs and Os.
*****************************************************************************/
#ifndef TTTVIEW_INCLUDED
#define TTTVIEW_INCLUDED

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef WIN_INCLUDED
#include <win.h>
#endif

#ifndef VIEW_INCLUDED
#include <view.h>
#endif

#ifndef SYSGRAF_INCLUDED
#include <sysgraf.h>
#endif

#ifndef TTTPRIV_INCLUDED
#include <tttpriv.h>
#endif

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Defines                                        *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

//
// Tags used by the view's option sheet.
//
#define tagTttViewOptionSheet MakeTag(clsTttView, 0)
#define tagTttViewCard        MakeTag(clsTttView, 1)
#define tagCardLineThickness  MakeTag(clsTttView, 2)

//
//    Tags for TTT's quick help.
//
#define  tagTttQHelpForView      MakeTag(clsTttView, 3)
#define  tagTttQHelpForLineCtrl  MakeTag(clsTttView, 4)
// tagCardLineTickness (defined above) is also used for quick help.


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Common Typedefs                                *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

typedef OBJECT
TTT_VIEW, * P_TTT_VIEW;

typedef struct {
     U32     lineThickness;
     U32     spare1;
     U32     spare2;
} TTT_VIEW_METRICS, * P_TTT_VIEW_METRICS,
  TTT_VIEW_NEW_ONLY, * P_TTT_VIEW_NEW_ONLY;

typedef struct TTT_VIEW_INST {
     TTT_VIEW_METRICS        metrics;
     SYSDC                   dc;
     // AKN - currentCell is used to hold hit row/col
     TTT_DATA_SET_SQUARE currentCell;
     RECT32                  selectedRange;
} TTT_VIEW_INST,
  * P_TTT_VIEW_INST,
  * * PP_TTT_VIEW_INST;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Private Functions                              *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************
 TttViewOptions  returns STATUS
     Called in response to msgSelOptions and the "Check" gesture.
*/
STATUS PASCAL
TttViewOptions(
     OBJECT             self,
     PP_TTT_VIEW_INST   pData);

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Exported Functions                             *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************
 ClsTttViewInit  returns STATUS
     Initializes / installs clsTttView.
     This routine is only called during installation of the class.
*/
STATUS PASCAL
ClsTttViewInit (void);


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Messages for clsTttView                        *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************
 msgNew      takes P_TTT_VIEW_NEW, returns STATUS
     category: class message
     Creates an instance of clsTttView.
*/

#define  tttViewNewFields    \
     viewNewFields           \
     TTT_VIEW_NEW_ONLY       tttView;

typedef struct TTT_VIEW_NEW {
     tttViewNewFields
} TTT_VIEW_NEW, * P_TTT_VIEW_NEW;
```

```
/*****************************************************************************
 msgNewDefaults      takes P_TTT_VIEW_NEW, returns STATUS
     category: class message
     Initializes the TTT_VIEW_NEW structure to default values.
     pArgs->view.createDataObject = true;
     pArgs->tttView.lineThickness = 5L;
     pArgs->tttView.spare1 = 0;
     pArgs->tttView.spare2 = 0;
*/
/*****************************************************************************
 msgTttViewGetMetrics takes P_TTT_VIEW_METRICS, returns STATUS
     Gets TTT_VIEW metrics.
*/
#define msgTttViewGetMetrics            MakeMsg(clsTttView, 0)

/*****************************************************************************
 msgTttViewSetMetrics takes P_TTT_VIEW_METRICS, returns STATUS
     Sets the TTT_VIEW metrics.
*/
#define msgTttViewSetMetrics            MakeMsg(clsTttView, 1)

/*****************************************************************************
 msgTttViewToggleSel     takes nothing, returns STATUS
     Causes the view to toggle whether or not it holds the selection.
*/
#define msgTttViewToggleSel                 MakeMsg(clsTttView, 2)

/*****************************************************************************
 msgTttViewTakeSel          takes nothing, returns STATUS
     Causes the view to toggle whether or not it holds the selection.
*/
#define msgTttViewTakeSel               MakeMsg(clsTttView, 3)
#endif   // TTTVIEW_INCLUDED
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      AKN - defines for constants (r,w)                  *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#define  numberOfRows                       5
#define  numberOfColumns                5
```

TTTVIEW.C

```
/*****************************************************************************
 File: tttview.c
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.6  $
 $Date:   07 Jan 1992 17:18:56  $
 This file contains the implementation of clsTttView.
*****************************************************************************/
#ifndef CONTROL_INCLUDED
#include <control.h>
#endif
#ifndef RESFILE_INCLUDED
#include <resfile.h>
#endif
#ifndef XGESTURE_INCLUDED
#include <xgesture.h>
#endif
#ifndef PEN_INCLUDED
#include <pen.h>
#endif
#ifndef STDIO_INCLUDED
#include <stdio.h>
#endif
#ifndef SYSGRAF_INCLUDED
#include <sysgraf.h>
#endif
#ifndef TTTDATA_INCLUDED
#include <tttdata.h>
#endif
#ifndef TTTVIEW_INCLUDED
#include <tttview.h>
#endif
#ifndef TTTPRIV_INCLUDED
#include <tttpriv.h>
#endif
#ifndef OSHEAP_INCLUDED
#include <osheap.h>
#endif
#ifndef PREFS_INCLUDED
#include <prefs.h>
#endif
#ifndef SEL_INCLUDED
#include <sel.h>
#endif
#ifndef KEY_INCLUDED
#include <key.h>
#endif
#include <string.h>
#include <methods.h>
```

```
#ifndef APPTAG_INCLUDED
#include <apptag.h>
#endif
#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Defines, Types, Globals, Etc                       *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

//
// CURRENT_VERSION is the file format version written by this implementation.
// MIN_VERSION is the minimum file format version readable by this
// implementation.  MAX_VERSION is the maximum file format version readable
// by this implementation.
//
#define CURRENT_VERSION 0
#define MIN_VERSION     0
#define MAX_VERSION     0
//
// The desired size for the tic-tac-toe view
//
#define desiredWidth  200
#define desiredHeight 200
typedef struct TTT_VIEW_FILED_0 {
    U32 lineThickness;
} TTT_VIEW_FILED_0, * P_TTT_VIEW_FILED_0;

//
// This struct defines the positions and sizes of all of the interesting
// pieces of the window.
//
// It is a relatively large structure, but since it is only used as a
// stack variable, its size isn't of much concern.
//
typedef struct {
    U32     normalBoxWidth;
    U32     normalBoxHeight;
    RECT32  vertLines[2];
    RECT32  horizLines[2];
    RECT32  r[3][3];
    SCALE   scale;

} TTT_VIEW_SIZES, * P_TTT_VIEW_SIZES;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Utility Routines                                   *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
```



```
/*****************************************************************************
    TttViewFiledData0FromInstData
    Computes filed data from instance data.
*****************************************************************************/
STATIC void PASCAL
TttViewFiledData0FromInstData(
    P_TTT_VIEW_INST     pInst,
    P_TTT_VIEW_FILED_0  pFiled)
{
    pFiled->lineThickness = pInst->metrics.lineThickness;
} /* TttViewFiledData0FromInstData */

/*****************************************************************************
    TttViewInstDataFromFiledData0
    Computes instance data from filed data.
*****************************************************************************/
STATIC void PASCAL
TttViewInstDataFromFiledData0(
    P_TTT_VIEW_FILED_0  pFiled,
    P_TTT_VIEW_INST     pInst)
{
    pInst->metrics.lineThickness = pFiled->lineThickness;
} /* TttViewInstDataFromFiledData0 */

/*****************************************************************************
    TttViewNeedRepaint
    Marks for repaint.
*****************************************************************************/
STATIC STATUS PASCAL
TttViewNeedRepaint(
    OBJECT  self)
{
    return ObjCallWarn(msgWinDirtyRect, self, pNull);
} /* TttViewNeedRepaint */

/*****************************************************************************
    TttViewCreateDC

    Constructs and initializes dc.
    This routine uses the system font as of the time the dc is
    created.  No effort is made to track changes to the system font.
*****************************************************************************/
#define DbgTttViewCreateDC(x) \
    TttDbgHelper("TttViewCreateDC",tttViewDbgSet,0x1,x)
STATIC STATUS PASCAL
TttViewCreateDC(
    OBJECT          self,
    P_OBJECT        pDC)
{
    SYSDC_NEW           dcNew;
```

```
    RES_READ_DATA       resRead;
    PREF_SYSTEM_FONT    font;
    STATUS              s;
    DbgTttViewCreateDC(("self=0x%lx",self))
    //
    // Initialize for error recovery
    //
    dcNew.object.uid = objNull;
    *pDC = objNull;
    //
    // Create the dc
    //
    ObjCallJmp(msgNewDefaults, clsSysDrwCtx, &dcNew, s, Error);
    ObjCallJmp(msgNew, clsSysDrwCtx, &dcNew, s, Error);
    //
    // Set the dc's font to the current system font.
    //
    resRead.resId = prSystemFont;
    resRead.heap = Nil(OS_HEAP_ID);
    resRead.pData = &font;
    resRead.length = SizeOf(font);
    ObjCallJmp(msgResReadData, theSystemPreferences, &resRead, s, Error);
    ObjCallJmp(msgDcOpenFont, dcNew.object.uid, &(font.spec), s, Error);
    //
    // Bind dc to self
    //
    ObjCallJmp(msgDcSetWindow, dcNew.object.uid, (P_ARGS)self, s, Error);
    *pDC = dcNew.object.uid;
    DbgTttViewCreateDC(("return stsOK"))
    return stsOK;
Error:
    if (dcNew.object.uid) {
        ObjCallWarn(msgDestroy, dcNew.object.uid, pNull);
    }
    DbgTttViewCreateDC(("Error; returns 0x%lx",s))
    return s;
} /* TttViewCreateDC */

/*****************************************************************************
    TttViewGestureSetSquare

    Handles all gestures that set the value
    of a single square.
*****************************************************************************/
#define DbgTttViewGestureSetSquare(x) \
    TttDbgHelper("TttViewGestureSetSquare",tttViewDbgSet,0x2,x)
STATIC STATUS PASCAL
TttViewGestureSetSquare(
    VIEW                    self,
    P_GWIN_GESTURE          pGesture,
    TTT_SQUARE_VALUE        value)      // Either tttX or tttO
{
    TTT_DATA_SET_SQUARE     set;
    OBJECT                  dataObject;
    WIN_METRICS             wm;
    STATUS                  s;
    DbgTttViewGestureSetSquare(("hot=[%ld %ld]",pGesture->hotPoint))
    //
    // Compute row and col
    //
    ObjCallJmp(msgWinGetMetrics, self, &wm, s, Error);
    if (pGesture->hotPoint.x < (wm.bounds.size.w / 3)) {
        set.col = 0;
    } else if (pGesture->hotPoint.x < (2 * (wm.bounds.size.w / 3))) {
        set.col = 1;
    } else {
        set.col = 2;
    }
    if (pGesture->hotPoint.y < (wm.bounds.size.h / 3)) {
        set.row = 0;
    } else if (pGesture->hotPoint.y < (2 * (wm.bounds.size.h / 3))) {
        set.row = 1;
    } else {
        set.row = 2;
    }
    //
    // Set new square value.
    //
    set.value = value;
    ObjCallJmp(msgViewGetDataObject, self, &dataObject, s, Error);
    ObjCallJmp(msgTttDataSetSquare, dataObject, &set, s, Error);
    DbgTttViewGestureSetSquare(("return stsOK"))
    return stsOK;
Error:
    DbgTttViewGestureSetSquare(("Error; returns 0x%lx",s))
    return s;
} /* TttViewGestureSetSquare */

/*****************************************************************************
    TttViewInitAndRestoreCommon

    Has code common to Init and Restore
*****************************************************************************/
#define DbgTttViewInitAndRestoreCommon(x) \
    TttDbgHelper("TttViewInitAndRestoreCommon",tttViewDbgSet,0x4,x)
STATIC STATUS PASCAL
TttViewInitAndRestoreCommon(
    VIEW            self,
    P_TTT_VIEW_INST pInst)
```

```
{
    STATUS          s;
    DbgTttViewInitAndRestoreCommon(("self=0x%lx",self))
    //
    // Initialize for Error Recovery
    //
    pInst->dc = objNull;
    //
    // Recreate the dc
    //
    StsJmp(TttViewCreateDC(self, &(pInst->dc)), s, Error);
    DbgTttViewInitAndRestoreCommon(("return stsOK"))
    return stsOK;
Error:
    DbgTttViewInitAndRestoreCommon(("Error; returns 0x%lx",s))
    return s;
} /* TttViewInitAndRestoreCommon */

/*****************************************************************************
    TttViewSingleKey
    Utility routine to handle single key.

    Actions:
        * x sets upper left cell to X
        * y sets upper left cell to Y
        * space sets upper left cell to Blank
*****************************************************************************/
#define DbgTttViewSingleKey(x) \
    TttDbgHelper("TttViewSingleKey",tttViewDbgSet,0x8,x)
STATIC STATUS PASCAL
TttViewSingleKey(
    OBJECT              self,
    U16                 keyCode)
{
    TTT_DATA_SET_SQUARE set;
    OBJECT              dataObject;
    BOOLEAN             inputIgnored;
    STATUS              s;
    DbgTttViewSingleKey(("keyCode=%ld=%c", (U32)keyCode, (U8)keyCode))
    ObjCallJmp(msgViewGetDataObject, self, &dataObject, s, Error);
    inputIgnored = TRUE;
    if ((keyCode == 'x') OR (keyCode == 'X')) {
        inputIgnored = FALSE;
        set.value = tttX;
    } else if ((keyCode == 'o') OR (keyCode == 'O')) {
        inputIgnored = FALSE;
        set.value = tttO;
    } else if (keyCode == ' ') {
        inputIgnored = FALSE;
        set.value = tttBlank;
    }
    if (inputIgnored) {
        s = stsInputIgnored;
    } else {
        set.row = 2;
        set.col = 0;
        ObjCallJmp(msgTttDataSetSquare, dataObject, &set, s, Error);
        s = stsInputTerminate;
    }
    DbgTttViewSingleKey(("return 0x%lx",(U32)s))
    return s;
Error:
    DbgTttViewSingleKey(("Error; return 0x%lx",s))
    return s;
} /* TttViewSingleKey */

/*****************************************************************************
    TttViewKeyInput
    Utility routine that handles all clsKey input events.
    Assumes ancestor is not interested in keyboard input.
    Note that one and only one of msgKeyMulti and msgKeyChar should be
    handled.
*****************************************************************************/
#define DbgTttViewKeyInput(x) \
    TttDbgHelper("TttViewKeyInput",tttViewDbgSet,0x10,x)
STATIC STATUS PASCAL
TttViewKeyInput(
    OBJECT          self,
    P_INPUT_EVENT   pArgs)
{
    STATUS          s;
    DbgTttViewKeyInput(("self=0x%lx",self))
    ASSERT((ClsNum(pArgs->devCode) == ClsNum(clsKey)), \
            "KeyInput gets wrong cls");
    if (MsgNum(pArgs->devCode) == MsgNum(msgKeyMulti)) {
        U16 i;
        U16 j;
        P_KEY_DATA pKeyData = (P_KEY_DATA)(pArgs->eventData);
        for (i=0; i < pKeyData->repeatCount; i++) {
            for (j=0; j < pKeyData->multi[i].repeatCount; j++) {
                s = TttViewSingleKey(self, pKeyData->multi[i].keyCode);
                if (s < stsOK) {
                    break;
                }
            }
            if (s < stsOK) {
                break;
            }
```

```
        }
    } else {
        s = stsInputIgnored;
    }
    DbgTttViewKeyInput(("return 0x%lx",s))
    return s;
} /* TttViewKeyInput */

/*****************************************************************************
    TttViewPenInput
    Utility routine that handles all clsPen input events.
*****************************************************************************/
#define DbgTttViewPenInput(x) \
    TttDbgHelper("TttViewPenInput",tttViewDbgSet,0x20,x)
STATIC STATUS PASCAL
TttViewPenInput(
    MESSAGE             msg,
    OBJECT              self,
    P_INPUT_EVENT       pArgs,
    CONTEXT             ctx,
    PP_TTT_VIEW_INST    pData)
{
    STATUS              s;
    DbgTttViewPenInput(("self=0x%lx",self))
    ASSERT((ClsNum(pArgs->devCode) == ClsNum(clsPen)), "PenInput gets wrong cls");
    if (MsgNum(pArgs->devCode) == MsgNum(msgPenHoldTimeout)) {
        P_PEN_DATA pPen = (P_PEN_DATA)(pArgs->eventData);
        ObjCallJmp(msgTttViewTakeSel, self, pNull, s, Error);
        ObjCallJmp(msgWinUpdate, self, pNull, s, Error);
        ObjCallJmp((pPen->taps == 0) ? msgSelBeginMove : msgSelBeginCopy,
                self, &pArgs->xy, s, Error);
        s = stsInputTerminate;
    } else {
        s = ObjectCallAncestorCtx(ctx);
    }
    DbgTttViewPenInput(("return 0x%lx",s))
    return s;
Error:
    DbgTttViewPenInput(("Error; returns 0x%lx",s))
    return s;

    Unused(msg);Unused(pData);
} /* TttViewPenInput */

/*****************************************************************************
    TttViewComputeSizes
*****************************************************************************/
#define DbgTttViewComputeSizes(x) \
    TttDbgHelper("TttViewComputeSizes",tttViewDbgSet,0x40,x)
#define FONT_SIZE_FUDGE 5
STATIC void PASCAL
TttViewComputeSizes(
    P_TTT_VIEW_SIZES    p,
    PP_TTT_VIEW_INST    pData,
    P_RECT32            pBounds)    // window bounds
{
    U32                 thickness;
    S16                 t;
    DbgTttViewComputeSizes(("bounds=[%ld %ld %ld %ld]", *pBounds))
    thickness = Max(1L, (*pData)->metrics.lineThickness);
    p->normalBoxWidth = Max(1L, (pBounds->size.w - (2 * thickness)) / 3);
    p->normalBoxHeight = Max(1L, (pBounds->size.h - (2 * thickness)) / 3);
    //
    // x and width of horiztonal stripes
    //
    p->horizLines[0].origin.x =
    p->horizLines[1].origin.x = 0;
    p->horizLines[0].size.w =
    p->horizLines[1].size.w = Max(1L, pBounds->size.w);
    //
    // y and height of vertical stripes
    //
    p->vertLines[0].origin.y =
    p->vertLines[1].origin.y = 0;
    p->vertLines[0].size.h =
    p->vertLines[1].size.h = Max(1L, pBounds->size.h);
    //
    // x and width of left column.
    //
    p->r[0][0].origin.x =
    p->r[1][0].origin.x =
    p->r[2][0].origin.x = 0;
    p->r[0][0].size.w =
    p->r[1][0].size.w =
    p->r[2][0].size.w = p->normalBoxWidth;
    //
    // x and width of left vertical stripe.
    //
    p->vertLines[0].origin.x = p->r[0][0].size.w;
    p->vertLines[0].size.w = thickness;
    //
    // x and width of middle column.
    //
    p->r[0][1].origin.x =
    p->r[1][1].origin.x =
    p->r[2][1].origin.x = p->vertLines[0].origin.x + p->vertLines[0].size.w;
    p->r[0][1].size.w =
```

```
p->r[1][1].size.w =
p->r[2][1].size.w = p->normalBoxWidth;
//
// x and width of right vertical stripe.
//
p->vertLines[1].origin.x = p->r[0][1].origin.x + p->r[0][1].size.w;
p->vertLines[1].size.w = thickness;
//
// x and width of right column.  Accumlate all extra width here.
//
p->r[0][2].origin.x =
p->r[1][2].origin.x =
p->r[2][2].origin.x = p->vertLines[1].origin.x + p->vertLines[1].size.w;
p->r[0][2].size.w =
p->r[1][2].size.w =
p->r[2][2].size.w = Max(1L, (pBounds->size.w -
        (p->vertLines[0].size.w + p->vertLines[1].size.w +
        p->r[0][0].size.w + p->r[0][1].size.w)));
//
// y and height of bottom row.
//
p->r[0][0].origin.y =
p->r[0][1].origin.y =
p->r[0][2].origin.y = 0;
p->r[0][0].size.h =
p->r[0][1].size.h =
p->r[0][2].size.h = p->normalBoxHeight;
//
// y and height of bottom horizontal stripe.
//
p->horizLines[0].origin.y = p->r[0][0].size.h;
p->horizLines[0].size.h = thickness;
//
// y and height of middle row.
//
p->r[1][0].origin.y =
p->r[1][1].origin.y =
p->r[1][2].origin.y = p->horizLines[0].origin.y + p->horizLines[0].size.h;
p->r[1][0].size.h =
p->r[1][1].size.h =
p->r[1][2].size.h = p->normalBoxHeight;
//
// y and height of top horizontal stripe.
//
p->horizLines[1].origin.y = p->r[1][0].origin.y + p->r[1][0].size.h;
p->horizLines[1].size.h = thickness;
//
// y and height of top row.  Accumulate all extra height here.
//
p->r[2][0].origin.y =
    p->r[2][1].origin.y =
    p->r[2][2].origin.y = p->horizLines[1].origin.y + p->horizLines[1].size.h;
    p->r[2][0].size.h =
    p->r[2][1].size.h =
    p->r[2][2].size.h = Max(1L, (pBounds->size.h -
            (p->horizLines[0].size.h + p->horizLines[1].size.h +
            p->r[0][0].size.h + p->r[1][0].size.h)));
    //
    // Compute font scale info.
    //
    if ((p->normalBoxWidth - FONT_SIZE_FUDGE) > 0) {
        t = (S16)(p->normalBoxWidth - FONT_SIZE_FUDGE);
    } else {
        t = 0;
    }
    p->scale.x = FxMakeFixed(t, 0);
    if ((p->normalBoxHeight - FONT_SIZE_FUDGE) > 0) {
        t = (S16)(p->normalBoxHeight - FONT_SIZE_FUDGE);
    } else {
        t = 0;
    }
    p->scale.y = FxMakeFixed(t, 0);
    DbgTttViewComputeSizes(("nBW=%ld nBH=%ld",p->normalBoxWidth,
p->normalBoxHeight))
    DbgTttViewComputeSizes(("vert [%ld %ld %ld %ld]  [%ld %ld %ld %ld]",
            p->vertLines[0], p->vertLines[1]));
    DbgTttViewComputeSizes(("horiz [%ld %ld %ld %ld]  [%ld %ld %ld %ld]",
            p->horizLines[0], p->horizLines[1]));
    DbgTttViewComputeSizes((
            "r0 [%ld %ld %ld %ld]  [%ld %ld %ld %ld]  [%ld %ld %ld %ld]",
            p->r[0][0], p->r[0][1], p->r[0][2]));
    DbgTttViewComputeSizes((
            "r1 [%ld %ld %ld %ld]  [%ld %ld %ld %ld]  [%ld %ld %ld %ld]",
            p->r[1][0], p->r[1][1], p->r[1][2]));
    DbgTttViewComputeSizes((
            "r2 [%ld %ld %ld %ld]  [%ld %ld %ld %ld]  [%ld %ld %ld %ld]",
            p->r[2][0], p->r[2][1], p->r[2][2]));
} /* TttViewComputeSizes */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Message Handlers                               *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************
    TttViewNewDefaults

    Respond to msgNewDefaults.
*****************************************************************************/
#define DbgTttViewNewDefaults(x) \
    TttDbgHelper("TttViewNewDefaults",tttViewDbgSet,0x80,x)
```

```
MsgHandlerWithTypes(TttViewNewDefaults, P_TTT_VIEW_NEW, PP_TTT_VIEW_INST)
{
    DbgTttViewNewDefaults(("self=0x%lx",self))
    pArgs->win.flags.input |= inputHoldTimeout;
    pArgs->gWin.helpId = tagTttQHelpForView;
    pArgs->embeddedWin.style.moveable = true;
    pArgs->embeddedWin.style.copyable = true;
    pArgs->view.createDataObject = true;
    pArgs->tttView.lineThickness = 5L;
    pArgs->tttView.spare1 = 0;
    pArgs->tttView.spare2 = 0;
    DbgTttViewNewDefaults(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* TttViewNewDefaults */

/*****************************************************************************
    TttViewInit

    Initialize instance data of new object.

    Note: clsmgr has already initialized instance data to zeros.
*****************************************************************************/
#define DbgTttViewInit(x) \
    TttDbgHelper("TttViewInit",tttViewDbgSet,0x100,x)
MsgHandlerWithTypes(TttViewInit, P_TTT_VIEW_NEW, PP_TTT_VIEW_INST)
{
    P_TTT_VIEW_INST pInst;
    TTT_DATA_NEW    tttDataNew;
    STATUS          s;
    BOOLEAN         responsibleForDataObject;
    DbgTttViewInit(("self=0x%lx",self))
    //
    // Initialize for error recovery
    //
    pInst = pNull;
    tttDataNew.object.uid = objNull;
    responsibleForDataObject = false;
    //
    // Allocate instance data and initialize those parts of it
    // that come from pArgs.
    //
    StsJmp(OSHeapBlockAlloc(osProcessHeapId, SizeOf(*pInst), &pInst), \
            s, Error);
    pInst->metrics.lineThickness = pArgs->tttView.lineThickness;
    //
    // Create the data object, if appropriate.
    //
    if ((pArgs->view.dataObject == Nil(OBJECT)) AND
            (pArgs->view.createDataObject)) {
        ObjCallJmp(msgNewDefaults, clsTttData, &tttDataNew, s, Error);
        ObjCallJmp(msgNew, clsTttData, &tttDataNew, s, Error);
        responsibleForDataObject = true;
        pArgs->view.createDataObject = false;
        pArgs->view.dataObject = tttDataNew.object.uid;
    }
    //
    // Now let ancestor finish initializing self.
    // clsView will make self an observer of the data object.
    //
    ObjCallAncestorCtxJmp(ctx, s, Error);
    responsibleForDataObject = false;
    //
    // Use the utility routine to handle things common to Init and Restore.
    //
    StsJmp(TttViewInitAndRestoreCommon(self, pInst), s, Error);
    ObjectWrite(self, ctx, &pInst);
    DbgTttViewInit(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    if (responsibleForDataObject AND tttDataNew.object.uid) {
        ObjCallWarn(msgDestroy, tttDataNew.object.uid, pNull);
        // clsView will notice that the data object has been destroyed
        // and will update its instance data, so no need to ObjectWrite.
    }
    if (pInst) {
        OSHeapBlockFree(pInst);
    }
    DbgTttViewInit(("Error; returns 0x%lx",s))
    return s;
} /* TttViewInit */

/*****************************************************************************
    TttViewFree

    Respond to msgFree.

    Note:  Always return stsOK, even if a problem occurs.  This is
    (1) because there's nothing useful to do if a problem occurs anyhow
    and (2) because the ancestor is called after this function if and
    only if stsOK is returned, and it's important that the ancestor
    get called.
*****************************************************************************/
#define DbgTttViewFree(x) \
    TttDbgHelper("TttViewFree",tttViewDbgSet,0x200,x)
MsgHandlerWithTypes(TttViewFree, P_ARGS, PP_TTT_VIEW_INST)
{
    OBJECT  dataObject;
    DbgTttViewFree(("self=0x%lx",self))
```

```
    if ((*pData)->dc) {
        ObjCallWarn(msgDestroy, (*pData)->dc, pNull);
    }
    if (ObjCallWarn(msgViewGetDataObject, self, &dataObject) >= stsOK) {
        if (dataObject) {
            ObjCallWarn(msgViewSetDataObject, self, objNull);
            ObjCallWarn(msgDestroy, dataObject, pNull);
        }
    }
    OSHeapBlockFree(*pData);
    DbgTttViewFree(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* TttViewFree */

/*****************************************************************************
    TttViewSave

    Save self to a file.
*****************************************************************************/
#define DbgTttViewSave(x) \
    TttDbgHelper("TttViewSave",tttViewDbgSet,0x400,x)
MsgHandlerWithTypes(TttViewSave, P_OBJ_SAVE, PP_TTT_VIEW_INST)
{
    TTT_VIEW_FILED_0    filed;
    STATUS              s;
    DbgTttViewSave(("self=0x%lx",self))
    StsJmp(TttUtilWriteVersion(pArgs->file, CURRENT_VERSION), s, Error);
    TttViewFiledData0FromInstData(*pData, &filed);
    StsJmp(TttUtilWrite(pArgs->file, SizeOf(filed), &filed), s, Error);
    DbgTttViewSave(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewSave(("Error; return 0x%lx",s))
    return s;
} /* TttViewSave */

/*****************************************************************************
    TttViewRestore

    Restore self from a file.
    Note: clsmgr has already initialized instance data to zeros.
*****************************************************************************/
#define DbgTttViewRestore(x) \
    TttDbgHelper("TttViewRestore",tttViewDbgSet,0x800,x)
MsgHandlerWithTypes(TttViewRestore, P_OBJ_RESTORE, PP_TTT_VIEW_INST)
{
    P_TTT_VIEW_INST     pInst;
    TTT_VIEW_FILED_0    filed;
    TTT_VERSION         version;
    STATUS              s;
    DbgTttViewRestore(("self=0x%lx",self))
    //
    // Initialize for error recovery.
    //
    pInst = pNull;
    //
    // Read version, then read filed data.  (Currently there's only
    // only one legitimate file format, so no checking of the version
    // need be done.)
    //
    // The allocate instance data and convert filed data.
    //
    StsJmp(TttUtilReadVersion(pArgs->file, MIN_VERSION, MAX_VERSION, \
            &version), s, Error);
    StsJmp(TttUtilRead(pArgs->file, SizeOf(filed), &filed), s, Error);
    StsJmp(OSHeapBlockAlloc(osProcessHeapId, SizeOf(*pInst), &pInst), \
            s, Error);
    TttViewInstDataFromFiledData0(&filed, pInst);
    //
    // Use the utility routine to handle things common to Init and Restore.
    //
    StsJmp(TttViewInitAndRestoreCommon(self, pInst), s, Error);
    ObjectWrite(self, ctx, &pInst);
    DbgTttViewRestore(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    if (pInst) {
        OSHeapBlockFree(pInst);
    }
    DbgTttViewRestore(("Error; return 0x%lx",s))
    return s;
} /* TttViewRestore */

/*****************************************************************************
    TttViewDump

    Respond to msgDump.
*****************************************************************************/
#ifdef DEBUG
MsgHandlerWithTypes(TttViewDump, P_ARGS, PP_TTT_VIEW_INST)
{
    Debugf("TttViewDump: dc=0x%lx lineThickness=%ld",
            (*pData)->dc,(U32)((*pData)->metrics.lineThickness));
    return stsOK;
    MsgHandlerParametersNoWarning;
```

```
} /* TttViewDump */
#endif // DEBUG

/*****************************************************************************
    TttViewDataChanged

    Respond to changes in viewed object.
*****************************************************************************/
#define DbgTttViewDataChanged(x) \
    TttDbgHelper("TttViewDataChanged",tttViewDbgSet,0x1000,x)
MsgHandlerWithTypes(TttViewDataChanged, P_TTT_DATA_CHANGED, PP_TTT_VIEW_INST)
{
    STATUS s;
    DbgTttViewDataChanged(("self=0x%lx pArgs=0x%lx",self,pArgs))
    if (pArgs == pNull) {
        ObjCallJmp(msgWinDirtyRect, self, pNull, s, Error);
    } else {
        WIN_METRICS     wm;
        TTT_VIEW_SIZES sizes;
        DbgTttViewDataChanged(("row=%ld col=%ld",(U32)(pArgs->row), \
                (U32)(pArgs->col)))
        ObjCallJmp(msgWinGetMetrics, (*pData)->dc, &wm, s, Error);
        TttViewComputeSizes(&sizes, pData, &(wm.bounds));
        ObjCallJmp(msgWinDirtyRect, (*pData)->dc, \
                &(sizes.r[pArgs->row][pArgs->col]), s, Error);
    }
    DbgTttViewDataChanged(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewDataChanged(("Error; return 0x%lx",s))
    return s;
} /* TttViewDataChanged */

/*****************************************************************************
    TttViewRepaint

    Respond to msgRepaint.
    This handler demonstrates how to do "smart repainting." It asks the
    window manager for the "dirty" rectangle, and then only repaints those
    objects that intersect with the rectangle.
    Smart repainting should be used by applications that have expensive
    repainting procedures. Although Tic-Tac-Toe's repainting is not even
    close to being prohibitive, it uses smart repainting for the sake
    of demonstrating how to do it.
    Instead of using smart repainting, many applications simply redraw
    their entire window. For an example of this approach, take a look
    at Hello World (Custom Window).
*****************************************************************************/
#define DbgTttViewRepaint(x) \
    TttDbgHelper("TttViewRepaint",tttViewDbgSet,0x2000,x)
MsgHandlerWithTypes(TttViewRepaint, P_ARGS, PP_TTT_VIEW_INST)
{
    TTT_VIEW_SIZES      sizes;
    SYSDC_TEXT_OUTPUT   tx;
    XY32                sizeX;
    XY32                sizeO;
    OBJECT              dataObject;
    BOOLEAN             endRepaintNeeded;
    WIN_METRICS         wm;
    RECT32              dirtyRect;
    TTT_DATA_METRICS    dm;
    U16                 row;
    U16                 col;
    U16                 i;
    OBJECT              owner;
    STATUS              s;
    BOOLEAN             drawIt;
    DbgTttViewRepaint(("self=0x%lx",self))
    //
    // General initialization and intialization for error recovery.
    // Also collect miscellaneous info needed to paint.
    //
    endRepaintNeeded = false;
    ObjCallJmp(msgViewGetDataObject, self, &dataObject, s, Error);
    ObjCallJmp(msgTttDataGetMetrics, dataObject, &dm, s, Error);
    ObjCallJmp(msgWinGetMetrics, (*pData)->dc, &wm, s, Error);
    TttViewComputeSizes(&sizes, pData, &(wm.bounds));
    //
    // Must do msgWinBeginRepaint before any painting starts, and
    // to get dirtyRect.
    //
    ObjCallJmp(msgWinBeginRepaint, (*pData)->dc, &dirtyRect, s, Error);
    endRepaintNeeded = true;
    // ImagePoint ROUNDS from LWC to LUC, but DrawRectangle TRUNCATES.
    // Therefore, increase the size of the dirtyRect so as to be sure to
    // cover all the pixels that need to be painted.
    // Another solution would be to use finer LUC than points.
    dirtyRect.origin.x--;
    dirtyRect.origin.y--;
    dirtyRect.size.w += 2;
    dirtyRect.size.h += 2;
    //
    // Fill the dirty rect with the appropriate background.  If we hold the
    // selection, the appropriate background is grey, otherwise it is white.
    //
    ObjCallJmp(msgSelOwner, theSelectionManager, &owner, s, Error);
    if (owner == self) {
        DbgTttViewRepaint(("owner is self"))
```

```
    ObjectCall(msgDcSetBackgroundRGB, (*pData)->dc, \
            (P_ARGS)sysDcRGBGray33);
} else {
    DbgTttViewRepaint(("owner is not self"))
    ObjectCall(msgDcSetBackgroundRGB, (*pData)->dc, \
            (P_ARGS)sysDcRGBWhite);
}
ObjectCall(msgDcSetFillPat, (*pData)->dc, (P_ARGS)sysDcPatBackground);
ObjectCall(msgDcSetLineThickness, (*pData)->dc, (P_ARGS)0L);
ObjectCall(msgDcDrawRectangle, (*pData)->dc, &dirtyRect);
ObjectCall(msgDcSetFillPat, (*pData)->dc, (P_ARGS)sysDcPatForeground);
//
// Paint the vertical lines
//
for (i=0; i<2; i++) {
    if (Rect32sIntersect(&dirtyRect, &(sizes.vertLines[i]))) {
        DbgTttViewRepaint(("vertical i=%ld; overlap",(U32)i))
        ObjectCall(msgDcDrawRectangle, (*pData)->dc, \
                &(sizes.vertLines[i]));
    } else {
        DbgTttViewRepaint(("vertical i=%ld; no overlap",(U32)i))
    }
}
//
// Paint the horizontal lines
//
for (i=0; i<2; i++) {
    if (Rect32sIntersect(&dirtyRect, &(sizes.horizLines[i]))) {
        DbgTttViewRepaint(("horizontal i=%ld; overlap",(U32)i))
        ObjectCall(msgDcDrawRectangle, (*pData)->dc, \
                &(sizes.horizLines[i]));
    } else {
        DbgTttViewRepaint(("horizontal i=%ld; no overlap",(U32)i))
    }
}
//
// Scale the font to the box size.
//
// Note: This could be done once when the window size
// changes rather than each time the window is painted.
//
ObjCallJmp(msgDcIdentityFont, (*pData)->dc, pNull, s, Error);
ObjCallJmp(msgDcScaleFont, (*pData)->dc, &(sizes.scale), s, Error);
//
// Measure X and O in the font.
//
TttUtilInitTextOutput(&tx, sysDcAlignChrTop, pNull);
tx.pText = "X";
tx.lenText = strlen(tx.pText);
ObjectCall(msgDcMeasureText, (*pData)->dc, (P_ARGS)&tx);
sizeX = tx.cp;
DbgTttViewRepaint(("measure X=[%ld %ld]",sizeX.x, sizeX.y))
TttUtilInitTextOutput(&tx, sysDcAlignChrTop, pNull);
tx.pText = "O";
tx.lenText = strlen(tx.pText);
ObjectCall(msgDcMeasureText, (*pData)->dc, (P_ARGS)&tx);
sizeO = tx.cp;
DbgTttViewRepaint(("measure O=[%ld %ld]",sizeO.x, sizeO.y))
//
// Paint the cells.
//
for (row=0; row<3; row++) {
    for (col=0; col<3; col++) {
        if (Rect32sIntersect(&dirtyRect, &(sizes.r[row][col]))) {
            DbgTttViewRepaint(("row=%ld col=%ld; overlap", \
                    (U32)row, (U32)col));
            if (dm.squares[row][col] == tttX) {
                drawIt = TRUE;
                tx.pText = "X";
                tx.lenText = strlen(tx.pText);
                tx.cp.x = sizes.r[row][col].origin.x +
                        ((sizes.r[row][col].size.w - sizeX.x) / 2);
                tx.cp.y = sizes.r[row][col].origin.y + sizeX.y +
                        ((sizes.r[row][col].size.h - sizeX.y) / 2);
            } else if (dm.squares[row][col] == tttO) {
                drawIt = TRUE;
                tx.pText = "O";
                tx.lenText = strlen(tx.pText);
                tx.cp.x = sizes.r[row][col].origin.x +
                        ((sizes.r[row][col].size.w - sizeO.x) / 2);
                tx.cp.y = sizes.r[row][col].origin.y + sizeO.y +
                        ((sizes.r[row][col].size.h - sizeO.y) / 2);
            } else {
                DbgTttViewRepaint(("blank cell"))
                drawIt = FALSE;
            }
            if (drawIt) {
                ObjCallJmp(msgDcDrawText, (*pData)->dc, &tx, s, Error);
            }
        } else {
            DbgTttViewRepaint(("row=%ld col=%ld; no overlap", \
                    (U32)row, (U32)col));
        }
    }
}
//
// Balance the msgWinBeginRepaint
//
ObjCallWarn(msgWinEndRepaint, (*pData)->dc, pNull);
DbgTttViewRepaint(("return stsOK"))
return stsOK;
MsgHandlerParametersNoWarning;
```

```
Error:
    if (endRepaintNeeded) {
        ObjCallWarn(msgWinEndRepaint, (*pData)->dc, pNull);
    }
    DbgTttViewRepaint(("Error; return 0x%lx",s))
    return s;
} /* TttViewRepaint */

/*****************************************************************************
    TttViewGetDesiredSize

    Respond to msgGetDesiredSize.
    The desired size is an appropriate minimum size for the drawing.
*****************************************************************************/
#define DbgTttViewGetDesiredSize(x) \
    TttDbgHelper("TttViewGetDesiredSize",tttViewDbgSet,0x2000,x)
MsgHandlerArgType(TttViewGetDesiredSize, P_WIN_METRICS)
{
    pArgs->bounds.size.w    = desiredWidth;
    pArgs->bounds.size.h    = desiredHeight;
    DbgTttViewGetDesiredSize(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* TttViewGetDesiredSize */
/*****************************************************************************
    TttViewGesture

    Let ancestor handle unrecognized gestures.
*****************************************************************************/
#define DbgTttViewGesture(x) \
    TttDbgHelper("TttViewGesture",tttViewDbgSet,0x8000,x)
MsgHandlerWithTypes(TttViewGesture, P_GWIN_GESTURE, PP_TTT_VIEW_INST)
{
    STATUS      s;
    OBJECT      owner;
#ifdef DEBUG
    {
        P_CLS_SYMBUF    mb;
        DbgTttViewGesture(("self=0x%lx msg=0x%lx %s", self, pArgs->msg,
                ClsMsgToString(pArgs->msg,mb)))
    }
#endif // DEBUG
    switch(ClsNum(pArgs->msg)) {
        case ClsNum(clsXGesture):
            switch(MsgNum(pArgs->msg)) {
                case MsgNum(xgs1Tap):
                    ObjCallJmp(msgTttViewToggleSel, self, pNull, \
                            s, Error);
                    break;
                case MsgNum(xgsCross):
                    StsJmp(TttViewGestureSetSquare(self, pArgs, tttX), \
                            s, Error);
                    break;
                case MsgNum(xgsCircle):
                    StsJmp(TttViewGestureSetSquare(self, pArgs, tttO), \
                            s, Error);
                    break;
                case MsgNum(xgsPigtailVert):
                case MsgNum(xgsPigtailHorz):
                    StsJmp(TttViewGestureSetSquare(self, pArgs, tttBlank), \
                            s, Error);
                    break;
                case MsgNum(xgsCheck):
                    // Make sure there is a selection.
                    ObjCallJmp(msgSelOwner, theSelectionManager, \
                            &owner, s, Error);
                    if (owner != self) {
                        ObjCallJmp(msgTttViewTakeSel, self, pNull, s, Error);
                        ObjCallJmp(msgWinUpdate, self, pNull, s, Error);
                    }
                    // Then call the ancestor.
                    ObjCallAncestorCtxJmp(ctx, s, Error);
                    break;
                default:
                    DbgTttViewGesture(("Letting ancestor handle gesture"))
                    return ObjCallAncestorCtxWarn(ctx);
            }
            break;
        default:
            DbgTttViewGesture(("Letting ancestor handle gesture"))
            return ObjCallAncestorCtxWarn(ctx);
    }
    DbgTttViewGesture(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewGesture(("Error; return 0x%lx",s))
    return s;
} /* TttViewGesture */

/*****************************************************************************
    TttViewSelYield

     msgSelYield from selection manager.
*****************************************************************************/
#define DbgTttViewSelYield(x) \
    TttDbgHelper("TttViewSelYield",tttViewDbgSet,0x10000,x)
MsgHandlerWithTypes(TttViewSelYield, P_ARGS, PP_TTT_VIEW_INST)
```

```
{
    STATUS      s;
    DbgTttViewSelYield(("self=0x%lx",self))
    StsJmp(TttViewNeedRepaint(self), s, Error);
    DbgTttViewSelYield(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewSelYield(("Error; return 0x%lx",s))
    return s;
} /* TttViewSelYield */
/*****************************************************************************
    TttViewSelDelete
    In this particular application, deleting is a poorly defined concept.
    Rather than do nothing, though, we clear the board.
*****************************************************************************/
#define DbgTttViewSelDelete(x) \
    TttDbgHelper("TttViewSelDelete",tttViewDbgSet,0x80000,x)
MsgHandlerWithTypes(TttViewSelDelete, P_ARGS, PP_TTT_VIEW_INST)
{
    TTT_DATA_METRICS    dm;
    OBJECT              dataObject;
    U16                 row;
    U16                 col;
    STATUS              s;
    DbgTttViewSelDelete(("",))
    ObjCallJmp(msgViewGetDataObject, self, &dataObject, s, Error);
    ObjCallJmp(msgTttDataGetMetrics, dataObject, &dm, s, Error);
    for (row=0; row<3; row++) {
        for (col=0; col<3; col++) {
            dm.squares[row][col] = tttBlank;
        }
    }
    dm.undoTag = tagTttDataUndoDelete;
    ObjCallJmp(msgTttDataSetMetrics, dataObject, &dm, s, Error);
    DbgTttViewSelDelete(("returns stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewSelDelete(("Error; return 0x%lx",s))
    return s;
} /* TttViewSelDelete */

/*****************************************************************************
    TttViewGetMetrics
*****************************************************************************/
#define DbgTttViewGetMetrics(x) \
    TttDbgHelper("TttViewGetMetrics",tttViewDbgSet,0x100000,x)
MsgHandlerWithTypes(TttViewGetMetrics, P_TTT_VIEW_METRICS, PP_TTT_VIEW_INST)
{
    DbgTttViewGetMetrics(("self=0x%lx",self))
    *pArgs = (*pData)->metrics;
    DbgTttViewGetMetrics(("returns stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* TttViewGetMetrics */

/*****************************************************************************
    TttViewSetMetrics
*****************************************************************************/
#define DbgTttViewSetMetrics(x) \
    TttDbgHelper("TttViewSetMetrics",tttViewDbgSet,0x200000,x)
MsgHandlerWithTypes(TttViewSetMetrics, P_TTT_VIEW_METRICS, PP_TTT_VIEW_INST)
{
    DbgTttViewSetMetrics(("self=0x%lx",self))
    (*pData)->metrics = *pArgs;
    TttViewNeedRepaint(self);
    DbgTttViewSetMetrics(("returns stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* TttViewSetMetrics */

/*****************************************************************************
    TttViewToggleSel
     msgTttViewToggleSel
*****************************************************************************/
#define DbgTttViewToggleSel(x) \
    TttDbgHelper("TttViewToggleSel",tttViewDbgSet,0x400000,x)
MsgHandlerWithTypes(TttViewToggleSel, P_ARGS, PP_TTT_VIEW_INST)
{
    OBJECT      owner;
    STATUS      s;
    DbgTttViewToggleSel(("self=0x%lx",self))
    ObjCallJmp(msgSelOwner, theSelectionManager, &owner, s, Error);
    if (owner == self) {
        DbgTttViewToggleSel(("View is the owner; set to be objNull"))
        StsJmp(TttUtilGiveUpSel(self), s, Error);
    } else {
        DbgTttViewToggleSel(("View is not the owner; set to be self"))
        StsJmp(TttUtilTakeSel(self), s, Error);
    }
    StsJmp(TttViewNeedRepaint(self), s, Error);
    DbgTttViewToggleSel(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
```

```
    DbgTttViewToggleSel(("Error; return 0x%lx",s))
    return s;
} /* TttViewToggleSel */

/*****************************************************************************
    TttViewTakeSel
*****************************************************************************/
#define DbgTttViewTakeSel(x) \
    TttDbgHelper("TttViewTakeSel",tttViewDbgSet,0x800000,x)
MsgHandlerWithTypes(TttViewTakeSel, P_ARGS, PP_TTT_VIEW_INST)
{
    OBJECT  owner;
    STATUS  s;
    DbgTttViewTakeSel(("self=0x%lx",self))
    ObjCallJmp(msgSelOwner, theSelectionManager, &owner, s, Error);

    if (owner != self) {
        DbgTttViewTakeSel(("owner is not self; taking"))
        StsJmp(TttUtilTakeSel(self), s, Error);
        StsJmp(TttViewNeedRepaint(self), s, Error);
    } else {
        DbgTttViewTakeSel(("owner is already self; doing nothing"))
    }
    DbgTttViewTakeSel(("returns stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewTakeSel(("Error; return 0x%lx",s))
    return s;
} /* TttViewTakeSel */

/*****************************************************************************
    TttViewInputEvent
     msgInputEvent.
*****************************************************************************/
#define DbgTttViewInputEvent(x) \
    TttDbgHelper("TttViewInputEvent",tttViewDbgSet,0x1000000,x)
MsgHandlerWithTypes(TttViewInputEvent, P_INPUT_EVENT, PP_TTT_VIEW_INST)
{
    STATUS      s;
    switch (ClsNum(pArgs->devCode)) {
        case ClsNum(clsPen):
            s = TttViewPenInput(msg, self, pArgs, ctx, pData);
            break;

        case ClsNum(clsKey):
            s = TttViewKeyInput(self, pArgs);
            break;
        default:
            s = ObjectCallAncestorCtx(ctx);
            break;
    }
    return s;
    MsgHandlerParametersNoWarning;
} /* TttViewInputEvent */
/*****************************************************************************
    TttViewProvideEnable

    Respond to msgControlProvideEnable.
*****************************************************************************/
MsgHandlerWithTypes(TttViewProvideEnable, P_CONTROL_PROVIDE_ENABLE,
PP_TTT_VIEW_INST)
{
    switch (pArgs->tag) {
        case (tagAppMenuMove):
        case (tagAppMenuCopy):
            pArgs->enable = true;
            break;
        default:
            return ObjectCallAncestorCtx(ctx);
    }
    return stsOK;
    MsgHandlerParametersNoWarning;
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Installation                                       *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************
    ClsTttViewInit

    Install the class.
*****************************************************************************/
STATUS PASCAL
ClsTttViewInit (void)
{
    CLASS_NEW       new;
    STATUS          s;
    ObjCallJmp(msgNewDefaults, clsClass, &new, s, Error);
    new.object.uid          = clsTttView;
    new.object.key          = 0;
    new.cls.pMsg            = clsTttViewTable;
    new.cls.ancestor        = clsView;
    new.cls.size            = SizeOf(P_TTT_VIEW_INST);
    new.cls.newArgsSize = SizeOf(TTT_VIEW_NEW);
    ObjCallJmp(msgNew, clsClass, &new, s, Error);
    return stsOK;
```

```
Error:
     return s;
} /* ClsTttViewInit */
```

TTTVOPT.C

```
/****************************************************************************
 File: tttvopt.c
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.2  $
 $Date:   04 Dec 1991 13:39:18  $
 This file contains the implementation of clsTttView's Option Sheets.
 Notes:

 [1] The Option Sheet protocol allows any class of an
         object to create an option sheet and/or add cards.
         Therefore this code carefully validates that it only
         operates on Option Sheets and Cards that it knows about.
         This could be overkill.
****************************************************************************/
#ifndef OPTION_INCLUDED
#include <option.h>
#endif
#ifndef TTTVIEW_INCLUDED
#include <tttview.h>
#endif
#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif
#ifndef OPTTABLE_INCLUDED
#include <opttable.h>
#endif
#ifndef TTTDATA_INCLUDED
#include <tttdata.h>
#endif
#ifndef OS_INCLUDED
#include <os.h>
#endif
#ifndef SEL_INCLUDED
#include <sel.h>
#endif

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Defines, Types, Globals, Etc                       *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
//
// The following static maps to the TK_TABLE_ENTRY struct, in tktable.h
// It is short-hand for defining a TkTable.
//
STATIC TK_TABLE_ENTRY cardEntries[] = {
     {"Line Thickness:", 0, 0, 0, 0, 0, tagTttQHelpForLineCtrl},
         {"1", 0, 0, tagCardLineThickness, 0, clsIntegerField,
             tagTttQHelpForLineCtrl},
     {pNull}
};

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Utility Routines                                   *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Message Handlers                                   *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/****************************************************************************
     TttViewOptionAddCards

     Handles msgOptionAddCards.

     Note on error handling: Once a card has been added to the sheet,
     destroying the sheet will destroy the card.
****************************************************************************/
#define DbgTttViewOptionAddCards(x) \
     TttDbgHelper("TttViewOptionAddCards",tttViewOptsDbgSet,0x4,x)
MsgHandlerWithTypes(TttViewOptionAddCards, P_OPTION_TAG, PP_TTT_VIEW_INST)
{
     OPTION_CARD         card;
     STATUS              s;
     DbgTttViewOptionAddCards(("entry"))
     //
     // Create the card.
     //
     card.tag = tagTttViewCard;
     card.win = objNull;
     card.pName = "TTT Card";
     card.client = self;
     ObjCallJmp(msgOptionAddLastCard, pArgs->option, &card, s, Error);
     DbgTttViewOptionAddCards(("return stsOK"))
     return stsOK;
```

```
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewOptionAddCards(("Error; return 0x%lx",s))
    return s;
} /* TttViewOptionAddCards */
/*****************************************************************************
    TttViewOptionProvideCard

    Handles msgOptionProvideCardWin.
*****************************************************************************/
#define DbgTttViewOptionProvideCard(x) \
    TttDbgHelper("TttViewOptionProvideCard",tttViewOptsDbgSet,0x8,x)
MsgHandlerWithTypes(TttViewOptionProvideCard, P_OPTION_CARD, P_UNKNOWN)
{
    STATUS              s;
    OPTION_TABLE_NEW    otn;
    DbgTttViewOptionProvideCard(("",))
    pArgs->win = objNull;
    if (pArgs->tag == tagTttViewCard)
    {
        ObjCallJmp(msgNewDefaults, clsOptionTable, &otn, s, Error);
        otn.tkTable.client = self;
        otn.tkTable.pEntries = cardEntries;
        otn.win.tag = pArgs->tag;
        otn.gWin.helpId = tagCardLineThickness;
        ObjCallJmp(msgNew, clsOptionTable, &otn, s, Error);
        pArgs->win = otn.object.uid;
    }
    return(stsOK);
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewOptionProvideCard(("Error; return 0x%lx",s))
    return s;
} /* TttViewOptionProvideCard */
/*****************************************************************************
    TttViewOptionRefreshCard
    Handles msgOptionRefreshCard
*****************************************************************************/
#define DbgTttViewOptionRefreshCard(x) \
    TttDbgHelper("TttViewOptionRefreshCard",tttViewOptsDbgSet,0x10,x)
MsgHandlerWithTypes(TttViewOptionRefreshCard, P_OPTION_CARD, PP_TTT_VIEW_INST)
{
    OBJECT              view;
    TTT_VIEW_METRICS    vm;
    OBJECT              control;
    STATUS              s;
    DbgTttViewOptionRefreshCard(("",))
    //
    // See note [1] at the beginning of this file.
    //
    if (pArgs->tag != tagTttViewCard) {
        DbgTttViewOptionRefreshCard(("unrecognized card; call ancestor"))
        return ObjCallAncestorCtxWarn(ctx);
    }
    //
    // Collect info needed to refresh card.
    //
    StsJmp(TttUtilGetComponents(OSThisApp(), tttGetView, \
            objNull, &view, objNull), s, Error);
    ObjCallJmp(msgTttViewGetMetrics, view, &vm, s, Error);
    DbgTttViewOptionRefreshCard(("refreshing card"))
    control = (OBJECT) ObjectCall(msgWinFindTag, pArgs->win, \
            (P_ARGS)tagCardLineThickness);
    ObjCallJmp(msgControlSetValue, control, (P_ARGS)(vm.lineThickness), \
            s, Error);
    //
    // The whole card is clean now.
    //
    ObjCallJmp(msgControlSetDirty, pArgs->win, (P_ARGS)false, s, Error);
    DbgTttViewOptionRefreshCard(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewOptionRefreshCard(("Error; return 0x%lx",s))
    return s;
} /* TttViewOptionRefreshCard */

/*****************************************************************************
    TttViewOptionApplyCard
    Handles msgOptionApplyCard
    Note:  Perhaps this should be an undoable operation.
*****************************************************************************/
#define DbgTttViewOptionApplyCard(x) \
    TttDbgHelper("TttViewOptionApplyCard",tttViewOptsDbgSet,0x20,x)
MsgHandlerWithTypes(TttViewOptionApplyCard, P_OPTION_CARD, PP_TTT_VIEW_INST)
{
    OBJECT              view;
    TTT_VIEW_METRICS    vm;
    BOOLEAN             dirty;
    OBJECT              control;
    U32                 value;
    OBJECT              owner;
    STATUS              s;
    DbgTttViewOptionApplyCard(("",))
    //
    // See note [1] at the beginning of this file.
    //
    if (pArgs->tag != tagTttViewCard) {
```

```
        DbgTttViewOptionRefreshCard(("unrecognized card; call ancestor"))
        return ObjCallAncestorCtxWarn(ctx);
    }
    //
    // Collect info needed to apply card.
    //
    StsJmp(TttUtilGetComponents(OSThisApp(), tttGetView, objNull, \
            &view, objNull), s, Error);
    DbgTttViewOptionApplyCard(("applying card"))
    control = (OBJECT) ObjectCall(msgWinFindTag, pArgs->win, \
            (P_ARGS)tagCardLineThickness);
    ObjCallJmp(msgControlGetDirty, control, &dirty, s, Error);
    if (dirty) {
        // Promote the view's selection, if it is not already promoted.
        ObjCallJmp(msgSelOwner, theSelectionManager, &owner, s, Error);
        if (owner != self) {
            ObjCallJmp(msgSelSetOwnerPreserve, theSelectionManager, \
                    pNull, s, Error);
        }
        ObjCallJmp(msgControlGetValue, control, &value, s, Error);
        DbgTttViewOptionApplyCard(("\"Line Thickness\" is dirty; value=%ld",value))
        ObjCallJmp(msgTttViewGetMetrics, view, &vm, s, Error);
        vm.lineThickness = value;
        ObjCallJmp(msgTttViewSetMetrics, view, &vm, s, Error);
    } else {
        DbgTttViewOptionApplyCard(("\"Line Thickness\" is not dirty"))
    }

    DbgTttViewOptionApplyCard(("return stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewOptionApplyCard(("Error; return 0x%lx",s))
    return s;
} /* TttViewOptionApplyCard */

/*****************************************************************************
    TttViewOptionApplicableCard
    Handles msgOptionApplicableCard
*****************************************************************************/
#define DbgTttViewOptionApplicableCard(x) \
    TttDbgHelper("TttViewOptionApplicableCard",tttViewOptsDbgSet,0x80,x)
MsgHandlerWithTypes(TttViewOptionApplicableCard, P_OPTION_CARD, \
        PP_TTT_VIEW_INST)
{
    OBJECT      owner;
    STATUS      s;
    DbgTttViewOptionApplicableCard(("\"Line Thickness\" is not dirty"))
    //
    // See note [1] at the beginning of this file.  Also, don't use
    // ObjCallAncestorCtxWarn();  it is not an error for the ancestor to
    // return stsFailed, and we don't want to generate a debugging message.
    //
    if (pArgs->tag != tagTttViewCard) {
        DbgTttViewOptionApplicableCard(("unrecognized card; call ancestor"))
        return ObjectCallAncestorCtx(ctx);
    }
    //
    // So it's a ttt card.  Decide if it's consistent with the current seln.
    //
    ObjCallJmp(msgSelOwner, theSelectionManager, &owner, s, Error);
    if (owner == self) {
        DbgTttViewOptionApplicableCard(("owner is self; return stsOK"))
        return stsOK;
    } else {
        DbgTttViewOptionApplicableCard(("owner is not self; return stsFailed"))
        return stsFailed;
    }
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewOptionApplicableCard(("Error; return 0x%lx",s))
    return s;
} /* TttViewOptionApplicableCard */
```

TTTVXFER.C

```
/*****************************************************************************
 File: tttvxfer.c
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.1  $
 $Date:   02 Dec 1991 19:08:02  $
 This file contains the implementation of clsTttView's Data Transfer
*****************************************************************************/
#ifndef TTTVIEW_INCLUDED
#include <tttview.h>
#endif
#ifndef LIST_INCLUDED
#include <list.h>
#endif
```

```
#ifndef XFER_INCLUDED
#include <xfer.h>
#endif
#ifndef SEL_INCLUDED
#include <sel.h>
#endif
#ifndef TTTDATA_INCLUDED
#include <tttdata.h>
#endif
#ifndef EMBEDWIN_INCLUDED
#include <embedwin.h>
#endif
#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Defines, Types, Globals, Etc                     *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Utility Routines                                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Message Handlers                                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************
     TttViewSelBeginMoveAndCopy
     Handles both msgSelBeginMove and msgSelBeginCopy
*****************************************************************************/
#define DbgTttViewSelBeginMoveAndCopy(x) \
     TttDbgHelper("TttViewSelBeginMoveAndCopy",tttViewXferDbgSet,0x1,x)
MsgHandlerWithTypes(TttViewSelBeginMoveAndCopy, P_XY32, PP_TTT_VIEW_INST)
{
     EMBEDDED_WIN_BEGIN_MOVE_COPY    bmc;
     STATUS                          s;
     DbgTttViewSelBeginMoveAndCopy(("self=0x%lx",self))
     if (pArgs) {
         bmc.xy = *pArgs;
     } else {
         bmc.xy.x =
         bmc.xy.y = 0;
     }
     bmc.bounds.origin.x =
     bmc.bounds.origin.y =
     bmc.bounds.size.w =
     bmc.bounds.size.h = 0;
     ObjCallJmp(MsgEqual(msg, msgSelBeginMove) ?
             msgEmbeddedWinBeginMove : msgEmbeddedWinBeginCopy,
             self, &bmc, s, Error);
     DbgTttViewSelBeginMoveAndCopy(("returns stsOK"))
     return stsOK;
     MsgHandlerParametersNoWarning;
Error:
     DbgTttViewSelBeginMoveAndCopy(("Error; return 0x%lx",s))
     return s;
} /* TttViewSelBeginMoveAndCopy */

/*****************************************************************************
     TttViewXferGet
*****************************************************************************/
#define DbgTttViewXferGet(x) \
     TttDbgHelper("TttViewXferGet",tttViewXferDbgSet,0x2,x)
MsgHandlerWithTypes(TttViewXferGet, P_XFER_FIXED_BUF, PP_TTT_VIEW_INST)
{
     STATUS  s;
     DbgTttViewXferGet(("self=0x%lx",self))
     if (pArgs->id == xferString) {
         OBJECT dataObj;
         TTT_DATA_METRICS dm;
         U16 row;
         U16 col;
         P_XFER_FIXED_BUF p = (P_XFER_FIXED_BUF)pArgs;
         ObjCallJmp(msgViewGetDataObject, self, &dataObj, s, Error);
         ObjCallJmp(msgTttDataGetMetrics, dataObj, &dm, s, Error);

         //
         // initialize the length to the number of squares (9) plus 1
         // to allow for a string termination character (just in case
         // the user copies/moves the string into a text processor.
         //
         p->len = 10;
         p->data = 0L;
         for (row=0; row<3; row++) {
             for (col=0; col<3; col++) {
                 p->buf[(row*3)+col] = dm.squares[row][col];
             }
         }
         p->buf[9] = '\0';
         s = stsOK;
     } else {
         s = ObjectCallAncestorCtx(ctx);
     }
     DbgTttViewXferGet(("returns 0x%lx",s))
     return s;
     MsgHandlerParametersNoWarning;
Error:
     DbgTttViewXferGet(("Error; return 0x%lx",s))
```

```
    return s;
} /* TttViewXferGet */

/*****************************************************************************
    TttViewXferList
*****************************************************************************/
static TAG
sourceFormats[] = {xferString};
#define N_SOURCE_FORMATS (SizeOf(sourceFormats) / SizeOf(sourceFormats[0]))
#define DbgTttViewXferList(x) \
    TttDbgHelper("TttViewXferList",tttViewXferDbgSet,0x4,x)
MsgHandlerWithTypes(TttViewXferList, OBJECT, PP_TTT_VIEW_INST)
{
    STATUS  s;
    DbgTttViewXferList(("self=0x%lx",self))
    //
    // Don't let ancestor add types.  We aren't interested in
    // moving/copying the window, which is the only type the
    // ancestor supports.
    //
    StsJmp(XferAddIds(pArgs, sourceFormats, N_SOURCE_FORMATS), s, Error);
    DbgTttViewXferList(("returns stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewXferList(("Error; return 0x%lx",s))
    return s;
} /* TttViewXferList */

/*****************************************************************************
    TttViewSelMoveAndSelCopy
    Handles both msgSelMoveSelection and msgSelCopySelection
*****************************************************************************/
static TAG
receiverFormats[] = {xferString};
#define N_RECEIVER_FORMATS (SizeOf(receiverFormats) /
SizeOf(receiverFormats[0]))
#define DbgTttViewSelMoveAndSelCopy(x) \
    TttDbgHelper("TttViewSelMoveAndSelCopy",tttViewXferDbgSet,0x8,x)
MsgHandlerWithTypes(TttViewSelMoveAndSelCopy, P_XY32, PP_TTT_VIEW_INST)
{
    TAG             transferType;
    OBJECT          owner;
    XFER_LIST_NEW   listNew;
    STATUS          s;
    DbgTttViewSelMoveAndSelCopy(("self=0x%lx",self))
    //
    // Initialize for error recovery
    //
    listNew.object.uid = NULL;
    //
    // Get source of move/copy.
    //
    ObjCallJmp(msgSelOwner, theSelectionManager, &owner, s, Error);
    if (! owner) {
        DbgTttViewSelMoveAndSelCopy(("no owner!"))
        s = stsFailed;
        goto Error;
    }
    //
    // Don't bother doing move/copy to self.  Use the Error exit out of
    // this routine even though this really isn't really an error.
    // FIXME: Inform the user or not?  Wait for UI Guidelines.
    //
    if (owner == self) {
        DbgTttViewSelMoveAndSelCopy(("owner == self"))
        s = stsOK;
        goto Error;
    }

    //
    // Get list of available types.
    //
    ObjCallJmp(msgNewDefaults, clsXferList, &listNew, s, Error);
    ObjCallJmp(msgNew, clsXferList, &listNew, s, Error);
    ObjCallJmp(msgXferList, owner, listNew.object.uid, s, Error);
    StsJmp(XferListSearch(listNew.object.uid, receiverFormats,
            N_RECEIVER_FORMATS, &transferType), s, Error);
    //
    // This only handles one transfer type now, but we expect to handle
    // more in the future.  So code it in that style.
    //
    if (transferType == xferString) {
        TTT_DATA_METRICS metrics;
        OBJECT dataObj;
        XFER_FIXED_BUF xfer;
        U16 i;
        DbgTttViewSelMoveAndSelCopy(("transferType is xferString"))
        ObjCallJmp(msgViewGetDataObject, self, &dataObj, s, Error);
        ObjCallJmp(msgTttDataGetMetrics, dataObj, &metrics, s, Error);
        xfer.id = xferString;
        ObjSendUpdateJmp(msgXferGet, owner, &xfer, SizeOf(xfer), s, Error);
        DbgTttViewSelMoveAndSelCopy(("data=%ld len=%ld",
                (U32)(xfer.data), (U32)(xfer.len)))
        for (i=0; i < (U16)Min(xfer.len,9L); i++) {
            metrics.squares[i/3][i%3] =
                    TttUtilSquareValueForChar(xfer.buf[i]);
        }
```

```
        metrics.undoTag = tagTttDataUndoMoveCopy;
        ObjCallJmp(msgTttDataSetMetrics, dataObj, &metrics, s, Error);
    } else {
        goto Error;
    }
    //
    // If this was a move, delete the source.
    //
    if (MsgEqual(msgSelMoveSelection, msg)) {
        ObjSendU32Jmp(msgSelDelete, owner, (P_ARGS)SelDeleteNoSelect, s,
                Error);
    }
    //
    // Take the selection.  Be sure to do this AFTER deleting the
    // selection because the source may "forget" what to delete when
    // the selection is pulled from it.
    //
    ObjCallJmp(msgTttViewTakeSel, self, pNull, s, Error);
    ObjCallWarn(msgDestroy, listNew.object.uid, pNull);
    DbgTttViewSelMoveAndSelCopy(("returns stsOK"))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    if (listNew.object.uid) {
        ObjCallWarn(msgDestroy, listNew.object.uid, pNull);
    }
    DbgTttViewSelMoveAndSelCopy(("Error; return 0x%lx",s))
    return s;
} /* TttViewSelMoveAndSelCopy */
```

Resource Files

TTTMISC.RC

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 File: tttmisc.rc
 Copyright 1991, 1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#ifndef RESCMPLR_INCLUDED
#include <rescmplr.h>
#endif
#ifndef TTTPRIV_INCLUDED
#include "tttpriv.h"
#endif
static P_STRING tttTKStrings[] = {
     "Undo Delete",       // 0 - special undo menu string for deletes
     "Undo Move/Copy",    // 1 - special undo menu string for moves & copies
     pNull
};
static RC_INPUT  tttTKResStrings = {
     MakeListResId(clsTttData, resGrpTK, 0),
     tttTKStrings,
     sizeof(tttTKStrings),
     resStringArrayResAgent
};
P_RC_INPUT    resInput [] = {
     &tttTKResStrings,
     pNull
};
```

TTTQHELP.RC

```
/*****************************************************************************
 File: tttqhelp.rc
 Copyright 1990-1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 Tic-Tac-Toe Quick Help resources.
*****************************************************************************/
#ifndef RESCMPLR_INCLUDED
#include <rescmplr.h>
#endif
#ifndef  QHELP_INCLUDED
#include <qhelp.h>
#endif
#ifndef TTTVIEW_INCLUDED
#include "tttview.h"
#endif
//
// Quick Help string for ttt's option card.
//
```

```
static CHAR tttOptionString[] = {
     // Title for the quick help window
     "TTT Card||"
     // Quick help text
     "Use this option card to change the thickness of the lines "
     "on the Tic-Tac-Toe board."
};
//
// Quick Help string for the line thickness control in ttt's option card.
//
static CHAR tttLineThicknessString[] = {
     // Title for the quick help window
     "Line Thickness||"
     // Quick help text
     "Change the line thickness by writing in a number from 1-9."
};
//
// Quick Help string for the view.
//
static CHAR tttViewString[] = {
     // Title for the quick help window
     "Tic-Tac-Toe||"
     // Quick help text
     "The Tic-Tac-Toe window lets you to make X's and O's in a Tic-Tac-Toe "
     "grid. You can write X's and O's and make move, copy "
     "and pigtail delete gestures.\n\n"
     "It does not recognize a completed game, either tied or won.\n\n"
     "To clear the game and start again, tap Select All in the Edit menu, "
     "then tap Delete."
};
// Define the quick help resource for the view.
static P_RC_TAGGED_STRING tttViewQHelpStrings[] = {
     tagCardLineThickness,   tttOptionString,
     tagTttQHelpForLineCtrl, tttLineThicknessString,
     tagTttQHelpForView,     tttViewString,
     pNull
};
static RC_INPUT tttViewHelp = {
     MakeListResId(clsTttView, resGrpQhelp, 0),
     tttViewQHelpStrings,            // Name of the string array
     0,
     resTaggedStringArrayResAgent    // Use string array resource agent
};
/*****************************************************************************
     The glue that ties everything together -- resInput.
*****************************************************************************/
// resInput is an exported variable that the resource compiler expects.
// Each element is a pointer to a structure describing the next resource.
// The list is terminated with a null pointer.
P_RC_INPUT    resInput [] = {
     &tttViewHelp,   // this is the one defined in this example
     // any other resource pointers would go here
     pNull
};
```

MAKEFILE

```
###############################################
#
#  WMake Makefile for TTT (Tic-tac-Toe)
#
# Copyright 1990-1992 GO Corporation. All Rights Reserved.
#
# You may use this Sample Code any way you please provided you
# do not resell the code and that this notice (including the above
# copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
# IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
# EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
# LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
# PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
# FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
# THE USE OR INABILITY TO USE THIS SAMPLE CODE.
#
# $Revision:   1.10  $
#      $Date:   07 Jan 1992 17:18:40  $
#
###############################################
# Set PENPOINT_PATH to your environment variable, if it exists.
# Otherwise, set it to \penpoint
!ifdef %PENPOINT_PATH
PENPOINT_PATH = $(%PENPOINT_PATH)
!else
PENPOINT_PATH = \penpoint
!endif

MODE = debug

#  The DOS name of your project directory.
PROJ = ttt

#  Standard defines for sample code
!INCLUDE $(PENPOINT_PATH)\sdk\sample\sdefines.mif

#  The PenPoint name of your application
EXE_NAME = Tic-Tac-Toe

#  The linker name for your executable : company-name-V<major>(<minor>)
EXE_LNAME= GO-TIC_TAC_TOE-V1(1)

#  The app's version (if undefined, srules.mif defines it to be 1.0)
APP_VERSION = 1.1

#  Object files needed to build your app
EXE_OBJS =    methods.obj &
              tttapp.obj &
              tttdata.obj &
```

```
            tttview.obj &
            tttvopt.obj &
            tttdbg.obj &
            tttmbar.obj &
            tttutil.obj &
            tttvxfer.obj &
            s_ttt.obj
#  Libs needed to build your app
EXE_LIBS = penpoint app win input xfer xtemplt
#  The .res files for your project; typically these will be bitmap resources
#  and/or compiled resource compiler files.  If you have resources, add
#  $(APP_DIR)\app.res to the "all" target.
RES_FILES = lgicon.res smicon.res tttqhelp.res tttmisc.res
#  Targets
all: $(APP_DIR)\$(PROJ).exe $(APP_DIR)\app.res .SYMBOLIC
#  Install the help files. Note that you should copy the help files
#    in the reverse order from how you want them to appear in the help
#    notebook.
help :: .SYMBOLIC
 mkdir $(PENPOINT_PATH)\app\ttt\help
 mkdir $(PENPOINT_PATH)\app\ttt\help\ttthelp2
 copy strat.txt $(PENPOINT_PATH)\app\ttt\help\ttthelp2\help.txt
 mkdir $(PENPOINT_PATH)\app\ttt\help\ttthelp1
 copy rules.txt $(PENPOINT_PATH)\app\ttt\help\ttthelp1\help.txt
 -$(STAMP) $(PENPOINT_PATH)\app\ttt\help /g "Tic-Tac-Toe Rules" /d ttthelp1
 -$(STAMP) $(PENPOINT_PATH)\app\ttt\help /g "Tic-Tac-Toe Strategy" /d ttthelp2
#  Install the stationery files. Each stationery file must be in a separate
#     directory off of statnry. Here, we also stamp the stationery directories
#     with meaningful names. Note that the "filled" stationery directory is also
#     stamped with attributes so that it appears in the "Create" menu.
stationery :: .SYMBOLIC
 mkdir $(PENPOINT_PATH)\app\ttt\statnry
 mkdir $(PENPOINT_PATH)\app\ttt\statnry\tttstat1
 copy filled.txt $(PENPOINT_PATH)\app\ttt\statnry\tttstat1\tttstuff.txt
 mkdir $(PENPOINT_PATH)\app\ttt\statnry\tttstat2
 copy xsonly.txt $(PENPOINT_PATH)\app\ttt\statnry\tttstat2\tttstuff.txt
 -$(STAMP) $(PENPOINT_PATH)\app\ttt\statnry /g "Tic-Tac-Toe (filled)" /d
tttstat1 /a 00800274 1
 -$(STAMP) $(PENPOINT_PATH)\app\ttt\statnry /g "Tic-Tac-Toe (X's)" /d tttstat2
#  The clean rule must be :: because it is also defined in srules
clean :: .SYMBOLIC
 -del methods.h
 -del $(APP_DIR)\help\ttthelp1\*.*
 -del $(APP_DIR)\help\ttthelp2\*.*
 -del $(APP_DIR)\help\*.*
 -rmdir $(APP_DIR)\help\ttthelp1
 -rmdir $(APP_DIR)\help\ttthelp2
 -rmdir $(APP_DIR)\help
 -del $(APP_DIR)\statnry\tttstat1\*.*
 -del $(APP_DIR)\statnry\tttstat2\*.*
 -del $(APP_DIR)\statnry\*.*
 -rmdir $(APP_DIR)\statnry\tttstat1
 -rmdir $(APP_DIR)\statnry\tttstat2
 -rmdir $(APP_DIR)\statnry
#  Dependencies
methods.obj: methods.tbl tttpriv.h tttdata.h tttview.h tttapp.h
tttapp.obj: methods.h tttapp.c tttview.h tttapp.h tttdata.h tttpriv.h
tttdata.obj: methods.h tttdata.c tttdata.h tttpriv.h
tttview.obj: methods.h tttview.c tttdata.h tttview.h tttpriv.h
tttvopt.obj: tttvopt.c tttview.h
tttdbg.obj: tttdbg.c tttpriv.h tttdata.h
tttmbar.obj: tttmbar.c tttpriv.h tttapp.h
tttvxfer.obj: tttvxfer.c tttview.h tttdata.h
#  Standard rules for sample code
!INCLUDE $(PENPOINT_PATH)\sdk\sample\srules.mif
```

Stationery Source Files

XSONLY.TXT

```
x x x x x stationery for tttapp
```

FILLED.TXT

```
xoxoxoxox stationery for tttapp
```

Help Text Source Files

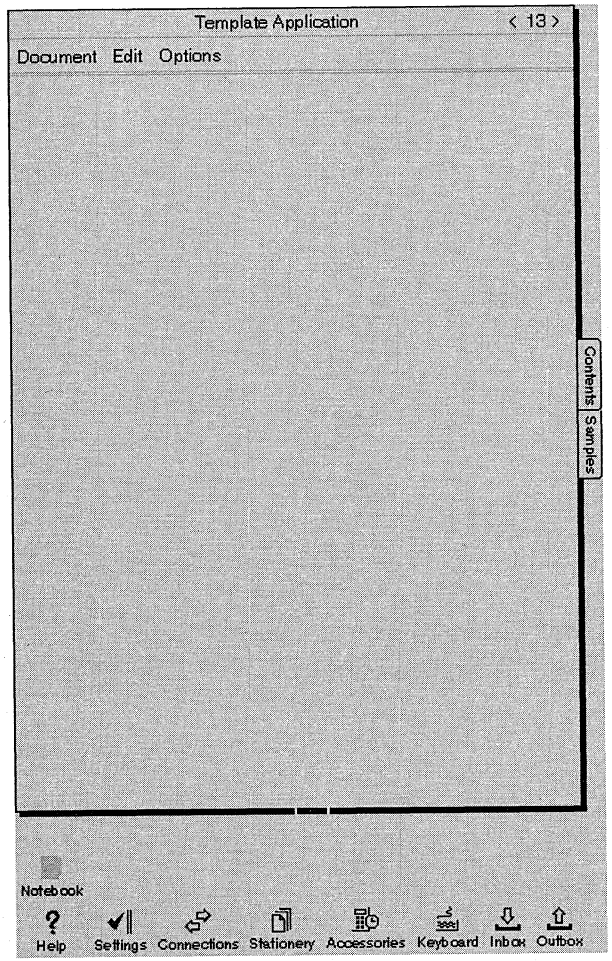
RULES.TXT

```
Tic-Tac-Toe is a simple game for two players. The players take turns writing
X's and O's in the grid.
The player that gets three X's or three O's in a row (across, down, or
diagonally) wins.
```

STRAT.TXT

```
The first player should put her X (or O) in the center square. By doing so,
she increases the
possibility of getting three X's (or O's) in a row.
```

Template Application



The template application provides a template for a full-featured application.

As its name implies, Template Application is a template, "cookie cutter" application. As such, it does not exhibit much functionality. However, it does handle many "typical" application messages. This aspect makes Template Application a good starting point for building a real application.

Objectives

Template Application serves as a shell of an application and can be used as a starting point for a real application.

This sample application also shows how to:

- File instance data
- Create the standard menu bar and add applicationspecific menus
- Create an icon window as a client window.

Class Overview

Template Application defines two classes: **clsTemplateApp** and **clsFoo**. It makes use of the following classes:

clsApp

clsAppMgr

clsClass

clsIconWin

clsMenu

clsObject

Compiling

To compile TemplateApp, just

```
cd \penpoint\sdk\sample\templtap
wmake
```

This compiles the application and creates TEMPLTAP.EXE and APP.RES in \PENPOINT\APP\TEMPLTAP.

Running

After compiling TemplateApp, you can run it by

1. Adding \\BOOT\PENPOINT\APP\Template Application to \PENPOINT\BOOT\APP.INI
2. Booting PenPoint
3. Creating a new Template Application document, and turning to it.

Alternatively, you can boot PenPoint and then install Template Application through the Connections Notebook.

Files Used

The code for Template Application is in \PENPOINT\SDK\SAMPLE\TEMPLTAP. The files are:

FOO.C the source code for the foo class

FOO.H the header file for the foo class

METHODS.TBL the list of messages that the classes respond to, and the associated message handlers to call

TEMPLATE.RC resource file containing error message strings

TEMPLTAP.C the source code for the application class

TEMPLTAP.H the header file for the application class.

METHODS.TBL

```
/*****************************************************************************
 File: methods.tbl
 Copyright 1991, 1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.0  $
     $Date:   07 Jan 1992 16:57:42  $
 This file contains the method table definitions for templtap.exe.
*****************************************************************************/
#include <app.h>
#include <templtap.h>
#include <foo.h>
MSG_INFO clsTemplateAppMethods[] = {
{msgDump,            "TemplateAppDump",          objCallAncestorBefore    },
{msgInit,            "TemplateAppNew",           objCallAncestorBefore    },
{msgFree,            "TemplateAppFree",          objCallAncestorAfter     },
{msgSave,            "TemplateAppSave",          objCallAncestorBefore    },
{msgRestore,         "TemplateAppRestore",       objCallAncestorBefore    },
{msgAppInit,         "TemplateAppInit",          objCallAncestorBefore    },
{msgAppOpen,         "TemplateAppOpen",          objCallAncestorAfter     },
{msgAppClose,        "TemplateAppClose",         objCallAncestorBefore    },
{msgAppCreateClientWin,  "TemplateAppCreateClientWin",                    },
{msgAppCreateMenuBar,"TemplateAppCreateMenuBar",                          },
{msgTemplateAppGetMetrics,       "TemplateAppGetMetrics",                 },
{0}
};
MSG_INFO clsFooMethods[] = {
{msgNewDefaults,     "FooNewDefaults",       objCallAncestorBefore        },
{msgDump,            "FooDump",              objCallAncestorBefore        },
{msgInit,            "FooNew",               objCallAncestorBefore        },
{msgFree,            "FooFree",              objCallAncestorAfter         },
{msgSave,            "FooSave",              objCallAncestorBefore        },
{msgRestore,         "FooRestore",           objCallAncestorBefore        },
{msgFooGetStyle,     "FooGetStyle",                                       },
{msgFooSetStyle,     "FooSetStyle",                                       },
{msgFooGetMetrics,       "FooGetMetrics",                                 },
{0}
};
CLASS_INFO classInfo[] = {
{"clsTemplateAppTable",      clsTemplateAppMethods   },
{"clsFooTable",              clsFooMethods           },
{0}
};
```

TEMPLTAP.H

```
/*****************************************************************************
 File: templtap.h
 Copyright 1991, 1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.2  $
     $Date:   13 Nov 1991 18:19:46  $
 This file contains the templateapp application API.
*****************************************************************************/
#ifndef TEMPLATEAPP_INCLUDED
#define TEMPLATEAPP_INCLUDED
#include <clsmgr.h>
#define clsTemplateApp                      MakeWKN(3513, 1, wknGlobal)
// Status codes.
#define stsTemplateAppError1            MakeStatus(clsTemplateApp, 1)
#define stsTemplateAppError2            MakeStatus(clsTemplateApp, 2)
```

```
// Quick Help codes.
#define qhTemplateAppQuickHelp1              MakeTag(clsTemplateApp, 1)
#define qhTemplateAppQuickHelp2              MakeTag(clsTemplateApp, 2)

typedef OBJECT TEMPLATEAPP, *P_TEMPLATEAPP;

typedef struct TEMPLATEAPP_METRICS {
     U32     dummy;
     U32     reserved;
} TEMPLATEAPP_METRICS, *P_TEMPLATEAPP_METRICS;

/*****************************************************************************
 msgTemplateAppGetMetrics takes P_TEMPLATEAPP_METRICS, returns STATUS
     Get TemplateApp metrics.
*/
#define msgTemplateAppGetMetrics                  MakeMsg(clsTemplateApp, 1)

#endif    // TEMPLATEAPP_INCLUDED
```

TEMPLTAP.C

```
/*****************************************************************************
 File: templtap.c

 Copyright 1991, 1992 GO Corporation. All Rights Reserved.

 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL, INCIDENTAL, OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.

 $Revision:   1.2  $
     $Date:   07 Jan 1992 16:57:28  $

 This file contains the templtap application.
*****************************************************************************/
#include <methods.h>
#include <templtap.h>
#include <app.h>
#include <appmgr.h>
#include <resfile.h>
#include <string.h>
#include <frame.h>
#include <os.h>
#include <debug.h>
#include <iconwin.h>

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Defines, Types, Globals, Etc                     *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

static char *company = "GO Corporation";
static char *copyright    = "Copyright \213 1991, 1992\nby GO Corporation,\nAll
Rights Reserved.";
static char *version = "1.0";
static char *defaultDocName  = "Template Application";

typedef struct TEMPLATEAPP_INST {
     TEMPLATEAPP_METRICS metrics;
} TEMPLATEAPP_INST, *P_TEMPLATEAPP_INST;

typedef struct FILED_DATA {
     TEMPLATEAPP_METRICS metrics;
} FILED_DATA, *P_FILED_DATA;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Message Handlers                                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************
     TemplateAppDump

     Respond to msgDump.
*****************************************************************************/
MsgHandlerArgType(TemplateAppDump, P_ARGS)
{
     Debugf("templtap: msgDump");
     return stsOK;
     MsgHandlerParametersNoWarning;
}    // TemplateAppDump

/*****************************************************************************
     TemplateAppNew

     Respond to msgInit.
*****************************************************************************/
MsgHandlerArgType(TemplateAppNew, P_APP_NEW)
{
     return stsOK;
     MsgHandlerParametersNoWarning;
}    // TemplateAppNew

/*****************************************************************************
     TemplateAppFree

     Respond to msgFree.
*****************************************************************************/
MsgHandlerArgType(TemplateAppFree, P_ARGS)
{
     return stsOK;
     MsgHandlerParametersNoWarning;
}    // TemplateAppFree

/*****************************************************************************
     TemplateAppSave

     Respond to msgSave.
```

```
*****************************************************************************/
MsgHandlerArgType(TemplateAppSave, P_OBJ_SAVE)
{
    P_TEMPLATEAPP_INST      pInst;
    STREAM_READ_WRITE       fsWrite;
    FILED_DATA              filed;
    STATUS                  s;

    pInst = IDataPtr(pData, TEMPLATEAPP_INST);
    memset(&filed, 0, sizeof(FILED_DATA));

    filed.metrics = pInst->metrics;

    // Write filed data to the file.
    fsWrite.numBytes    = SizeOf(FILED_DATA);
    fsWrite.pBuf        = &filed;
    ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s);

    return stsOK;
    MsgHandlerParametersNoWarning;
}   // TemplateAppSave
/*****************************************************************************
    TemplateAppRestore

    Respond to msgRestore.
*****************************************************************************/
MsgHandlerArgType(TemplateAppRestore, P_OBJ_RESTORE)
{
    STREAM_READ_WRITE   fsRead;
    TEMPLATEAPP_INST    inst;
    FILED_DATA          filed;
    STATUS              s;

    memset(&inst, 0, sizeof(TEMPLATEAPP_INST));

    // Read instance data from the file.
    fsRead.numBytes = SizeOf(FILED_DATA);
    fsRead.pBuf     = &filed;
    ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s);

    inst.metrics = filed.metrics;

    // Update instance data.
    ObjectWrite(self, ctx, &inst);

    return stsOK;
    MsgHandlerParametersNoWarning;
}   // TemplateAppRestore
/*****************************************************************************
    TemplateAppInit

    Respond to msgAppInit.  Perform one-time initializations.
*****************************************************************************/
MsgHandlerArgType(TemplateAppInit, DIR_HANDLE)
{
    APP_METRICS     am;
    OBJECT          win;
    STATUS          s;

    // Create the client win.
    win = objNull;
    ObjCallRet(msgAppCreateClientWin, self, &win, s);

    // Get the main window.
    ObjCallRet(msgAppGetMetrics, self, &am, s);

    // Set the client win.
    ObjCallRet(msgFrameSetClientWin, am.mainWin, (P_ARGS)win, s);

    return stsOK;
    MsgHandlerParametersNoWarning;
}   // TemplateAppInitApp
/*****************************************************************************
    TemplateAppOpen

    Respond to msgAppOpen.
*****************************************************************************/
MsgHandlerArgType(TemplateAppOpen, P_APP_OPEN)
{
    FRAME_METRICS       fm;
    APP_METRICS         am;
    OBJECT              menuBar;
    STATUS              s;

    // Create the menu bar.
    menuBar = objNull;
    ObjCallRet(msgAppCreateMenuBar, self, &menuBar, s);

    // Get the main window.
    ObjCallRet(msgAppGetMetrics, self, &am, s);

    // Insert the menu bar.
    ObjCallRet(msgFrameSetMenuBar, am.mainWin, (P_ARGS)menuBar, s);

    // Set the childAppParentWin.
    ObjCallRet(msgFrameGetMetrics, am.mainWin, &fm, s);

    pArgs->childAppParentWin = fm.clientWin;

    return stsOK;
    MsgHandlerParametersNoWarning;
}   // TemplateAppOpen
/*****************************************************************************
    TemplateAppClose

    Respond to msgAppClose.
*****************************************************************************/
MsgHandlerArgType(TemplateAppClose, P_ARGS)
{
    APP_METRICS     am;
    STATUS          s;

    ObjCallRet(msgAppGetMetrics, self, &am, s);

    ObjCallRet(msgFrameDestroyMenuBar, am.mainWin, pNull, s);

    return stsOK;
```

```
    MsgHandlerParametersNoWarning;
}   // TemplateAppClose
/*****************************************************************************
    TemplateAppCreateClientWin

    Respond to msgAppCreateClientWin.
*****************************************************************************/
MsgHandlerArgType(TemplateAppCreateClientWin, P_OBJECT)
{
    ICON_WIN_NEW    iwn;
    // If the client win has already beed provided, return.
    if (*pArgs != objNull) {
        return stsOK;
    }
    // Create an iconwin.
    ObjectCall(msgNewDefaults, clsIconWin, &iwn);
    iwn.iconWin.style.showOptions       = true;
    iwn.iconWin.style.allowOpenInPlace  = true;
    iwn.iconWin.style.constrainedLayout = false;
    ObjCallWarn(msgNew, clsIconWin, &iwn);
    // Return the client win.
    *pArgs = iwn.object.uid;
    return stsOK;
    MsgHandlerParametersNoWarning;
}   // TemplateAppCreateClientWin
/*****************************************************************************
    TemplateAppCreateMenuBar

    Respond to msgAppCreateMenuBar.
*****************************************************************************/
MsgHandlerArgType(TemplateAppCreateMenuBar, P_OBJECT)
{
    STATUS          s;
    //MENU_NEW      mn;

    // If a sub-class provided the menu bar...
    if (*pArgs != objNull) {
        // Pass message to ancestor - Add SAMs.
        return ObjectCallAncestor(msg, self, pArgs, ctx);
    }
    // Create your menu bar here...
    //ObjectCall(msgNewDefaults, clsMenu, &mn);
    //mn.win.flags.style    &= ~wsSendFile;
    //mn.tkTable.pEntries   = menuBar;
    //mn.tkTable.client     = self;
    //ObjCallRet(msgNew, clsMenu, &mn, s);
    //*pArgs = mn.object.uid;
    // Pass message to ancestor - Add SAMs.
    ObjCallAncestorRet(msg, self, pArgs, ctx, s);
    // Fixup the menu bar here...
    return stsOK;
    MsgHandlerParametersNoWarning;
}   // TemplateAppCreateMenuBar

/*****************************************************************************
    TemplateAppGetMetrics

    Get TemplateApp metrics.
*****************************************************************************/
MsgHandlerArgType(TemplateAppGetMetrics, P_TEMPLATEAPP_METRICS)
{
    P_TEMPLATEAPP_INST      pInst;
    pInst = IDataPtr(pData, TEMPLATEAPP_INST);
    *pArgs = pInst->metrics;
    return stsOK;
    MsgHandlerParametersNoWarning;
}   // TemplateAppGetMetrics
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Installation                                       *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************
    ClsTemplateAppInit

    Install the application.
*****************************************************************************/
STATUS PASCAL
ClsTemplateAppInit (void)
{
    APP_MGR_NEW     new;
    STATUS          s;
    ObjectCall(msgNewDefaults, clsAppMgr, &new);
    new.object.uid          = clsTemplateApp;
    new.object.key          = 0;
    new.cls.pMsg            = clsTemplateAppTable;
    new.cls.ancestor        = clsApp;
    new.cls.size            = SizeOf(TEMPLATEAPP_INST);
    new.cls.newArgsSize     = SizeOf(APP_NEW);
    new.appMgr.copyright    = copyright;
    RectInit(&new.appMgr.defaultRect, 0, 0, 216, 108);
    strcpy(new.appMgr.defaultDocName, defaultDocName);
    strcpy(new.appMgr.company, company);
    strcpy(new.appMgr.version, version);
    ObjCallRet(msgNew, clsAppMgr, &new, s);
    return stsOK;
}
STATUS ClsFooInit (void);
```

```
/*****************************************************************************
    main

    Main application entry point.
*****************************************************************************/
void CDECL
main (
    int     argc,
    char    *argv[],
    int     processCount)
{
    if (processCount == 0) {
        // Create the application class.
        StsWarn(ClsTemplateAppInit ());
        // Create global classes.
        StsWarn(ClsFooInit());
        // Start msg dispatching.
        AppMonitorMain(clsTemplateApp, objNull);
    }
    else {
        // Start msg dispatching.
        AppMain();
    }
    Unused(argc); Unused(argv);
}   // main
```

FOO.H

```
/*****************************************************************************
 File: foo.h
 Copyright 1991, 1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.2  $
    $Date:   13 Nov 1991 18:19:48  $
 This file contains the API definition for clsFoo.
*****************************************************************************/
#ifndef FOO_INCLUDED
#define FOO_INCLUDED
#include <clsmgr.h>
typedef OBJECT FOO, *P_FOO;
#define clsFoo                          MakeWKN(3514,1,wknGlobal)
// Status codes.
#define stsFooError1                MakeStatus(clsFoo, 1)
#define stsFooError2                MakeStatus(clsFoo, 2)
// Quick Help codes.
#define qhFooQuickHelp1                 MakeTag(clsFoo, 1)
#define qhFooQuickHelp2                 MakeTag(clsFoo, 2)
typedef struct FOO_STYLE {
    U16     style1      :1;
    U16     style2      :1;
    U16     reserved    :14;
} FOO_STYLE, *P_FOO_STYLE;
typedef struct FOO_METRICS {
    FOO_STYLE   style;
    U32         reserved1[2];           // Reserved.
    U16         reserved2       :16;    // Reserved.
} FOO_METRICS, *P_FOO_METRICS;
/*****************************************************************************
 msgNew  takes P_FOO_NEW, returns STATUS
    Create a new object.
*/
typedef struct FOO_NEW_ONLY {
    // Your new parameters here...
    FOO_STYLE   style;
    U32         reserved;
} FOO_NEW_ONLY, *P_FOO_NEW_ONLY;
#define  fooNewFields    \
    objectNewFields     \
    FOO_NEW_ONLY        foo;
typedef struct FOO_NEW {
    fooNewFields
} FOO_NEW, *P_FOO_NEW;
/*****************************************************************************
 msgFooGetMetrics takes P_FOO_METRICS, returns STATUS
    Get foo metrics.
*/
#define msgFooGetMetrics                MakeMsg(clsFoo, 1)
/*****************************************************************************
 msgFooGetStyle      takes P_FOO_STYLE, returns STATUS
    Get foo style.
*/
#define msgFooGetStyle                      MakeMsg(clsFoo, 2)
/*****************************************************************************
 msgFooSetStyle      takes P_FOO_STYLE, returns STATUS
    Set foo style.
*/
#define msgFooSetStyle                      MakeMsg(clsFoo, 3)
```

```
#endif    // FOO_INCLUDED
```

FOO.C

```
/*****************************************************************************
 File: foo.c
 Copyright 1991, 1992 GO Corporation. All Rights Reserved.
 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.2  $
    $Date:   07 Jan 1992 16:57:22  $
 This file contains the class definition and methods for clsFoo.
*****************************************************************************/
#include <methods.h>
#include <foo.h>
#include <string.h>
#include <debug.h>
#include <resfile.h>

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Defines, Types, Globals, Etc                       *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
typedef struct FOO_INST {
    FOO_METRICS     metrics;
} FOO_INST, *P_FOO_INST;
typedef struct FILED_DATA {
    // Your filed instance data here...
    FOO_STYLE   style;
    U32         reserved1;
    U16         reserved2       :16;    // Reserved.
} FILED_DATA, *P_FILED_DATA;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                      Message Handlers                                   *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*****************************************************************************
    FooNewDefaults

    Respond to msgNewDefaults.
*****************************************************************************/
MsgHandlerArgType(FooNewDefaults, P_FOO_NEW)
{
    memset(&(pArgs->foo), 0, sizeof(FOO_NEW_ONLY));
    pArgs->foo.style.style1     = false;
    pArgs->foo.style.style2     = false;
    pArgs->foo.reserved         = 0;
    return stsOK;
    MsgHandlerParametersNoWarning;
}    // FooNewDefaults
/*****************************************************************************
    FooDump
    Respond to msgDump.
*****************************************************************************/
MsgHandlerArgType(FooDump, P_ARGS)
{
    Debugf("foo: msgDump");
    return stsOK;
    MsgHandlerParametersNoWarning;
}    // FooDump
/*****************************************************************************
    FooNew

    Respond to msgInit.  Create a new object.
*****************************************************************************/
MsgHandlerArgType(FooNew, P_FOO_NEW)
{
    FOO_INST    inst;
    memset(&inst, 0, sizeof(FOO_INST));
    // Update instance data.
    ObjectWrite(self, ctx, &inst);
    return stsOK;
    MsgHandlerParametersNoWarning;
}  // FooNew
/*****************************************************************************
    FooFree

    Destroy an object.
*****************************************************************************/
MsgHandlerArgType(FooFree, P_ARGS)
{
    return stsOK;
    MsgHandlerParametersNoWarning;
}  // FooFree
/*****************************************************************************
    FooSave

    Save self to a file.
*****************************************************************************/
MsgHandlerArgType(FooSave, P_OBJ_SAVE)
```

```
{
    P_FOO_INST          pInst;
    STREAM_READ_WRITE   fsWrite;
    FILED_DATA          filed;
    STATUS              s;

    pInst = IDataPtr(pData, FOO_INST);
    memset(&filed, 0, sizeof(FILED_DATA));

    filed.style = pInst->metrics.style;

    // Write filed instance data to the file.
    fsWrite.numBytes    = SizeOf(FILED_DATA);
    fsWrite.pBuf        = &filed;
    ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s);

    return stsOK;
    MsgHandlerParametersNoWarning;
}   // FooSave
/*****************************************************************************
    FooRestore

    Restore self from a file.
*****************************************************************************/
MsgHandlerArgType(FooRestore, P_OBJ_RESTORE)
{
    STREAM_READ_WRITE   fsRead;
    FOO_INST            inst;
    FILED_DATA          filed;
    STATUS              s;

    memset(&inst, 0, sizeof(FOO_INST));

    // Read instance data from the file.
    fsRead.numBytes = SizeOf(FILED_DATA);
    fsRead.pBuf     = &filed;
    ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s);

    inst.metrics.style  = filed.style;

    // Update instance data.
    ObjectWrite(self, ctx, &inst);

    return stsOK;
    MsgHandlerParametersNoWarning;
}   // FooRestore

/*****************************************************************************
    FooGetStyle

    Get foo style.
*****************************************************************************/
MsgHandlerArgType(FooGetStyle, P_FOO_STYLE)
{
    P_FOO_INST      pInst = IDataPtr(pData, FOO_INST);
    MsgHandlerParametersNoWarning;

    *pArgs = pInst->metrics.style;

    return stsOK;
}   // FooGetStyle
/*****************************************************************************
    FooSetStyle

    Set foo style.
*****************************************************************************/
MsgHandlerArgType(FooSetStyle, P_FOO_STYLE)
{
    FOO_INST        selfInst;
    P_FOO_INST      pInst;

    selfInst = IDataDeref(pData, FOO_INST);
    pInst   = &selfInst;

    // Update instance data.
    pInst->metrics.style = *pArgs;
    ObjectWrite(self, ctx, pInst);

    return stsOK;
    MsgHandlerParametersNoWarning;
}   // FooSetStyle
/*****************************************************************************
    FooGetMetrics

    Get foo metrics.
*****************************************************************************/
MsgHandlerArgType(FooGetMetrics, P_FOO_METRICS)
{
    P_FOO_INST      pInst;

    pInst = IDataPtr(pData, FOO_INST);
    *pArgs = pInst->metrics;

    return stsOK;
    MsgHandlerParametersNoWarning;
}   // FooGetMetrics
/*****************************************************************************
    ClsFooInit

    Install the class.
*****************************************************************************/
STATUS ClsFooInit (void)
{
    CLASS_NEW       new;
    STATUS          s;

    // Create the class.
    ObjectCall(msgNewDefaults, clsClass, &new);
    new.object.uid      = clsFoo;
    new.object.key      = (OBJ_KEY)clsFooTable;
    new.cls.pMsg        = clsFooTable;
    new.cls.ancestor    = clsObject;
    new.cls.size        = SizeOf(FOO_INST);
    new.cls.newArgsSize = SizeOf(FOO_NEW);
```

```
    ObjCallRet(msgNew, clsClass, &new, s);
    return stsOK;
}  // ClsFooInit
```

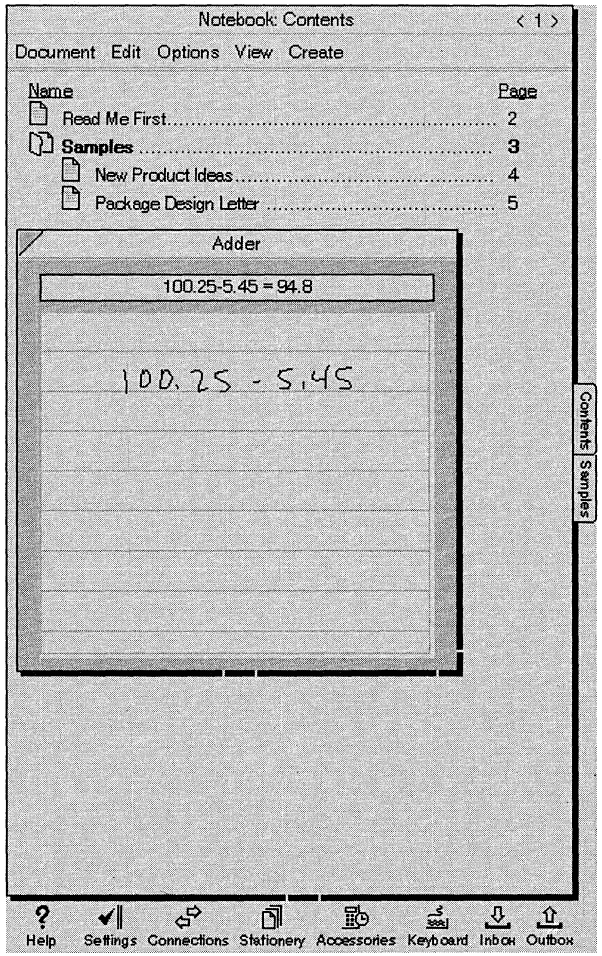
TEMPLATE.RC

```
/******************************************************************************
 templt.rc

 Copyright 1991, 1992 GO Corporation. All Rights Reserved.

 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.

 $Revision:   1.0  $
     $Date:   13 Nov 1991 18:19:50  $

 This file contains the string table resources used by the standard
 system message facility, StdMessage.  Each resource table represents
 the strings for the status values for a given class.  The administered
 portion of the status value is used as the resource identifier.  The
 value portion of the status value is used as an index into the string
 table resource.
******************************************************************************/
#include <rescmplr.h>
#include <stdmsg.h>
#include <apptag.h>
#include <templtap.h>
#include <foo.h>

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
//                          Std Messages - clsTemplateApp                    //
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

static P_STRING clsTemplateAppStrings[] = {
"",                                 // None
"Error 1",                          // stsTemplateAppError1
"Error 2",                          // stsTemplateAppError2
pNull
};

static RC_INPUT  clsTemplateAppStringTable = {
resForStdMsgError(clsTemplateApp),
clsTemplateAppStrings,
0,
resStringArrayResAgent
};

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
//                          Std Messages - clsFoo                            //
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

static P_STRING clsFooStrings[] = {
"",                                 // None
"Error 1",                          // stsFooError1
"Error 2",                          // stsFooError2
pNull
};

static RC_INPUT  clsFooStringTable = {
resForStdMsgError(clsFoo),
clsFooStrings,
0,
resStringArrayResAgent
};

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
//                          Resource Tables                                  //
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

P_RC_INPUT   resInput [] = {
&clsTemplateAppStringTable,
&clsFooStringTable,
pNull
};
```

MAKEFILE

```
##############################################
#
#  WMake Makefile for the templtap application
#
# Copyright 1991, 1992 GO Corporation. All Rights Reserved.
#
# You may use this Sample Code any way you please provided you
# do not resell the code and that this notice (including the above
# copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
# IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
# EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
# LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
# PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
# FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
# THE USE OR INABILITY TO USE THIS SAMPLE CODE.
#
# $Revision:   1.3  $
#     $Date:   07 Jan 1992 16:57:32  $
#
##############################################
!ifdef %PENPOINT_PATH
PENPOINT_PATH = $(%PENPOINT_PATH)
```

```
!else
PENPOINT_PATH = \penpoint
!endif
#  The DOS name of your project directory.
PROJ = templtap
#  Standard defines for sample code (needs the PROJ) definition
!INCLUDE $(PENPOINT_PATH)\sdk\sample\sdefines.mif
#  The PenPoint name of your application
EXE_NAME  = Template Application
#  The linker name for your executable : company-name-V<major>(<minor>)
EXE_LNAME= GO-TEMPLATE_APP-V1
#  Object files needed to build your app
EXE_OBJS = methods.obj templtap.obj foo.obj
#  Libs needed to build your app
EXE_LIBS = penpoint app
RES_FILES = template.res
# Targets
all: $(APP_DIR)\$(PROJ).exe $(APP_DIR)\app.res .SYMBOLIC
#  The clean rule must be :: because it is also defined in srules
clean :: .SYMBOLIC
 -del methods.h
 -del methods.tc
 -del app.res
#  Dependencies
templtap.obj: templtap.c templtap.h methods.h
foo.obj: foo.c foo.h methods.h
methods.obj: methods.tbl templtap.h foo.h
#  Standard rules for sample code
!INCLUDE $(PENPOINT_PATH)\sdk\sample\srules.mif
```

Adder is a simple pencentric calculator, limited to addition and subtraction. The user can write "4+5" and Adder will print "4+5=9" at the top of its window. In addition, Adder can handle slightly more complicated expressions, such as "42 + 8 + 3 -2.5".

Objectives

This sample application shows how to:

- Create an insertion pad for handwritten input
- Create a translator and a custom template for the insertion pad
- Translate the insertion pad ink when the user lifts the pen out of proximity

- Disable some of the handwriting engine's assumptions to improve arithmetic recognition
- Create a custom layout window
- Construct a simple parser.

Class Overview

Adder defines two classes: **clsAdderApp** and **clsAdderEvaluator**. It makes use of the following classes:

clsApp
clsAppMgr
clsClass
clsCustomLayout
clsIP
clsLabel
clsObject
clsXText

Compiling

To compile Adder, just

```
cd \penpoint\sdk\sample\adder
wmake
```

This compiles the application, and creates ADDER.EXE in \PENPOINT\APP\ADDER.

Running

After compiling Adder, you can run it by

1. Adding \\boot\penpoint\app\Adder to \PENPOINT\BOOT\APP.INI
2. Booting PenPoint
3. Tapping on the Accessories icon to open it, and then tapping on "Adder."

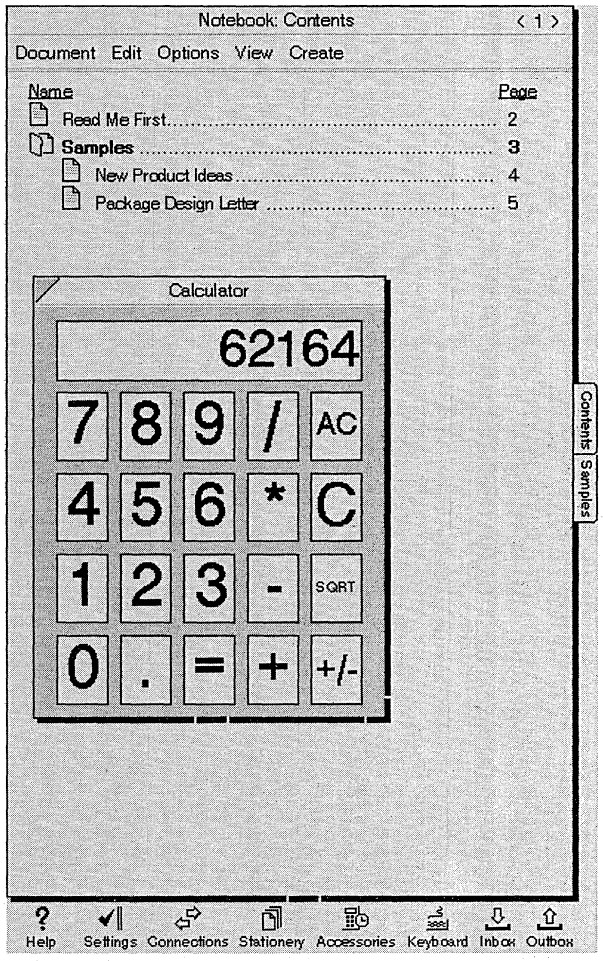
Alternatively, you can boot PenPoint and then install Adder via the Connections Notebook.

Files Used

The code for Adder is in \PENPOINT\SDK\SAMPLE\ADDER. The files are:

ADDERAPP.C the source code for the application class
ADDEREVL.C the source code for the adder evaluator engine class
ADDEREVL.H the header file for the evaluator class
METHODS.TBL the method tables for the adder classes.

Calculator



The calculator application implements a typical pushbutton calculator. This program is split into an application, which handles the user interface, and a calculator engine, which performs the computations.

Objectives

This sample application shows how to:

- Separate out part of an application into a reuseable dynamic link library
- Have an application be an accessory
- Use ClsSymbolsInit()
- Use table layout and custom layout
- Use labels
- Use **TK_TABLE_ENTRY** struct to create a collection of buttons in one fell swoop
- Handle ButtonNotifystyle messages
- Change the default window size
- File data.

Class Overview

Calc defines two classes: **clsCalcApp** and **clsCalcEngine**. It makes use of the following classes:

clsAppMgr
clsClass
clsCustomLayout
clsLabel
clsObject
clsTkTable

When **clsCalcApp** receives **msgAppOpen**, it creates a set of windows (all of which are standard UI components):

- A table of buttons (**clsTkTable**) for the calculator's push buttons
- A label (**clsLabel**) for the calculator's display
- A window (**clsCustomLayout**) to hold the label and the button table.

The application destroys these windows when it receives **msgAppClose**.

In its **msgAppInit** handler, the application creates an instance of **clsCalcEngine**, the calculator engine. This class performs arithmetic operations.

Although **clsCalcApp** does not file any of its views, it does file the string that is displayed in its label. It also files the calculator engine object by sending it **msgResPutObject** (in response to **msgSave**) and **msgResGetObject** (in response to **msgRestore**).

Compiling

To compile Calc, just

```
cd \penpoint\sdk\sample\calc
wmake
```

This compiles the dynamic link library and the application, and creates CALC.DLL and CALC.EXE in \PENPOINT\APP\CALC. It also copies CALC.DLC to \PENPOINT\APP\CALC.

Running

After compiling Calc, you can run it by

1. Adding \\boot\penpoint\app\Calculator to \PENPOINT\BOOT\APP.INI
2. Booting PenPoint
3. Tapping on the Accessories icon to open it, and then tapping on "Calculator."

Alternatively, you can boot PenPoint and then install Calculator via the Connections Notebook.

Files Used

The code for Calc is in \PENPOINT\SDK\SAMPLE\CALC. The files are:

CALC.DLC indicates the dependency of the calculator application upon the calculator engine

CALCAPP.C clsCalcApp's code and initialization

CALCAPP.H header file for the application class

CALCENG.C clsCalcEng's code and initialization

CALCENG.H header file for the calculator engine

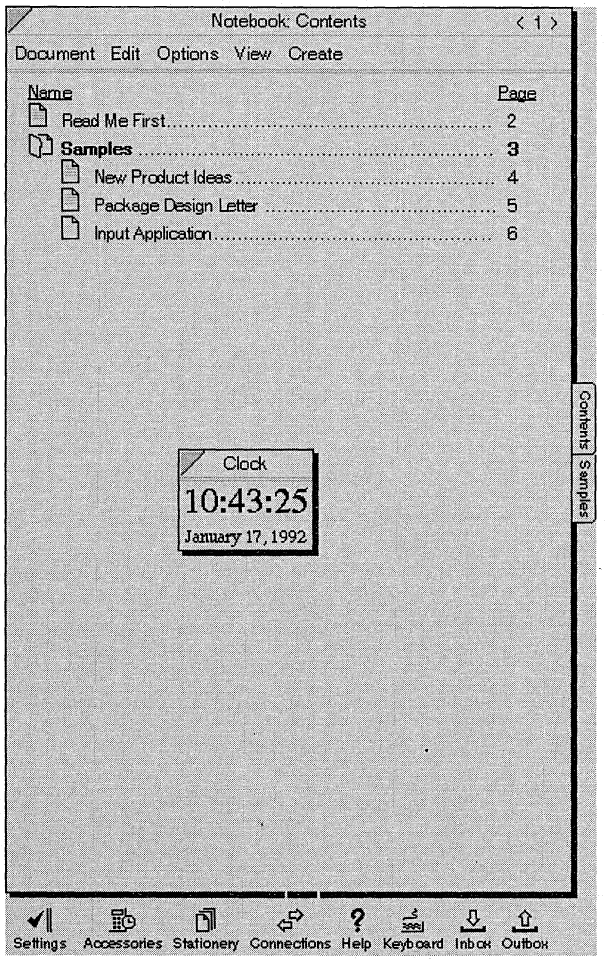
CAPPMETH.TBL method table for the application

CENGMETH.TBL method table for the engine

DLL.LBC list of exported functions for the Watcom linker

S_CALC.C symbol name definitions and call to ClsSymbolsInit().

Clock



Clock is an application that serves two purposes: a digital alarm clock distributed as a part of PenPoint, and a sample application.

The enduser can configure the clock's display by changing the placement of the time and date and by specifying things like whether the time should include seconds. The enduser can also set up an alarm. Depending on how the user configures an alarm, it might beep at a certain time on a certain day or display a note every day at the same time.

SAMPLE CODE

Objectives

This sample application shows how to:

- Observe the system preferences for changes to date/time formats
- Observe the power switch to refresh on powerup
- Provide option cards for an application
- Destroy unneeded controls on a default application option card
- Disable inappropriate controls on a default application option card
- Respond to **msgGWinForwardedGesture** (including handling, forwarding, and ignoring gestures)
- Provide quick help
- Make use of **clsTimer**.

Class Overview

Clock defines four classes: **clsClockLabel**, **clsClockApp**, **clsClockWin**, and **clsNoteIconWin**. It makes use of the following classes:

clsApp
clsAppMgr
clsAppWin
clsClass
clsCommandBar
clsDateField
clsGotoButton
clsIconWin
clsIntegerField
clsLabel
clsNote
clsOptionTable
clsPopupChoice
clsPreferences
clsString
clsTableLayout

clsTextField

clsTimer

clsTkTable

clsToggleTable

clsClockApp

clsNoteIconWin appears as a corkboard on the popup note that Clock displays when an alarm goes off. The note needs to be dismissed when the user opens one of the icons in the window. To provide this functionality, **clsNoteIconWin** observes objects inserted into its window.

Clock uses a table layout of several windows. There can be up to four child windows (time digits, am/pm indicator, alarm indicator, and the date). All of these are labels. **clsLabel** only repaints the rightmost characters that change. So, to minimize flashing as time ticks away, Clock displays the time digits in a separate window from the am/pm indicator.

clsClockApp does not file its labels or client window. It does, however, file most of the settings of the controls in its Display and Alarm option cards. It also files its **clsNoteIconWin** corkboard window.

Compiling

To compile Clock, just

```
cd \penpoint\sdk\sample\clock
wmake
```

This compiles the application and creates CLOCK.EXE and APP.RES in \PENPOINT\APP\CLOCK.

Running

After compiling Clock, you can run it by

1. Adding \\boot\penpoint\app\clock to \PENPOINT\BOOT\APP.INI
2. Booting PenPoint
3. Tapping on the Accessories icon to open it, and then tapping on "Clock."

Note: Most PenPoint configurations list the Clock application in \PENPOINT\BOOT\SYSAPP.INI. If this is done in your configuration, then you can skip step 1 above.

Files Used

The code for Clock is in \PENPOINT\SDK\SAMPLE\CLOCK. The files are:

BITMAP.RES resource file (compiled) for the clock accessory icon

CLABEL.C source code for **clsClockLabel**

CLABEL.H header file for **clsClockLabel**

CLOCK.RC resource file (uncompiled) for the clock's quick help

CLOCKAPP.C source for **clsClockApp**, the application class

CLOCKAPP.H header file for the application class

CWIN.C source code for **clsClockWin**

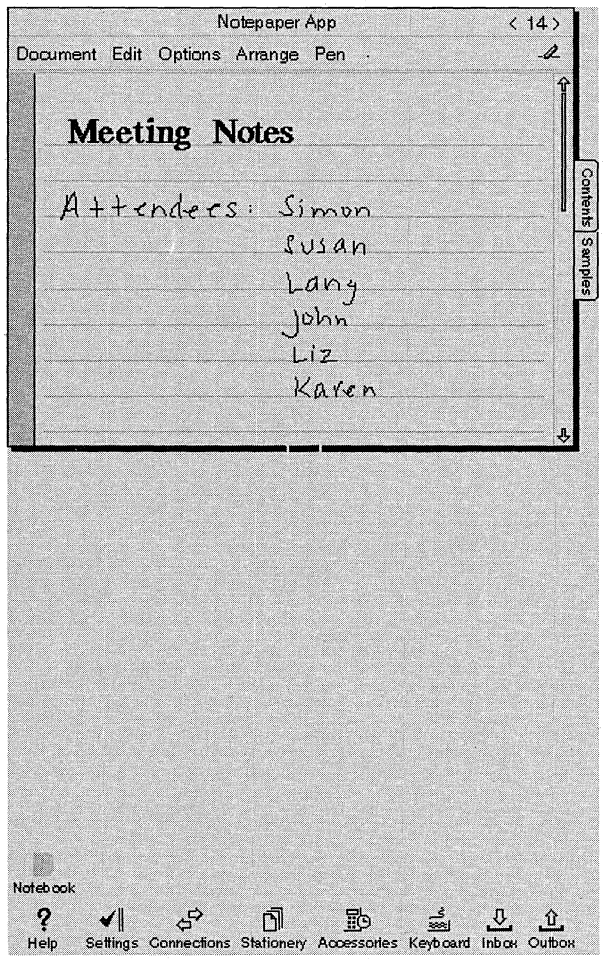
CWIN.H header file for **clsClockWin**

METHOD.TBL the method tables for the four classes

NOTEIWIN.C source code for **clsNoteIconWin**

NOTEIWIN.H header file for **clsNoteIconWin.**

Notepaper App



Notepaper App is a simple notetaking application. It relies on the NotePaper DLL for most of its functionality.

Objectives

This sample application shows how to use the NotePaper DLL.

Class Overview

Notepaper App defines one class: **clsNotePaperApp**. It makes use of the following classes:

clsApp

clsAppMgr

clsGestureMargin

clsNotePaper

Compiling

To compile Notepaper App, just

```
cd \penpoint\sdk\sample\npapp
wmake
```

This compiles the application and creates NPAPP.EXE and APP.RES in \PENPOINT\APP\NPAPP.

Running

After compiling Notepaper App, you can run it by

1. Adding \\boot\penpoint\app\Notepaper App to \PENPOINT\BOOT\APP.INI
2. Booting PenPoint
3. Creating a new Notepaper App document, and turning to it.

Alternatively, you can boot PenPoint and then install Notepaper App via the Connections Notebook.

Files Used

The code for Notepaper App is in \PENPOINT\SDK\SAMPLE\NPAPP. The files are:

BITMAP.RES resource file for the document bitmap

METHODS.TBL method table for the notepaper application class

MODES.RES resource file for the mode icons

NPAPP.C source code for the notepaper application

PAPER.RES resource file for the different paper icons

PENS.RES resource file for the different pen icons.

Paint

Paint is a simple painting application. The user can choose different nibs (square, circle, or italic) and different paint colors (white, light gray, dark gray, and black) with which to paint. The user can easily clear the window to start painting all over again.

Objectives

This sample application shows how to:

- Read and write data and strings to a file
- Provide a totally applicationspecific menu bar (no SAMS)
- Place a button on a menu line
- Create a scroll win, and have a gray border displayed around its client window
- Use pixel maps
- Handle pen input events.

While Paint does demonstrate these topics, it is far from being a perfect sample application, for these reasons:

- Pixelmaps are inherently device dependent, so Paint documents are also device dependent
- When the user changes the screen orientation, Paint does not flush and rebuild its pixelmaps
- Paint's pixelmaps cannot be printed (which is why the Print menu item and the "P" gesture are not supported).

Within the field of sampled image programming, there are wellunderstood ways to overcome these problems. However, Paint does not implement them.

Class Overview

Paint defines three classes: **clsPaintApp**, **clsPaintWin**, and **clsPixWin**. It makes use of the following classes:

clsApp
clsAppMgr
clsChoice
clsClass
clsFileHandle
clsImgDev
clsMenu
clsScrollWin
clsSysDrwCtx
clsToggleTable
clsWin

Compiling

To compile Paint, just

```
cd \penpoint\sdk\sample\paint
wmake
```

This compiles the application and creates PAINT.EXE and APP.RES in \PENPOINT\APP\PAINT.

Running

After compiling Paint, you can run it by

1. Adding \\boot\penpoint\app\Paint Demo App to \PENPOINT\BOOT\APP.INI
2. Booting PenPoint
3. Creating a new Paint document, and turning to it.

Alternatively, you can boot PenPoint and then install Paint Demo App via the Connections Notebook.

Files Used

The code for Paint is in \PENPOINT\SDK\SAMPLE\PAINT. The files are:

BITMAP.RES resource file for the document icon

CUTIL.C utility routines for reading & writing data and strings

CUTIL.H header file for reading/writing utility routines

METHODS.TBL method table for the paint classes

PAPP.C source code for the paint application class

PIXELMAP.C utility routines for using pixel maps

PIXELMAP.H header file for pixel map utility routines

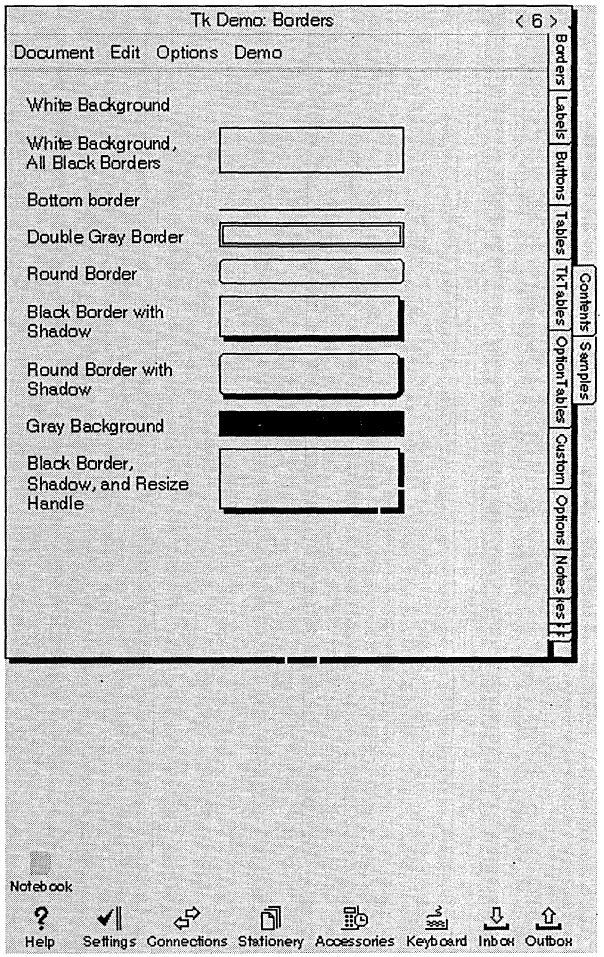
PIXWIN.C source code for the PixWin class

PIXWIN.H header file for the PixWin class

PWIN.C source code for the PaintWin class

PWIN.H header file for the PaintWin class.

Toolkit Demo



Toolkit Demo (also known as tkdemo) shows how to use many of the classes in PenPoint's UI Toolkit. Although it is not exhaustive, it does provide many examples of using the Toolkit's APIs and setting fields to get different functionality and visual effects.

Tkdemo does not show how to use trackers, grab boxes, or progress bars.

Objectives

This sample application shows how to:

- Use most of the classes in the PenPoint UI Toolkit
- Create a table layout

- Create a custom layout
- Provide multiple option cards for an application subclass
- Determine if an option card should be applicable, based on the current selection
- Provide a bitmap for **clsIcon**
- Create a scrollwin as the app's frame client window
- Specify an application version number.

Class Overview

Toolkit Demo defines one class: **clsTkDemo**. It makes use of the following classes:

clsApp
clsAppMgr
clsBitmap
clsBorder
clsButton
clsChoice
clsCustomLayout
clsDateField
clsField
clsFixedField
clsFontListBox
clsIcon
clsIntegerField
clsLabel
clsListBox
clsMenu
clsMenuButton
clsNote
clsOptionTable
clsPopupChoice
clsScrollWin
clsStringListBox
clsTabBar
clsTabButton
clsTableLayout
clsTextField
clsTkTable
clsToggleTable

clsTkDemo creates a scrollwin as its frame client window. This lets the demonstration's windows be larger than the frame. A scrollwin can have many client windows, but it shows only one at a time. So, tkdemo creates several child windows, one for each of the topics it demonstrates. **clsTkDemo** also creates a tab that corresponds to each window. The tab buttons are set up so that their instance data includes the UID of the associated window. When the user taps on the tab, the tab sends **msgTkDemoShowCard** to the application. Tkdemo then switches to that window by sending **msgScrollWinShowClintWin** to the scrollwin.

An interesting point is that, to avoid receiving messages while it is creating windows, tkdemo only sets itself as the client of its tab bar after it has created all the windows.

clsTkDemo's instance data is the tag of the current selection and the UIDs of its option sheets. It files all of this in response to **msgSave**.

Compiling

To compile Toolkit Demo, just

```
cd \penpoint\sdk\sample\tkdemo
wmake
```

This compiles the application and creates TKDEMO.EXE in \PENPOINT\APP\TKDEMO.

Running

After compiling Toolkit Demo, you can run it by

1. Adding \\boot\penpoint\app\Toolkit Demo to \PENPOINT\BOOT\APP.INI
2. Booting PenPoint
3. Creating a new Toolkit Demo document, and turning to it.

Alternatively, you can boot PenPoint and then install Toolkit Demo via the Connections Notebook.

Files Used

The code for Toolkit Demo is in \PENPOINT\SDK\SAMPLE\TKDEMO. The files are:

BORDERS.C code for creating different kinds of borders

BUTTONS.C code for creating different kinds of buttons

CUSTOMS.C code for creating **clsCustomLayout** instances

FIELDS.C code for handwriting fields of **clsField** and its descendants **clsDateField**, **clsFixedField**, **clsIntegerField**, and **clsTextField**

GOLOGO.INC include file containing a handcoded GO logo bitmap

ICON.RES resource file for a smiley face, created with PenPoint's bitmap editor

ICONS.C code for creating different kinds of icons

LABELS.C code for creating **clsLabel** instances with different fonts, rows, columns, etc.

LBOXES.C code for making list boxes (**clsListBox**, **clsStringListBox**, and **clsFontListBox**)

METHODS.TBL the method table for the tkdemo app

NOTES.C creates different kinds of instances of **clsNote**

OPTABLES.C creates a sample option table

OPTIONS.C demonstrates option cards and their protocol, and also creates an instance of **clsPopUpChoice**

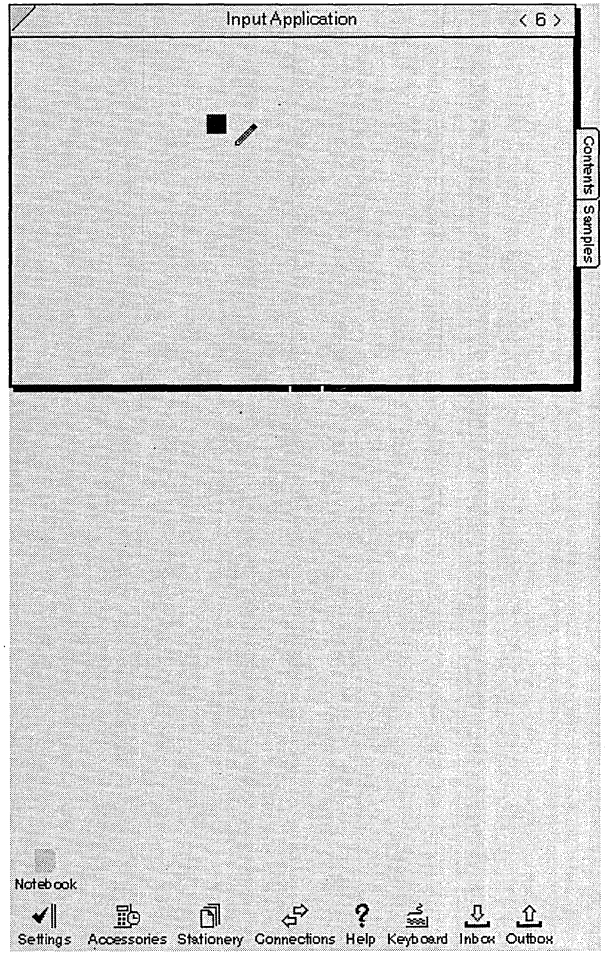
TABLES.C creates various tables (instances of **clsTableLayout**)

TKDEMO.C code for the overall Toolkit Demo application

TKDEMO.H header file for the application class

TKTABLES.C code for creating several subclasses of **clsTkTable**, including **clsMenu**, **clsChoice**, and **clsTabBar**.

Inputapp



Inputapp is a simple application that demonstrates penbased input event handling. As the user drags the pen around, inputapp draws a small square in the window. To provide this functionality, it creates a descendant of **clsWin** (**clsInWin**), which looks for pen input events.

Inputapp's window tracks the pen by drawing a small box at the xy location provided by the event. It also erases the previous box by first setting its DC's raster op to **sysDcRopXOR**. Then it redraws the box at the previous location, thereby erasing it.

Note: If you want your windows to respond to gestures or handwriting, you usually do not look for input events yourself. Instead, you use specialized window classes such as **clsGWin** and **clsIP**, which "hide" lowlevel input event processing

from their descendants. These classes send higherlevel notifications such as **msgGWinGesture** and **msgIPDataAvailable**.

Objectives

This sample application shows how to:

- Create a drawing context in a window
- Set a window's input flags to get pen tip & move events
- Handle pen events (PenUp, PenDown, etc.) in a window
- Use BeginPaint/EndPaint messages
- Turn on message tracing for a class.

Class Overview

Inputapp defines two classes: **clsInputApp** and **clsInWin**. It makes use of the following classes:

clsApp
clsAppMgr
clsClass
clsSysDrwCtx
clsWin

The only function of **clsInputApp** is to create **clsInWin** as its client window. It does this in its **msgAppInit** handler.

clsInWin is a descendant of **clsWin**. Because **clsWin** does not turn on any window input flags, **clsInWin** must set window flags to get certain pen events.

Since **clsInputApp** recreates the input window from scratch and has no other instance data, it does not need to file itself. The input window does not need to save state either. When called upon to restore its state (**msgRestore**), it simply reinitializes.

Compiling

To compile inputapp, just

```
cd \penpoint\sdk\sample\inputapp
wmake
```

This compiles the application, and creates INPUTAPP.EXE in \PENPOINT\APP\INPUTAPP.

Running

After compiling inputapp, you can run it by

1. Adding \\boot\penpoint\app\inputapp to \PENPOINT\BOOT\APP.INI
2. Booting PenPoint
3. Creating a new inputapp document, and turning to it.

Alternatively, you can boot PenPoint and then install inputapp via the Connections Notebook.

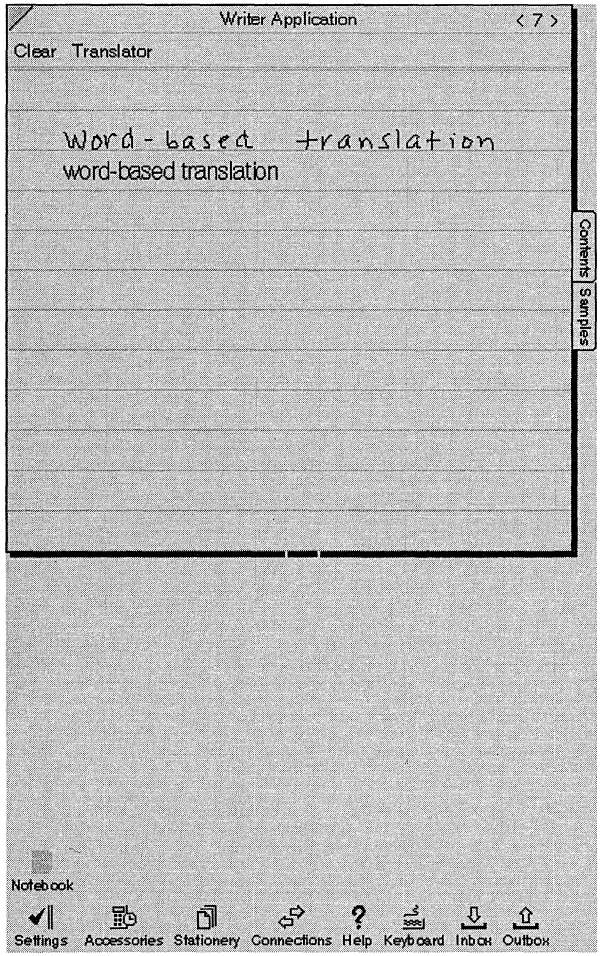
Files Used

The code for inputapp is in \PENPOINT\SDK\SAMPLE\INPUTAPP. The files are:

INPUTAPP.C the source code for the inputapp classes

METHODS.TBL the method tables for the classes.

Writerap



Writerap provides a ruled sheet for the user to write on. When the user lifts the pen out of proximity, writerap translates what the user wrote, and places the translated text on the line below the ink. The user can change the translation algorithm from wordbased, to text or numberbased.

Objectives

This sample application shows how to:

- Use the SPaper and Translation objects
- Make a translator for words, text, or numbers only
- Use the xlist returned by the translation objects and how to interpret its data
- Put a choice control in a menu.

Class Overview

Writerap defines two classes: **clsWriter** and **clsWriterApp**. It makes use of the following classes:

clsApp
clsAppMgr
clsChoice
clsClass
clsMenu
clsSPaper
clsSysDrwCtx
clsXText
clsXWord

Compiling

To compile writerap, just

```
cd \penpoint\sdk\sample\writerap
wmake
```

This compiles the application and creates INPUTAPP.EXE in \PENPOINT\APP\WRITERAP.

Running

After compiling writerap, you can run it by

1. Adding \\boot\penpoint\app\writerap to \PENPOINT\BOOT\APP.INI
2. Booting PenPoint
3. Creating a new writerap document, and turning to it.

Alternatively, you can boot PenPoint and then install writerap via the Connections Notebook.

Files Used

The code for writerap is in \PENPOINT\SDK\SAMPLE\WRITERAP. The files are:

METHODS.TBL the method table for the classes

WRITERAP.C the source code for the classes

WRITERAP.H the header file for the classes.

Basic Service

Basic Service is the absolute minimum code required to be a service. This is the Hello World for service writers. It only requires the service writer to implement one message: **msgNewDefaults**. This approach helps to get someone up and running as a service in the shortest time.

For more complex services, see the TESTSVC and MILSVC samples.

Objectives

This sample service shows how to make a service (the makefile differs from application makefiles).

Class Overview

Basic Service defines one class: **clsBasicService**. It makes use of the following classes:

clsClass

clsService

Compiling

To compile Basic Service, just

```
cd \penpoint\sdk\sample\basicsvc
wmake
```

This compiles the service, and creates BASICSVC.DLL in \PENPOINT\SERVICE\BASICSVC.

Running

After compiling Basic Service, you can run it by

1 Booting PenPoint

2 Tapping on the Connections Notebook to open it, and turning to the Services page

3 Installing Basic Service.

Testing

To test Basic Service, check to make sure that you can install and deinstall it without getting any errors in PENPOINT.LOG.

Files Used

The code for Basic Service is in \PENPOINT\SDK\SAMPLE\BASICSVC. The files are:

BASICSVC.C clsBasicSvc's code and initialization

BASICSVC.H header file for **clsBasicSvc**

DLL.LBC list of exported functions for the Watcom linker

METHOD.TBL method table for **clsBasicSvc**.

Test Service

Test Service is a starter kit for most service writers. It has message handler stubs for the most common service messages.

For other examples of services, see the BASICSVC and MILSVC samples.

Objectives

This sample service shows how to:

- Make a service (the makefile differs from application makefiles)
- Define handlers for messages sent to a class.

Class Overview

Test Service defines two classes: **clsTestService** and **clsTestOpenObject**. It makes use of the following classes:

clsButton

clsClass

clsFileHandle

clsOpenServiceObject

clsOptionTable

clsService

Compiling

To compile Test Service, just

```
cd \penpoint\sdk\sample\testsvc
wmake
```

This compiles the service and creates TESTSVC.DLL in \PENPOINT\SERVICE\TESTSVC.

Running

After compiling Test Service, you can run it by

1. Booting PenPoint
2. Tapping on the Connections Notebook to open it, and turning to the Services page
3. Installing Test Service

Testing

To test Basic Service, check to make sure that you can install and deinstall it without getting any errors in PENPOINT.LOG.

Files Used

The code for Test Service is in \PENPOINT\SDK\SAMPLE\TESTSVC. The files are:

DLL.LBC list of exported functions for the Watcom linker

METHOD.TBL method table for the classes defined in Test Service

OPENOBJ.C clsTestOpenObject's code and initialization

OPENOBJ.H header file for **clsTestOpenObject**

TESTSVC.C clsTestService's code and initialization

TESTSVC.H header file for **clsTestService**.

MIL Service

Test MIL Service is a starter kit for device driver writers.

For other examples of services, see the BASICSVC and TESTSVC samples.

Objectives

This sample service shows how to make a service (the makefile differs from application makefiles).

Class Overview

Test MIL Service defines one class: **clsTestMILService**. It makes use of the following classes:

clsClass

clsMILService

Compiling

To compile Test MIL Service, just

```
cd \penpoint\sdk\sample\milsvc
wmake
```

This compiles the service, and creates MILSVC.DLL in \PENPOINT\SERVICE\MILSVC.

Running

After compiling Test MIL Service, you can run it by

1. Booting PenPoint
2. Tapping on the Connections Notebook to open it, and turning to the Services page
3. Installing Test MIL Service.

Testing

To test Test MIL Service, check to make sure that you can install and deinstall it without getting any errors in PENPOINT.LOG.

Files Used

The code for Test MIL Service is in \PENPOINT\SDK\SAMPLE\MILSVC. The files are:

DLL.LBC list of exported functions for the Watcom linker

METHOD.TBL method table for **clsTestMILService**

MILSVC.C clsTestMILService's code and initialization

MILSVC.H header file for **clsTestMILService.**

PENPOINT APPLICATION WRITING GUIDE
GLOSSARY

Here are some of the terms used throughout the PenPoint Software Developer's Kit. Glossary terms appear **in bold** for emphasis. Terms that are only used within one subsystem may not appear in this Glossary. The glossary entry is followed by the name of the manual part that discusses the topic in more detail. Use the Master Index found at the end of the *PenPoint Development Tools* volume to locate more information.

---

abstract class A class that defines messages or provides useful functionality, however, clients do not create *instances* of an abstract class. *Class Manager*

accessory An accessory document "floats" on the Desktop when active, appearing over pages in the Notebook. Most accessories appear in the Accessories auxiliary notebook. *Application Framework*

acetate plane, acetate layer The window system maintains a global, screen-wide display plane called the **acetate plane**, which is where ink from the pen is normally "dribbled" by the pen-tracking software as the user writes on the screen. *Windows, Input*

activation The transition of a document to an active state, with a running process, an application instance, etc. *Application Framework*

activation What happens when the user actually operates a control, often by lifting the pen up. The user can **preview** a control without activating it. *UI Toolkit*

advisory message Messages that an object's class sends to other objects (possibly including *self*) informing them of changes. Often the class sending the message does not itself do anything in response. *Class Manager*

aliasing Multiple tasks can access the same piece of physical memory by aliasing **selectors**. *System Services*

ancestor Every class has one immediate ancestor. When a class receives a message, the class can elect to pass the message on to its immediate ancestor, and in turn the ancestor may pass on the message to its own ancestor. Hence a class can "pick up," or *inherit* the behavior of its ancestors. *Class Manager*

API (Application Programmer's Interface) The programmatic interface to a software system. The PenPoint API is covered in depth in the *Architecture Reference*; the functions and messages comprising the PenPoint API are listed in the *API Reference* and in the header files in \PENPOINT\SDK\INC from which the *API Reference* was derived. *all*

application Usually, the program code and support files of an end-user program—the software that you create to run under PenPoint. *all*

application class A PenPoint class that contains the code and initialization data used to create running applications. *Application Framework*

application framework See "PenPoint Application Framework." (Application Framework)

application hierarchy Whenever the user creates, moves, or copies a document, PenPoint creates a directory for the document in memory (usually on the RAM volume, under \PENPOINT\SYS\DT). These directories make up the application hierarchy. Every document has a place in the application hierarchy even if it isn't running, so that the state of the system can survive application shutdown and reboots. The Notebook and Section Table of Contents display a view of part of the application hierarchy. *Application Framework*

application home An external volume on which an application is permanently stored. *Installation*

application instance Loosely, a single, activated document, in a running process, complete with its objects and application data. Strictly speaking, the application instance is only the **instance** of the application's class, also known as the **application object**. *Application Framework*

application object The instance of an application that receives PenPoint Application Framework messages when a **document** of that application is **activated**. *Application Framework*

at-least-once delivery Guarantees that datagrams are delivered to their destination at least once. *Remote Interfaces*

atom The Text subsystem provides a mechanism for creating a database of unique strings and accessing those strings through a globally-valid, compact, unique identifier, called an atom. *Text*

attribute Various typographical properties affecting the appearance of glyphs in a font, such as bold, italic, condensed, encoding, and so on. *Windows and Graphics*

attributes Data associated with a **node** in the File System. There is a fixed set of file-system attributes, and clients can define and set additional arbitrary client-defined attributes. *File System*

attributes Text attributes are either default attributes, which apply to an entire object, or local attributes, which apply to contiguous ranges of characters or paragraphs. *Text*

baseline Windows have a baseline in both the x and y dimensions which you can use to achieve more pleasing alingment. *Windows and Graphics*

Bezier curve A curved line formed from two end points and two control points, supported by **SysDC**. *Graphics*

bind You must bind a **DC** to a window (and hence a pixel device) before you can draw using it. *Windows and Graphics*

binding The process that joins a device driver to a port or another device driver; for example, a client binding to a **service** makes the client known to that service. *Remote Interfaces*

bitmap An array of pixels, with an optional **mask** and **hot spot**. *Graphics*

blocking protocol A transfer protocol used to tranfer large amounts of data, in which *clsXfer* may block the sender until the receiver empties the buffer. *Utility Classes*

bounding box The smallest rectangle that fully encloses a region. *UI Toolkit*

bridge A computer or other smart device that can link two networks and route traffic from one zone to another. *Remote Interfaces*

browser A **component** that displays file system contents to the user; used in disk viewers, installers, and **TOC**s. *Utility Classes*

button Buttons are **labels** that the user can **activate**. *UI Toolkit*

cached image A **pixelmap** in memory which is in the optimum form for quickly displaying in a window. *Windows and Graphics*

capability A flag controlling an action on an object (such as sending it *msgNew*, creating a descendant class from it, etc.). If it is not set, the action may require that the sender know the object's **key** (*classes*)

child window A window that is a child, or grandchild, of another in the **window tree**. *Windows*

choice A table that displays several alternatives and allows the user to pick only one. *UI Toolkit*

class A special kind of **object** that implements a particular style of behavior in response to **messages**. Most classes act as factories for objects: you can create *instances* of a class by sending that class the message *msgNew*. In the PenPoint Class Manager, a class has a **method table**, which determines which messages sent to objects of that class that the class responds to. A class's processing of a message often involves passing the message to the class's **ancestor** in order to **inherit** appropriate behavior. *Class Manager*

Class Manager The code that supports the object-oriented, message-passing, class-based programming style used throughout the PenPoint operating system and in all PenPoint applications. The Class Manager itself implements two classes, *clsObject* and *clsClass*. *Class Manager*

client The general term for any software using some feature of PenPoint. *all*

client The object that receives messages from toolkit components notifying it of important user actions and events. *UI Toolkit*

client window Frames and scrollwins manage a window that is supplied by, or of interest to, the **client** of the frame or scrollwin. *UI Toolkit*

clipping The process by which drawing operations in a window are prevented from affecting certain **pixels** on the **windowing device**, for example because those pixels are part of another window. *Windows and Graphics*

clsApp Class for all applications; provides a "head start" framework by responding to the *PenPoint Application Framework*'s protocol of messages. *Application Framework*

column A Table Server table has a fixed number of columns and a variable number of **rows**. *Components*

component A piece of system or application software functionality with a well-defined external interface, packaged as a DLL, which can be used or dynamically replaced by a third-party developer. Some components are unbundled and must be licensed separately from PenPoint. *all*

component layer The **component** layer of PenPoint consists of general-purpose subsystems offering significant functionality that can be shared among applications. *Overview*

connected A file system volume that is accessible from a PenPoint computer is connected: a volume may be known yet disconnected. *File System*

connection An association between two sockets. *Remote Interfaces*

connection-oriented communication In this style of interface, you establish a connection with another **socket**, exchange data, and then break the connection. *Remote Interfaces*

connectionless communication In this style of interface, data is transferred from one **socket** to another without explicitly establishing a connection. *Remote Interfaces*

constraints Specifications for the sizing and positioning of child windows during **layout**. The UI Toolkit includes *clsTableLayout* and *clsCustomLayout*, which implement tabular layout and relative positioning respectively. *Windows and Graphics*

container application An application whose primary purpose is to contain (**embed**) other documents. *Application Framework*

context Information maintained by the driver and slave in a traversal engine episode. *Application Framework*

context Information maintained by the Class Manager to keep track of messages passed up and down the class hierarchy during message processing. *Class Manager*

context The PenPoint Source Debugger maintains a context indicating the current procedure body, which controls the lexical scope of variables. *Development Tools*

control dirty Indicates whether or not the control has been "touched." (*UI Toolkit*)

copy A mode of **ITC** that copies the entire message; see also **transfer**. *System Services*

cork margin An optional thin strip in the default application frame that knows how to embed applications. *Application Framework*

corrupt When a handle's target node is deleted or destroyed, the File System marks the handle **corrupt**. *File System*

current byte position Each **file handle** maintains a position in the file it refers to, the position at which the next byte will be read or written. *File System*

current directory entry Each **directory handle** maintains a reference to the next directory entry it will use when the directory is read one entry at a time. *File System*

current grafic A picture segment maintains an index in its set of grafics which is the grafic relative to which the next operation will take place. *Windows and Graphics*

data object An **object** that maintains, manipulates and can recursively file data. Any descendant of *clsObject* can do this. Often used in Applications together with a **view** that *observes* the data object. *Application Framework*

data resource Contains information saved as a stream of bytes; see also **object resource**. *Resources*

DB The PenPoint Source-level Debugger. *DB*

DC (Drawing Context) An **object** that implements an imaging model; it draws on the device of the window to which it is bound. GO's *SysDC* is the imaging model used by all PenPoint's visual components. *Graphics*

deactivate Deactivating an application removes its code from the user's PenPoint computer, but the Installer still keeps track of the application's **UID** and where its **home** is. *PenPoint Application Framework, Installation*

debug flags A set of flags in PenPoint used to control the behavior of code. You can set each flag to different bit values before and while PenPoint is running, and your code can examine flags programmatically. The lowercase flags are available for outside developers. *Application Writing Guide*

default resource agent A resource agent that treats data resources as a stream of bytes and which treats object resources as a simple object. *Resources*

descendant class A class that **inherits** from another, either directly, or through a chain of **subclasses**. *Class Manager*

device driver Provides an interface between programs and devices or other device drivers. *Remote Interfaces*

dimensions A custom layout specification has four "dimensions": horizontal location, vertical location, width, and height. *UI Toolkit*

directory A **directory** is like a catalog; it contains references to zero or more *nodes*, called **directory entries**. *File System*

directory handle An object that references either a new or existing directory *node* in the File System. *File System*

dirty control See "control dirty." (*UI Toolkit*)

dirty layout A client can mark a window's layout dirty to indicate that it needs to be laid out. *Windows*

dirty window The window system marks regions of a window "dirty" when they need to be repainted. Dirty windows later receive *msgWinRepaint* telling them to repaint their contents. You can mark windows as dirty yourself to make them repaint. *Windows*

document A filed instance of some application. A document has a directory in the *application hierarchy*, but at any given time it may not actually have a running process and a live *application instance*—these usually are destroyed when the user turns the page. Most documents live in the *Notebook*, but running copies of *floating* "applications" such as the Calculator and Installer are also documents. *Application Framework*

dribble The "ink" from the pen when the user writes over windows which support gestures and/or handwriting. *Input*

driver The object requesting the traversal (such as the traversal engine or the search and replace application); see also **slave**. *Application Framework*

DU4 (Device Units 4th quadrant) The coordinate system of pixels on the screen. Usually you perform window operations in *LWC* and specify drawing coordinates in *LUC*. *Windows*

em A typographical term for the height of a square roughly the size of characters in a font; also known as the point size of the font. *Windows and Graphics*

embed The PenPoint Application Framework provides facilities for applications and components to display and operate inside of other applications and components without detailed knowledge about each other. For example, every page in the *Notebook* is actually a *document* embedded in the Notebook's window. As another example, a business graphic document or component could be embedded within a text document. *Application Framework*

embedded document An **embedded document** is a document that is contained within another document. *Application Framework*

embedded window mark *clsEmbeddedWin* provides an **embedded window mark** that indicates the location of an embedded window. *Application Framework*

entries The items in a list box. List boxes are scrolling windows that support very large numbers of items, not all of which need to exist as windows at all times. *UI Toolkit*

event The occurrence of some an activity, such as the user moving the pen or pressing a key. *input*

exactly-once delivery A style of datagram communication that guarantees that the datagrams are delivered to their destination exactly once. *Remote Interfaces*

explicit locator A **locator** that includes both a starting point and a path. *File System*

extract The removal of a window and its children from the tree of windows on some device. It makes the window invisible but does not destroy it. *Windows and Graphics*

fields Labels that you can handwrite in. *UI Toolkit*

file A file is a repository for data. *File System*

file export A mechanism of the Browser that presents the user with a choice of file format translators to export the selection. *Utility Classes*

file handle The object with which you access a file node and its data (the handle is not a file itself). *File System*

file import A mechanism of the browser that presents the user with a list of available applications that can accept the imported file. *Utility Classes*

filing Objects must ordinarily file their state in the File System so that the user is not aware of documents **activating** and terminating on page turns. *Application Framework*

filter A means of restricting the kinds of messages an object or process receives. *Input*

fixed-point number A 32-bit number composed of an integer and fractional component. *Windows and Graphics*

floating A floating window appears above the main Notebook; unlike documents on pages in the Notebook, the user can move and resize a floating window. *all*

flow control Some protocol is necessary to avoid buffer overflow in communication. The PenPoint serial driver utilizes two flow control protocols: XON/XOFF flow control and hardware RTS/CTS flow control. *Remote Interfaces*

font cache After ImagePoint **renders** a font glyph into a bitmap, it keeps the bitmap in a font cache to speed future drawing of the character at that size. *Windows and Graphics*

frame The border surrounding documents and option sheets, which often includes a title bar, resize corner, move box, etc. *UI Toolkit*

function prototype The declaration of a function and its parameters in a PenPoint header file. *all*

gateway A computer that translates network requests from one protocol to another. *Remote Interfaces*

gesture A simple shape or figure that the user draws on the screen with the pen to invoke an action or command. *Also see **scribble**. Handwriting*

global memory Memory accessible from all **tasks** -- you can pass pointers to objects in global memory between tasks. *all*

glyph A symbol or character in a font. *Windows and Graphics*

grab Getting exclusive notification of events in the system, for example when tracking the pen. *Input*

grafic The individual figure drawing operations stored in a **picture segment**. *Windows and Graphics*

graphics state The current scale, rotation, units, foreground and background colors, line thickness, and so on, maintained by a **SysDC** object. *Graphics*

hardware task A **task** that is not scheduled by the operating system but by hardware events. This category includes **interrupt tasks**. *System Services*

heap A pool of memory; individual chunks of memory in it aren't protected. *System Services*

hot mode A state in which the PenPoint Application Framework will not terminate (*shut down*) an application. *Application Framework*

hot spot The pixel at the "origin" of a **bitmap** that is drawn at the location of the bitmap. *Windows and Graphics*

image device A **windowing device** whose image memory is under the control of the client (instead of on a screen or printer). *Windows and Graphics*

in-line In-line fields provide full handwriting and gesture recognition, allowing the user to write with the pen directly into the field itself. *UI Toolkit*

Internationalization The process of generalizing a program so that it is suitable for use in more than one country. *Application Writing Guide*

InBox PenPoint's **InBox** and **OutBox** services allow the user to defer and batch data transfer operations for later execution; they appear as iconic notebooks. *Remote Interfaces*

inheritance A class inherits the behavior of its immediate *ancestor* class. Through inheritance, all classes forms a tree with *clsObject* at the top. *Class Manager*

input registry The set of objects that have expressed interest in receiving input events. *Input and Handwriting Translation*

insertion pad A window that supports character entry. It may contain windows supporting different kinds of character entry such as character boxes, ruled paper, and a virtual keyboard. *Handwriting*

installation Usually refers to the process of installing some item onto a PenPoint computer, especially an application, but also fonts, handwriting prototypes, and services. *Installation*

instance Every *object* is an immediate instance of the *class* that created it. It is also an instance of that class's ancestors. For example, a button is an instance of *clsButton*, but it is also an instance of *clsLabel*, of *clsWin*, and of *clsObject*. *Class Manager*

instance data Data stored in an object; it is normally only accessable by the object's *class*, which uses instance data in responding to messages sent to that object. The class defines the format of the instance data. Classes often choose to have instance data include pointers to instance "information" stored outside the object. *Class Manager*

inter-task message An ITC message that contains miscellaneous information for a task along with an optional, variable-length message buffer. *System Services*

interrupt task A **hardware task** that executes in response to some hardware interrupt, such as the interval timer. *System Services*

ITC (Inter-Task Communications Manager) Handles sending **data** and **semaphores** between tasks. Note that ITC messages are different from Class Manager messages. *System Services*

kernel The portion of the PenPoint operating system that interacts directly with the hardware; the core memory and task management code that is the first code loaded when PenPoint boots. Most **System Services** are implemented in the kernel. *System Services*

key If an object's class doesn't have the **capability** set for an operation, the sender may still be able to perform it by supplying the correct key. *Class Manager*

label A window that displays a string or another window. *UI Toolkit*

layout The process of sizing and positioning a **tree** of windows. Windows and Graphics implements a protocol through which a client can tell windows to lay out and windows can ask each other for their desired sizes. Instead of specifying the exact position and size of all its windows, you need only supply a set of **constraints** on their relative positions. *UI Toolkit*

list An object that holds an ordered collection of items. *Utility Classes*

list box A scrolling window that displays a subset of entries from a potentially very large set. *UI Toolkit*

local memory Per-**process** memory; pointers to objects in local memory can only be passed between **tasks** in the same **task family**. *all*

local volume Volumes on hard or floppy disk drives attached to the PenPoint Computer through its built-in SCSI port. *File System*

localization The process of modifying an application so that it is usable in a specific language or region. *Application Writing Guide*

locator Specifies a **node** in the **File System; it is a directory handle**: **path** pair, in which the path is the path from that directory handle to the node. *File System*

logical device driver A device driver that controls another device driver, rather than an actual device. *Remote Interfaces*

LUC (Local Unit Coordinates) Arbitrary coordinates associated with a **DC**. You can specify different units, scaling, rotation, and transformation for LUC. *Windows & Graphics*

LWC (Local Window Coordinates) The coordinates of a window in pixels, with the origin at the lower-left corner of the window. *Windows & Graphics*

main window The window of an application which the PenPoint Application Framework inserts on-screen in the page location or as a floating window. An application's main window is usually a frame. *PenPoint Application Framework, UI Toolkit*

manager An **installation manager** is an instance of **clsInstallMgr** that manages the installation, activation, deactivation, and "Update from Home" of a set of similar items. *Installation*

manager An object that coordinates the previewing and activation of one or more UI components in order to create, for example, an exclusive choice or pop-up menu. *UI Toolkit*

mask **Bitmaps** have a 1-bit deep mask that controls which **pixels** in the bitmap are drawn. *Windows and Graphics*

memory-mapped You can map a file into memory so that you read and write to it simply by accessing memory. *File System*

memory-resident Volume residing in RAM. *File System*

menu bar A **frame** has an optional menu bar below its title bar. The PenPoint Application Framework defines standard application menu items (**SAMS**) for an application's main window frame. *UI Toolkit*

menu button A button that displays a pop-up menu when the user taps on it. *UI Toolkit*

message A 32-bit value you send to an object requesting it to perform some action. Messages are constants representing some action that an object can perform. The type MESSAGE is a **tag** which identifies the class defining the message and guarantees uniqueness. When you send a message to an object, if that message is mentioned in the class's **method table**, then the Class Manager calls a **message handler** routine in the class's code which responds to the message. *Class Manager*

message argument A U32 parameter to sending a message which lets the sender specify additional information. *Class Manager*

message argument(s) The information needed by a class to respond to a message. Often the message argument parameter is a pointer to a separate message arguments structure: this is the only way a class can pass back information to the sender. *Class Manager*

message handler A function in a class's code that implements appropriate behavior for some message or messages; called by the Class Manager in response to message associated with it in the class's **method table**. *Class Manager*

message tracing Facility that prints out all messages received by a particular **instance** or **class**. *Class Manager*

method Synonym for **message handler**. *Class Manager*

method table An array of message-function name pairs (plus some flags) that determines which **message handler** function (if any) will handle messages sent to objects of that class. *Class Manager*

Method Table Compiler DOS program that compiles a file of method tables into an object file that you link with your classes' code. *Class Manager*

metrics Information made public about instances of a class is often called **metrics**, and many classes provide a pair of messages to set and get metrics. *all*

name Most names in PenPoint, such as the names of **nodes** in the File System, are strings containing 1 to 31 characters. *all*

node A location in the File System; can be a directory or file. The PenPoint file system is organized as a tree of nodes. *File System*

note A window that presents transient information to the user. *UI Toolkit*

Notebook The main notebook on-screen, usually the user's personal notebook. *all*

notebook metaphor The visual paradigm in PenPoint of a physical notebook containing pages documents and sections, with tabs, a page turn effect, and so on. *PenPoint Application Framework, UI Toolkit*

notification The act of sending a message to some client telling it about a change, such as a tap on a button, or a change to an **observed** object. *Class Manager*

object An entity that maintains private data and can receive **messages**. Each object is an **instance** of some class, created by sending **msgNew** to the class. *Class Manager*

object resource Contains information required for creating or restoring a PenPoint object; see also **data resource**. *Resources*

observer An object that has asked the Class Manager to notify it when changes occur to another object. Objects maintain a list of their observers. *Class Manager*

open A document currently displayed on-screen. *Application Framework*

option sheet A floating **frame** that displays attributes of the **selection** in one or more "card" windows. *UI Toolkit*

outline ImagePoint stores fonts as size and resolution-independent **outlines**. It must convert the outline of a glyph into a bitmap before drawing the glyph. *Windows and Graphics*

owner An object is **owned** by the task that creates it. *Class Manager*

owner The process that creates a subtask **owns** that subtask and any sibling subtasks created by it. *System Services*

parent window Every window in the **window tree** but the **root window** has a parent window. Conversely, when you extract a window from the window tree, it no longer has a parent and so it and all its child windows are no longer visible on-screen. *Windows*

pass back Nearly all classes return a STATUS value in response to a messages. In addition, a class may pass back information to the sender in the message's **message arguments**. *Class Manager*

path The sequence of **node** names between a **directory handle** and a particular node. The path to a **node** in the File System is the sequence of directory names and the node's own name that the File System must traverse to get to that node from a particular **directory handle**; the directory handle:path pair is called a **locator**. *File System*

PenPoint GO's operating system that supports pen-based computing in a document-oriented notebook-like interface. *all*

PenPoint Application Framework Both the protocol supporting multiple, embeddable, concurrent applications in the **Notebook**, and the support code that implements most of your application's default response to the protocol for you. The protocol and code provide a "head start" for building applications in the pen-based, document-oriented Notebook environment. *Application Framework*

picture segment An object in which you can store and replay sequences of drawing operations. *Windows and Graphics*

pixel A picture element with a value. *Windows and Graphics*

pixel device An **abstract class** that defines the minimal behavior for a pixelmap graphics device. A pixel device corresponds either to actual hardware capable of displaying graphics, or memory that simply stores an image (an **image device**). *Windows and Graphics*

pixelmap A rectangular array of pixels. *Windows and Graphics*

point A unit of measure, approximately equal to 1/72 of an inch. *Windows and Graphics*

pop-up A window (usually a menu or field) that temporarily appears on top of all other windows. *UI Toolkit*

previewing The feedback provided by a control while the user is "fiddling" with the control, before the user **activates** (if at all) the control. *UI Toolkit*

privilege A level of trust that PenPoint has in code. Some data and functions are **protected** in that they can only be accessed by code at a high privilege level. A task always executes at some privilege level. *System Services*

process An operating system context with its own **local memory**. *System Services*

process ID String that identifies a process in DB. *DB*

protected Usually refers to memory locations which can't be accessed, either because they require system privilege or are "hidden" in another process's **local memory**. *286*

prototype A shape template with which sets of **strokes** are compared in handwriting **recognition**. *handwriting*

proximity A state reported by the pen hardware on some PenPoint computers when the user has the pen **near** the screen. It's independent of the pen tip being "down." Using a mouse, you simulate this by pressing the MIDDLE mouse button to go out of proximity. *input*

PWC (Parent Window Coordinates) The **LWC** (Local Window Coordinates of a window's parent). *Windows and Graphics*

raster op Determines how the values of a source and destination **pixel** combine to affect the value of the destination pixel in graphic operations. *Graphics*

recognition Matching a set of user **strokes** with the most likely **prototype**(s) during handwriting translation. *handwriting*

region Area of a window that is visible or requires repainting (**dirty** region). *Windows & Graphics*

remote server A program or device on another computer that can communicate over the network. *Remote Interfaces*

remote volume Volumes available over a network or other communication channel. *File System*

render The generation of a bitmap of a font **glyph** from an **outline**. ImagePoint can synthesize certain **attributes** of the font, for example rendering bold characters from a regular outline. *Windows and Graphics*

repaint The **pixels** of a window need to be repainted in various circumstances: when the window first appears on screen, when the window is covered by another window and then exposed, when the window changes size, and so on. When a window needs repainting, the window system marks it **dirty**. When you repaint a window, the pixels affected are the visible portions of the dirty **region**. *Windows and Graphics*

resource A uniquely identified collection of data. Resources allow applications to separate data from code in a clean, structured way. *Resources*

resource definition file A C-like file which when run through the PenPoint Resource Compiler creates or appends to a **resource file**. *Resources*

resource file A file containing typed **resources**, each identified by UUID. Every application and document has a **well-known** resource file. Clients can ask a resource file to read and write resources, and to search for a particular resource. *Resources*

resource list A list of **resource files**. Each document has a resource list composed of its own resource file, its application's resource file, and the system resource file. Clients can ask a resource list to search for a particular resource. *Resources*

RGB (Red, Green, Blue) A means of specifying colors by the amount of these primary colors. *Windows and Graphics*

Ring An area of memory protection on the Intel 80386. Most application code runs in Ring 3 memory; crucial parts of the PenPoint kernel and device drivers run in Ring 0 memory.

root directory Top-most node of the file system hierarchy on a volume. *File System*

root window Top of the **window tree** on a **windowing device**. *Windows and Graphics*

router A task that decides which window object to send input events to. *input*

row A Table Server table has a fixed number of **columns** and a variable number of rows. *Components*

sampled image An image made up of **pixels**. *Windows and Graphics*

SAMs (Standard Application Menus) The PenPoint Application Framework supplies a set of SAMS (the Document and Edit menus), to which applications can add their own menu items. *PenPoint Application Framework, UI Toolkit*

scope The component, documents, set of documents, or partial area of the same upon which a **traversal** acts. *PenPoint Application Framework, Utility Classes*

scribble A collection of **strokes** which **translators** can translate into either text characters or command **gestures**. *input*

SDK (Software Developerment Kit) A development package to assist developers in writing applications for a system. The PenPoint SDK provides the code required to develop applications, and documentation and tools to assist development. *all*

selection PenPoint maintains a system-wide selection, which is the target for all editing operations. The Notebook UI lets the user select applications and icons; applications and **components** may allow the user to select words, shapes, and other entities within their windows. *Utility Classes*

self The **object** that originally receives a message. Code that processes a **message** is passed the UID of self. *Class Manager*

semaphore A lock that lets cooperating tasks synchronize access to resources, or guarantee exclusive access to one process or subtask at a time. *System Services*

service A general, non-application DLL that enables PenPoint clients to communicate with a device or to access a function, such as a database engine. *Remote Interfaces*

service section A section in the Inbox or Outbox; each is associated with a specific service and represents a queue to or from that service. *Remote Interfaces*

shut down The **PenPoint Application Framework** shuts down a **document** to conserve memory by destroying its application **object** and terminating its process. Thereafter the document only exists as a directory and files in the **application hierarchy**. *Application Framework*

socket To access the network, you must create a **socket**, through which you send and receive transport messages. *Remote Interfaces*

software task A **task** that is scheduled for execution by software. This category includes **processes** and **subtasks**. *System Services*

source-level debugger A debugger (such as PenPoint's Source Debugger) that lets you use line numbers, variables, and structures from the original source code instead of from the assembly language in executable code. *DB*

SPaper (sketchpad) A view window that stores the user's handwriting in a **scribble**. *Handwriting*

standard message A procedural interface to put up standardized notifications, warnings, and requests to the user in the form of **notes**. The text of standard messages is stored in resources. *UI Toolkit*

stateful Most objects are stateful, that is they change behavior over time. This means that they have state which they need to maintain, and often file as well. *all*

stationery Application-specific template documents. *Application Framework*

status value Most functions and messages in PenPoint return a value of type **STATUS**, indicating success, an error of some sort, or some other status. Status values are constant **tags** in order to indicate the class (or pseudo-class) returning the status, and to guarantee uniqueness. *all*

stream A stream operation is one in which files or data items are treated as a series of individual bytes. *Utility Classes*

stroke **SysDC** strokes a line when drawing lines and the borders of figures using the current line pattern, width, end style, corner style, etc. of the **graphics state**. *Graphics*

stroke Data structure that stores the path traced by the pen when the user holds it against the screen and "writes" with it. Note that the pen hardware supplies stroke coordinates at much higher resolution than that of the "ink" **dribbled** by the pen on-screen. *input*

style fields You can control the appearance and behavior of many UI Toolkit components by specifying different parameters in the style fields of its class and ancestor classes. *UI Toolkit*

subclass To create a new class that **inherits** from an existing class. You subclass a class in order to take pick up its behavior, while modifying or extending its behavior to do what you want. *Class Manager*

subtask A task that shares the address space (**local memory** as well as **global memory**) of its parent **process**. *System Services*

SysDC (System Drawing Context) PenPoint's standard **DC** which implements the imaging model used by all of PenPoint's visual components. It supports polylines, splines, arcs, outline fonts, arbitrary units, scaling, transformation, and many other features. It unifies text with other graphics primitives in a single, PostScript-like imaging model. *Graphics*

system layer The **system** layer of PenPoint provides windowing, graphics, and user interface support in addition to common operating system services such as filing and networking. *Overview*

system privilege A high privilege level associated with executing code; particular data may only be accessible by **tasks** running at this level. Only PenPoint code executes at this level. *System Services*

tag A unique 32-bit number that uses the administrated value of a **well-known** UID to ensure uniqueness. *Class Manager*

tag An arbitrary 32-bit number that you can associate with any window. You can check a window's tag and search for a particular tag in the **window tree**, which makes tags useful for identifying components in shared option sheets and menus. *Windows and Graphics*

tags The PenPoint **SDK** includes the PenPoint Online Reference Tags facility, which lets you quickly jump to message and structure definitions in the PenPoint header files. *Developer Tools*

tap A pen down **event** followed by a pen up, with no significant motion in between. *user interface*

task Generic term for a thread of control executing code in PenPoint; includes **software tasks** and **hardware tasks**. *System Services*

task family A **process** and all its **subtasks**. *System Services*

task ID Hexadecimal identifier of a task in DB. *DB*

testing UID A **well-known** UID, predefined by GO, that you can use while testing a prototype application. These UIDs have the names **wknGDTa** through **wknGDTg**.

TOC (Table of Contents) The **browser** page at the beginning of a notebook or section that shows its contents. *Application Framework*

toolkit table Workhorse class in the UI Toolkit for a tabular collection of other components; its descendants include choices, option tables, menus, tab bars, and command bars. You can define toolkit tables statically, so they form a simple user interface specification language. *UI Toolkit*

transfer A mode of **ITC** which queues an alias for the message; see also **copy**. *System Services*

transfer type A data transfer type identifies a specific data format (such as a string or a structure) and transfer protocol. *Utility Classes*

translator An object that when hooked up to a handwriting window receives captured **scribbles** and translates them into ASCII characters or gestures. *Input and Handwriting Translation*

UI (User Interface) *all*

UI component Any window implemented by one of the UI Toolkit's many classes. *UI Toolkit*

UI Toolkit PenPoint's **User Interface Toolkit** provides many different kinds of window subclasses to support a wide variety of on-screen controls, such as labels, buttons, menus, frames, option sheets, etc. *UI Toolkit*

UID **Status values**, messages, **tags**, and input device codes borrow the format of well-known Class Manager UID's to associate themselves with the class that defines them, and to guarantee uniqueness. *all*

UID (Unique Identifier) A 32-bit number that is the handle on an **object**. When you send a message to an object, you send it to the object's UID. *Class Manager*

update region The portion of a window whose pixels are affected by drawing operations. Usually it's either the visible pixels in the window, or the visible pixels in the window that are **dirty**. *Windows*

user The human being interacting with the software. *all*

UUID Universal Unique Identifier. A 64-bit number that is guaranteed to be unique across all PenPoint computers, usually used to identify **resources** in **resource files**. *Resources*

view A window that presents a user interface and **observes** a **data object**; when the data change, the data object notifies its observers and the view updates its display of the object. *Application Framework*

view-data model An approach to designing applications and components that divides the presentation and storage of state into separate **view** and **data** objects. *Application Framework*

volume A physical medium or a network entity that supports a file system. *File System*

well-known An **object** whose **UID** is statically defined for all PenPoint computers. Access may still not be possible if the object is not correctly installed on a particular PenPoint Computer. Most PenPoint classes and globally-accessible objects (such as **theScreen**, **theWorkingDir**, etc.) have well-known UID's. *Class Manager*

well-known Well-known resource IDs identify data and object **resources** that can be used by any client. *Resources*

window tree The hierarchy of windows formed by a window, its child windows, their child windows, and so on. The on-screen window tree starts with a **root window** on a **windowing device**. *Windows*

windowing device A **pixel device** that supports multiple overlapping windows. All windows are associated with some windowing device, even if the window is not currently inserted in the **window tree** on that device. *Windows and Graphics*

working directory A **directory handle** pointing to the directory in the **application hierarchy** for a **document**. This is where the application should file itself. *Application Framework*

wrapper application A special type of application that handles output for a specific **service**. *Remote Interfaces*

you In the SDK "you" generally refers to the reader, who we assume to be a programmer who wishes to develop software for PenPoint. Hello there! *all*

zones A network is divided into two or more zones when it is joined by a bridge. *Remote Interfaces*

PENPOINT CONTRIBUTORS

Todd Agulnick
Bonnie Albin
Roland Alden
Arvind Arakeri
Kenji Armstrong
Josh Axelrod
Ayse Aysoy
Jennifer Bailey
Gary Barg
Steve Bartlett
Trudy Bartlett
Simon Bate
Sandy Benett
John Bennett
Keith Bentley
Debbie Biondolillo
Sharin Blair
Joe Bosurgi
Murray Bowles
Bill Bradley
Alex Brown
Howard Brunnings
Kurt Buecheler
Bill Campbell
Lynn Carpenter
Robert Carr
Karen Catlin
Chia-Lun Chang
Atri Chatterjee
Shirley Chu
Mark Cogdill
Kevin Doren
Gary Downing
Andy Felong
Jim Floyd
Robb Foster
Bob Fraik
Dan Fraisl
John Garrett
Alex Gerson
Regis Gratacap
Steve Grey
Ken Guzik
Philip Haine
Julie Hawthorne
Rob Hayes
June Hirai
Deborah Hodge
Tony Hoeber
Thom Hogan
Mike Homer
Greg Hullender
Steve Isaac
Jerry Kaplan
Ram Kedlaya
Susan Keohan
Nora Kim
Kevin Kitagawa
Sheridan Klenk
Randy Komisar
Omid Kordestani
Merri-Lynn Kubota
Roy Kumar
Ilona Leale
James Lee
Mark Lentczner
Charlie Leu
Debra Levesque
Dan'l Lewin
Debbie Lovell
David Low
Pete Ma
Arjen Maarleveld
Paul Marcos
Cary Masatsugu
Holli Maxwell
Janet McCandless
Devin McKinney
Carl Meyer
Norm Meyrowitz
George Mills
Miki Murdoch
Ravi Narayanan
Mark Nudelman
Lisa Oatman
S Page
Grace Pariante
Elizabeth Parker
Craig Payne
David Polnaszek
James Roseborough
Mark Sapsford
Michael Sattler
Steve Schoettler
Don Schuerholz
Bruce Schwartz
Mike Schwartz
David Serna
Tetsuo Seto
Danny Shader
Pam Shear
Sandra Shmunis
Siew Sim
Eddie Soloko
Peter Stahl
Peggy Stok
Craig Taylor
Sue Toeniskoetter
Mike Tyler
Catherine Valentine
Bob Vallone
Doug Van Duyne
Joe Vierra
Scott Walker
Tony Wang
Darrah Westrup
Tim Wiegman
Rick Wolfe
Phil Ydens
Darren Yee
Eddie Yee
Satya Yenigalla
Peter Yorgin
Lang Zerner
John Zussman

And Thanks To
Henry Allen
Rolando America
David Baird
Celeste Baranski
Fred Benz
Timothy W. Bishop
Judy Bligh
Haydon Boone
Adrian Britt
Harrilyn Callon
David Chavez
Ratha Chea
John Croll
Art Crumpler
Richard Dickson
Lisa Forman
Harmon Gee
Christopher Glazek
Michael Gonzales
Joanne Gozawa
Clifford Gross
Paul Hammel
Brian Hurley
Josh Jacobs
Madeleine Kahn
Lili Kanda
Donna Kelley
Henry Korman
Liz Landreth
Kristina Lincicome
Mike McGeoy
Henry Madden
Marcia Mason
Scott Merkle
Robert Moncrieff
Jerilee Morse
Linda Norris
Alex Osborne
Michael Ouye
Doug Pasos
Jonathan Price
Paul Quin
Steve Rath
Pat Rogondino
John Russell
Alan Saichek
Pam Shandrick
Robbie Shepard
Greg Stikeleather
Andres Tong
Ben Wiseman

READER'S COMMENTS

Your comments on our software documentation are important to us. Is this manual useful to you? Does it meet your needs? If not, how can we make it better? Is there something we're doing right and you want to see more of?

Make a copy of this form and let us know how you feel. You can also send us marked up pages. Along with your comments, please specify the name of the book and the page numbers of any specific comments.

Please indicate your previous programming experience:

☐ MS-DOS ☐ Mainframe ☐ Minicomputer

☐ Macintosh ☐ None ☐ Other _____

Please rate your answers to the following questions on a scale of 1 to 5:

	1 Poor	2	3 Average	4	5 Excellent
How useful was this book?	☐	☐	☐	☐	☐
Was information easy to find?	☐	☐	☐	☐	☐
Was the organization clear?	☐	☐	☐	☐	☐
Was the book technically accurate?	☐	☐	☐	☐	☐
Were topics covered in enough detail?	☐	☐	☐	☐	☐

Additional comments:

_____

_____

_____

_____

_____

_____

_____

_____

_____

Your name and address:

Name _____

Company _____

Address _____

City _____ State _____ Zip _____

Mail this form to:

Team Manager, Developer Documentation
GO Corporation
919 E. Hillsdale Blvd., Suite 400
Foster City, CA 94404–2128

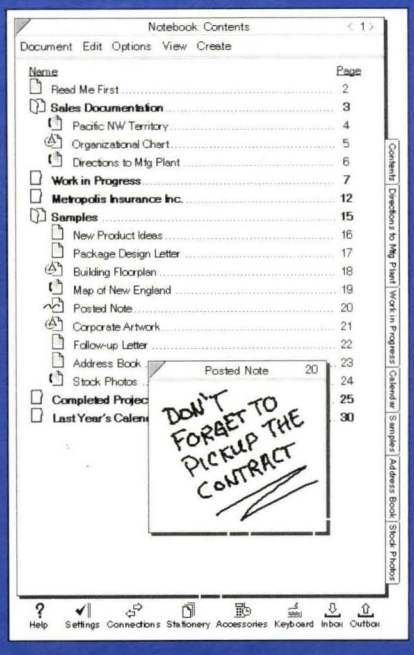
Or fax it to: (415) 345-9833

Operating Systems

> $24.95 USA
> $31.95 CANADA

PenPoint™ Application Writing Guide

The *PenPoint Application Writing Guide* teaches you how to write an application for the PenPoint operating system. It begins by introducing the PenPoint operating system and the Application Framework, which is the framework for all PenPoint applications. After discussing how to design an application, the book describes the environment in which you develop applications, and GO's coding conventions. The remainder of the book is devoted to tutorials and sample programs that show you how to build a variety of sample applications—beginning with an extremely simple application and concluding with a comprehensive one that implements most of the features required by a complete PenPoint application.



The PenPoint operating system is a compact, 32-bit, fully object-oriented, multitasking operating system designed expressly for mobile, pen computers. GO Corporation designed the PenPoint operating system as a productivity tool for people who may never have used computers before, including salespeople, service technicians, managers and executives, field engineers, insurance agents and adjustors, and government inspectors.

Other volumes in the GO Technical Library are:

PenPoint User Interface Design Reference describes the elements of the PenPoint Notebook User Interface, sets standards for using those elements, and describes how PenPoint uses the elements.
PenPoint Development Tools describes the environment for developing, debugging, and testing PenPoint applications.
PenPoint Architectural Reference, Volume I presents the concepts of the fundamental PenPoint classes.
PenPoint Architectural Reference, Volume II presents the concepts of the supplemental PenPoint classes.
PenPoint API Reference, Volume I provides a complete reference to the fundamental PenPoint classes, messages, and data structures.
PenPoint API Reference, Volume II provides a complete reference to the supplemental PenPoint classes, messages, and data structures.

GO Corporation was founded in September 1987 and is a leader in pen computing technology for mobile professionals. The company's mission is to expand the accessibility and utility of computers by establishing its pen-based operating system as a standard.

GO Corporation

919 East Hillsdale Blvd.
Suite 400
Foster City, CA 94404

Addison-Wesley Publishing Company

52495
9 780201 608571
ISBN 0-201-60857-X
60857