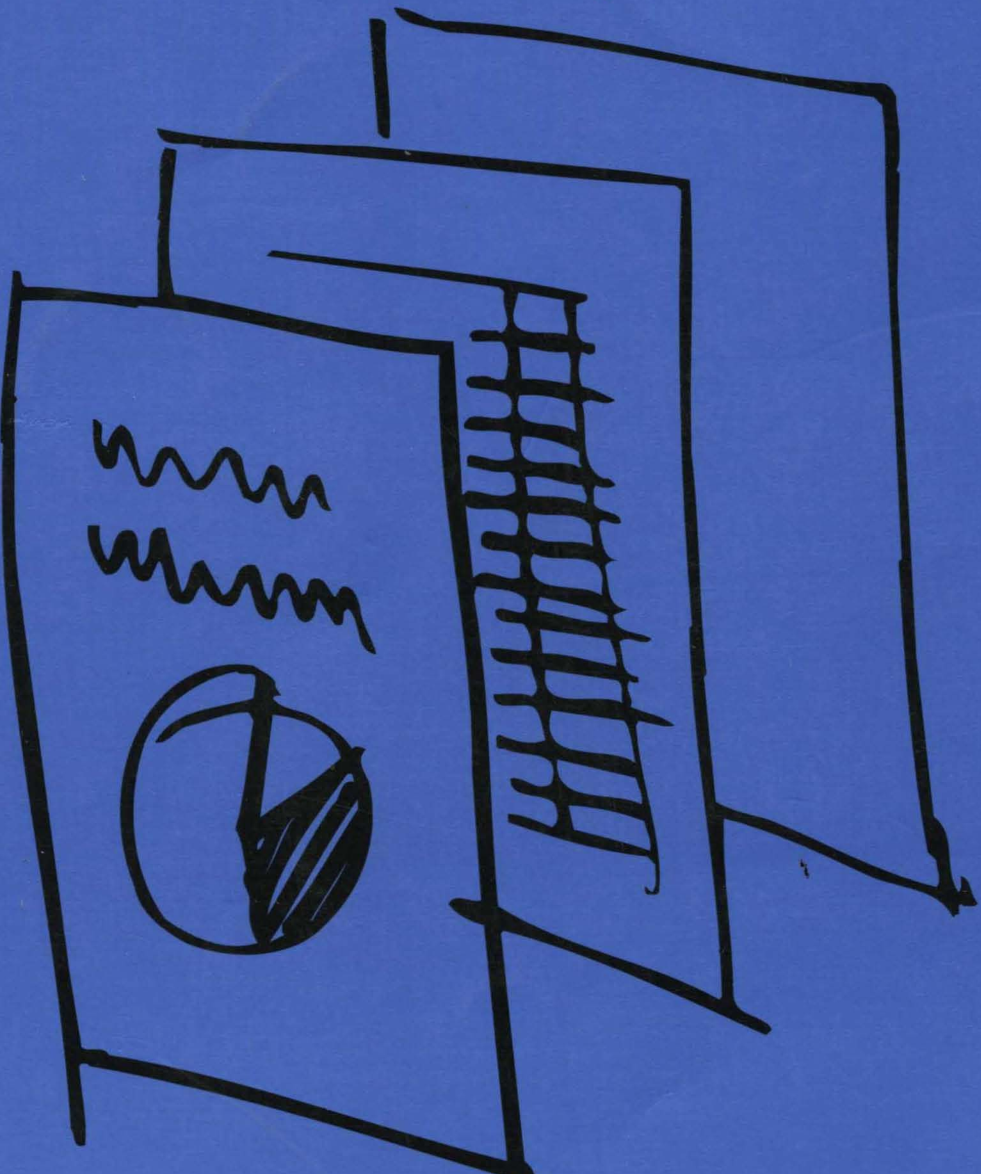
GO Corporation

GO Technical Library

PenPoint® Application Writing Guide Expanded Second Edition



PenPoint®

Updated to include Versions 1.0 and 2.0

PenPoint® Application Writing Guide

EXPANDED SECOND EDITION

PenPoint®

PenPoint Application Writing Guide

EXPANDED SECOND EDITION

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

QA
76.76
O63
P49
1993

ISBN 0-201-62299-8

123456789—AL—9796959493

First Printing, February 1993

Preface

The *PenPoint Application Writing Guide: Expanded Second Edition* is an updated version of the *PenPoint Application Writing Guide* printed in March 1992. This expanded edition includes new information that reflects GO Corporation's enhancements to the PenPoint™ operating system for PenPoint 2.0 Japanese and the PenPoint Software Development Kit (SDK) 2.0 Japanese.

This book is an up-to-date introduction to the PenPoint SDK 2.0 Japanese operating system and contains errata and additional information for earlier PenPoint SDKs. Unless stated otherwise, discussions of the PenPoint operating system and the PenPoint SDK in this book are valid for PenPoint 1.0, PenPoint 1.01, and PenPoint 2.0 Japanese.

PenPoint 2.0 Japanese is a Japanese product with Japanese system resources (on-line help, menu choices, and so on). Developers who do not read Japanese can run the PenPoint SDK 2.0 Japanese with U.S. English resources, but there are some parts of the system that show Japanese text even with the U.S. English feature enabled.

Intended audience

This book 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 utilities.

Document structure

This book contains several parts:

- *Part 1: PenPoint Application Writing Guide* is an introduction to the PenPoint operating system and the PenPoint SDK. It introduces you to basic PenPoint programming concepts, then illustrates those concepts by examining some of the sample applications included with the PenPoint SDK.
- *Part 2: PenPoint Internationalization Handbook* describes the features PenPoint provides that allow you to write an application that is easy to port from one national locale to another.
- *Part 3: PenPoint Japanese Localization Handbook* shows how to use features of the PenPoint SDK 2.0 Japanese that support Japanese-language application development.
- *Part 4: PenPoint Development Tools Supplement* provides new information about the development tools you use with the PenPoint SDK. This part is an update to *PenPoint Development Tools* originally published for the PenPoint SDK 1.0.

- *Part 5: PenPoint Architectural Reference Supplement* provides new information about the architecture of the PenPoint operating system and the classes that it provides. This part is an update to *PenPoint Architectural Reference* originally published for the PenPoint SDK 1.0.
- *Part 6: PenPoint User Interface Design Reference Supplement* provides new information about user interface design and the user interface classes that PenPoint provides. This part is an update to *PenPoint User Interface Design Reference* originally published for the PenPoint SDK 1.0.
- *Part 7: Sample Code* provides descriptions and listings of the sample applications used as examples in this book, and descriptions of the other sample code included with the PenPoint SDK.

Each of these parts was at one time intended to be a separate document, but they have been bound together into a single volume for your convenience. Be aware that you may still find some cross-references that refer to a part of this volume as though it were still a separate book.

Other sources of information

Several parts of this book supplement existing books published for the PenPoint SDK 1.0. These books include *PenPoint Development Tools, PenPoint Architectural Reference,* and *PenPoint User Interface Design Reference.*

For information on the classes, messages, macros, functions, and structures that the PenPoint SDK header files define, see the header files themselves. Many of the header files have changed since the *PenPoint API Reference* was published for the PenPoint SDK 1.0.

CONTENTS

List of Tables

List of Figures

Part 1 / PenPoint Application Writing Guide

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 many other environments.

We sometimes use the names "PenPoint 2.0" and "PenPoint SDK 2.0" in this document. Because this release of PenPoint has been localized only to Japan, these terms refer to PenPoint 2.0 Japanese and PenPoint 2.0 SDK Japanese.

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.

PenPoint also 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

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 utilities.

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.

Other sources of information

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

For information on running PenPoint on a PC, see the *Running PenPoint on a PC* document that comes with the PenPoint SDK.

To learn how to use the PenPoint development tools and utilities, such as the PenPoint source-level debugger, see *PenPoint Development Tools.*

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

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.
- 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

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.
- The use of a notebook metaphor that is natural and easy to use.

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

The pen

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

Gestures are simple shapes or figures that the user draws directly on the screen to invoke an action or command. For example, a cross out X 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	X
Flick left	—	Flick right	—
Flick up	|	Flick down	|
Insert space	∟	Pigtail	?
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

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 mouse 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 one is available as an optional user preference.

Notebook metaphor

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

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:

- They are compact, providing a body of shared code that need not be duplicated by all applications.
- They are consistent, since all applications share the same implementation of common system and user interface functions.
- They are 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

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

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 called the machine interface layer (MIL) 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. These parts are defined below.

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.

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

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

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.

Resource manager

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

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. With the appropriate TOPS software, users can configure their systems to connect directly to desktop 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, edit an existing document, and export it to 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 returning 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

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

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. The ImagePoint API 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

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 Epson-compatible dot-matrix printers and HP Laserjet-compatible laser 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

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

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

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.

Also called the PenPoint UI Toolkit.

Input and handwriting translation

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	Out of proximity
Tip down	Tip up
Move down	Move up
Window enter	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 recognizes hand-printed characters and has the following characteristics:

- Operates in real time (shape matcher operates at 60 characters per second on 33 Mhz 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.
- Uses context to distinguish nonunique character forms such as the letter "O" and the numeral "0".
- Tolerates inconsistency by same user (that is, the user may shape the same character in different ways at different times).
- Accepts 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 accepts 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

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 Rich Text Format (RTF), structured graphics, and Tagged Image File Format (TIFF); 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 Dynamic Data Exchange (DDE) linking are still possible in PenPoint, such live application embedding obviates the need for most of them.

Component layer

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 multifont, 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 paying any license fee to 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

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.

Application layer

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. In addition:

- 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, In box, and Out box 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 online 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

With the exception of some hardware-dependent code, PenPoint and the applications it supports are written 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 OS/2 and Macintosh System 7, will find the development environment familiar.

Software development kit

The PenPoint SDK provides developers with the documentation and tools to develop applications. The kit includes a source-level 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 pen 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

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

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 solid-state, pocket pen 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. Because of techniques such as hardware memory protection, object-oriented programming, rigorous modularization, and extensive sharing of code, PenPoint is a highly reliable operating system.

PenPoint design guidelines

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

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 that meets key needs and requires very little memory.

Think small

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

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 offers several components itself, including Text View™ and the TableServer™ .

Your application must recover

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

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

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, with a separate back-end or a core engine, you'll need to have the big 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. A straightforward application may not need to concern itself with any of the kernel features. It just interacts with PenPoint subsystems, which make careful use of the kernel. For example, none of the sample programs use any advanced kernel features.

Use document orientation

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 start 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

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

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) 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

The Notebook User Interface (NUI) differs from other graphical user interfaces. If you are porting a DOS or Macintosh-based program to PenPoint, rather than create new user interface classes, try to create a user interface that takes advantage of the PenPoint UI Toolkit.

The *PenPoint User Interface Design Reference* describes the PenPoint user interface, its rationale, and how and when to use its components. You should not deviate from the PenPoint interface. Remember that a consistent user interface allows users to learn your application quickly; an 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, PenPoint 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 may have an notable effect on your approach to application design.

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

- At the architectural level, where you design your application.
- At the implementation level, where you write and test 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. Chapter 4, PenPoint Class Manager 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 both the architecture and implementation of simple PenPoint programs and applications. 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

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 found in PenPoint application environments:

- Stylus-based user interaction.
- Object-oriented programming.
- Disk storage unnecessary.
- Multitasking.
- Cooperating, simultaneously active, embeddable applications.
- Graphics-intensive user interface.
- Notebook metaphor.
- Document orientation instead of application and file orientation.

Dealing with these aspects of PenPoint requires you to observe a number of guidelines, described in the following sections. The benefit is that the software architecture of PenPoint eliminates much of the work 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 that 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 that comprise the PenPoint Application Framework.

How applications work

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 does little 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 house-keeping functions that Macintosh or MS-DOS programs must perform from boiler-plate code. Meanwhile, the PenPoint Application Framework presents the PenPoint user with multiple small, concurrent documents as part of a consistent, rich note-book 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:

PenPoint Application Framework Both the protocol for supporting multi-ple, 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 envi-ronment, this section walks through some important stages in the life of an applica-tion. By the end of this section 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

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 oper-ating system (MS-DOS). Then we'll compare these operations with installing and running an application on PenPoint.

MS-DOS installation

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

Some MS-DOS programs require the user to copy the files from distribution disks to the hard disk; others provide their own installation programs that 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

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 under two conditions: 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

During installation, the Installer calls a standard entry point (called **main()**) in your program 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 Create 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 the system until the user removes the application.

Running a PenPoint application

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

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 the following 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

When the user activates the document, PenPoint finds out from the document what application it requires and creates a process that "runs" the application (see "Application classes and instances" on the following page for more details). When the user deactivates the document, PenPoint saves all of the document's information and then destroys the application process.

A PenPoint 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

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 **main**()

entry point in your code 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 when the user chooses accelerated access speed (sometimes called hot mode) from the Access document option sheet, 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

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 fact for now. We will spend pages and pages in this and other manuals explaining how this works.

PenPoint drives your application

Because of all the 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. In additon, your application can't 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

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, for windows, and for data. They provide a skeleton of correct behavior, although obviously GO's code doesn't create the user interface and data classes needed to implement behavior specific to your application. To get the behavior you want, you often need to create **descendant** classes that **inherit** from existing classes.

A descendant of clsApp

Lots of the behavior that is common to all applications is already implemented for you.

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

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.

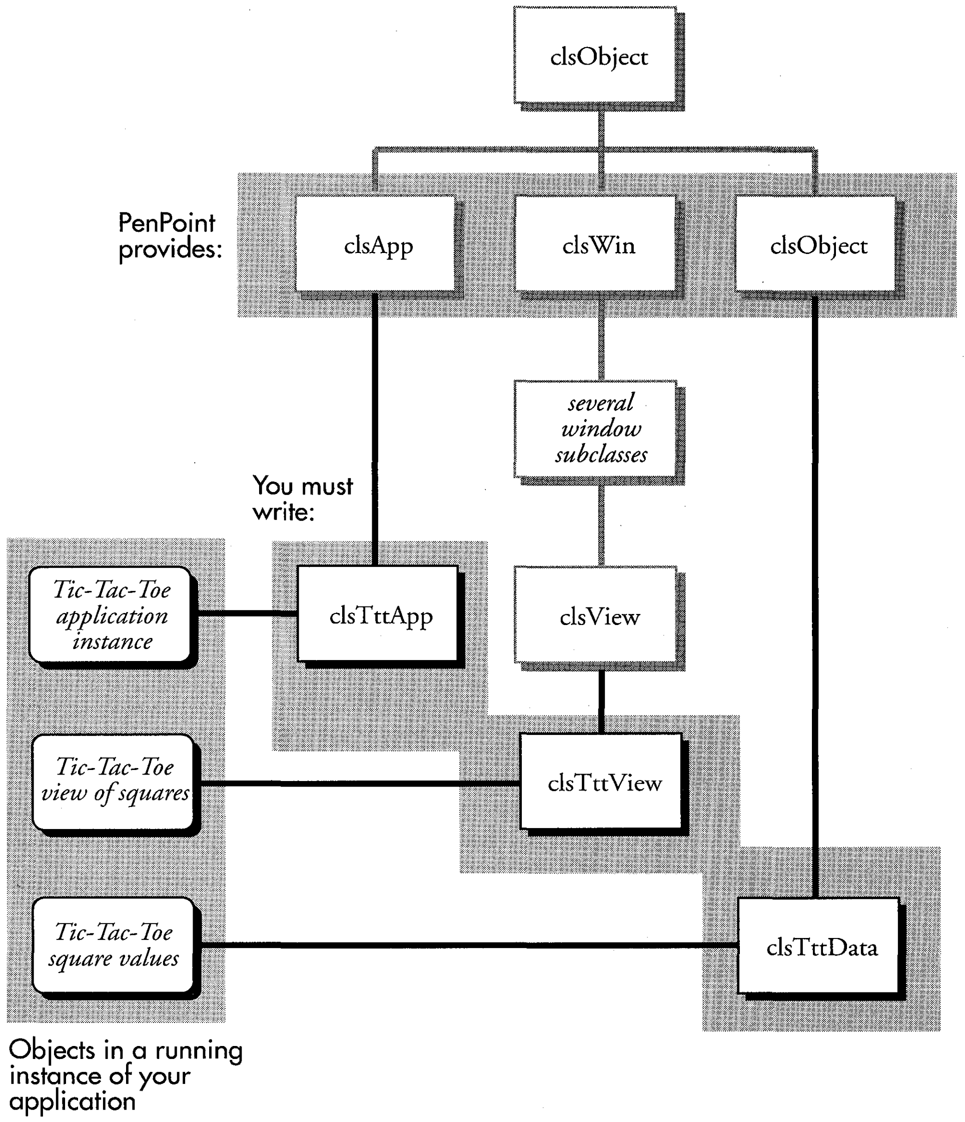
You must write a descendant class of **clsApp** and create it during installation. In the example shown here, the descendant is **clsTttApp**. At the appropriate time, the PenPoint Application Framework sends this class a message to create an instance of the class (Tic-Tac-Toe application instance in the figure). However, you must decide when to create your application's other objects (windows and filing objects).

An instance of clsWin

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

Application, view, and object classes FIGURE 3-1


clsObject
PenPoint provides:
clsApp
clsWin
clsObject
several window subclasses
You must write:
Tic-Tac-Toe application instance
clsTttApp
clsView
Tic-Tac-Toe view of squares
clsTttView
Tic-Tac-Toe square values
clsTttData
Objects in a running instance of your application

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 that draws what you want or you must use some of the many window descendant classes in PenPoint.

Some window classes

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

- 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**, **clsInteger-Field**) 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

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

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) is maintained by a separate object, Tic-Tac-Toe square values, an instance of the specialized class **clsTttData**.

Understanding the application hierarchy

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 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 (see Figure 3-3).

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

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

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!

- The 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.

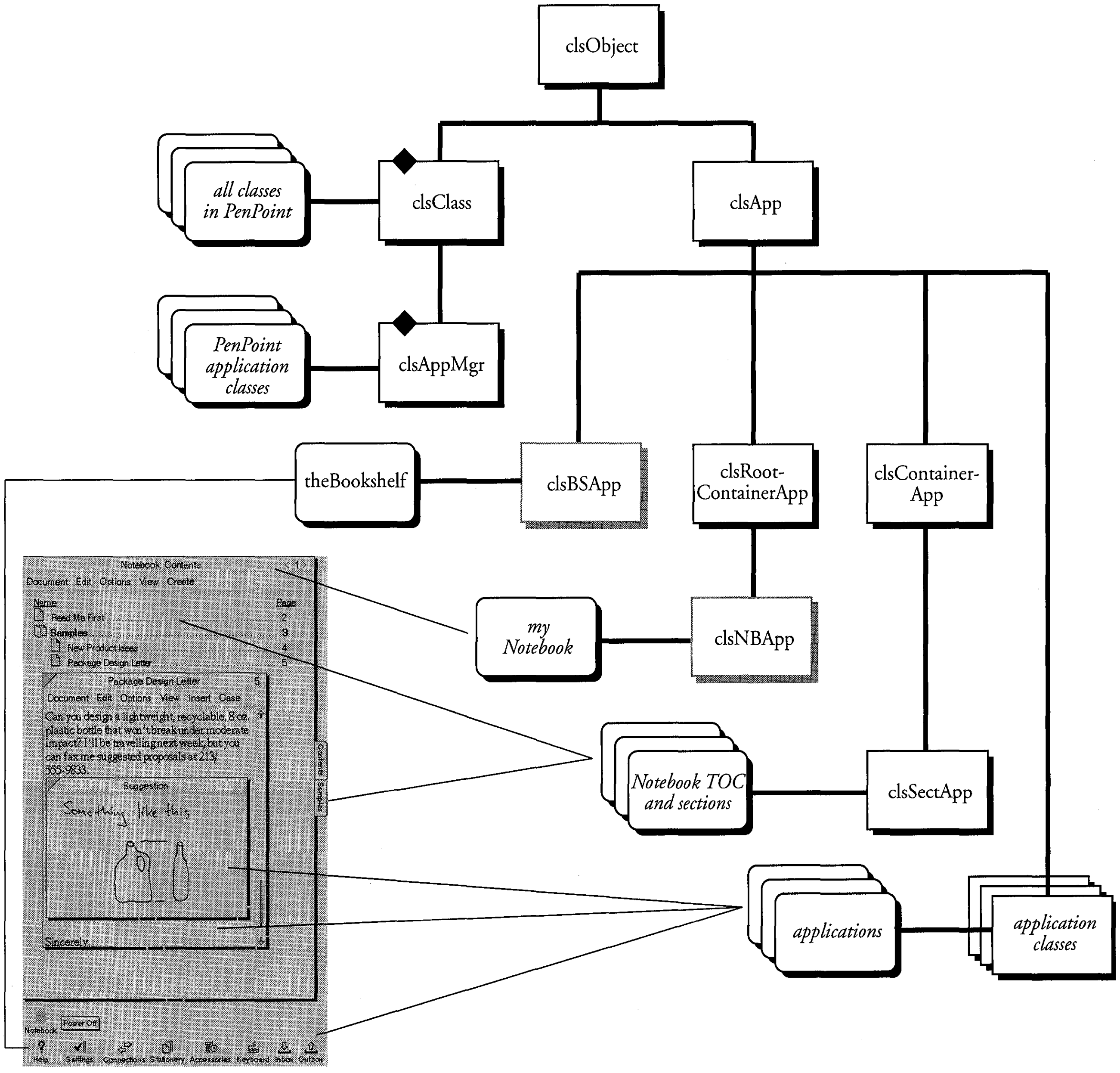
Section name and page number location in a notebook combine to form a location in the file system.

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*.

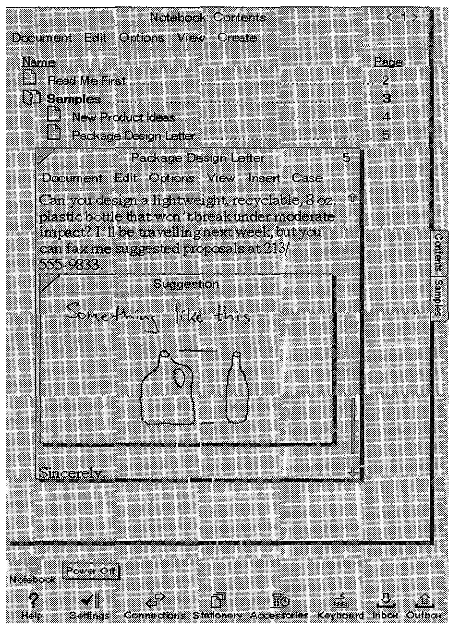
Application Framework and Notebook hierarchy

FIGURE 3-2



Can you design a lightweight, recyclable, 8 oz. plastic bottle that won't break under moderate impact? I'll be travelling next week, but you can fax me suggested proposals at 213/ 555-9833.

The Notebook hierarchy as mirrored by the file system FIGURE 3-3



Name	Page
Read Me First	2
Samples	**3**
New Product Ideas	4
Package Design Letter	5

Can you design a lightweight, recyclable, 8 oz. plastic bottle that won't break under moderate impact? I'll be travelling next week, but you can fax me suggested proposals at 213/ 555-9833.

Sincerely,

```
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...
```

Figure 3-3 and Figure 3-4 indicate how the same visual components exist in the file system, and as processes and objects.

You can use the Connections notebook to explore the relationship between documents and the file system yourself. To view the running PenPoint file system in the Connections notebook, you need to set the B debug flag to hexadecimal 800 in order to view the contents of the boot file system. The easiest way to do this is to modify the **DebugFlag** line in ENVIRON.INI.

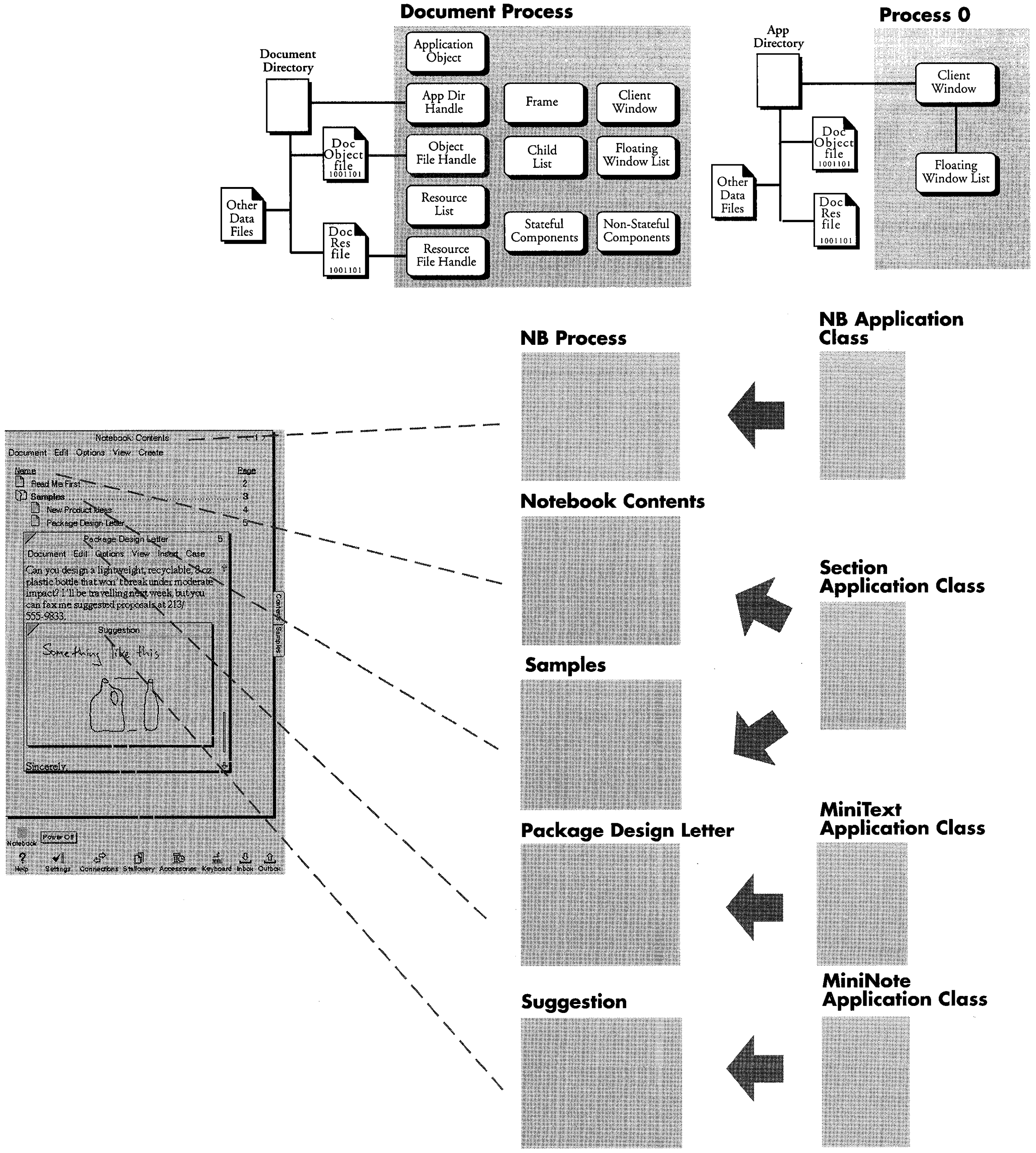
The Bookshelf

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

The Notebook

Below the Bookshelf's'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.

Notebook hierarchy and application processes FIGURE 3-4



Can you design a lightweight, recyclable, 8-oz plastic bottle that won't break under moderate impact? I'll be travelling next week, but you can fax me suggested proposals at 213/555-9833.

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

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 3-3), the directory in which each of the items are stored, and the class from which each item is instantiated.

Notebook organization and the file system

TABLE 3-1

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

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 attribute 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.

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.

Floating accessories

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 Bookshelf when active, appearing over pages in the Notebook. Their parent window is the Bookshelf, 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

It is possible to **embed** documents in other documents that permit it. For example, an on-line "electronic newspaper" document might embed 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 that can embed, and can 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 Bookshelf 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 that is the main window for the crossword document. The crossword document window contains a window that is the main window for the text editor.

Application data

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

In the section "Application classes and instances" on page 28, 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

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

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 closing

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)

In an MS-DOS or Macintosh program, the user explicitly quits 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

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...

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.
- 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 or exporting data that will be used by a file-oriented program on a file-oriented operating system.

Stationery

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

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

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

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

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

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 DLLs 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. Still 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 check ✓ 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

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 The term "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

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 **iconWindow** object to a routine called **WindowSetString()**.

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 **IconSetString()** 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 nonerror completion status), reject the message (by returning an error completion status), or give the message to its ancestor.

Classes instead of code sharing

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, but not quite, everything you want, 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

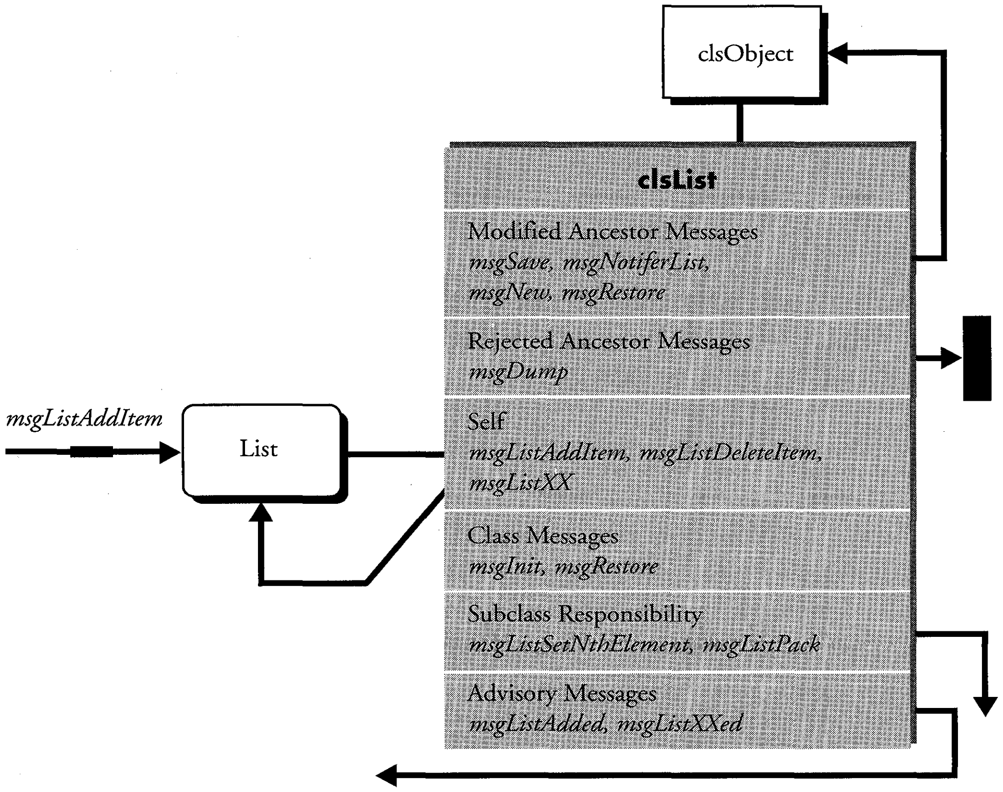
An object can send a message to its ancestor class either when it doesn't recognize the message or when it chooses to allow its ancestor class to handle the message.

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

Message handling by a class and its ancestors

FIGURE 4-1



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 fully class-based (in fact, for an application to run under PenPoint, it *must* be class-based).

Sending a message

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

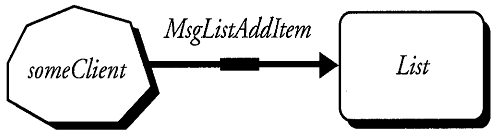
Here's a real-life example of sending a message. PenPoint provides a utility class, **clsList**, which maintains a list object. The messages that **clsList** responds to are documented in Part 9: Utility Classes, of the *PenPoint Architectural Reference* and in the **clsList** header file (PENPOINT\SDK\INC\LIST.H).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

Sending msgListAddItemAt to a list FIGURE 4-2



Message arguments

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:

```
typedef struct LIST_ENTRY {
      U16        position;
      LIST_ITEM  item;
} LIST_ENTRY, *P_LIST_ENTRY;
```

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.

U16 is an unsigned 16-bit number, P_UNKNOWN means a 32-bit pointer to an unknown. (Chapter 5, Developing an Application, 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           = (LIST_ITEM)'G';
// 2. Now send the message and message arguments to the object.
if ((s = ObjectCall(msgListAddItemAt, list, &add)) != stsOK) {
      Dbg(Debugf(U_L("add item failed: status is: 0x%lX", s)));
      }
```

ObjectCall() parameters

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 UIDs, unique 32-bit ID numbers. UIDs are discussed in more detail later.
- The arguments for the message (add). Not all messages take arguments (**msg-FrameClose**, 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, **msgList-AddItem** takes a P_LIST_ENTRY, a pointer to a LIST_ENTRY structure. (The Pen-Point 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

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

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.

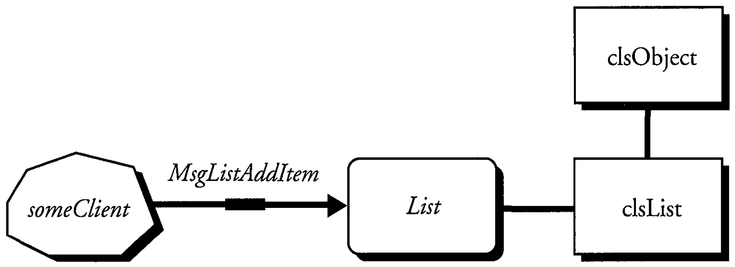
The Class Manager finds out which class created the object and looks for the method table for that class. The method table tells the Class Manager that the class has a function entry point for that message, so the Class Manager calls that function entry point, passing in the message and the message argument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

How messages to instances are processed by classes FIGURE 4-3



Creating an object

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

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

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" on page 54. 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

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

The exceptions are pseudo-classes and abstract classes.

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 _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;
      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

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 definition for the _NEW structure (**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 (**list-NewFields**) has two parts:

- The **#define** name for the **objectNewFields** 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

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

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

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.
ObjCallWarn(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, **ObjCallWarn()**. **ObjCallWarn()** 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**, **ObjCallWarn()** prints a warning to the debugging output device (when compiled with the DEBUG flag). 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

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 previously, 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 referring to the new list object.

You refer to objects using UIDs. 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. Contact GO Customer Services at 1-415-358-2040 (or by Internet electronic mail at gocustomer@go.com) for information on how to get a unique 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

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;

      ObjCallWarn(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

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(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

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 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

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 that 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).
- The internal context the Class Manager uses to keep track of classes (**ctx**).
- A pointer to the instance data of the instance.

Self

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 descendants the opportunity to override the message, now handles the message itself (or even passes the message to its ancestor!).

Possible responses to messages

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 that 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 messages are defined
MSG_INFO ListMethods [] =
{
      /* clsObject methods */
      msgNewDefaults, "ListNewDefaults", 0,
      msgInit, "ListInit", 0,
      msgFree, "ListMFree", 0,
      msgSave, "ListSave", 0,
      msgRestore, "ListRestore", 0,
          /* clsList methods */
          msgListFree, "ListMFree", 0,
          // Functions for the rest of the clsList methods...
          ...,
          ...,
          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 the PenPoint™ operating system 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

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

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 Reference.*

Designing classes

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

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

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

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

PenPoint 2.0 Japanese contains 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 Japanese. The first is making your application port easily to PenPoint 2.0 Japanese. The second is internationalizing your application.

Porting from PenPoint 1.0 to PenPoint 2.0 Japanese

PenPoint 2.0 Japanese incorporates some major changes that will cause applications compiled for PenPoint 1.0 to be incompatible with PenPoint 2.0 Japanese. The data created by 1.0 applications should still work under PenPoint 2.0 Japanese, and properly writtern 1.0 applications should be portable to PenPoint 2.0 Japanese 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 Japanese. 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 United States English application to PenPoint 2.0 Japanese.

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

Character types

PenPoint provides three character types: **CHAR8**, **CHAR16**, and **CHAR**. The first two provide 8- and 16-bit characters, respectively. In PenPoint 1.0, the plain **CHAR** type is 8 bits long; in PenPoint 2.0 Japanese, **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 Japanese (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

All of the familiar C string routines (**strcmp**, **strcpy**, and so on) still exist in PenPoint 2.0 Japanese, and they still work only 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 Japanese, they are 16-bit routines. In other words, the old routines only work on **CHAR8** strings, while the U...() routines work on **CHAR8** strings in PenPoint 1.0 and on **CHAR16** strings in PenPoint 2.0 Japanese. If you use the U...() versions and **CHAR** strings in your PenPoint 1.0 code, you will not have to change anything for PenPoint 2.0 Japanese, because CHAR is an 8-bit value in PenPoint 1.0 and a 16-bit value in PenPoint 2.0 Japanese.

You should use the U...() versions wherever you use **CHAR** strings, which should be for every string you display on the screen or debugging ourput device.

Character and string constants

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 precede the character or string constant with the letter L, which tells the compiler you are using a 16-bit (long) character, as:

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

When you use the **CHAR** type, you must precede 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 Japanese, this tells the compiler to use 16-bit characters.

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

Preparing for internationalization

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 to reduce the work needed to internationalize. This section lists these facilities.

Move strings into resource files

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. 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

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.

PenPoint 2.0 Japanese provides a number of **services** to perform functions that vary by language, such as 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 of PenPoint 2.0 Japanese can install several different sort engines and choose one to use with the current language.

You should identify and flag any language-dependent routines, such as text manipulation, in your PenPoint 1.0 application. When you port the code to PenPoint 2.0 Japanese, use services to replace them wherever possible.

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 PenPoint 2.0 Japanese will be much easier.

New text composition routines

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 resource ID 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

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, A Simple Application (Empty Application), the PenPoint SDK includes sources for an empty application called Empty Application. You can copy, compile, install, and run Empty Application.

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

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

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 the 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

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

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 nonstateful objects.

Creating component classes

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

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

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

- The make file.
- The application's method table files.
- The application's C source and header files.
- The PenPoint SDK header and library files.

Method table files

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

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.

PenPoint SDK files

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

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 PENPOINT\SDK\BOOT\locale\APP.INI file (where *locale* is USA for United States English and JPN for Japanese).

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

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

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

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 nonessential items

Use this checklist to ensure that you have considered all possible nonessential 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

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

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

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 variable names are prefixed with a lowercase *p*. The letter following the *p* is capitalized.

```
P_U16        pColorMap;
```

Functions

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)

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

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

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

Class names

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

Well-known objects

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

Messages

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**, **msg-AddrBookChanged**, 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

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* of the *PenPoint Architectural Reference.*

Exported names

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

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 comment
#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 (for application classes)
```

File header comment

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.

Include directives

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

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 reading even the first few lines of a header file a second time:

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

Common header files

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

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

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

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(clsXxx, 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

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

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

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

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.
- 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

In developing PenPoint, we found it useful to establish a "base" environment that 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

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 PenPoint 2.0 Japanese, which includes support for international character sets, we've changed CHAR to U16. Simply stated, you should use CHAR instead of char to ensure an easier transition to PenPoint 2.0 Japanese.

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

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. "Return values" on page 80 describes status values in greater detail.

Basic constants

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

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

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

Function qualifiers

GO.H introduces a layer in between the special function qualifier keywords, such as STATIC, 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, for example, to experiment with changing the 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.

Enumerated values

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 **OR**ed to specify several flags. If a PenPoint header file uses the **flag0**-style bit flags, assume that you can OR these flags.

Data conversion and checking

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

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

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* of the *PenPoint Architectural Reference* for an explanation of how classes, tags, and administered numbers relate to each other.

Return values

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

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

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

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 "Error-handling macros" on page 82.

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 **clsGO**.

Generic status values TABLE 5-1

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	The message is not supported.
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

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

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 in *Part 7: Sample Code.*

Error-handling macros

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

Status-checking macros TABLE 5-2

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

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 Development 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**() lines in your code.

Printing debugging strings

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

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

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.

Debugging flags

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 flag 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 flag sets 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 be:

```
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 within a **Dbg()** macro so that the flag-testing code is only compiled while in the testing version of their program.

Setting debugging flag sets

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

Isolate debugging messages

In general, always isolate all debugging code using preprocessor directives:

```
#ifdef DEBUG
Debugf(U_L("Debugging output string"))
#endif
```

DEBUG is the conventional flag for debugging code, used by much of PenPoint. If you have a short statement that you want to isolate for debugging purposes, you can use the **Dbg()** macro, which has the effect of using the preprocessor directives shown above:

```
Dbg(Debugf(U_L("Debugging output string")))
```

Use the status-checking macros

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

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

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

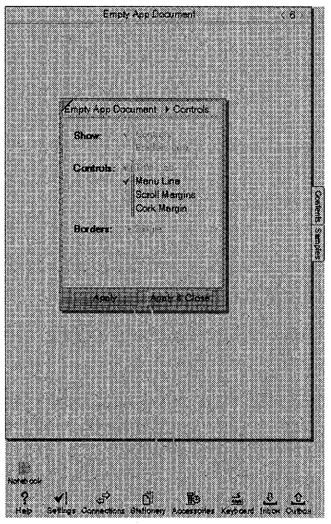
Chapter 6, A Simple Application (Empty Application), explains how to compile and run programs using Empty Application. The chapter is quite long because it teaches the general development cycle:

- 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

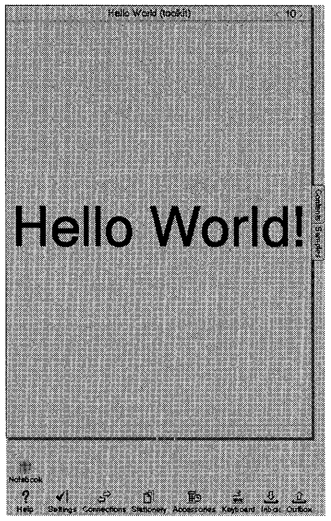
The tutorial starts off with an extremely simple application, Empty Application. Chapter 6 explains how to build and run it and how the application works. Empty Application has no view, no data, and no application-specific behavior (apart from printing a debugging message). It only responds to one message 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)

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.

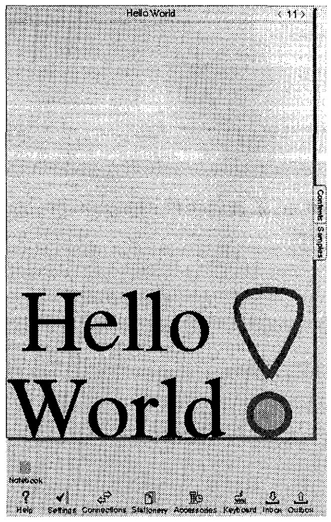
Hello World (toolkit) is described more fully in Chapter 7, Creating Objects (Hello World: Toolkit).



Hello World (Custom Window)

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.



Hello
World

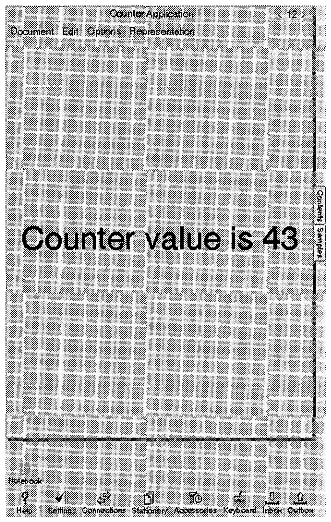
Counter Application

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 that 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.



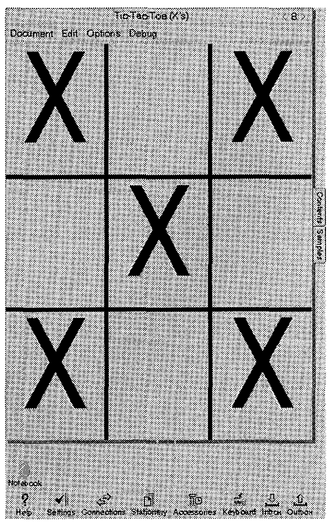
Counter value is 43

Tic-Tac-Toe

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, selection, data import and export, option cards, undo support, stationery, help text, and so on.



Template 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.

Other code available

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, Sample Code. Excerpts from sample programs also appear and are described in those parts of the *PenPoint Architectural Reference* that cover related subsystems.

Chapter 6 / A Simple Application (Empty Application)

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 the PenPoint™ operating system, 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. Like all PenPoint applications, Empty Application is a subclass of **clsApp**, so Empty Application inherits all of the Application Framework behavior. 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

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 rules that tell the make utility how to build Empty Application.

There is also a text file file called README.TXT that describes Empty Application, but the README.TXT file is not required to compile and link the application.

Not the simplest

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 methods inherited from **clsApp**. Empty Application handles one message by printing a string to the debug stream, so it needs a method table.

Compiling and linking the code

The source code for sample applications is in subdirectories of PENPOINT\SDK\SAMPLE. Each subdirectory contains a "makefile" that tells the make utility how to build the application. All you need to do to compile and link Empty Application is make PENPOINT\SDK\SAMPLE\EMPTYAPP the current directory and start the make utility, but you 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

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 that the make utility uses include a default rule for compiling method tables. MT produces an object file and a header file for the method table. You use these files when you compile and link the application.

Installing and running Empty Application

As described in the "How applications work" on page 24, you must install an application in PenPoint before you can run it. To install Empty Application, you either install it at boot time or use the Settings notebook 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\APP\Empty Application to PENPOINT\BOOT\locale\APP.INI (where *locale* is USA or JPN).
- 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" on page 101 explains the difference between Installation and Activation, and the relationship between PenPoint processes and application classes.

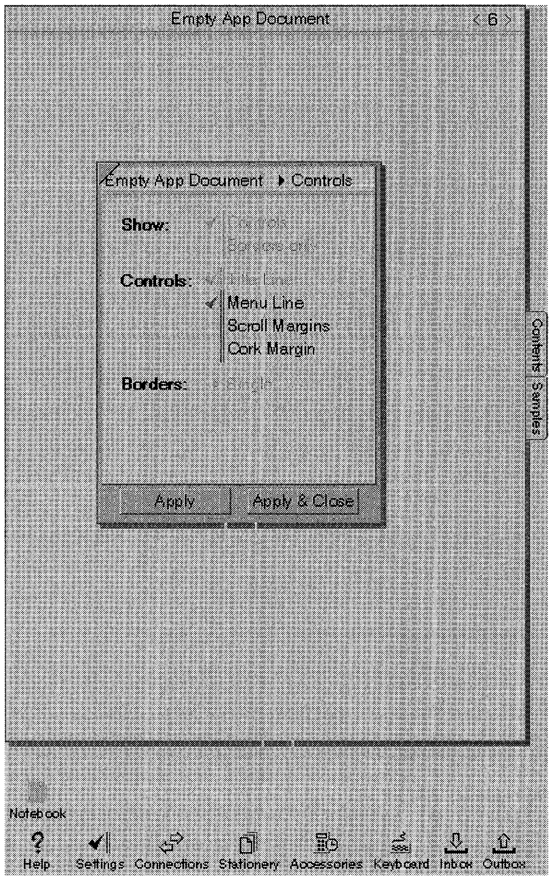
Interesting things you can do with Empty Application

Although Empty Application doesn't do any useful 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 $\wedge$ 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 of 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 check ✓ in its title bar. An option sheet for the document appears, with several cards in it for the document's appearance.
- 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.
- 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.
- As you turn the pages, note the sequence of messages sent to each instance of **clsEmptyApp** by the PenPoint Application Framework.
- 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 a cross out X over it.

Empty Application option sheet

FIGURE 6-1



- 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 Application 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.

Code run-through

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

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 organization of the C source code is described in the sections below.

Method table file

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.

The method table file always has the suffix .TBL. It looks like C code, but you process it with the method table compiler MT before linking it into your program.

A single method table file can have method tables for several different classes. The names of the method tables are usually pretty self-explanatory, typically the name of the class with the word **Methods** appended. For example, Empty Application's class is **clsEmptyApp**, and Empty Application's method table is **clsEmptyAppMethods**.

Although the normal practice is to define a method for each message, you can use the wild-card feature of method tables to have one method handle several messages. 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 the *PenPoint Architectural Reference.*

Application C code file

By convention, an application source code file is usually organized into the following sections:

- **#include** directives for the header files required by the application.
- **#defines** and **typedefs**.
- Utility routines.
- Message handlers.
- Class initialization routine.
- **main**() entry point.

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 nonapplication 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

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 declarations. The **MsgHandler()** macro generates a message handler declaration based on the name of the function. The **MsgHandlerWithTypes()** macro generates a message handler declaration based on the name of the function and the types to which to cast its arguments.

Empty Application's source code

This section presents an overview of Empty Application's method table and C source code.

Method table

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

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 (
      S32         argc,
      CHAR *      argv[],
      U32         processCount)
{
      Dbg(Debugf(U_L("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.
      //
      ObjCallWarn (msgNewDefaults, clsAppMgr, &new);
      new.object.uid    = clsEmptyApp;
      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;
      Ustrcpy(new.appMgr.company, U_L("GO Corporation"));
      Ustrcpy(new.appMgr.defaultDocName, U_L("Empty App Document"));
      ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);
      //
      // Turn on message tracing if flag is set.
      //
      if (DbgFlagGet('F', 0x1L)) {
            Debugf(U_L("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(U_L("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

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:

Common header files

TABLE 6-1

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

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 obtain a unique administered value from GO. Contact GO Customer Services at 1–415–358–2040 (or by Internet electronic mail at gocustomer@go.com) for information on how to get a unique administered value. Until you get an administered value for your application, you can use the pre-defined 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.

Be on the lookout for conflicts with other test software when using the well-known testing UIDs (**wknGDTa** through **wknGDTg**). If another application happens to use the same well-known testing UID for one of its classes, you will have problems installing your application because it has the same UID as another class.

Class creation

The initialization routine **ClsEmptyAppInit()** creates the **clsEmptyApp** class. It 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_NEWnew;
      STATUS      s;
      //
      // Install the Empty App class as a descendant of clsApp.
      //
      ObjCallWarn (msgNewDefaults, clsAppMgr, &new);
      new.object.uid         = clsEmptyApp;
...
      Ustrcpy(new.appMgr.defaultDocName, U_L("Empty App Document"));
      ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);
...
```

clsAppMgr explained

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 It is better to specify the company name, default document name, and other locale-dependent information in the USA.RES or JPN.RES file. This helps when porting your application to other locales.

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;
Ustrcpy(new.appMgr.company, U_L("GO Corporation"));
Ustrcpy(new.appMgr.defaultDocName, U_L("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.

```
ObjCallWarn (msgNewDefaults, clsAppMgr, &new);
new.object.uid          = clsEmptyApp;
new.cls.pMsg            = clsEmptyAppTable;
new.cls.ancestor  = clsApp;
//
// This class has no instance data, so its size is zero.
//
new.cls.size            = Nil(SIZEOF);
```

The **cls.pMsg** argument to **msgNew** establishes the connection between the new class and its method table. More on this later.

Documents, accessories and stationery

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?

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

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 (
      S32         argc,
      CHAR *      argv[],
      U32         processCount)
{
      Dbg(Debugf(U_L("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 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()

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();
        }
    // Suppress compiler's "unused parameter" warnings
    Unused(argc); Unused(argv);
    } /* main */
```

Most applications follow these simple steps and have a **main**() routine similar to the one in EMPTYAPP.C.

A complex explanation of main()

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 Installed Applications page of the Settings notebook. PenPoint or the Settings notebook 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 nonsystem 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

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 **AppMonitor-Main**() 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 in an indirect fashion when the user chooses Empty Application from the Tools notebook or the Stationery notebook. 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

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 **OSProgram-Instantiate()**. **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 an ordinary 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, and so on—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. **AppMain()** does not return until the user turns away from the document and the application instance can be terminated.

Handling a message

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 except one, 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

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's 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.cls.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.
//
ObjCallWarn (msgNewDefaults, clsAppMgr, &new);
new.object.uid    = clsEmptyApp;
new.cls.pMsg      = clsEmptyAppTable;
new.cls.ancestor  = clsApp;
...
```

msgDestroy

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

The message handler (also known as a **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

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:

```
#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
#ifndef INTL_INCLUDED
#include <intl.h>      // for international routines
#endif
#include <methods.h>  // method function prototypes generated by MT
#include <string.h>    // for strcpy().
...
```

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()

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(U_L("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

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

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 debugger stream output

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 debugging tools designed to run under DOS because these packages require your executable file to be running 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

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 that this application could have used instead; the message tracing and status warning debugging facilities use these fancier output formats.

Seeing debug output

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 directory specified with **PenPointPath** 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 Development Tools.*

Chapter 7 / Creating Objects (Hello World: Toolkit)

Although Empty Application shows that the Application Framework can do many things for an application, Empty Application 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." With the PenPoint™ operating system, 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 UI elements, 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 that 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

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

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 as HELLOTK.C before building the application.

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\locale\APP.INI (where *locale* is a locale such as JPN or USA) 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

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

HELLOTK1.C creates a single label in its frame.

Highlights of HELLOTK1

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 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

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()

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

Because messages return a status value, you should usually check their return values. This would ordinarily lead to lots and lots of constructs in your code, such as the following:

```
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 **ObjCall-Warn**(), **ObjCallRet**(), **ObjCallJmp**(), **ObjCallChk**(), and **ObjCallOK**().

The return value of **ObjCallWarn**() is the status value returned by **ObjectCall**(). If compiled with the DEBUG preprocessor variable 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

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?

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. **clsWin** implements the standard window behavior of multiple overlapping regions on a pixel device, but **clsWin** does not draw images in a window. The descendants of **clsWin** inherit **clsWin**'s behavior and add the ability to draw images, handle input, and so on. All of the UI toolkit components inherit from **clsBorder**, a special kind of window which knows how to draw a border.

Tip Some of the key decisions you make in any object-oriented programming system are choosing which 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 multilevel glory. For a hint of what it can do, Figure 7-1 shows a screen shot with all the different kinds of UI Toolkit components present.

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:

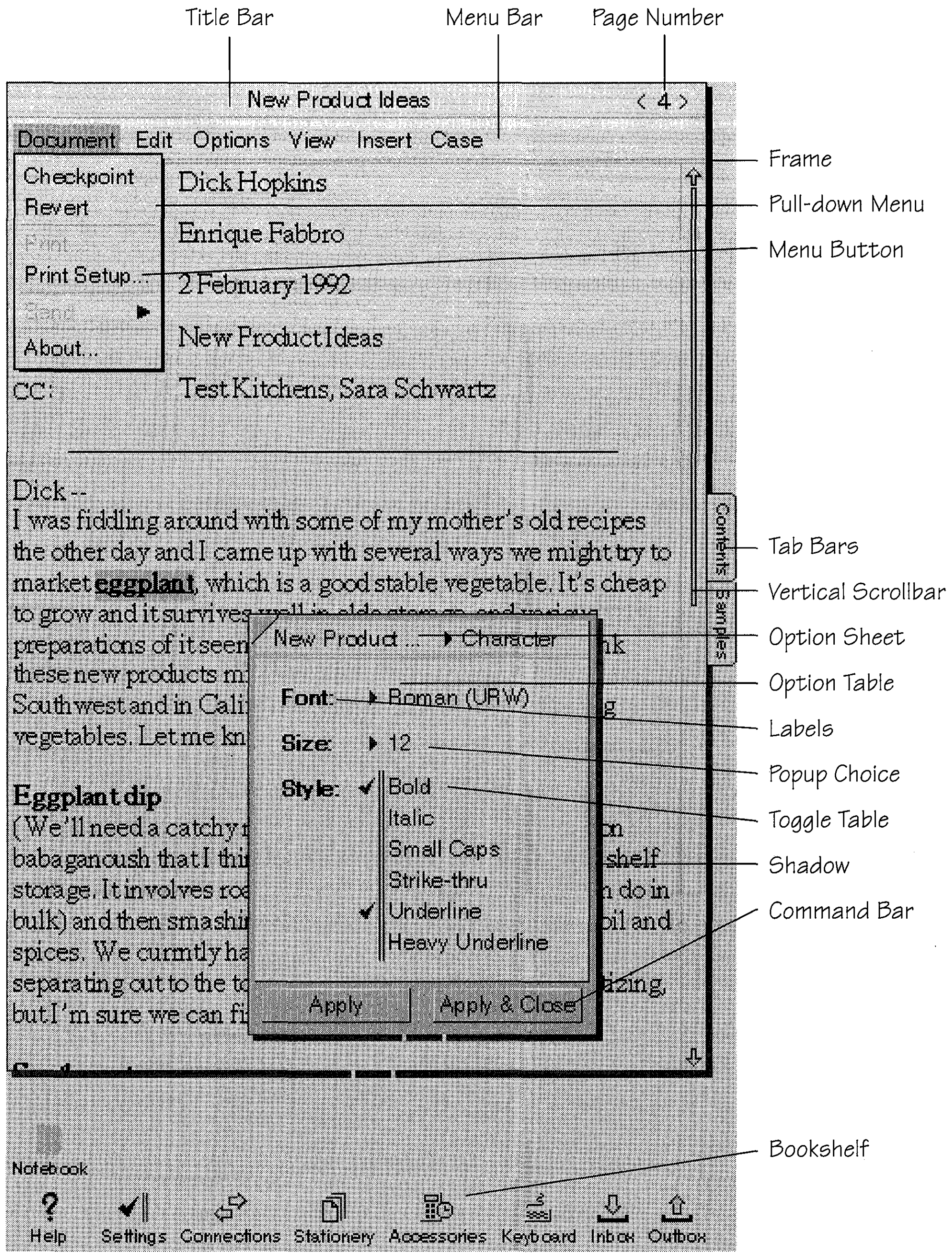
- Use the class browser to get a brief description of it and all its messages.
- Read about it in its subsystem's part of the *PenPoint Architectural Reference.*
- Look up its "datasheets" in the *PenPoint API Reference.*
- Look at its header file in PENPOINT\SDK\INC.

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

As you learned in Chapter 3, 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

UI Toolkit components FIGURE 7-1



Dick Hopkins
Enrique Fabbro
2 February 1992
New Product Ideas
CC: Test Kitchens, Sara Schwartz

Dick --
I was fiddling around with some of my mother's old recipes the other day and I came up with several ways we might try to market **eggplant**, which is a good stable vegetable. It's cheap to grow and it survives

Eggplant dip

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) names it **ln**, the first letter of each part of the structure name. The sample code follows this naming convention consistently.

```
/****************************************************************************
      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);
```

Note the use of **ObjCallWarn**() instead of **ObjectCall**(). As mentioned earlier, **ObjCallWarn**(), when compiled with DEBUG set, sends a warning message to the debugging output device when it returns a non-zero status value.

Now you're ready to give values to those fields in the structure that 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 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 = U_L("Hello World!");
```

You can look up the hierarchy for a class by looking in the *PenPoint API Reference* section 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_ONLYembeddedWin;
      BORDER_NEW_ONLY   border;
      CONTROL_NEW_ONLYcontrol;
      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** that 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.
ObjCallWarn(msgNewDefaults, clsLabel, &ln);
ln.label.style.scaleUnits     = bsUnitsFitWindowProper;
ln.label.style.xAlignment     = lsAlignCenter;
ln.label.style.yAlignment     = lsAlignCenter;
ln.label.pString              = U_L("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 show up on-screen.

Where the window goes

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 UI Toolkit components. A frame can include other windows within it. 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 specific 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 that received the message. This is actually very common. The Class Manager provides a parameter to methods, **self**, which identifies the object that received the message.

Why msgAppInit?

Earlier you turned on message tracing to Empty Application. This causes 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, whenever 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?

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

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

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

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, 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

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 **msg-WinDelta**. 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 that support these styles, **clsCustom-Layout** 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) that 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)**.

```
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

You might consider trying to add one of the following to HELLOTK2.C.

Fields

Change the label to be an editable field. There are several ways of handling hand-writing in PenPoint. One way is to use a UI component that allows editing, **cls-Field**. 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

Add some more controls, using different custom layout constraints. You should be able to put together a simple control panel.

General model of controls

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 to display the string "Hello World!" on screen by creating and drawing in custom windows.

Hello World (Custom Window)

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 create new classes of windows to implement the special behavior you require.

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

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 class of the client window (**clsHelloWin**).

You can build Hello World (custom window) by changing the directory to PENPOINT\SDK\SAMPLE\HELLO and running the make utility.

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

When you link DLLs (dynamic link libraries), the information you provide to the linker is slightly different from the information you provide when linking an executable image. In addition to the object code, the linker 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.'lname'
```

Letter case doesn't matter in the DLL.LBC file.

The **entry-point** is the name of the exported function. In PenPoint, this is **DLLMain**(), the entry point for the DLL. The PenPoint Installer uses the *lname* to identify code modules. An **lname** is composed of a company ID, a project name, and a revision number. The revision number takes the form V*major*(*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-V2(0)'
```

In this example, the entry point is **DLLMain**() and the lname is **GO-HELLO_DLL-V2(0)**. The lname indicates that the company is GO, the project is **HELLO_DLL**, and the version is 2(**0**).

DLC files

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-V2(0) hello.dll
GO-HELLO_EXE-V2(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 2(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.

Highlights of clsHelloWorld

The method table for **clsHelloWorld** (in HELTBL.TBL) 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

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, *P_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

To draw in a window you need to create a **drawing context** object (often abbreviated to 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

The standard **system drawing context** (sometimes abbreviated to **sysDC**) 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 system drawing context 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 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

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

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 will receive **msgWinRepaint**. **clsHelloWin** has no need to dirty itself since it doesn't change what it paints.

When to create things

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 waits until it needs an object before it creates the object, it will use less memory before it creates the object. But creating objects takes time, so you may want to create the object at initialization time, before the user interacts with the application, to reduce the time it takes your application to respond to the user. As is often the case, you must strike a balance between memory and performance.

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 in terms of time. 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

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?

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

Here's the **HelloWinInit**() code that 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(data));
```

clsHelloWin then creates a DC:

```
// Create a dc.
ObjCallRet(msgNewWithDefaults, clsSysDrwCtx, &dn, s);
```

When **msgNewWithDefaults** 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;
```

msgNewWithDefaults works like **msgNewDefaults** followed immediately with **msgNew**. Use **msgNewWithDefaults** when you don't need to modify the defaults before creating the object.

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)) {
      Dbg(Debugf(U_L("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

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 de-reference 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

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 that 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

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

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)

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-V2(0)" \\boot\penpoint\sdk\sample\hello\hello.exe
srcdir "go-hello_exe-V2(0)" \\boot\penpoint\sdk\sample\hello
sym "go-hello_dll-V2(0)" \\boot\penpoint\sdk\sample\hello\hello.dll
srcdir "go-hello_dll-V2(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)

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 maintain information about what is on screen, how the user last interacted with the application, 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

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 the application instance is coming and going "behind" what is seen on-screen.

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

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 that model the natural structure of the system.

It's natural to think of the application 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** remembers the format.

Note the difference between Counter Application 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 Counter Application'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 a 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 Counter Application, so we create a separate counter class. Figure 9-1 shows the classes defined by Counter Application and their ancestors.

Because the UI Toolkit uses the symbol **clsCounter** already, Counter Application uses the symbol **clsCntr** for its counter class.

Compiling and installing the application

To compile Counter Application, change to the PENPOINT\SDK\SAMPLE\CNTRAPP directory and start the MAKE utility. This creates a PENPOINT\APP\CNTRAPP directory and compiles CNTRAPP.EXE in that directory.

Install Counter Application either by adding \\BOOT\PENPOINT\APP\Counter Application to PENPOINT\BOOT\locale\APP.INI (where *locale* is USA for United States English or JPN for Japanese) before starting PenPoint or by installing the application using the Installer.

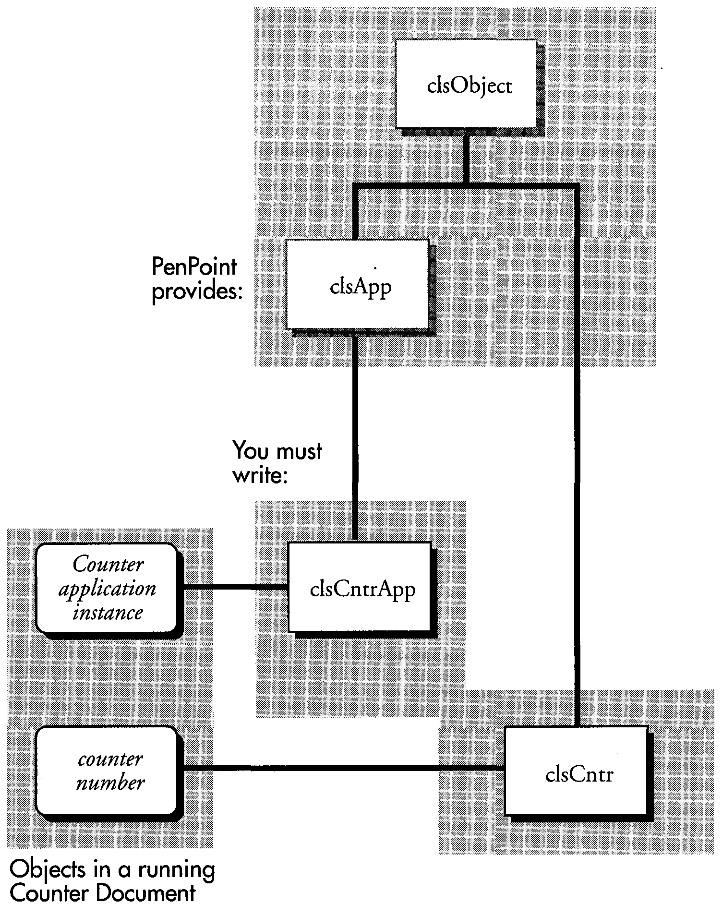
Counter Application highlights

The method table for **clsCntrApp** handles a number of interesting messages:

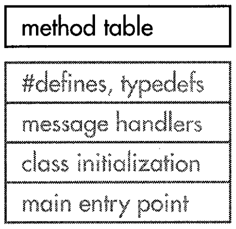
```
msgInit,                  "CntrAppInit",          objCallAncestorBefore,
msgSave,                  "CntrAppSave",          objCallAncestorBefore,
msgRestore,               "CntrAppRestore",       objCallAncestorBefore,
msgFree,                  "CntrAppFree",          objCallAncestorAfter,
msgAppInit,               "CntrAppAppInit",       objCallAncestorBefore,
msgAppOpen,               "CntrAppOpen",          objCallAncestorAfter,
msgAppClose,              "CntrAppClose",         objCallAncestorBefore,
msgCntrAppChangeFormat,   "CntrAppChangeFormat", 0,
```

Counter Application objects

FIGURE 9-1



clsCntrApp creates an instance of clsCntr at msgAppInit time.



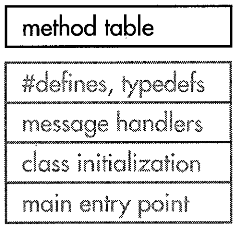
clsCntrApp responds to **msgAppOpen** by incrementing the counter, creating a label containing the counter value, making the label the client window, and creating the menu bar.

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



The method table for **clsCntr** is also defined in METHODS.TBL and handles these messages:

```
msgNewDefaults,        "CntrNewDefaults",    objCallAncestorBefore,
msgInit,               "CntrInit",           objCallAncestorBefore,
msgSave,               "CntrSave",           objCallAncestorBefore,
msgRestore,            "CntrRestore",        objCallAncestorBefore,
msgFree,               "CntrFree",           objCallAncestorAfter,
msgCntrGetValue,       "CntrGetValue",       0,
msgCntrIncr,           "CntrIncr",           0,
```

Instance data

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 **msgCntr-GetValue**. 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 **msgNew-Defaults**:

```
MsgHandlerArgType(CntrNewDefaults, P_CNTR_NEW)
{
      Dbg(Debugf(U_L("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;
      Dbg(Debugf(U_L("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

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

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)
{
      Dbg(Debugf(U_L("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

msgCntrIncr increments the value. It doesn't take any arguments.

```
MsgHandlerWithTypes (CntrIncr, P_ARGS, P_CNTR_INST)
{
      CNTR_INST inst;
      Dbg(Debugf(U_L("Cntr:CntrIncr"));)
      inst = *pData;
      inst.currentValue++;
      ObjectWrite(self, ctx, &inst);
      return stsOK;
      MsgHandlerParametersNoWarning;
} /* CntrIncr */
```

There are a couple of things to note here. First, 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. If the code had tried to update the protected **pData->currentValue** directly, PenPoint would have generated a general protection fault. That is why the code assigns the instance data (the implicit **pData** argument) to a variable (**inst**) before modifying it. After modifying the copy of the instance data, the code calls **ObjectWrite**(), which directs the Class Manager to update the protected instance data stored in the object.

Note Two frequent sources of programming error are trying to modify protected, read-only instance data (instead of a copy of the instance data), and forgetting to update instance data with **ObjectWrite**() after modifying the copy of the instance data.

Second, the code uses the **MsgHandlerWithTypes**() macro to define the function. **MsgHandlerWithTypes**() works like **MsgHandler**(), but lets you specify data types for the message argument (**pArgs**) and instance data (**pData**). In addition to the name of the method, the **MsgHandlerWithTypes()** macro takes two arguments that indicate the types of the message arguments (**pArgs**) and the instance data (**pData**).

In the **CntrIncr()** example above, the **MsgHandlerWithTypes()** casts the message argument as a P_ARGS pointer (the default for **pArgs**), and the instance data as a P_CNTR_INST, a pointer to a CNTR_INST structure.

Object filing

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 \\BOOT\PENPOINT\SYS\Bookshelf\Notebook\CONTENTS, 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

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

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:

```
    Dbg(Debugf(U_L("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 */
```

Counter Application's instance data

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 Counter Application document's page, the Application Framework destroys the counter object (and the application instance). When the user turns back to the Counter Application 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 (in which case there's no reason to file it), or the UID is dynamic (in which case the UID will be different when the object is restored).

Memory-mapped file

Counter Application 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.

Counter Application 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

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

When Counter Application receives **msgAppInit**, it creates the counter object:

```
MsgHandler(CntrAppInit)
{
      CNTRAPP_INST inst;
      Dbg(Debugf(U_L("CntrApp:CntrAppInit"));)
      inst.counter = pNull;
      inst.fileHandle = pNull;
      inst.pFormat = pNull;
```

```
        // Update instance data.
        ObjectWrite(self, ctx, &inst);
        return stsOK;
        MsgHandlerParametersNoWarning;
    } /* CntrAppInit */
```

Opening to restore

CntrAppRestore() opens a file (called FORMATFILE) where it stores the document's instance data, then maps the file to memory. This makes the format data available in **inst.pFormat**.

```
    MsgHandlerWithTypes(CntrAppRestore, P_OBJ_RESTORE, P_CNTRAPP_INST)
    {
        FS_NEW       fsn;
        CNTRAPP_INST inst;
        STATUS       s;
        Dbg(Debugf(U_L("CntrApp:CntrAppRestore"));)
        // Get handle for format file, and save the handle.
        // The default for fsn.fs.locator.uid is theWorkingDir, which
        // is the document's directory.
        ObjCallWarn(msgNewDefaults, clsFileHandle, &fsn);
        fsn.fs.locator.pPath = U_L("formatfile");
        ObjCallRet(msgNew, clsFileHandle, &fsn, s);
        inst.fileHandle = fsn.object.uid;
        // Map the file to memory
        ObjCallRet(msgFSMemoryMapSetSize, fsn.object.uid, \
                            (P_ARGS)(SIZEOF)cntrAppMemoryMapSize, s);
        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 */
```

Using a memory-mapped file handle lets you maintain just one copy of the instance data when the document is open. The alternative is to create an instance data structure in memory when the document opens, copy the filed data to the structure in memory, then copy changes from the in-memory data structure to the file before closing the document.

Closing on msgFree

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;
        Dbg(Debugf(U_L("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

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

When Counter Application 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;
        Dbg(Debugf(U_L("CntrApp:CntrAppSave"));)
        // Save the counter object.
        ObjCallRet(msgResPutObject, pArgs->file, pData->counter, s);
        return stsOK;
        MsgHandlerParametersNoWarning;
} /* CntrAppSave */
```

Counter Application doesn't have to write the instance data to a file, because the data is a memory-mapped to a file.

Restoring the counter object

When Counter Application receives **msgRestore**, it sends **msgResGetObject** 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;
```

```
      Dbg(Debugf(U_L("CntrApp:CntrAppRestore"));)
      // Get handle for format file, and save the handle.
      // The default for fsn.fs.locator.uid is theWorkingDir, which
      // is the document's directory.
      ObjCallWarn(msgNewDefaults, clsFileHandle, &fsn);
      fsn.fs.locator.pPath = U_L("formatfile");
      ObjCallRet(msgNew, clsFileHandle, &fsn, s);
      inst.fileHandle = fsn.object.uid;
      // Map the file to memory
      ObjCallRet(msgFSMemoryMapSetSize, fsn.object.uid, \
                  (P_ARGS)(SIZEOF)cntrAppMemoryMapSize, s);
      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 */
```

Menu support

clsCntrApp creates a menu by specifying the contents of the menu statically in a toolkit table. **clsTkTable** is the ancestor of several UI components that 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 **msgApp-CreateMenuBar.**

Buttons

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, or a resource ID tag that refers to a string in a resource file.
- The notification message the button should send when the user activates it.
- A value for the button.
- If the string is specified as resource ID, the type of resource (usually **tkLabel-StringID** to specify a string resource).

There are other fields in a TK_TABLE_ENTRY, but you can rely on their defaults of 0.

The menu bar used in Counter Application is described by the TK_TABLE_ENTRY structure named **CntrAppMenuBar()** in CNTRAPP.C. **CntrAppMenuBar()** specifies user-readable strings, such as the name of the menu and the menu items, with resource ID tags.

```
typedef enum CNTRAPP_DISPLAY_FORMAT {
      dec, oct, hex
}  CNTRAPP_DISPLAY_FORMAT,
   *P_CNTRAPP_DISPLAY_FORMAT;
...
/*
 * Here we use tags that are associated with strings in a resource file
 * for the name of our menu and the menu items.
 *
 * When using tags in a TKTable, the fifth field must be an id that gives
 * the type of the tag.  If there is an item already in the fifth field,
 * you can 'or' the two field items, and the system will know which one
 * to use.
 */
static const TK_TABLE_ENTRY CntrAppMenuBar[] = {
      {tagCntrMenu, 0, 0, 0, tkMenuPullDown | tkLabelStringId, clsMenuButton},
            {tagCntrDec, msgCntrAppChangeFormat, dec, 0, tkLabelStringId},
            {tagCntrOct, msgCntrAppChangeFormat, oct, 0, tkLabelStringId},
            {tagCntrHex, msgCntrAppChangeFormat, hex, 0, tkLabelStringId},
            {pNull},
      {pNull}
};
```

When the user taps one of the menu buttons, the menu button sends **msgCntr-AppChangeFormat** 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[MAXSTRLEN];
      Dbg(Debugf(U_L("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 that 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

Tic-Tac-Toe, also known as Naughts and Crosses, is a game where two players take turns placing markers on a 3 x 3 grid. The object is to place three markers in a row while preventing your opponent from placing three markers in a row.

The rest of this chapter details the architecture of Tic-Tac-Toe, its files, classes, objects, etc, and describes some of the enhanced application features implemented in Tic-Tac-Toe.

Tic-Tac-Toe objects

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 will probably have to redesign your object hierarchy at least once. (Here at GO, we redesigned our class hierarchy countless times in the first two years—now it is 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.

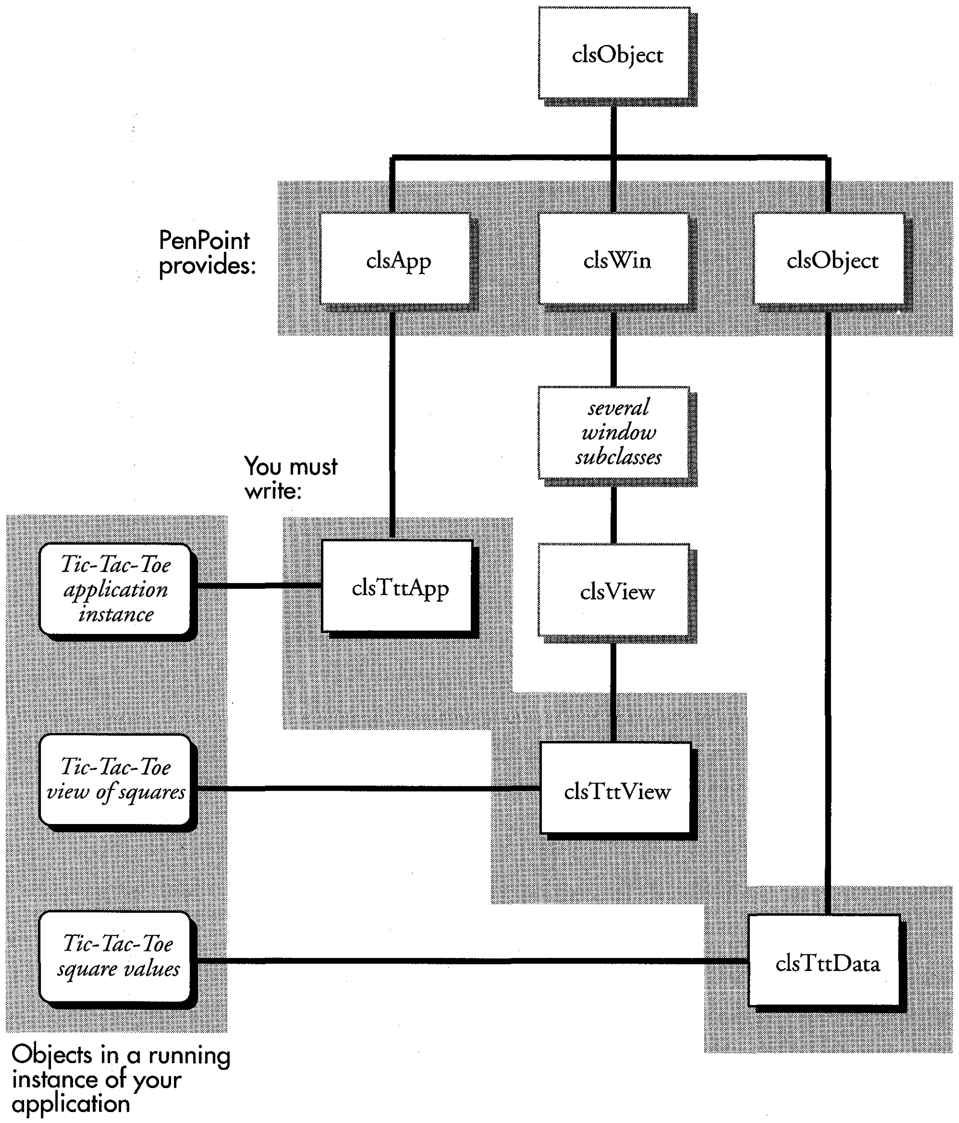
Each section from now on will discuss the various design choices made.

Application components

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**.

Tic-Tac-Toe classes and instances FIGURE 10-1


clsObject
PenPoint provides:
clsApp
clsWin
clsObject
several window subclasses
You must write:
Tic-Tac-Toe application instance
clsTttApp
clsView
Tic-Tac-Toe view of squares
clsTttView
Tic-Tac-Toe square values
clsTttData
Objects in a running instance of your application

Separate stateful data objects

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 the thickness of the lines on the Tic-Tac-Toe board.

Tic-Tac-Toe structure

Table 10-1 lists the files in PENPOINT\SDK\SAMPLE\TTT that you use to build the Tic-Tac-Toe application (the directory also contains some text files that provide information about the application)

Tic-Tac-Toe files

TABLE 10-1

File name	*Purpose*
MAKEFILE	Dependency definitions for the MAKE utility.
METHODS.TBL	Message tables for **clsTttApp**, **clsTttView**, and **clsTttData**.
USA.RC	Resource file containing the strings and other resources for the United States English localization.
JPN.RC	Resource file containing the strings and other resources for the Japanese localization.
S_TTT.C	Sets up UID-to-string translation tables that the Class Manager uses to provide more informative debugging output.
TTTPRIV.H	**TttDbgHelper()** support macro and debugger 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

There is no pre-existing class that draws letters in a rectangular grid. So, some work is needed here. The PenPoint UI Toolkit provides labels that can have borders, along with **clsTableLayout** that lets you position windows in a regular grid. So, you could create the Tic-Tac-Toe board by creating nine 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. A window in PenPoint is fairly lightweight (that is, it has a small system resource requirement), but if we were to change to a 16 x 16 board, it would use 256 single-character label windows.

A even more efficient way to draw the grid is to create a window and use the ImagePoint™ graphics system to draw the lines of the 3 x 3 grid.

Coordinate system

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

As explained in Hello World, the window system tells windows to repaint. When a window receives **msgWinRepaint**, it always self-sends **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

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

Tic-Tac-Toe's data object class, **clsTttData**, is similar to Counter Application'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.)

Instance data by value vs. by reference

The instance data for each of **clsTttApp**, **clsTttView**, and **clsTttData** is a pointer that points to a data structure outside the instance. The outside data structure is where the class stores the information. Storing instance data by reference in this way has some advantages:

- You don't have to use **ObjectWrite()** to update instance data every time state changes, since the pointer never changes.
- The size of the instance data can vary.

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

clsTttApp tells its view to file, and an instance of **clsView** automatically files its data object.

Handling failures during msgInit and msgRestore

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
      STATUS           s;
      DbgTttDataInit((U_L("")))
      // 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((U_L("return stsOK")))
      return stsOK;
      MsgHandlerParametersNoWarning;
Error:
      if (pInst) {
            OSHeapBlockFree(pInst);
      }
      DbgTttDataInit((U_L("Error; returns 0x%lx"),s))
      return s;
} /* TttDataInit */
```

The selection and keyboard input

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 that the selected object may receive). So, to allow typing, the Tic-Tac-Toe view must "be selectable."

How selection works

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?

Because the Tic-Tac-Toe view is the object that 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?

The application developer must decide what input event causes a selection in the Tic-Tac-Toe 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, and in white when it does not have the selection. It determines whether it has the selection by sending **msgSelIsSelected** to **self**. Here is the code from **TttViewRepaint()** (**msgSelIsSelected** returns **stsOK** if the receiver of the message has the selection):

```
// Fill the dirty rect with the appropriate background.  If we hold the
// selection, the appropriate background is grey, otherwise it is white.
//
s = ObjectCall(msgSelIsSelected, self, pNull);
if (s == stsOK) {
      DbgTttViewRepaint((U_L("self is selected")))
      ObjectCall(msgDcSetBackgroundRGB, (*pData)->dc, \
                  (P_ARGS)sysDcRGBGray33);
} else {
      DbgTttViewRepaint((U_L("self is not selected")))
      ObjectCall(msgDcSetBackgroundRGB, (*pData)->dc, \
                  (P_ARGS)sysDcRGBWhite);
      }
```

Supporting selections

When a Tic-Tac-Toe view receives a **msgPenHoldTimeout** input event, **clsTttView** self-sends **msgTttViewTakeSel** telling it to acquire the selection, and self-sends **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

The selection holder receives a variety of messages, including **msgSelYield** 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

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

Views inherit from **clsGWin**, so there is little extra work required to make **clsTttView** respond to input events and gestures.

Input event handling

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 handles **msgInputEvent** with a routine called **TttViewInputEvent()**.

TttViewInputEvent() calls a routine to process keyboard events, and lets **clsTttView**'s ancestor, **clsEmbeddedWin**, handle pen events. **clsEmbeddedWin** includes code for handling many pen input events. For example, **clsEmbeddedWin** watches for pen-hold timeouts and, if **self** is moveable or copyable, self-sends **msgSelSelect** and **msgSelBeginMoveCopy**.

```
MsgHandlerWithTypes(TttViewInputEvent, P_INPUT_EVENT, PP_TTT_VIEW_INST)
{
      STATUS      s;
      switch (ClsNum(pArgs->devCode)) {
            case ClsNum(clsKey):
                  s = TttViewKeyInput(self, pArgs);
                  break;
            default:
                  s = ObjectCallAncestorCtx(ctx);
                  break;
      }
      return s;
      MsgHandlerParametersNoWarning;
} /* TttViewInputEvent */
```

Gesture handling

When **clsTttView**'s ancestor gets a pen event, 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** are different in PenPoint 1.0 than in PenPoint 2.0 Japanese. For PenPoint 1.0, the arguments 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**. In PenPoint 2.0 Japanese, the arguments include the gesture in the form of a Unicode character that represents the gesture.

TttViewGesture(), the handler for **msgGWinGesture**, uses a switch statement to take appropriate action based on the gesture argument. Because the **msgGWin-Gesture** API is different in PenPoint 1.0 than in PenPoint 2.0 Japanese, the code uses an **#ifdef** directive that lets you compile the correct switch statement for PenPoint 1.0 by defining PP1_0, or setting it on the compiler command line. The switch statement handles the gestures that are meaningful to **clsTttView**, and lets the ancestor class handle all other gestures:

```
MsgHandlerWithTypes(TttViewGesture, P_GWIN_GESTURE, PP_TTT_VIEW_INST)
{
      STATUS      s;
#ifdef PP1_0
      switch(pArgs->msg) {
#else
      switch(pArgs->gesture) {
#endif

            case xgs1Tap:
                  ObjCallJmp(msgTttViewToggleSel, self, pNull, s, Error);
                  break;
            case xgsCross:
                  StsJmp(TttViewGestureSetSquare(self, pArgs, tttX), s, Error);
                  break;
            case xgsCircle:
                  StsJmp(TttViewGestureSetSquare(self, pArgs, tttO), s, Error);
                  break;
            case xgsPigtailVert:
                  StsJmp(TttViewGestureSetSquare(self, pArgs, tttBlank), \
                              s, Error);
                  break;
            case xgsCheck:
            case xgsUGesture:
                  // Make sure there is a selection.
                  s = ObjectCall(msgSelIsSelected, self, pNull);
                  if (s == stsNoMatch) {
                        ObjCallJmp(msgTttViewTakeSel, self, pNull, s, Error);
                        ObjCallJmp(msgWinUpdate, self, pNull, s, Error);
                  }
                  // Then call the ancestor.
                  ObjCallAncestorCtxJmp(ctx, s, Error);
                  break;
            default:
                  DbgTttViewGesture((U_L("Letting ancestor handle gesture")))
                  return ObjCallAncestorCtxWarn(ctx);
      }
      DbgTttViewGesture((U_L("return stsOK")))
      return stsOK;
      MsgHandlerParametersNoWarning;
Error:
      DbgTttViewGesture((U_L("Error; return 0x%lx"),s))
      return s;
} /* TttViewGesture */
```

Keyboard handling

clsTttView's keyboard input routine handles multikey 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 the X, O, and Space keys 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

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. It uses the following facilities for debugging:

- **msgTrace** to trace messages.
- **Debugf**() to print debugging messages.
- **msgDump** to dump the state of objects.
- **ClsSymbolsInit**() to give the Class Manager symbolic names for Tic-Tac-Toe's objects, messages, and status values.

The complexity in Tic-Tac-Toe arises because it lets you turn features on and off while the program is running.

Tracing

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}
};
```

Note that the strings in this TK_TABLE_ENTRY array, unlike the one used for the menu bar in Counter Application, are not specified with resource IDs. That's acceptable programming practice in this case, because the user will never see the trace menu. That means that you'll never have to translate the strings to another language, so there is no benefit to putting the strings into a resource file (and no cost to hard-coding them).

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

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 can 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((U_L("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, **DbgTttApp-CheckStationery**().

```
#define DbgTttAppCheckStationery(x) \
      TttDbgHelper(U_L("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 Unicode format string and any arguments ((U_L("**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 (U_L("**TttAppCheckStationery**")) together with any **printf**()-style format string passed in (x).

There are 256 debugging flag sets, each with 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're interested, 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

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

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

All the Class Manager's macros (**ObjCallRet**(), **ObjCallWarn**(), **ObjCall-AncestorChk**(), 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,    U_L("msgTttAppChangeDebugFlag"),
  msgTttAppChangeDebugSet,     U_L("msgTttAppChangeDebugSet"),
...
  msgTttViewTakeSel,           U_L("msgTttViewTakeSel"),
  0, 0};
const CLS_SYM_OBJ tttObjSymbols[] = {
  clsTttApp,      U_L("clsTttApp"),
  clsTttData,     U_L("clsTttData"),
  clsTttView,     U_L("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**():

```
STATUS EXPORTED TttSymbolsInit(void)
{
      return ClsSymbolsInit(
                  U_L("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 if DEBUG is not set.

Generating symbols automatically

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 on the unsupported SDK Goodies disk.

Printing symbol names yourself

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

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 separately 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 are covered in detail in *Part 12: Installation API* of the *PenPoint Architectural Reference.* This section only covers what Tic-Tac-Toe does.

Stationery

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 (see Figure 11-1).

Creating stationery

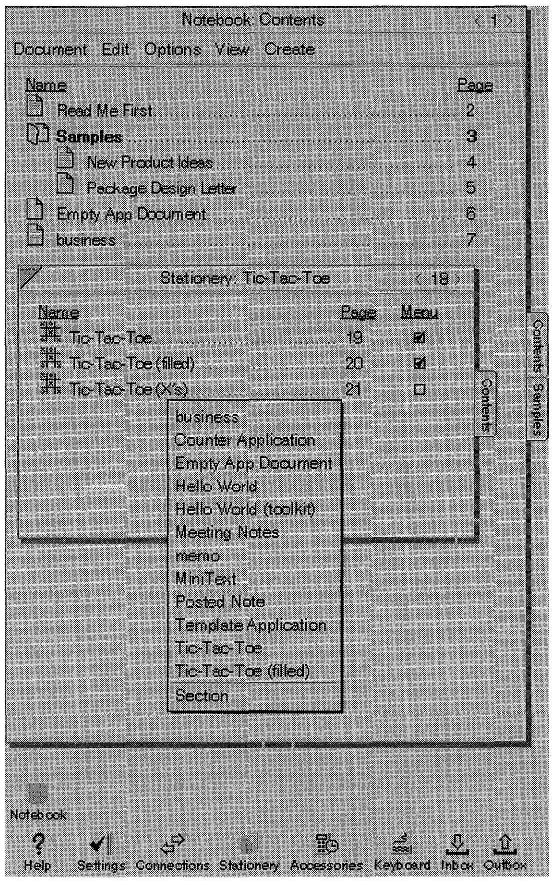
The Installer looks for Stationery in a subdirectory called STATNRY. Each Stationery document should be in a separate directory in STATNRY. You can stamp the directories with long PenPoint names, and in PenPoint 2.0 Japanese you can stamp the directory with a locale and read the long name from a resource file. 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.

How Tic-Tac-Toe handles stationery

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.

Stationery notebook and Stationery menu FIGURE 11-1



Name	Page
Read Me First	2
Samples	3
New Product Ideas	4
Package Design Letter	5
Empty App Document	6
business	7

Name	Page	Menu
Tic-Tac-Toe	19	☑
Tic-Tac-Toe (filled)	20	☑
Tic-Tac-Toe (X's)	21	☐

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\TTTSTAT1) 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 TTTSTAT1 and TTTSTAT2. The makefile copies the file FILLED.TXT to TTTSTAT1 and names it TTTSTUFF.TXT; it then copies the file XSONLY.TXT to TTTSTAT2 and also names it TTTSTUFF.TXT

Help notebook

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

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 TTTHELP1 and TTTHELP2 directories in TTTHELP. The makefile copies STRAT.TXT to TTTHELP1 and names it HELP.TXT; it then copies RULES.TXT to TTTHELP2 and names it also HELP.TXT.

Help documents can either be complete instances of filed documents (of any type, such as MiniText or MiniNote, 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

Quick Help is the other form of help in PenPoint. The Quick Help window appears when the user makes the question mark ? gesture in a window, or taps on a window when Quick Help is up (see Figure 11-2).

clsGWin, the gesture window class, automatically handles the Quick 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((U_L("self=0x%lx"),self))

      pArgs->win.flags.input |= inputHoldTimeout;
      pArgs->gWin.helpId = tagTttView;
      ...
      pArgs->view.createDataObject = true;
      ...
```

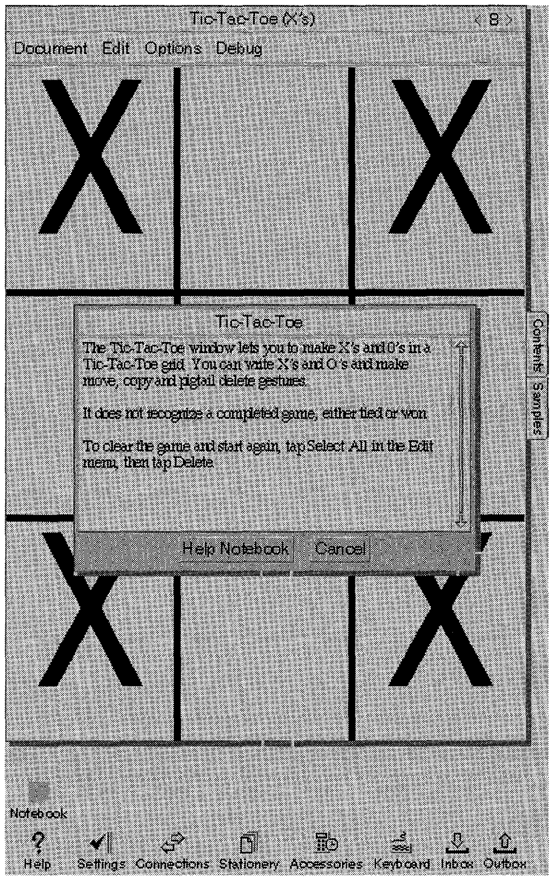
This is the only thing **clsTttView** must do to handle Quick Help.

Creating Quick Help resources

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

Quick Help FIGURE 11-2



Tic-Tac-Toe

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.

It does not recognize a completed game, either tied or won.

To clear the game and start again, tap Select All in the Edit menu, then tap Delete.

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 gesture window **helpIds** to associate a gesture window 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 case **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 (**resTaggedString-ArrayResAgent**).

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.

These are the United States English Quick Help strings for the Tic-Tac-Toe application, defined in USA.RC (there is also a Japanese version in JPN.RC):

```
// Define the Quick Help resource for TTT.
static RC_TAGGED_STRING tttViewQHelpStrings[] = {
      // Quick help for TTT's option card to change the line thickness.
      tagTttViewCard,
      U_L("TTT Card||")
      U_L("Use this option card to change the thickness of the lines ")
      U_L("on the Tic-Tac-Toe board."),
      // Quick help for the line thickness control in TTT's option card.
      tagCardLineThickness,
      U_L("Line Thickness||")
      U_L("Change the line thickness by writing in a number from 1-9."),
      // Quick Help for the TTT window.
      tagTttView,
      U_L("Tic-Tac-Toe||")
      U_L("The Tic-Tac-Toe window lets you to make X's and O's in a"
            "Tic-Tac-Toe ")
      U_L("grid. You can write X's and O's and make move, copy ")
      U_L("and pigtail delete gestures.\n\n")
      U_L("It does not recognize a completed game, either tied or won.\n\n")
      U_L("To clear the game and start again, tap Select All in the Edit menu, ")
      U_L("then tap Delete."),
      Nil(TAG)
};
static RC_INPUTtttViewQHelp = {
      resTttViewQHelp,
      tttViewQHelpStrings,
      0,
      resTaggedStringArrayResAgent
};
```

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 during installation. Hence the makefile tells the resource compiler to append the Quick Help resources to the application resource file. The name of the application resource file includes a three-letter code that indicates the locale for which you've compiled the resource file. For example, for the United States, the resource file is called USA.RES; for Japan, it is called JPN.RES. You define in the makefile which locales to compile resource files for.

Standard message facility

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, ...)**

With **StdMsg()**, **StdError()**, **StdSystemError()**, and **StdProgressUp()**, 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

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 resource file. Here's the resource file definition of all of the error notes for the CLOCK sample application:

```
static RC_TAGGED_STRING errorStrings[] = {
    // Error: user has set alarm to before the current time
    stsClockAlarmInvalid,
    U_L("You can't set the alarm date and time to be earlier ")
    U_L("than the current date and time."),
    // Error: user has specified an out-of-bounds number, like 12:72
    stsClockFieldRangeError,
    U_L("The ^1s you specified is invalid. Choose a number ")
    U_L("between ^2d and ^3d."),
            // Error: user has specified a string with an illegal character
            stsClockIntFieldInvalid,
            U_L("The ^1s you specified is invalid because it is ")
            U_L("blank or contains an invalid character."),
            // Error: user has specified a date not using month/day/year format
            stsClockDateFieldInvalid,
            U_L("The date ^1s you specified is invalid because it is ")
            U_L("blank or does not follow the format mm/dd/yy."),
            Nil(TAG)
      };
      static RC_INPUT stdError = {
            resClockAppStdMsgError,
            errorStrings,
            0,
            resTaggedStringArrayResAgent
      };
```

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 stsClockAlarmInvalid MakeStatus(clsClockApp, 0)
#define stsClockFieldRangeError MakeStatus(clsClockApp, 1)
#define stsClockIntFieldInvalid MakeStatus(clsClockApp, 2)
#define stsClockDateFieldInvalid MakeStatus(clsClockApp, 3)
```

To create a note from the items defined above, simply call **StdMsg()** or **StdError()**.

```
if ( (low <= value ) && (value <= high) ) {
      return stsOK;
} else {
      // else field out of range
      StdMsg(stsClockFieldRangeError, pFieldName, low, high);
      return stsFailed;
}
```

Progress notes are slightly different from the message functions. Your application displays a progress note 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

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**:

```
U_L("[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:

```
U_L("Here's the first paragraph.\n\nHere's the second one.")
```

StdMsg() and resource files or lists

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(resource_file, tag, ...)**
- **StdErrorRes(resource_file, status, ...)**

StdMsg() customization function

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**.

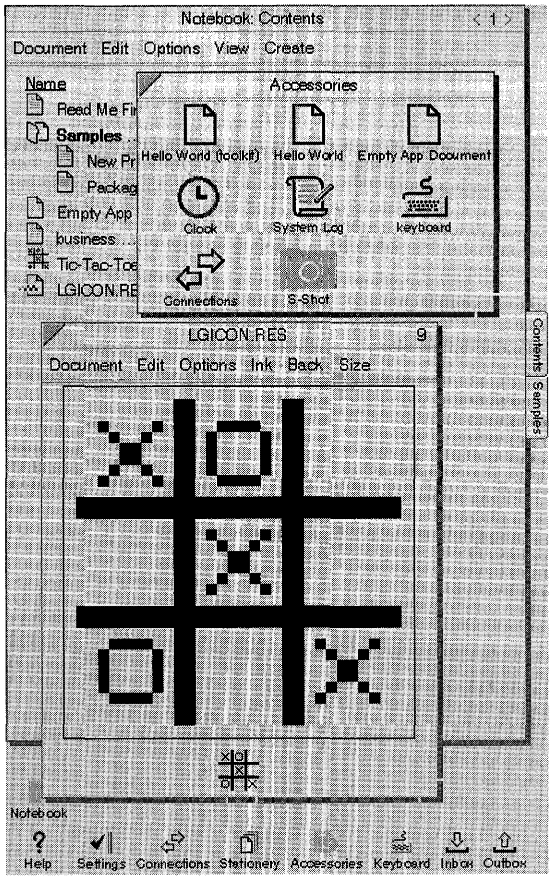
Bitmaps (icons)

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 (see Figure 11-3).

When it needs a bitmap, the Application Framework searches for it by resource ID in your application's resource list. If you do not specify an application icon in your application's resource file, 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.

Application and document icons

FIGURE 11-3



Creating icons

The bitmap editor application is available in the 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 16 x 16 and 32 x 32 sizes. Although you can export them to the APP.RES file in your application's installation directory, it is often preferable to create separate SMICON.RES and LGICON.RES files for the large and small icons and use the resource compiler to append these to your application's resource file.

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

While developing an application, you can identify your classes with the well-known UIDs **wknGDTa** through **wknGDTg**. Of course, if you use these 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 obtain an administered 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). Contact GO Customer Services at 1-415-358-2040 (or by Internet electronic mail at gocustomer@go.com) for information on how to get a unique administered value.

Documenting the application

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

For more information on documenting your application, contact GO Customer Services at 1-415-358-2040 (or by Internet electronic mail at gocustomer@go.com) 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 Notes also give print specifications, if you want your documentation to appear similar to GO's.

Screen shots

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

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 GO Customer Services.

On-disk structure

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

If you have created a component class that might be useful to other PenPoint developers, you should consider licensing the class.

Part 2 / PenPoint Internationalization Handbook

Chapter 13 / Introduction

The worldwide software market is growing at an exciting rate. Recognizing this trend, GO Corporation has designed PenPoint™ to be a global operating system. Specifically, the PenPoint operating system provides many objects, functions, and tools that help you prepare your application for an international market.

Modifying an application for use in a specific country (or locale, because countries like Canada, Switzerland, and Singapore use more than one language) is called **localization**. The process of preparing an application so that it is ready to localize is called **internationalization**.

This handbook provides a step-by-step guide through the process of internationalizing your code. The result is code that is ready to be adapted to particular markets.

This handbook does *not* contain specific guidelines on how to design a successful version of your application for a particular market. It does not, for example, offer specific suggestions on how to design an appropriate user interface for Japan or Germany or Italy.

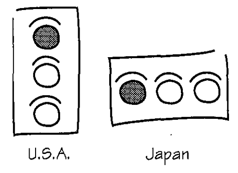
Because this handbook does not discuss localization, you may want to work with your marketing and sales departments, local software partners, localization houses, users, and other resources to create an appropriate product for a given locale. Also, Chapter 19, Additional Resources, lists resources that may help you with the localization process.

The PenPoint operating system currently supports only American English and Japanese. Future releases will support more languages and countries.

Intended audience

This handbook is for developers designing original PenPoint applications for an international market and for developers porting existing PenPoint 1.0 applications to PenPoint 2.0 Japanese. When used in this document, the terms "PenPoint 2.0" or "PenPoint SDK 2.0" refer to "PenPoint 2.0 Japanese" or "PenPoint 2.0 SDK Japanese."

Many issues that you need to consider while designing international applications are less important if you are developing only for a local market. For example, consider an icon designed to signal "stop." American users might expect such an icon to look like a traffic light with a red light on top. This icon would be inappropriate in many countries outside of the United States. In Japan, for example, traffic lights are horizontal so that the red light is on the far left.



This handbook assumes that you are familiar with PenPoint programming. *Part 1: PenPoint Application Writing Guide* is the best place to start if you are new to PenPoint programming. You will also get the most out of this handbook if you understand what your target locale is and what kind of application you plan to market there. Once you've identified the target locales for your application, you can tailor the general recommendations in this handbook to your specific needs.

This handbook also assumes you have installed PenPoint SDK 2.0 Japanese. See *Installing and Running the PenPoint SDK 2.0,* a document included with the SDK, for details on how to install the SDK.

For details on localizing your application to Japan, read *Part 3: PenPoint Japanese Localization Handbook.* This part assumes you have internationalized your code according to the guidelines in this handbook.

Handbook structure

This chapter describes the handbook's purpose and organization.

Chapter 14, Overview, describes the general process of how to write an international application.

Chapter 15, PenPoint Support for International Software, describes PenPoint's routines, functions, and utilities that support international applications.

Chapter 16, Procedures, provides step-by-step instructions on how to write code for international applications.

Chapter 17, Porting to PenPoint 2.0, discusses how to port existing PenPoint 1.0 code to the latest version of PenPoint.

Chapter 18, Localization Guidelines, lists general issues to consider while localizing your application.

Chapter 19, Additional Resources, describes other helpful resources, including dictionaries, books, and contacts for standards organizations.

How to use this handbook

If you are new to internationalization issues, begin with Chapter 14, Overview. If you are familiar with internationalization issues and want to learn how the PenPoint operating system helps you produce internationalized applications, read Chapter 15, PenPoint Support for International Software. Both of these conceptual chapters place the procedures covered in Chapter 16, Procedures, in perspective.

If you are looking for specific directions on how to perform a task such as supporting Unicode, see "Supporting Unicode" on page 209 in Chapter 16, Procedures. Each procedure refers to other information, usually in this handbook, that you need to understand before doing the procedure.

Chapter 14 / Overview

This chapter provides a general overview of how to write international software. Chapter 15, PenPoint Support for International Software, provides specific details on how some special features of the Japanese localization of PenPoint™ operating system 2.0 help you write international software.

Many books and organizations may also help you internationalize your application. Chapter 19, Additional Resources, lists some of these resources.

Overview of international software

The goal of internationalization is to make an application easily adaptable for a target locale. The internationalized version of your source code must support international character sets and behave appropriately for different **locales**.

In this handbook, we usually use the term locale to identify a particular country, language, and dialect. Often, as with the USA and Japan, the country name is enough to identify a locale because the country uses only one language and no major dialects. Sometimes, however, as in Canada, Switzerland, and Singapore, countries use multiple languages and even dialects. In such cases, a locale represents both a country and a language (and sometimes a dialect) such as French-speaking Canada or Chinese-speaking Singapore.

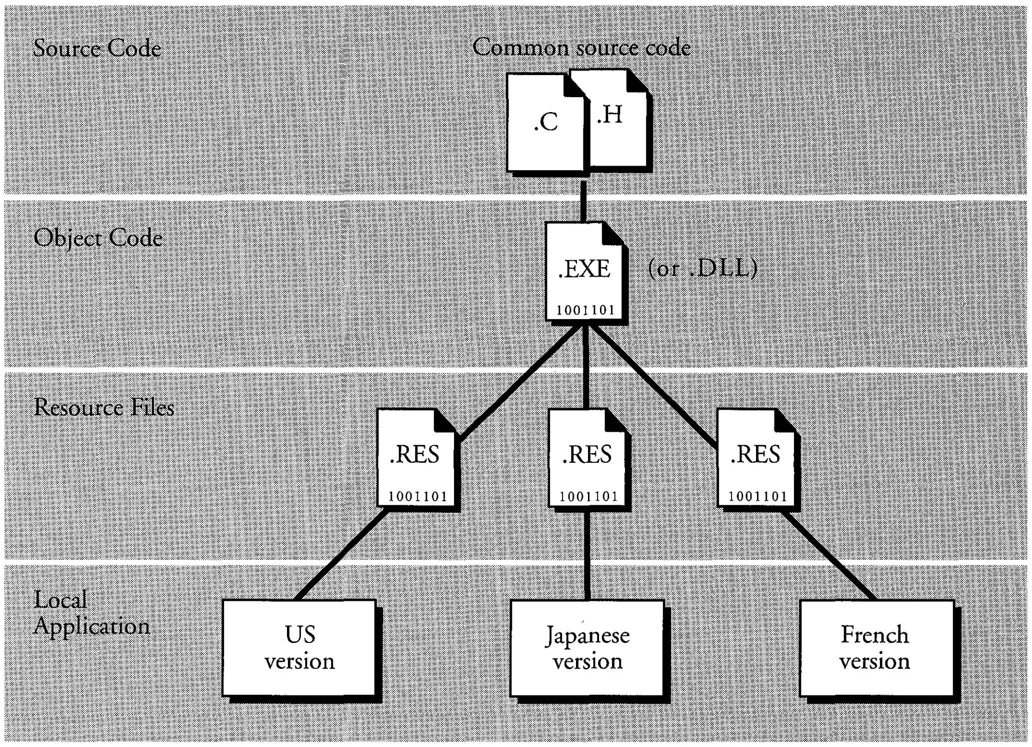
In the best case, you can maintain a single code base for your application and create localized versions by simply creating different **resource files**. Figure 14-1 shows this optimum design. Remember that resource files are collections of data, such as strings, that are cleanly separated from your application code.

Think of resource files as modular pieces that can be snapped in and out of your application as the locale requires. What kind of pieces might be stored as resource files? Anything that the user sees is a likely candidate. These include user interface strings, window layouts, icons, and bitmaps.

You can also use resource files to store binary data that your application interprets. For example, you might use a flag in a resource file to tell your application whether to calculate in English or metric units. Another set of flags might represent user preferences.

Sometimes local versions of your application differ too much for you to maintain a common code base. For example, a Japanese version of your application might need to provide context-sensitive handwriting recognition that a German version of your application does not need to provide.

Common source code for multiple localizations FIGURE 14-1



Machine-readable files, such as object files, are shown with binary number along their bottom edge; files you create and edit are shown without the binary numbers.

When local versions of your application differ too much for you to maintain a common code base, create different DLLs from different source code files as shown in Figure 14-2. In the example above, you might write all the code that provides additional Japanese handwriting support in one source file, compile the code into a DLL, and then link that library with the rest of your application's object files.

If you must resort to separate code bases for each locale, keep the differences isolated to as few files as possible. This strategy makes it easier to test and maintain your code.

Writing international software

This section introduces the process of writing international software. After this overview, you may want to look through Figure 14-1. This checklist is a detailed, step-by-step description of how to implement the general tasks described here.

Internationalizing an application requires these four general steps:

1 Prepare to handle international character sets by supporting Unicode.

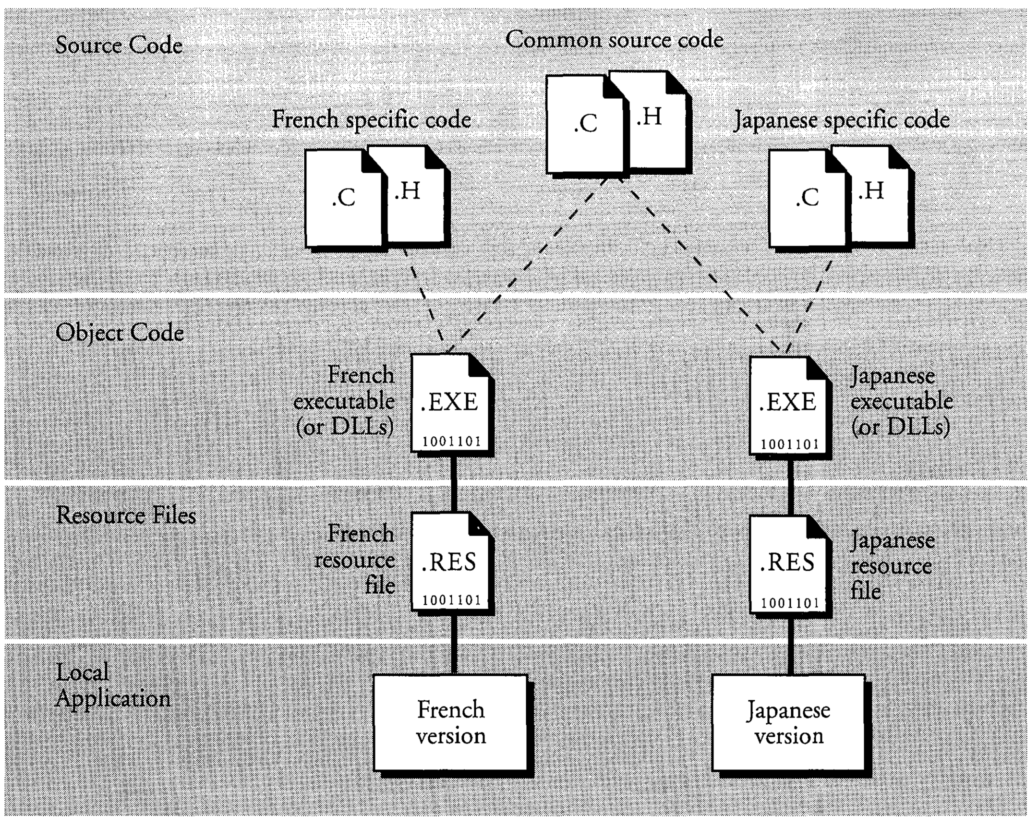
2 Write locale-independent code.

3 Move application components that vary with locale like strings, window layouts, and bitmaps into resource files.

4 Update your project files.

Multiple source files for multiple localizations

FIGURE 14-2



When you cannot maintain a single source code base for all your localizations, keep functionality specific to a locale in a separate file.

Notice that steps 1 and 2 involve producing the block of common code at the top of Figure 14-1. Step 3 creates the resource files in the middle tier, and step 4 brings all the pieces together.

Consider the example of a word processor. A U.S. version of the word processor should provide functionality and an interface tailored for U.S. users. This application would:

- Display, read, and write Roman characters.
- Delimit English words, sentences, and paragraphs based on U.S. English grammatical conventions.
- Sort words based on the order of the English alphabet.

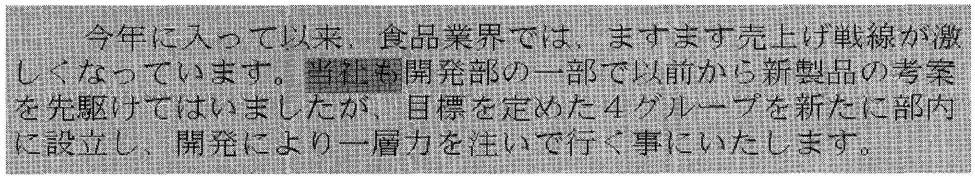
A Japanese version of the word processor, on the other hand, should use an interface tailored to Japanese users, and support Japanese functionality:

- Display, read, and write Japanese characters.
- Delimit Japanese phrases, sentences, and paragraphs based on Japanese grammatical conventions.
- Sort characters based on Japanese sorting conventions.

See Figure 14-3 for a MiniText document that shows how this might look.

Japanese Text in MiniText

FIGURE 14-3



　今年に入って以来、食品業界では、ますます売上げ戦線が激しくなっています。当社も開発部の一部で以前から新製品の考案を先駆けてはいましたが、目標を定めた4グループを新たに部内に設立し、開発により一層力を注いで行く事にいたします。

This screen shows the result of a user double-tapping to select a Japanese phrase. A PenPoint function provides the phrase-selection algorithm.

Again, the goal is to have a single code base for both local versions of your application. This single code base combined with an appropriate resource file yields a localized version of your application. The four general steps you must take to write an international version of your application or service are described in more detail below.

Step 1. Support Unicode

Your application must support multiple character sets such as Roman letters and Japanese characters. In order to maintain a single code base for all local versions of your application, you need a single character coding scheme that handles all the character sets you plan to support. The PenPoint operating system, beginning with version 2.0 Japanese, uses Unicode as its character coding scheme. Your first task is to provide Unicode support in your code.

- The Unicode character encoding scheme is discussed beginning with the section "International character sets" on page 183.
- The process of supporting Unicode is discussed beginning with the section "Supporting Unicode" on page 209.

Step 2. Write locale-independent code

The next step is to make your application behave correctly for a particular locale. Rather than rewriting major parts of your code to perform local functions (sorting, formatting, and filing, for example), write a single block of internationalized source code that behaves as expected in a given locale.

For example, rather than write separate formatting algorithms for English and Japanese dates, you can use a single PenPoint function called **IntlFormatDate**() in your code that appropriately formats English or Japanese dates depending on the locale. PenPoint provides a collection of functions whose behavior changes according to the locale (specified as an argument). These functions are discussed in detail in Chapter 15, PenPoint Support for International Software.

Whenever possible, your application should also use PenPoint-defined objects because these objects behave appropriately for any currently supported locale. For example, an input pad (**clsIP**) in the Japanese version of PenPoint handles Japanese handwriting recognition.

- The PenPoint international functions are discussed in Chapter 15, PenPoint Support for International Software.
- For step-by-step instructions on how to use the international functions, see Chapter 16, Procedures.

Step 3. Use resource files

Next, move application elements that vary with locale into resource files. The strings in your user interface are a good example, though other elements, such as bitmaps, may also belong in resource files.

Having strings and other elements that vary with locale in resource files makes it easier to localize your application.

- Resource files are discussed in the section "Resource files" on page 189.
- The process of moving things to resource files is discussed beginning with the section "Using the DOS utility INTLSCAN" on page 210. The discussion continues in "Moving strings to resource files" on page 216.

Step 4. Create your application

Finally, update your project directory and makefile to create localized versions of your application or service.

Makefiles are discussed in detail in Chapter 29 of *Part 4: PenPoint Development Tools Supplement*. For more details on how to update your makefiles, see "Updating your makefile" on page 224.

For recommendations on how to organize your project files, see "Managing your project" on page 205.

Internationalization checklist

These are the steps you should take to prepare your application for an international market. Don't worry if some of the terms in the checklist are unfamiliar. They will be explained in the section shown in the checklist.

- ☐ Understand the steps involved in the internationalization process by reading this chapter.
- ☐ Use INTLSCAN to help internationalize your code ("Using the DOS utility INTLSCAN" on page 210).
  - ☐ Declare character and string data as 16 bits long ("Supporting Unicode" on page 209).
  - ☐ Use 16-bit routines to handle Unicode characters ("Using the DOS utility INTLSCAN" on page 210).
  - ☐ Make literal strings Unicode strings ("Creating Unicode strings" on page 215).
  - ☐ Move most literal strings to resource files ("Moving strings to resource files" on page 216).
- ☐ Use resource files to store locale-dependent components ("Resource files" on page 189).
  - ☐ Use predefined Application Manager tags "Using predefined AppMgr tags" on page 219).
  - ☐ Use tags in your source code ("Using tags in source code" on page 222).
  - ☐ Use utility functions to read data out of resource files ("Using resource utility functions" on page 220).
- ☐ Write locale-independent code ("Locale-independent code" on page 196).
  - ☐ Take advantage of PenPoint's international functions ("PenPoint's international functions" on page 197).
- ☐ Manage your project files ("Updating your makefile" on page 224).
- ☐ If necessary, handle porting from 1.0 details (Chapter 17, Porting to PenPoint 2.0).
  - ☐ Update your code to reflect new PenPoint APIs ("Changed APIs" on page 229).
  - ☐ Update your gesture handling code ("Gesture handling code" on page 230).
  - ☐ Use the bitmap editor to design special characters ("Special characters" on page 231).
  - ☐ File class version information with your data ("File version data" on page 233).
  - ☐ Maintain a single code base ("Single code base" on page 233).

Chapter 15 / PenPoint Support for International Software

This chapter introduces the messages, functions, and utilities the PenPoint™ operating system provides to support international software. Currently, the PenPoint operating system 2.0 Japanese supports only Japanese and U.S. English. Future releases of PenPoint will support more locales.

International character sets

English text is composed of letters derived from the Roman alphabet. The Roman writing system is the most common in the modern world. Some 70 percent of the world's literate population write or understand a language based on the Roman writing system.

Languages based on Roman letters are relatively simple to represent. Indeed, 8-bit ASCII-based encoding schemes are sufficient to encode most European alphabets, Roman and non-Roman, as well as a large collection of punctuation marks.

Not all countries, of course, use letters from these European alphabets. The Cyrillic and Hebrew alphabets are two familiar examples.

Some languages do not use alphabets at all, or do not use them as the primary building blocks of language. Instead, these languages use **ideographs** (literally "idea symbols") to represent a thing or idea rather than letters to represent words.

Samples of Chinese and Japanese ideographs representing the words for Japanese (left) and document (right).

日本語　書類

Chinese, Japanese, and Korean are the most common of these ideographic writing systems, and these languages are written by almost a billion people in the world today. These character sets contain thousands rather than dozens of characters.

See *Part 3: PenPoint Japanese Localization Handbook* for more details on the Japanese language and its encoding.

Multibyte and wide characters

The problem with large character sets, from a programmer's point of view, is that they are difficult to represent. Clearly, 8 bytes are insufficient because you can represent only 2^8 or 256 characters if you use a one-to-one mapping between **code points** and represented characters. A code point is a number that represents a particular character. For example, the ASCII code point for the letter 'A' is 0x41.

Two possible solutions to this problem are multibyte characters and wide characters.

Multibyte character encoding schemes use one or more bytes to represent a single character. A good example of this scheme is the Japanese Industrial Standards (JIS) encoding of Japanese.

JIS encoding involves two states. In one state, a single byte represents a single ASCII character. In the other state, two bytes represent a single Japanese character. A special sequence of codes shifts a text stream between the two states.

Because many of the world's existing computer devices deal with bytes of information, such a scheme takes advantage of existing byte-sized system designs.

On the other hand, there are drawbacks to multibyte encoding. One of the obvious disadvantages is that you must know what state a given byte is in before you can manipulate it. For example, in an arbitrary stream of JIS text, you cannot be sure if a given byte is a single ASCII character or half a Japanese character without scanning for the "shift-in" and "shift-out" codes.

Because code that processes these multibyte character sets is complex and error-prone, programmers have developed an alternative scheme: using fixed-length codes wide enough to accommodate the required characters.

These wide codes allow more characters to be encoded without ambiguity. Character manipulation code is thus easier to write.

However, no widely adopted standard of wide character encoding has been established. Implementation of wide character sets has been private to a particular company. Though systems could depend on their internal characters being of some fixed length longer than a byte, they could not depend on other systems using the same fixed length in the same way.

Introduction to Unicode

The PenPoint operating system, beginning with version 2.0 Japanese, encodes its character using **Unicode**, a character encoding system that offers advantages of both multibyte and wide character schemes. It uses a wide, 16-bit code to encode each character, regardless of the language to which it belongs. For example, Roman letters and Japanese characters are both 16-bits long.

The Unicode Standard is supported by a nonprofit corporation called the Unicode Consortium. It is made up of companies such as Apple, IBM, HP, DEC, NeXT, and GO.

The uniform 16-bit length frees the programmer from the difficulties of multibyte encoding: a 16-bit Unicode code always represents a single character.

The Unicode standard aims to be comprehensive. Because Unicode characters are uniformly 16-bits long, there are 2^{16} or roughly 65,500 possible characters. Currently, some 34,400 characters and symbols have been assigned as part of Unicode 1.0. According to the Unicode Consortium, these characters are "more than sufficient for modern communication."

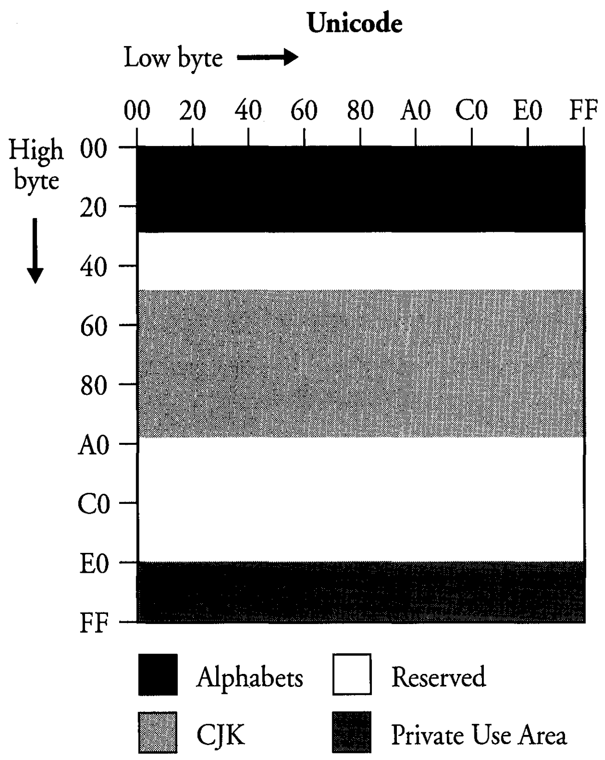
For more information on Unicode, see Chapter 19, Additional Resources, for more details on the two-volume book titled *The Unicode Standard, Version 1.0.*

Minor changes have been made to the Unicode standard since the publication of the two volumes. This revised standard is Unicode 1.0.1, and PenPoint uses this most current version of the Unicode standard.

Unicode architecture

The complete Unicode character set is divided into four major zones, as shown in Figure 15-1.

Unicode architecture FIGURE 15-1



The four zones in Unicode contain the following:

Alphabets contain all alphabets and all other nonideographic script characters, as well as miscellaneous symbols.

CJK contain all Chinese, Japanese, and Korean ideographs.

Reserved is a currently unassigned zone reserved for future use.

Private Use Area contains areas that corporations can use to define their own characters. For example, GO's gesture glyphs are in the private use area. This private area also includes characters retained for compatibility with previous character encoding standards.

The character set is laid out in successive blocks of 256 code points. A Unicode code point is a unique 16-bit number representing a particular character. For example, the code point 0x0041 represents the Latin letter 'A'.

Each block (or group of blocks) of 256 code points forms a linguistic or functional category. For example, there are blocks representing ASCII characters, Cyrillic letters, Arrows, Mathematical Operators, and Chinese, Japanese, and Korean (CJK) ideographs.

Each block is identified by the value of its upper byte. For example, ASCII characters are in block 00, Arabic is in block 06, and Thai is in 0E.

Chinese, Japanese, and Korean ideographs occupy the 76 blocks from hex 40 to 8B, representing a total of approximately 19,500 characters.

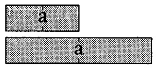
For more details on how Unicode compares with existing double-byte character sets, notably the popular JIS and Shift-JIS used in Japan, please see *The Unicode Standard 1.0* and the *Part 3: PenPoint Japanese Applications Handbook.*

Code supporting Unicode

Providing Unicode support in your code is a straightforward, one-time procedure. Once your code supports Unicode, you never need to rewrite substantial portions of your code to support different character sets.

ASCII-based encoding systems use 8 bits to encode characters while Unicode uses 16 bits. Supporting Unicode requires you to write code that deals with 16-bit rather than 8-bit characters. The PenPoint SDK 2.0 Japanese provides tools to help you make the transition, which impacts the following categories of code:

The letter 'a' encoded as 8-bit (top) and 16-bit (bottom) code points.



- Character types.
- String functions.
- Character and string constants.
- String formatting.

Each of these categories is discussed below. Table 15-1 below gives you a flavor of how the new code will look compared to the old.

How to work with strings TABLE 15-1

Attribute	*8-bit strings*	*16-bit strings*	*Both 8- and 16-bit*
Character types	CHAR8	CHAR16	CHAR
Character/string constants	"John"	L"John"	U_L("John")
String functions	strlen(&aString)	strlen16(&aString)	Ustrlen(&aString)
String formatting	"%hs"	"%ls"	"%s"
Library functions	isupper(aChar)	_uisupper(aChar)	Uisupper(aChar)

The last column is labelled "Both 8- and 16-bit." The code shown in this column works with 8-bit characters in PenPoint 1.0 and 16-bit characters in PenPoint 2.0 Japanese (and beyond).

For example, declaring a variable of type CHAR declares an 8-bit character (CHAR8) character in PenPoint 1.0 and a 16-bit character (CHAR16) in PenPoint 2.0 Japanese. Future releases of the PenPoint operating system will continue to use 16-bit Unicode characters. Use these hybrid functions and types whenever possible.

Most of the hybrid types are defined CTYPE.H. The hybrid functions are defined in the same C header file as the equivalent C function. For example, the **Ustrlen()** function is defined in STRING.H. The **U_L()** macro is defined in INTL.H.

Several procedures in Chapter 16, beginning with "Examples" on page 210, list step-by-step directions for writing code that supports Unicode. Read the following sections for an overview of the process.

The SDK includes a DOS utility called INTLSCAN that flags code that may need to be changed to support Unicode. The utility is on the Goodies disk in the directory \SDK\UTIL\DOS. See "Using the DOS utility INTLSCAN" on page 210 for more information.

Character types

The PenPoint operating system provides three type declarations for character data: CHAR8, CHAR16, and CHAR. The first two declare 8- and 16-bit characters, respectively. The CHAR type is defined for code portability: it is 8 bits wide in PenPoint 1.0 and 16 bits wide in PenPoint 2.0 Japanese.

If you have PenPoint 1.0 code that uses the **char** (lower case) type, convert all of your character data to use the CHAR (upper case) type. You may not need to change declarations of noncharacter data. Use types such as U8 to declare variables of fixed size.

Where noncharacter data depends on the size of CHAR being1 byte, you need to update your code because CHAR is 2 bytes longs in PenPoint 2.0 Japanese.

The DOS utility INTLSCAN, included on the Goodies disk in \SDK\UTIL\DOS, flags lines of code that may need to change to support Unicode. See "Using the DOS utility INTLSCAN" on page 210 for information on how to use INTLSCAN.

String functions

The familiar C string library functions (**strcmp**(), **strcpy**(), and so on) still exist in PenPoint 2.0, but they work only on 8-bit characters. A set of PenPoint macros such as **Ustrlen**() and **Ustrcmp**() allows you to work with 8-bit and 16-bit strings, depending on the PenPoint version. These macros are defined in STRING.H.

In PenPoint 2.0 Japanese, the macros are defined to call new functions provided by the WATCOM C compiler to work with 16-bit data. These functions all have the character **_u** prepended to the equivalent C function name. For example, the header file STRING.H defines prototypes for a set of string functions named **_ustrcmp**(), **_ustrcpy**(), and so on.

These 16-bit functions are defined in the same C header file you would find the equivalent 8-bit C function. Prototypes for **strlen**() and **_ustrlen**(), for example, are both defined in STRING.H.

One note before you replace all your 8-bit functions with the 16-bit or hybrid functions like **Ustrlen**(). Some functions like **isupper**() not only have 16-bit equivalents, but they also have equivalents in the PenPoint international package. In this particular case, the equivalent to **isupper**() is **IntlCharIsUpper**() defined in CHARTYPE.H.

The international functions also work on 16-bit characters or strings, but these functions are more likely to provide behavior appropriate for a particular language. Use these functions, discussed beginning with the section "Locale-independent code" on page 196, whenever you are processing linguistically meaningful text.

In summary, GO recommends that you use the following functions, in order of preference:

- Use the PenPoint international functions such as **IntlStrConvert**() when you are processing text the user sees.
- Use the U...() macros such as **Ustrcmp**() when an international function is unavailable or when you are processing internal data.

- Use the **_u...**() functions such as **_ustrlen**() provided by WATCOM when you are sure your data is 16-bits long.
- Use the C library functions such as **strlen**() when you are sure your data is 8-bits long.

Character and string constants

When you use CHAR8, you can use the standard C conventions for forming character and string constants. For example:

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

When you use the CHAR16 type, you must wrap the L"" modifier around your literal character or strings. This 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 put the character or string constant in the macro U_L().

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

Again, the U_L() macro is a hybrid. In PenPoint 1.0, U_L() tells the compiler to use 8-bit characters; in PenPoint 2.0, it tells the compiler to use 16-bit characters. GO recommends that you use the U_L() macro around all of your literal strings.

The L"" modifier is part of the C language, and the U_L() macro is defined in INTL.H.

You can specify particular Unicode characters in literal strings by typing \x *value* in the string, where *value* is a four-digit hexadecimal number. For example, here are some Quick Help strings from the TextView class:

```
U_L"\xF61F \\tab Pigtail.  Delete a character.\\par "
U_L"\xF60A \xF609 \xF60C \xF60B \\tab Flicks.  Scroll up, down, left, or
right.\\par "
```

This code uses the Unicode value for GO's gesture glyphs to specify them in a literal string. See Table 15-6 for a list of the Unicode value for all of the gesture glyphs.

String formatting

When you use the standard C formatting codes to format strings, make sure you use the correct format code. Note that the **Uprintf**() function requires the U_L() macro wrapped around its format code, as shown below"

```
Uprintf(U_L("%hs"), "I am an 8-bit string.");
Uprintf(U_L("%ls"), L"I am a 16-bit string.");
Uprintf(U_L("%s"), U_L("I can be either kind of string."));
```

Memory and file space

You may be concerned about the additional memory and file space required to support Unicode. Rest assured that your data files will not automatically double their size as a result of supporting Unicode.

Unicode does not demand much more storage space than popular multibyte encoding schemes like Shift-JIS, a standard popular in Japan. Japanese text requires 2 bytes in JIS and Shift-JIS just as it does in Unicode.

Although a Unicode representation of English-only text requires twice the memory space as an ASCII representation, you can compress the data efficiently when writing it to a file. In practice, a compressed Unicode file containing English-only text is less than 1% larger than the identical file stored in ASCII.

Compression affects only the size of your filed data. You will still need 2 bytes per character of memory when processing data.

Compressing Unicode

You can compress Unicode strings with the PenPoint functions **IntlCompress-Unicode()**, defined in \2_0\PENPOINT\SDK\INC\ISR.H. The function implements a compression scheme called **packed Unicode**. This scheme adds 1 byte to every 255 bytes of ASCII data and compresses a typical Shift-JIS file by roughly a quarter. You can compress data before writing it to a file.

You can also buy commercial compression algorithms to compress filed data. Be aware, however, that many commercially available compression algorithms are optimized for 8-bit data, and Unicode is 16 bits long. On the other hand, algorithms like the 16-bit Huffman algorithm that are optimized for 16-bit characters are often memory intensive.

Of course, the data in your application that does not represent text does not require any additional memory.

Resource files

PenPoint resource files store objects and data in a structured way that is isolated from source code. If you are unfamiliar with PenPoint resource files, read Part 11 of the *PenPoint Architectural Reference* for an overview.

You can use resource files to store elements of your application that vary from locale to locale. The most typical example of this is using resource files to store translated user interface strings.

The following list gives examples of when you might use resource files to store elements that differ between localized versions of an application.

- Text for menus, Quick Help, and **StdMsg...()** messages.
- Different, locally appropriate versions of a bitmap representing "stop."
- Two different window layouts for two different locales.
- Flags that your application reads and writes as binary data to save user preferences.

Resource files usually store user interface elements like strings, window layouts, and bitmaps. You can, however, use resource files to store things other than UI elements. The last example, for example, is binary data that influences how your application behaves.

Think of resource files as a place to store modular elements that can be plugged in and out of your application as appropriate to the locale.

GO recommends that you use a strategy for naming resources files to represent the specific localization. All of the PenPoint 2.0 sample code, for example, has a USA.RC file for the American localization and a JPN.RC file for the Japanese localization.

You must use the exact names JPN.RC and USA.RC if your makefile uses the standard makefile rules included with the sample applications (\2_0\PENPOINT\SDK\SAMPLE\SRULES.MIF). The standard makefile rules look for particular strings in USA.RC or JPN.RC to stamp the application directory with PenPoint information.

Resource files existed in PenPoint 1.0, although they were not used extensively in sample code. Resource file architecture in PenPoint 2.0 Japanese is unchanged, and there are additional utilities for working with resources in RESUTIL.H. The resource file architecture supports 16-bit strings.

Strings in resource files

If you have literal strings in your source code, consider moving the strings to resource files. The DOS utility INTLSCAN flags literal strings (as well as lines that may not be appropriate for international applications) in your code. See "Using the DOS utility INTLSCAN" on page 210 for details on how to use INTLSCAN.

While you may not need to move literal strings to resource files for a successful compile, we strongly encourage you to do so. The trade-offs involved in the move are described in the following sections.

Advantages of moving strings to resource files

- Strings in resource files are easier to translate because all the strings are in one place. You can simply pass the resource file to translators, and they can translate the strings without any programming knowledge.
- Applications with strings in resource files are easier to maintain because all user interface strings are in one place rather than scattered throughout various source and header files.
- Having strings in resource files makes it easier to maintain a single code base even if you have many localized versions of your application. Ideally, you can create new localized versions of your application by simply providing new resource files.

Disadvantages of moving strings to resource files

Moving strings to resource files makes your code harder to read. People who want to understand what your code does must follow the tag reference to another file. Some of the sample code included with the SDK, like EmptyApp, leaves strings in the source files for exactly this reason.

GO recommends that you use one resource file to contain all the strings for a particular localization.The resource file name should describe the locale, as in USA.RC and JPN.RC.

Resource file structure

This section describes the recommended structure for resource files. Before we describe the structure, you should understand the following about strings in resource files.

- Each string is associated with a tag that is defined in a header file. You use this tag in your source code when you need to use the string.
- Each string can be part of a group. In the resource file, the entire group is considered a single resource. The four predefined groups for each class are:
  - Toolkit strings.
  - Quick Help strings.
  - Miscellaneous strings (such as format strings for ComposeText functions).
  - Standard Message strings.

  You can define your own groups as needed.
- Each group can have up to four arrays that identify lists of indexed resources. Each array or list is identified as a **well-known list resource ID.** Because each array may contain up to 256 entries, your class can have up to 1,024 tags and corresponding strings just using the predefined groups.

The resource file from the sample Counter Application clearly shows the recommended file structure. You can find this code in \2_0\PENPOINT\SDK\SAMPLE\CNTRAPP.

Creating tags in header files

You must define a tag for each string you want to use. Remember that tags are just 32-bit numbers with a fixed structure. Define these tags in the header file that your source code includes. For example, CNTRAPP.H defines these tags:

```
#define tagCntrMenu     MakeTag(clsCntrApp, 0)
#define tagCntrDec      MakeTag(clsCntrApp, 1)
#define tagCntrOct      MakeTag(clsCntrApp, 2)
#define tagCntrHex      MakeTag(clsCntrApp, 3)
```

When you use resource utility functions from RESUTIL.H to read these strings from a resource file, use these tags when the functions expect a variable of type IX_RES_ID. See "Tags in source code" on page 193 for an example.

You must also define a RES_ID for each group. A RES_ID is a 32-bit number, defined in CLSMGR.H, that identifies a resource. Use a RES_ID to identify a particular resource in a resource file. The header file CNTRAPP.H defines these RES_IDs:

```
#define resCntrTK       MakeListResId (clsCntrApp, resGrpTK, 0)
#define resCntrMisc     MakeListResId (clsCntrApp, resGrpMisc, 0)
```

Defining tags and strings in resource files

You put the literal strings and their associated tags in a resource file. GO recommends that you put U.S. English strings in USA.RC, and Japanese strings in JPN.RC.

The data structure that contains a tag and its corresponding string is an array of structures of type RC_TAGGED_STRING.

```
/****************************************************************************
                         T o o l k i t   S t r i n g s
****************************************************************************/
/*
 * Strings used by toolkit elements in CNTRAPP.  In this case, there are
 * only the Representation menu and its menu items.
 */
static RC_TAGGED_STRING      tkStrings[] = {
      // Representation menu
      tagCntrMenu,      U_L("Representation"),
      // Decimal menu item
      tagCntrDec,       U_L("Dec"),
      // Octal menu item
      tagCntrOct,       U_L("Oct"),
      // Hexagonal menu item
      tagCntrHex,       U_L("Hex"),
      Nil(TAG)
};
```

Notice that the literal strings are surrounded by the **U_L()** macro which indicates the string contains 8-bit character data in PenPoint 1.0 and 16-bit character data in PenPoint 2.0 Japanese.

An RC_INPUT structure immediately follows the RC_TAGGED_STRING array.

```
static RC_INPUT   tk = {
      resCntrTK,
      tkStrings,
      0,
      resTaggedStringArrayResAgent
};
```

The macro **resCntrTK**, defined in CNTRAPP.H, is a 32-bit number that identifies the resource, in this case the group of strings defined in **tkStrings**

```
#define resCntrTK      MakeListResId(clsCntrApp, resGrpTK, 0)
```

The RC_INPUT structure also indicates how the Counter Application should interpret the **tkStrings** array. In this case, the tagged string array resource agent interprets the array. Every group has both of these structures: the RC_TAGGED_STRING structure and the RC_INPUT structure.

Finally, after all the groups have been similarly defined, one more structure of type P_RC_INPUT is required to identify all the groups.

```
P_RC_INPUT  resInput[] = {
      &app,                   // the Application Framework strings
      &tk,                    // the TK strings for CNTRAPP
      &misc,                  // the Misc strings for CNTRAPP
      pNull                   // End of list.
};
```

Note that Counter Application uses only three out of the four standard groups. This is fine. Groups may be left empty except for toolkit strings belonging to the Application Framework. The Application Framework uses those strings to display information about your application to the user.

Look at the sample code provided with the SDK for more examples of resource files. The Goodies disk also contains three files in \SDK\UTIL\TEMPLATE. The files, TEMPLATE.C, TEMPLATE.H, and TEMPLATE.RC, are examples of resource files and source code that uses resources.

Tags in source code

After defining tags in your resource file, you use them in one of three ways.

- Use tags directly if a function or message expects a tag as a parameter. Standard toolkit elements that inherit from **clsTkTable** often expect tags. This code sets up a standard toolkit menu, again in Counter Application.

```
static const TK_TABLE_ENTRY CntrAppMenuBar[] = {
      {tagCntrMenu, 0, 0, 0, tkMenuPullDown | tkLabelStringId, clsMenuButton},
            {tagCntrDec, msgCntrAppChangeFormat, dec, 0, tkLabelStringId},
            {tagCntrOct, msgCntrAppChangeFormat, oct, 0, tkLabelStringId},
            {tagCntrHex, msgCntrAppChangeFormat, hex, 0, tkLabelStringId},
            {pNull},
      {pNull}
};
```

  When you use tags instead of literal strings in a TK_TABLE_ENTRY, you must set (or add, using the bitwise OR operator) the **tkLabelStringId** flag. This flag directs the code to read the required string out of a resource file.

- Use tags instead of literal strings. Many user interface objects that inherit from **clsLabel** allow you to use tags in the place of literal strings. Let the object know that you are supplying a tag rather than a string by setting the **infoType** field of the LABEL_STYLE structure to **lsInfoStringId**. This constant is defined in LABEL.H.

Although this was not described in detail in the 1.0 documentation, you can use tags instead of strings in both PenPoint 1.0 and PenPoint 2.0 Japanese.

- Use resource utility functions to read the required string out of your resource file. A variety of resource utility functions are defined in RESUTIL.H.

```
size = sizeof(resStr) / sizeof(CHAR);
ResUtilGetListString (resStr, size, resGrpMisc, tagCntrMessage);
```

  The **ResUtilGetListString**() function expects a RES_ID to identify the group in which the string is defined; in the example shown here, **resGrpMisc** is the group defined in CNTRAPP.H. The function also expects a IX_RES_ID.

See "Using tags in source code" on page 222 for more detailed instructions and code samples.

Predefined tags

The Application Manager has predefined tags that you use to identify your company, application name, and copyright information.

In PenPoint 1.0, you did this by filling in fields of the APP_MGR_NEW structure. In PenPoint 2.0 Japanese, you must put these strings in a resource file and associate them with the predefined tags defined in APPTAG.H.

The strings defined in this resource file are used in two ways:

- The Application Framework reads these strings from your resource file when it needs to display information about your application to the user.
- Standard makefiles (such as those provided with the sample applications) use the application name and type to stamp your project directory.

This example is from the Counter Application.

```
static RC_TAGGED_STRING appStrings[] = {
      // Default document name
      tagAppMgrAppDefaultDocName,
            U_L("Counter Application"),
      // The company that produced the program.
      tagAppMgrAppCompany,
            U_L("GO Corporation"),
      // The copyright string.
      tagAppMgrAppCopyright,
            U_L("\x00A9 Copyright 1992 by GO Corporation, All Rights Reserved."),
      Nil(TAG)          // end of list marker
};
```

See "Makefiles" in Chapter 29 of *Part 4: PenPoint Development Tools Supplement* for more information on how the standard makefile rules use these tags. As usual, a RC_INPUT structure follows the RC_TAGGED_STRING structure.

```
static RC_INPUT   app = {
      tagAppMgrAppStrings,          // standard resource ID for APP strings
      appStrings,                   // pointer to string array
      0,                            // data length; ignored for string arrays
      resTaggedStringArrayResAgent  // How to interpret the data pointer
};
```

Working with resource files

The PenPoint operating system provides three DOS utilities to work with compiled resource files (for example, USA.RES). With these utilities, you can append (RESAPPND), view (RESDUMP), and delete (RESDEL) resources from a resource file.

For example, here is output of the utility RESDUMP on the Counter Application's resource file USA.RES. The DOS utilities work only on compiled resource files, so the following example shows the entire application being compiled and created in an application directory under \2_0\PENPOINT\APP.

```
C:\>2_0\PENPOINT\SDK\SAMPLES\CNTRAPP> wmake
...
C:\2_0\PENPOINT\SDK\SAMPLES\CNTRAPP> cd \2_0\penpoint\app\cntrapp
C:\2_0\PENPOINT\APP\CNTRAPP> resdump usa.res
DOS/4GW Protected Mode Run-time  Version 1.6
Copyright (c) Rational Systems, Inc. 1990-1992
File Header:
  file key=0100023A
  file format=3
  creator class=[0x0100023A WKN: Scope=Global Admin=285 Ver=1]
  file minimum system version=0
  file end=383
  reserved=00 00 00 00 00 00 00 00 00 00 00 00 00 00
Resource 0 is a well-known data resource
resId = [0x4640008A WKN List: Scope=Global Admin=69 Group=Misc List=0]
Wkn data agent = 8(String Array), data length=162
Min sys version = 0
     0:  05 00 00 01 00 00 00 00-16 00 00 00 27 00 00 00   *............'...*
    16:  62 00 00 00 78 00 00 00-86 00 00 00 00 14 43 6F   *b...x.........Co*
    32:  75 6E 74 65 72 20 41 70-70 6C 69 63 61 74 69 6F   *unter Applicatio*
    48:  6E 00 00 0F 47 4F 20 43-6F 72 70 6F 72 61 74 69   *n...GO Corporati*
    64:  6F 6E 00 00 39 A9 20 43-6F 70 79 72 69 67 68 74   *on..9. Copyright*
    80:  20 31 39 39 32 20 62 79-20 47 4F 20 43 6F 72 70   * 1992 by GO Corp*
    96:  6F 72 61 74 69 6F 6E 2C-20 41 6C 6C 20 52 69 67   *oration, All Rig*
   112:  68 74 73 20 52 65 73 65-72 76 65 64 2E 00 00 14   *hts Reserved....*
   128:  43 6F 75 6E 74 65 72 20-41 70 70 6C 69 63 61 74   *Counter Applicat*
   144:  69 6F 6E 00 00 0C 41 70-70 6C 69 63 61 74 69 6F   *ion...Applicatio*
   160:  6E 00                                             *n.*
Resource 1 is a well-known data resource
resId = [0x40400456 WKN List: Scope=Global Admin=555 Group=ToolKit List=0]
Wkn data agent = 8(String Array), data length=59
Min sys version = 0
     0:  04 00 00 01 00 00 00 00-11 00 00 00 17 00 00 00   *................*
    16:  1D 00 00 00 23 00 00 00-00 0F 52 65 70 72 65 73   *....#.....Repres*
    32:  65 6E 74 61 74 69 6F 6E-00 00 04 44 65 63 00 00   *entation...Dec..*
    48:  04 4F 63 74 00 00 04 48-65 78 00                 *.Oct...Hex.*
Resource 2 is a well-known data resource
resId = [0x46400456 WKN List: Scope=Global Admin=555 Group=Misc List=0]
Wkn data agent = 8(String Array), data length=103
Min sys version = 0
     0:  06 00 00 01 00 00 00 00-03 00 00 00 06 00 00 00   *................*
    16:  09 00 00 00 0C 00 00 00-28 00 00 00 47 00 00 00   *........(...G...*
    32:  00 01 00 00 01 00 00 01-00 00 01 00 00 1A 54 68   *..............Th*
    48:  65 20 63 6F 75 6E 74 65-72 20 76 61 6C 75 65 20   *e counter value *
    64:  69 73 3A 20 5E 31 73 00-00 1D 52 65 70 72 65 73   *is: ^1s...Repres*
    80:  65 6E 74 61 74 69 6F 6E-20 74 79 70 65 20 75 6E   *entation type un*
    96:  6B 6E 6F 77 6E 2E 00                              *known..*
```

Each group defined in CNTRAPP.H is a separate resource with its own RES_ID. Notice that the Application Manager group has a different administered number (**Admin=69**) than the Counter Application's groups (**Admin=555**). Look in CNTRAPP.H to see that 555 is the well-known UID identifying **clsCntrApp**.

```
#define clsCntrAppMakeWKN(555, 1, wknGlobal)
```

See *Part 4: PenPoint Development Tools Supplement* for more information on these DOS utilities.

Locale-independent code

Your application's behavior will likely vary between locales. Formatting, for example, is a behavior that varies between locales. Table 15-2 shows some examples of different formatting conventions.

Formatting differences between countries TABLE 15-2

Attribute	*American English example*	*Different country example*
Number formatting	1,234,567.89	Germany: 1.234.567,89
Time formatting	11:45 p.m.	Italy: h 23,45
Date formatting	3/31/92	Sweden: 92-03-31
Currency formatting	$1995.95	Norway: Kr. 1,995
Address formatting	John Smith Vice-President, Sales Acme Widgets Corporation 123 Industrial Boulevard Providence, RI 02913 U.S.A.	Denmark: Administrerende direktør Acme Corp. Sandtoften 39 DK-2820 Gentofte Danmark
Phone numbers	(416) 325-2061	France: (16) 2.25.20.61
Paper sizes	Letter, 8.5" x 11"	England: A4, 210 cm x 297 cm
Sort order begins	a A b B c C d D e E	Portugal: a A à À á Á â Â ã Ã

The following categories of behavior vary from locale to locale. If your application supports any of these behaviors, make sure local versions of your application implement the behavior appropriately. This list is not comprehensive.

- Formatting conventions
  - Number formatting.
  - Currency handling.
  - Time and date formatting.
  - Numbered items (for example, "3 files").
- Phone number formats.
- Fax dialing formats, cover sheets, and form letters.
- Paper sizes.
- Sorting and comparison rules.
- Word and sentence.
- Linguistic packages.
  - Dictionaries.
  - Heuristics for text processing.
  - Local handwriting translation engines.

Rather than write different code for each country, take advantage of PenPoint's collection of international functions. These international functions behave appropriately for a particular language or country. Using these functions frees you from implementing the locally appropriate version of a function yourself.

In the ideal case, the international functions allow you to write and maintain a single code base no matter how many local versions you create. This single code base would be **locale-independent code.**

Sometimes, you cannot write a single block of code to implement all the local variants your application requires. You have two alternatives in this case:

- You can create different DLLs from different source. You then load a different DLL for each local version of your application. See Figure 14-2 for a diagram of this situation.
- You can write a service to implement a specific function for a locale.

PenPoint's international functions

The PenPoint operating system provides a host of types, data structures, and functions that simplify your task of writing locale-independent code.

Consider a concrete example. Notice from Table 15-2 that Germans write 1.234.567,89 while the Americans prefer 1,234,567.89.

Rather than write your own formatting algorithm, you can simply call a function called **IntlFormatS32()** in your code. The functions accepts, among other arguments, a locale identifier (see "Locales" on page 199 for a discussion of locale identifiers), and returns the correctly formatted string.

Currently, PenPoint supports only U.S. English and Japanese versions of these international functions. Future releases of PenPoint will support more countries, languages, and functionality.

Table 15-3 describes the international functions PenPoint provides. The next section describes the most important functions, most of which are in ISR.H. For details on particular functions, see the on-line header files in \2_0\PENPOINT\SDK\INC.

PenPoint international functions

TABLE 15-3

Header file to include	*Contents*
ISR.H (stands for "International Services and Routines")	Types and functions such as word, sentence, and paragraph delimiting; line break calculation; time, date, number, and currency formatting; sorting and comparison; and Unicode manipulation. These functions deal primarily with strings.
ISRSTYLE.H	Styles that used to control how international functions behave. For example, styles control how to format date and negative numbers, how to sort a list (whether to consider spaces or not), and how to delimit words.
GOLOCALE.H	Constants for country, language, and currency names, as well as names for commonly used strings like days of the week, months of the year, time zones, and units of measurement.
CHARTYPE.H	Types, macros, and functions that work on individual characters. Sample operations include checking for spaces and uppercase letters.
INTL.H	Types and macros used by international functions.

PenPoint international functions TABLE 15-3 (continued)

Header file to include	Contents
GLPYH.H	Macros for Unicode code points, such as GO gesture glyphs, commonly used in PenPoint applications and services.
CMPSTEXT.H	Functions that compose text from strings and variable values, allowing free placement of the parameters throughout the text.

Remember to link the appropriate library with your source code if you use any of these functions. All of the functions described below are defined in INTL.LIB with the exception of **ComposeText**() functions, which are defined in SYSUTIL.LIB.

International functions in ISR.H

Most of PenPoint's international functions are defined in the header file ISR.H. Table 15-4 shows some of the most commonly used functions and their behavior.

Some functions from ISR.H TABLE 15-4

Function	Default behavior
IntlDelimitWord()	Delimits a word (or word-equivalent in languages with no words).
IntlDelimitSentence()	Delimits a sentence.
IntlBreakLine()	Calculates how to break a line of text that cannot fit on a single line.
IntlSecToTimeStruct()	Converts time from seconds since 1970 to international time structure.
IntlIntlTimeToOSDateTime()	Converts from international time structure to system time structure.
IntlFormatS32()	Formats a signed integer with the proper punctuation, as in 1,896.
IntlFormatNumber()	Formats a floating point number with the proper punctuation.
IntlFormatDate()	Formats a date, such as 26-Dec-1991.
IntlFormatTime()	Formats a time, such as 12:45 A.M.
IntlParseS32()	Parses a formatted signed integer, such as (1,592)
IntlParseNumber()	Parses a formatted floating point number, such as 12,572.78
IntlParseDate()	Parses a formatted date, such as 26-Dec-1991.
IntlParseTime()	Parses a formatted time, such as 12:45 A.M.
IntlCompare()	Compares Unicode value of characters.
IntlSort()	Sorts strings.
IntlConvertUnits()	Converts measures in different units, such as feet and meters.
IntlStrConvert()	Converts strings between various formats, such as lower- and upper-case.
IntlMBToUnicode()	Converts multibyte characters to Unicode characters.

Many of the functions come in pairs that reverse each other's functionality:

- The formatting functions such as **IntlFormatDate**() have parsing equivalents such as **IntlParseDate**().
- The conversion functions such as **IntlMBToUnicode**() have functions that reverse the conversion such as **IntlUnicodeToMB**(),

Many of the functions also have counted and uncounted version. Counted versions have the letter **N** in their names. For example, the counted version of

IntlDelimitWord() is **IntlNDelimitWord**(). The uncounted functions work on null-terminated strings. The counted versions work on strings with known length.

Most of the functions require a 32-bit argument that identifies a locale. Locales are explained in the next section.

Locales

This handbook uses the term **locale** rather than country or language because countries vary a great deal within their borders. Canada, Switzerland, and Singapore, for example, are countries that use more than one language. Even within a language, there are distinct variations called dialects. All of these differences influence the localization process.

The PenPoint international functions take these factors into account by introducing a type called LOCALE_ID. This 32-bit number contains three byte-long "fields" that correspond to the language, dialect, and country of a particular locale. Thus, a variable of type LOCALE_ID unique identifies a locale as a 32-bit number.

LOCALE_ID uses only 3 bytes of data. The remaining bits are reserved for future use. Fill those bits with 0s if you do custom manipulation of these identifiers. Usually, just use predefined macros in GOLOCALE.H to manipulate variables of type LOCALE_ID.

The following code uses a macro defined in INTL.H to create locale identifiers for two familiar locales. The arguments are constants defined in GOLOCALE.H. The three arguments correspond to the language, dialect, and country for each locale.

```
#define locUSA    intlLIDMakeLocaleId(ilcEnglish, 0, iccUnitedStates)
#define locJpn    intlLIDMakeLocaleId(ilcJapanese, 0, iccJapan)
```

The types and macros for creating locale identifiers are defined in INTL.H. Languages, dialects, and countries are assigned an 8-bit code and a corresponding mnemonic (like **iccUnitedStates**) in GOLOCALE.H.

Predefined locale identifiers

The PenPoint operating system identifies the current system locale by setting the a LOCALE_ID called **systemLocale**. This initialization is done at boot time, so by the time your application is running, **systemLocale** has been set.

If your application must behave differently in different locales, your code can check the value of **systemLocale** to control its behavior. The Clock sample application, for example, checks the value of **systemLocale** to determine how it should format the time. See "Checking the system locale" on page 227 for a code sample.

Most commonly, though, your application needs to behave appropriately for only a single locale. To accomplish this single-locale behavior, use a series of macros whose names begin with **Loc...**(). For example, use the macro called **LocDelimitWord**() to provide word selection functionality appropriate to PenPoint's current locale. Here is the definition of **LocDelimitWord** in ISR.H:

```
#define LocDelimitWord(tx,s,st)     IntlDelimitWord(tx,s,intlDefaultLocale,st)
```

Notice that the macros simply call the related international function with the predefined locale identifier **intlDefaultLocale** as an argument.

The international functions provide behavior to support Japanese and U.S. English. As shown above, two locale identifiers, **locUSA** and **locJpn**, are defined in GOLOCALE.H. You can send these identifiers as arguments to the international functions.

Styles

Often, even a LOCALE_ID is not enough to specify how a function should behave. There are, for example, at least four different ways to display a date in each Western language.

PenPoint introduces a 32-bit number called a **style** to control how functions should behave within locales. For example, the various styles associated with displaying a date are defined in ISRSTYLE.H as:

```
// Flags used with all Date format styles
#define intlFmtDateSpaceFill       flag16     // Space fill numeric fields
#define intlFmtDateZeroFill        flag17     // Zero fill numeric fields
// International Date format styles
#define intlFmtDateStyleNumeric    0x0001     // e.g. 1/14/92
#define intlFmtDateStyleAbbrv      0x0002     // e.g. 14-JAN-92
#define intlFmtDateStyleShort      0x0003     // e.g. Jan. 14, 1992
#define intlFmtDateStyleFull       0x0004     // e.g. January 14, 1992
```

You use these styles when calling the function **IntlNFormatDate()**. Note that the function expects, among other things, a locale and a style, as parameters:

```
S32 EXPORTED IntlFormatDate(
      P_INTL_TIME pTimeVal,    // Time to format
      P_CHAR      pString,     // Out: converted string
      U32         length,      // Length of buffer
      IX_RES_ID   format,      // Optional explicit format
      LOCALE_ID   locale,      // Locale to use, intlDefaultLocale for default
      U32         style        // Conversion style to use, or styleDefault
);
```

Styles are divided into two halves. The two halves represent major variations (flags) and more subtle variations (styles).

- A flag is a major variation that affects all the functions in a given category. You can specify only one flag at a time.
- A format style is a more subtle variation. You can sometimes use multiple variants simultaneously using the bitwise OR operator. If you specify an unsupported collection of styles, an international function returns the status **stsRequestNotSupported.**

The flag **intlFmtDateSpaceFill** is a good example of a major style. It directs the date formatting function to use spaces as a placeholder in dates, as in 12/ 3/92. Because you can only specify one flag at a time, you cannot specify **intlFmt-DateSpaceFill** and **intlFmtDateZeroFill** at the same time.

Unlike flags, you can specify a collection of format styles. For example, you can specify **intlFmtTimeStyleStandard** and **intlFmtTimeForce24Hour** simultaneously to format a time that looks like 13:57. Use the bitwise OR operator to specify multiple styles simultaneously. For example, define **myStyle** as follows to specify the two styles above:

```
U32   myStyle = intlFmtTimeStyleStandard | intlFmtTimeForce24Hour;
```

Use the predefined style **intlStyleDefault** as a parameter to an international function when you want to use what GO expects to be the most common variation for a given locale. Comments in ISRSTYLE.H identify the default style for a particular locale.

Query capability

Many of the PenPoint international functions require a buffer as an argument. A function that requires a buffer often offers clients a query capability in which the client requests the function to recommend a size for the buffer to pass in. For example, if you pass **pNull** as two of the arguments to the **IntlDelimitWord**() function, the function returns the recommended size of buffer to pass in.

```
U32   size;
U32   style = intlStyleDefault;
size = LocDelimitSentence(pNull, pNull, style);
```

Use the **size** returned by the function to determine how much of your buffer to send when you call the function again. See the procedure on delimiting words in*Part 3: PenPoint Japanese Localization Handbook* for a more detailed code sample.

International function structures

The international functions use three new structures as shown in Table 15-5. All of the structures are defined in ISR.H.

International function structures

TABLE 15-5

Structure name	*Description*
INTL_CNTD_STR	Contains a string and its count. Used by **IntlNSort**() to sort a collection of counted strings.
INTL_TIME	A time structure that is a superset of the standard **tm** structure. It contains two additional members to represent an era (for example, A.D., heisei) and time zone. The **year** member represents the year of an era rather than years since 1900. Valid eras are defined in GOLOCALE.H.
INTL_BREAK_LINE	Contains information on how to break a line, including the position of the break, the characters to delete from the end and start of the line, and the character to insert at the and start of the line.

The new time structure INTL_TIME introduces a new **era** member to accommodate international calendars. Many calendar systems use era information more heavily than the Western Gregorian calendar. For example, the Japanese imperial calendar specifies dates relative to the reign of the current emperor. The year 1992 is represented as heisei 4, the fourth year of the current emperor's reign.

There are international functions to convert between this international time structure and the system time structure OS_DATE_TIME.

The line break structure INTL_BREAK_LINE is used by the **IntlBreakLine**() (and its counted equivalent) to contain information about how a line should break. Different languages uses different rules about how lines should break.

For example, English permits words to break roughly at each syllable. A hyphen is used to indicate that a word continues to the next line. So ***running*** becomes ***run-ning.***

In Japanese, on the other hand, characters simply follow each other sequentially across lines. The only restriction is that certain characters, such as an open parenthesis, cannot end or begin a line.

As another example, when the German word ***backen*** breaks across lines, it becomes ***bak-ken.*** Notice that the trailing ***c*** becomes a trailing ***k.*** The **IntlBreakLine**() function uses the INTL_BREAK_LINE structure to return the necessary line break information. The structure is defined in ISR.H as follows:

This example is given to clarify the structure. The PenPoint 2.0 Japanese version of **IntlBreakLine**() supports only U.S. English and Japanese.

```
typedef struct INTL_BREAK_LINE {
      U32         breakAt;         // position of line break
      U32         deleteThis;      // chars to delete from end of this line
      CHAR  insertThis[intlBreakLineMaxInsert];
                                   // chars to insert at end of this line
      U32         deleteNext;      // chars to delete from start of next line
      CHAR  insertNext[intlBreakLineMaxInsert];
                                   // chars to insert at start of next line
} INTL_BREAK_LINE, *P_INTL_BREAK_LINE;
```

Unicode glyphs

The file GLYPH.H defines mnemonics for the Unicode values of various standard glyphs. Included are PenPoint user interface glyphs, GO gesture glyphs, Unicode control characters, and the Unicode values of PenPoint's standard gestures.

For example, you might use the mnemonics to assign the value of a character.

```
CHAR  myGlyph = glyphCheckMark;
```

After you make this assignment, use the standard drawing context messages to draw the gesture glyph on the screen. See Part 3 of the *Architectural Reference* for more information on drawing PenPoint graphics.

Table 15-6 lists the Unicode values of GO's gesture glyphs. The abbreviation "na" means the gesture glyph was undefined in PenPoint 1.0.

GO's gesture symbols

TABLE 15-6

Gesture tag	*Unicode*	*Code in PenPoint 1.0*	*#define*	*Symbol*
xgs1Tap	F600	46	glyph1Tap	Ỵ
xgs2Tap	F601	128	glyph2Tap	.Ỵ
xgs3Tap	F602	129	glyph3Tap	:Ỵ
xgs4Tap	F603	130	glyph4Tap	∷Ỵ
xgsPressHold	F604	138	glyphPressHold	!
xgsTapHold	F605	137	glyphTapHold	.!
xgs2TapHold	F606	244	glyph2TapHold	
xgs3TapHold	F607	245	glyph3TapHold	
xgs4TapHold	F608	246	glyph4TapHold	
xgsFlickUp	F609	174	glyphFlickUp	|
xgsFlickDown	F60A	175	glyphFlickDown	|
xgsFlickLeft	F60B	176	glyphFlickLeft	—
xgsFlickRight	F60C	177	glyphFlickRight	—
xgsDblFlickUp	F60D	178	glyphDblFlickUp	‖
xgsDblFlickDown	F60E	179	glyphDblFlickDown	‖
xgsDblFlickLeft	F60F	180	glyphDblFlickLeft	=
xgsDblFlickRight	F610	181	glyphDblFlickRight	=
xgsTrplFlickUp	F611	na	glyphTrplFlickUp	|||
xgsTrplFlickDown	F612	na	glyphTrplFlickDown	|||
xgsTrplFlickLeft	F613	na	glyphTrplFlickLeft	≡
xgsTrplFlickRight	F614	189	glyphTrplFlickRight	≡
xgsQuadFlickUp	F615	na	glyphQuadFlickUp	||||
xgsQuadFlickDown	F616	na	glyphQuadFlickDown	||||
xgsQuadFlickLeft	F617	na	glyphQuadFlickLeft	≣
xgsQuadFlickRight	F618	193	glyphQuadFlickRight	≣
xgsVertCounterFlick	F619	200	glyphVertCounterFlick	‖
xgsHorzCounterFlick	F61A	201	glyphHorzCounterFlick	=
xgsPlus '+'	F61B	43	glyphPlus	+
xgsLeftParens	F61C	40	glyphOpenBracket	[
xgsRightParens	F61D	41	glyphCloseBracket	]
xgsCross / xgsXGesture	F61E	88	glyphCross	X
xgsPigtailVert	F61F	141	glyphPigtail	9
xgsScratchOut	F620	140	glyphScratchOut	≈
xgsCircle xgsOGesture	F621	79	glyphCircle	○
xgsCircleTap	F622	142	glyphCircleTap	⊙
xgsCircleLine	F623	146	glyphCircleLine	⊖
xgsCircleFlickUp	F624	202	glyphCircleFlickUp	ძ
xgsCircleFlickDown	F625	203	glyphCircleFlickDown	ϙ
xgsDblCircle	F626	204	glyphDblCircle	○○
xgsCircleCrossOut	F627	207	glyphCircleCross	⊗

GO's gesture symbols TABLE 15-6 (continued)

Gesture tag	*Unicode*	*Code in PenPoint 1.0*	*#define*	*Symbol*
xgsUpCaret	F628	143	glyphCaret	∧
xgsUpCaretDot	F629	95	glyphCaretTap	∧̇
xgsDblUpCaret	F62A	161	glyphDblCaret	⩓
xgsCheck / xgsVGesture	F62B	86	glyphCheck	✓
xgsCheckTap	F62C	136	glyphCheckTap	✓·
xgsUpArrow	F62D	153	glyphUpArrow	↑
xgsDownArrow	F62E	155	glyphDownArrow	↓
xgsLeftArrow	F62F	na	glyphLeftArrow	←
xgsRightArrow	F630	na	glyphRightArrow	→
xgsUp2Arrow	F631	na	glyphUp2Arrow	⇑
xgsDown2Arrow	F632	na	glyphDown2Arrow	⇓
xgsLeft2Arrow	F633	na	glyphLeft2Arrow	⇐
xgsRight2Arrow	F634	na	glyphRight2Arrow	⇒
xgsUpLeft	F635	240	glyphUpLeft	┐
xgsUpRight	F636	173	glyphUpRight	┌
xgsDownLeft	F637	169	glyphDownLeft	┘
xgsDownRight / xgsLGesture	F638	76	glyphDownRight	└
xgsLeftUp	F639	209	glyphLeftUp	└
xgsLeftDown	F63A	210	glyphLeftDown	┌
xgsRightUp	F63B	165	glyphRightUp	┘
xgsRightDown	F63C	167	glyphRightDown	┐
xgsDownLeftFlick	F63E	170	glyphDownLeftFlick	╛
xgsDownRightFlick	F640	168	glyphDownRightFlick	╘
xgsRightUpFlick	F643	166	glyphRightUpFlick	╜
xgsNull	F6FF	255	glyphUnrecognized	

This table shows only some of the names defined in GLYPH.H. See the on-line header file for a complete listing.

Composed strings

Because strings created dynamically differ between locales, you need to be careful composing them. Many of the messages you display to the user are composed dynamically. For example, the file system dynamically composes the message "Delete MYFILE.DOC?" when the user makes the cross-out gesture over the file MYFILE.DOC.

The rules for composing strings differ between locales. Punctuation, word order, and capitalization rules, for example, vary between locales.

Because the familiar C library functions such as **sprintf()** and **printf()** fix the order of their parameters, they are not appropriate for composing strings where word order varies. PenPoint's compose text functions, defined in CMPSTEXT.H, allow you to place parameters where necessary.

The strategy is as follows:

1 Write the message interspersed with placeholders for each of the variables displayed in the message.

2 Place the entire message in a different resource file for each localized version of your application.

3 Read the message out of the resource file when you need to display it.

For example, here is part of the resource file TEMPLATE.RC from the Goodies disk. It shows the English version of a confirmation message. Versions of this message in other languages may put the variables in different places.

```
// Define the warning/informational message resource for EXAMPLE.
static RC_TAGGED_STRING stdMsgWarningStrings[] = {
   . . .
// Confirmation message used with the undo operation.  It allows
// the user to undo the last operation or all operations.
// Buttons: [Undo ^1s]  The last operation is undone
//          [Undo all]  Undo all operations since last checkpoint
//          [Cancel]    Cancels the operation, nothing undone
// Parameters:          ^1s   The type of the last operation (such as DRAW)
//                      ^2s   Name of the picture being worked on
stsExmplConfirmUndo,
U_L("[Undo ^1s] [Undo all] [Cancel] Undo the last operation (^1s) on ^2s?"),
. . .
Nil(TAG)
};
```

There many ComposeText functions that accept literal strings, pointers to format strings, and resource identifiers (RES_ID) as parameters. See the header file \2_0\PENPOINT\SDK\INC\CMPSTEXT.H for details. Remember to ink SYSUTIL.LIB with your source code if you use these functions.

Managing your project

Your project consists of a collection of files that comprise your application. It includes header files, source code, resource files, makefiles, and supporting files like Stationery and help documents. The following sections discuss strategies and tools you use to create localized versions of your application.

See Chapter 29 of *Part 4: PenPoint Development Tools Supplement* for more information on this topic.

Project organization

GO suggests that you keep all your project files in a single directory, including all the different resource files for your various localizations. Notice in the sample code, for example, that every application contains a USA.RC and a JPN.RC file. Each file corresponds to a particular localization.

When you build your application, compile the appropriate resource files and copy the compiled file into your application directory along with the executable image.

The makefiles provided with the sample applications show you how to set up a makefile to coordinate the process of producing different localizations of your application.

Makefiles

The makefiles provided with the sample applications contain a few lines that help create localized versions of your application. The information in this section applies only if you are using the WATCOM WMAKE application to make your application.

First, you can add a LOCALE flag to the command line to make a particular localized version of your application. For example, type one of the following to create the Japanese or American version of your application:

```
wmake LOCALE=jpn
wmake LOCALE=usa
```

Inside the makefile, you can use three resource variables to identify which resource files to include with the executable image:

Makefile variables TABLE 15-7

Variable	*Usage*
RES_FILES	For resource files that are the same for all locales
USA_RES_FILES	For resource files unique to the U.S. localization
JPN_RES_FILES	For resource files unique to the Japanese localization

See "Updating your makefile" on page 224 for details on how to use these makefiles.

Scanning your source code

INTLSCAN.EXE is a DOS utility located on the Goodies disk. It scans source code files and flags lines that may not be appropriate for international applications. The flagged lines fall into one of three categories.

- Code that deals with ASCII. These lines of code usually need to change as follows:
  - Code that performs ASCII (8-bit) manipulation must be changed to code that performs Unicode (16-bit) manipulation.
  - Literal ASCII strings must become literal Unicode strings.
  - Strings, including all the strings users see, should be moved into resource files to facilitate translation.
- Functions that are locale-dependent. Using these locale-dependent functions will make it difficult for you to localize your application. Consider replacing your locale-dependent function with a locale-independent equivalent.
- PenPoint 1.0 code that will no longer work under the latest PenPoint version because of API changes. Calls to the old APIs must be changed to reflect the new APIs.

INTLSCAN searches for particular declarations and function calls in your code. Because it cannot tell what you are doing with a particular variable or function, it may flag a line that does not need to be changed. If you are certain the code INTLSCAN flagged will work in all the locales you plan to market your application, leave it alone.

Conversely, do not assume that because INTLSCAN did not flag any lines in your code that your application is ready for localization. There are many internationalization issues that INTLSCAN cannot possibly detect. For example, INTLSCAN cannot tell you whether a particular piece of your application's functionality is appropriate to a particular locale.

Working with local users and getting familiar with popular local applications may help you understand the needs of a locale.

See "Using the DOS utility INTLSCAN" on page 210 for step-by-step directions on using INTLSCAN.

Other DOS utilities

A new DOS utility called UCONVERT on the Goodies disk converts between various character sets and Unicode. Chapter 24 of *Part 3: PenPoint Japanese Localization Handbook* contains instructions on using Unicode to convert between Shift-JIS, ASCII, and Unicode files.

Other utilities included with the PenPoint SDK 2.0 Japanese help you create localized versions of your application. See Chapter 31 of *Part 4: PenPoint Development Tools Supplement* for details on these utilities.

Missing functions

If your want to maintain a single code base for multiple local versions of your application, you need the international package unless you plan to implement a function not already in PenPoint.

If you do implement a new function, and you think the function you implement would be useful to many developers, contact GO Technical Services with your suggestion.

Chapter 16 / Procedures

This chapter provides step-by-step details on how to write internationalized code. To help you use this chapter more efficiently, each procedure begins with a list of references to prerequisite information and ends with a list of related information. If you have read the previous chapters already, don't worry about the prerequisite information.

The prerequisite information discusses the concepts and motivations for doing a particular procedure. If possible, an example is included with each task.

Supporting Unicode

Read this section if you want to support Unicode in a new application. If you want to add Unicode support to an existing PenPoint 1.0 application, see "Using the DOS utility INTLSCAN" on page 210.

If you followed the suggestions in "Designing for internationalization and localization" in Chapter 5 of *Part 1: PenPoint Application Writing Guide*, your code already handles 16-bit data. We still recommend that you work through this procedure since it provides new details like which string manipulation functions to call.

Prerequisite information

Read the following for an overview of Unicode and the code required to support it:

- "International character sets" on page 183.
- "Multibyte and wide characters" on page 183.
- "Unicode architecture" on page 185.
- "Code supporting Unicode" on page 186.

Procedure

1 Declare character and strings (pointer to characters) as CHAR, which is a 16-bit type in PenPoint 2.0 Japanese.

2 When you process text that a user sees, use the PenPoint international functions such as **IntlCharIsUpper**() and **IntlFormatS32**(). These functions are guaranteed to behave appropriately for the specified locale.

3 When no international functions are available, use the PenPoint macros **U...**() functions **Ustrcpy**() and **Uisupper**() rather than the standard C library functions to manipulate text. These functions work on 16-bit data in PenPoint 2.0 and on 8-bit data in PenPoint 1.0.

4 Use the WATCOM C compiler **_u...**() functions such as **_ustrcpy**() and **_uisupper**() for 16-bit data only. These functions may not be locale-independent, so use the PenPoint international functions whenever you are processing readable text.

5. Wrap the U_L() macro around literal strings, including format strings in the PenPoint U...() functions. See "Creating Unicode strings" on page 215 for details.
6. Do not depend on CHAR being 1 byte long because CHAR is 2 bytes long in PenPoint 2.0 Japanese.
7. Run INTLSCAN to help ensure your code supports Unicode. See "Using the DOS utility INTLSCAN" on page 210 for details.

Examples

See the following sections for code samples:

- "Unicode: 8-bit type—consider CHAR or P_CHAR" on page 212.
- "Unicode: 8-bit function—consider 16-bit replacement" on page 212.
- "Unicode: check mem size for sizeof(CHAR) != 1" on page 213.
- "CHAR8: fixed 8-bit type—are you sure?" on page 215.

Related procedures

- "Using the DOS utility INTLSCAN" on page 210.
- "Interpreting INTLSCAN messages" on page 211.

Using the DOS utility INTLSCAN

This procedure helps you use the DOS utility INTLSCAN.EXE. The utility identifies lines of code that may not be internationalized.

Prerequisite information

Read the following to understand why you should use INTLSCAN.

- "Overview of international software" on page 177.
- "Writing international software" on page 178.
- "International character sets" on page 183.
- "Resource files" on page 189.
- "Locale-independent code" on page 196.
- "Managing your project" on page 205.
- "Scanning your source code" on page 206.

Procedure

1. Copy INTLSCAN.EXE to your \2_0\PENPOINT\SDK\UTIL\DOS directory from the \SDK\UTIL\DOS directory on the Goodies disk.
2. If necessary, use the CONTEXT.BAT batch file to update your PATH variable to include \2_0\PENPOINT\SDK\UTIL\DOS in your DOS path. See *Installing and Running PenPoint SDK 2.0* for more information on the batch file.

```
context 2_0
```

3 Run INTLSCAN on your source (.C) files by typing:
```
intlscan *.C
```

4 List the error files generated by INTLSCAN:
```
dir *.ERR
```

5 Open any .ERR file with a non-zero size. The file contains a list of line numbers and corresponding INTLSCAN messages.

6 Make any necessary changes to your source code. The next procedure "Interpreting INTLSCAN messages" on page 211 shows you how to make the changes INTLSCAN recommends.

7 Repeat steps 4 through 6 for all of your project's header (.H) files.

8 Repeat steps 4 through 6 for all of your project's resource (.RC) files.

Tip *You can choose what kind of code INTLSCAN flags. Type INTLSCAN /H to see which switches are available.*

Tip *Many editors feature a "next error" command that moves you to the next line that needs attention.*

Related information

- "Interpreting INTLSCAN messages" on page 211.
- "Moving strings to resource files" on page 216.
- "Updating your makefile" on page 224.

Interpreting INTLSCAN messages

If INTLSCAN detects a line of code that may need to be changed, it writes the line number and one of the following messages to the file FILENAME.ERR. This section helps you interpret the message and make the recommended changes to your code.

Here is a list of INTLSCAN's messages. Each of these is discussed below.

- Unicode: 8-bit type—consider **CHAR** or **P_CHAR**.
- Unicode: Check mem size for **sizeof(CHAR)** != 1.
- Unicode: 8-bit function—consider 16-bit replacement.
- CHAR8: Fixed 8-bit type—Are you sure?
- Resource: Literal string.
- Resource: Literal character.
- ISR: USA function—consider ISR equivalent.

ISR stands for International Services and Routines.

Prerequisite information

Read the following to understand why INTLSCAN flags certain lines of code.

- "International character sets" on page 183.
- "Code supporting Unicode" on page 186.
- "Locale-independent code" on page 196.
- "Resource files" on page 189.

Procedure

1. Open the *FILENAME*.ERR file generated by INTLSCAN.
2. If your editor supports multiple windows, open your source file, *FILENAME*.C.
3. If appropriate, make the first recommended change. Remember, the changes are only recommendations. Some of the flagged lines may not need to change.
4. If your editor supports a "next error" function, use it to move to the next line INTLSCAN flagged.
5. You may want to run INTLSCAN after finishing your changes to make sure you responded to all of INTLSCAN's recommendations.

Examples

Each of the possible INTLSCAN messages is listed and described below along with old and rewritten code examples. The first sentence of each section describes why INTLSCAN flagged your code.

Unicode: 8-bit type—consider CHAR or P_CHAR

You see this message when your code contains a 8-bit variable with a type such as **U8**, **P_STRING**, or **char**. If appropriate, redeclare these variables as 16-bit strings or characters (using types such as CHAR or P_CHAR). Any text processing should be done in 16 bits. Actual byte-sized data should remain 8-bits long.

Remember that CHAR is 16-bits long in PenPoint 2.0 and 8-bits long in PenPoint 1.0. If you want 16-bit data all the time, use CHAR16.

Here are some examples of old and rewritten code. Old code is on top and grayed out.

```
typedef char    AM_PM_STR[5];
typedef CHAR    AM_PM_STR[5];
```

Strings need to be made of 16-bit characters.

```
P_STRING    tmpDate;
P_CHAR      tmpDate;
```

Pointers to (strings) also need to be 16 bits.

Note that declarations like:

```
U8   fontSize;
```

need not change because **fontSize** is real 8-bit data.

Unicode: 8-bit function—consider 16-bit replacement

You see this message when your code calls a function that works only with 8-bit data. You may want to replace the function with its 16-bit equivalent. Table 16-1 outlines some of your options.

8- and 16-bit functions

TABLE 16-1

If you want...	*Use...*
A specific 8-bit function	A function from the standard C library such as **isupper**().
A function that will work on 8-bit data in PenPoint 1.0 and 16-bit data in PenPoint 2.0.	A PenPoint macro such as **Uisupper**().
A function that works on 16-bit data only.	A WATCOM **_u...**() function such as **_uisupper**().
A function that works on 16-bit data and whose behavior is appropriate to any locale PenPoint supports.	A PenPoint international function such as **IntlCharIs-Upper**()..

If you want to maintain a single code base that compiles under PenPoint 1.0 and PenPoint 2.0, use the **U...**() functions rather than the **_u...**() functions. The **U...**() functions are 8-bit in PenPoint 1.0 and 16-bit in PenPoint 2.0. For more details on maintaining a single code base, see "Single code base" on page 233.

Here are some examples of old and new code:

*Standard C functions like **strlen**() work only on 8-bit arguments. Use **Ustrlen**() instead because it expects 16-bit arguments.*

```
char  s[];
U16   ix;
...
ix = strlen(s);
CHAR  s[];
U16   ix;
...
ix = Ustrlen(s);
```

Literal strings need to be 16 bits.

```
strcat(tmpStr, " ");
Ustrcat(tmpStr, U_L(" "));
```

Notice the last code example contains the **U_L**() macro. This macro, defined in INTL.H, makes the literal string inside a 16-bit string in PenPoint 2.0. In PenPoint 1.0, it allows strings to remain 8-bits long.

Unicode: check mem size for sizeof(CHAR) != 1

You see this message when your code calls a function like **OSHeapBlockAlloc**() that takes a size in bytes. When you call such functions, remember that 16-bit strings require twice the memory of 8-bit strings. Hence, if you depend on CHAR being 1 byte long, multiply your former memory request by the size of CHAR. For example:

```
#define MAX_DT_STR 60
P_CHAR      pBuf;
OSHeapBlockAlloc(osProcessSharedHeapId,
      (SIZEOF)(MAX_DT_STR), &pBuf);
#define MAX_DT_STR 60
P_CHAR     pBuf;
OSHeapBlockAlloc(osProcessSharedHeapId,
      (SIZEOF)(sizeof(CHAR)*MAX_DT_STR), &pBuf);
```

You need to do this multiplication for CHAR types only. You do not need to change the following code because the **SizeOf()** macro correctly computes the size of the structure CLOCK_APP_DATA, taking into account any 16-bit characters or strings it might have.

```
OSHeapBlockAlloc(osProcessHeapId, SizeOf(CLOCK_APP_DATA), &pInst);
```

The **SizeOf()** macro is defined in GO.H.

Resource: literal string

You see this when your code contains a literal quoted string. Consider putting these strings in a resource file, unless the string falls into one of these categories:

- Strings that are meaningful in all languages and in all countries. Universally recognized names like "Disneyland" or "Coca-Cola" might be examples.
- Debugging strings that you display using **Debugf()**.
- Hidden filenames that users will never see.

If you leave the literal string in your code because it falls into one of these three categories, consider wrapping the **U_L()** macro around the string. This makes it a 16-bit string in PenPoint 2.0 Japanese.

Move all other strings to resource files. See "Moving strings to resource files" on page 216 for the procedure.

Here are samples of old and new code that may help you change your code:

```
strcpy((*pData)->token.buf, "Error");
// where token.buf in pData is defined as CHAR[maxDigits]
ResUtilGetListString((*pData)->token.buf, maxDigits, resGrpMisc,
tagCalcAppError);
// Where tagCalcAppError is a tag with a corresponding string in a
// resGrpMisc string array
```

Use **ResUtilGetListString()** to read the string from a resource file into a pre-allocated buffer rather than use a literal string.

```
SYSDC_TEXT_OUTPUT tx;
tx.pText = "Hello";
SYSDC_TEXT_OUTPUT tx;
P_CHAR          helloStr;
helloStr =
      ResUtilAllocListString(osProcessHeapId, resGrpMisc, tagHelloStr);
// where tagHelloStr is a tag with a corresponding string in a
// resGrpMisc string array
// Free the string when you are finished
OSHeapBlockFree(helloStr);
```

Use **ResUtilAllocListString()** when you do not have a pre-allocated buffer, and you do not know how long the string might be. This function allocates a buffer on the heap, into which it loads the requested string from a resource file. Remember to free the string after using it.

Resource: literal character

You see this message when your code contains a literal character. Consider moving the literal character to a resource file, unless you are certain this character is valid in every language and every country in the world (and on all hardware, too). If a character stays literal, consider wrapping the **U_L()** macro around characters so that they are 16-bit.

For example, change this code:

```
#define BACK_SLASH_CHAR '\\'
```

to this:

```
#define BACK_SLASH_CHAR U_L('\\')
```

Typically, you may leave back slashes (in a filename, for example) as literals, but most other characters probably need to go into a resource file.

ISR: USA function—consider ISR equivalent

You see this message when your code uses a locale-dependent function. Unless you are sure this function will work in all the locales you sell this application, replace the function with a locale-independent function from PenPoint's international package.

See "Writing locale-independent code" on page 225 for details. Table 16-3, which lists locale-dependent functions and their locale-independent equivalents, may also be helpful.

CHAR8: fixed 8-bit type—are you sure?

You see this message when your code declares an 8-bit data type by declaring something to be of type CHAR8 or P_CHAR8. Make sure you intend the variable to contain only 8-bit data. If the variable stores the value of a Unicode character, use a 16-bit type like CHAR.

For example,

```
CHAR8 internalString = "private";
```

should be changed to

```
CHAR  internalString = U_L("private");
```

Remember to make sure the user does not see this string. If the user does see the string, it should go in a resource file.

Related information

- "Moving strings to resource files" on page 216.
- "Updating your makefile" on page 224.

Creating Unicode strings

This procedure writes Unicode characters and strings in your code.

Prerequisite information

- "International character sets" on page 183.
- "Code supporting Unicode" on page 186.

Procedure

1 Declare all characters and pointer to characters as a 16-bit type. Use CHAR16 for data that is always 16 bits and CHAR for data that will be 8 bits long in PenPoint 1.0 and 16 bits in later releases.

2 Wrap the U_L() macro around literal characters and strings. Use the L"" modifier only if you are certain your character data will always be 16 bits long.

3 To specify special Unicode characters, use a \x followed by its Unicode value, a 4-digit hexadecimal number.

Examples

These code fragments show examples of the U_L() macro and L"" modifier.

```
Uprintf(U_L("I am 8 bits long in PenPoint 1.0; 16 bits in PenPoint 2.0");
P_CHAR16 pTheString = L"I am always a 16-bit string.";
static RC_TAGGED_STRINGqHelpStrings[] = {
     tagTextView,      U_L("\xF61F \\tab Pigtail.  Delete a character.\\par "),
     Nil(TAG)
};
```

Related procedures

- "Using the DOS utility INTLSCAN" on page 210.
- "Interpreting INTLSCAN messages" on page 211.

Moving strings to resource files

If you read "Designing for internationalization and localization" in Chapter 5 of *Part 1: PenPoint Application Writing Guide,* your code may already have its strings in resource files.

This procedure explains how to put strings in a resource file and use those strings in source code.

Prerequisite information

- *Part 11: Resources* in *PenPoint Architectural Reference.*
- "Resource files" on page 189.
- "Strings in resource files" on page 190.
- "Resource file structure" on page 191.
- "Tags in source code" on page 193.

Procedure

1 Copy a resource file named USA.RC or JPN.RC from one of the sample applications into your project directory. Alternatively, copy the file \SDK\UTIL\TEMPLATE\TEMPLATE.RC from the Goodies disk. For example, type the following:

```
copy \2_0\penpoint\sdk\sample\cntrapp\usa.rc c:\myapp
```

2 Name the resource file to remind you of which localization the file is for: USA.RC and JPN.RC, for example.

3 Identify the file in which you plan to use a particular string, say PROJECT.C.

4 Define tags for each string in the corresponding header file, PROJECT.H. If you want to use an array of strings in a group such as the Toolkit group, you need a RES_ID for each group.

5 Replace the template resource file's strings and tags with your own strings and tags.

6 Modify your implementation in PROJECT.C to use your new tags rather than the literal string. See "Using tags in source code" on page 222 for details.

7 Update your makefile to identify which resources should be included with your application. See "Updating your makefile" on page 224 for more information.

Example

The following code comes from the Counter Application in 2_0\PENPOINT\SDK\SAMPLE\CNTRAPP. It shows the result of moving strings to resource files.

The following tags are defined in the header file CNTRAPP.H. The macro **MakeListResId**() is defined in RESFILE.H, and the macro **MakeTag**() is defined in GO.H. Each group must be identified by a RES_ID created by the **MakeListResID**() macro, and each string must be defined by a TAG created by the **MakeTag**() macro.

```
/* The RES_IDs for the resource lists used with the TAGs.
 */
#define resCntrTK      MakeListResId (clsCntrApp, resGrpTK, 0)
#define resCntrMisc    MakeListResId (clsCntrApp, resGrpMisc, 0)
/*
 * TAGs used to identify toolkit strings.
 */
#define tagCntrMenu    MakeTag(clsCntrApp, 0)
#define tagCntrDec     MakeTag(clsCntrApp, 1)
#define tagCntrOct     MakeTag(clsCntrApp, 2)
#define tagCntrHex     MakeTag(clsCntrApp, 3)
/*
 * TAGs used to identify miscellaneous CNTRAPP strings.
 */
#define tagCntrMessage MakeTag(clsCntrApp, 4)
#define tagCntrUnknown MakeTag(clsCntrApp, 5)
```

The next code fragment comes from the resource file USA.RC. The file uses three groups of strings, Application Framework strings, toolkit strings, and miscellaneous strings. Note that all the literal strings are enveloped in the U_L() macro, making them 16-bit Unicode strings.

```
/******************************************************************************
                       A P P   f r a m e w o r k   s t r i n g s
******************************************************************************/
static RC_TAGGED_STRING     appStrings[] = {
      // Default document name
      tagAppMgrAppDefaultDocName,
            U_L("Counter Application"),
      // The company that produced the program.
      tagAppMgrAppCompany,
            U_L("GO Corporation"),
      // The copyright string.
      tagAppMgrAppCopyright,
            U_L("\x00A9 Copyright 1992 by GO Corporation, All Rights Reserved."),
      // User-visible filename. 32 characters or less.
      tagAppMgrAppFilename,
            U_L("Counter Application"),
    // User-visible file type. 32 characters or less.
    tagAppMgrAppClassName,
        U_L("Application"),
    Nil(TAG)           // end of list marker
};
static RC_INPUT   app = {
    tagAppMgrAppStrings,          // standard resource ID for APP strings
    appStrings,                   // pointer to string array
    0,                            // data length; ignored for string arrays
    resTaggedStringArrayResAgent // How to interpret the data pointer
};
/***********************************************************************
                         T o o l k i t   s t r i n g s
***********************************************************************/
/*
 * Strings used by toolkit elements in CNTRAPP.  In this case, there are
 * only the Representation menu and its menu items.
 */
static RC_TAGGED_STRING      tkStrings[] = {
    // Representation menu
    tagCntrMenu,      U_L("Representation"),
    // Decimal menu item
    tagCntrDec,       U_L("Dec"),
    // Octal menu item
    tagCntrOct,       U_L("Oct"),
    // Hexagonal menu item
    tagCntrHex,       U_L("Hex"),
    Nil(TAG)
};
static RC_INPUT   tk = {
    resCntrTK,
    tkStrings,
    0,
    resTaggedStringArrayResAgent
};
/***********************************************************************
                          M i s c e l l a n e o u s   s t r i n g s
***********************************************************************/
static RC_TAGGED_STRING      miscStrings[] = {
    //
    // Message used to display counter value.  The '^1s' argument allows
    // the code to fill in the appropriate value based on the user's menu
    // choice.
    //
    tagCntrMessage,   U_L("The counter value is: ^1s"),
    //
    // Message indicating an unknown representation type.
    //
    tagCntrUnknown,   U_L("Representation type unknown."),
    Nil(TAG)
};
static RC_INPUT   misc = {
    resCntrMisc,
    miscStrings,
    0,
    resTaggedStringArrayResAgent
};
```

After each of the groups is defined with a RC_TAGGED_STRING and RC_INPUT structure, a P_RC_INPUT structure identifies all the groups. Each of the groups is a separate resource in the resource file.

```
/*****************************************************************************
                                L i s t   o f   r e s o u r c e s
****************************************************************************/
P_RC_INPUT  resInput [] = {
      &app,                  // the Application Framework strings
      &tk,                   // the TK strings for CNTRAPP
      &misc,                 // the Misc strings for CNTRAPP
      pNull                  // End of list.
};
```

Finally, Counter Application's source code needs to use these tags. Here is the code that creates the application's menu bar:

```
static const TK_TABLE_ENTRY CntrAppMenuBar[] = {
      {tagCntrMenu, 0, 0, 0, tkMenuPullDown | tkLabelStringId, clsMenuButton},
            {tagCntrDec, msgCntrAppChangeFormat, dec, 0, tkLabelStringId},
            {tagCntrOct, msgCntrAppChangeFormat, oct, 0, tkLabelStringId},
            {tagCntrHex, msgCntrAppChangeFormat, hex, 0, tkLabelStringId},
            {pNull},
      {pNull}
};
```

Related information

- "Using the DOS utility INTLSCAN" on page 210.
- "Updating your makefile" on page 224.

Using predefined AppMgr tags

The Application Manager has predefined tags that you should use to identify your company, application name, and copyright information. In PenPoint 1.0, you did this by filling in fields of the APP_MGR_NEW structure. You should now put these strings in a resource file and use the new predefined tags that are part of the Application Manager's toolkit group.

Prerequisite information

- "Resource files" on page 189.
- "Strings in resource files" on page 190.
- "Resource file structure" on page 191.
- "Tags in source code" on page 193.
- "Predefined tags" on page 193.

Procedure

1 If you have PenPoint 1.0 code that uses the fields of the APP_MGR_NEW structure to identify your company, application name, and copyright information, remove these lines.

2 Place this information in a resource file appropriate to the localized version. For example, American strings might go in USA.RC, Japanese strings into JPN.RC, and so on.

3 Your strings must be 32 characters or less.

Example

Remove these lines from PenPoint 1.0 code:

```
strcpy(new.appMgr.company, "GO Corporation");
strcpy(new.appMgr.defaultDocName, "Counter Application");
ObjCallRet(msgNew, clsAppMgr, &new, s);
```

and simply call:

```
ObjCallRet(msgNew, clsAppMgr, &new, s)
```

Then put these lines in your resource file:

```
/****************************************************************************
                        A P P   f r a m e w o r k   s t r i n g s
****************************************************************************/
static RC_TAGGED_STRING      appStrings[] = {
      // Default document name
      tagAppMgrAppDefaultDocName,
            U_L("Counter Application"),
      // The company that produced the program.
      tagAppMgrAppCompany,
            U_L("GO Corporation"),
      // The copyright string.
      tagAppMgrAppCopyright,
            U_L("\x00A9 Copyright 1992 by GO Corporation, All Rights Reserved."),
      // User-visible filename. 32 characters or less.
      tagAppMgrAppFilename,
            U_L("Counter Application"),
      // User-visible file type. 32 characters or less.
      tagAppMgrAppClassName,
            U_L("Application"),
      Nil(TAG)         // end of list marker
};
static RC_INPUT   app = {
      tagAppMgrAppStrings,         // standard resource ID for APP strings
      appStrings,                  // pointer to string array
      0,                           // data length; ignored for string arrays
      resTaggedStringArrayResAgent // How to interpret the data pointer
};
```

Related information

- "Moving strings to resource files" on page 216.
- "Updating your makefile" on page 224.

Using resource utility functions

This procedure uses functions defined in RESUTIL.H to read data out of resource files.

Prerequisite information

- "Resource files" on page 189.
- "Resource file structure" on page 191.
- "Tags in source code" on page 193.

Procedure

1 Find code where you use **msgResReadObject** or **msgResReadData** to read data out of a resource file.

2 Call Resource Utility functions as shortcuts to reading objects and strings out of resource files. Table 16-2 lists the available functions.

3 Call one of the first four functions in Table 16-2 to read a single object or string from **theProcessResList**, the application's standard list of resources stored in USA.RES or JPN.RES.

1 Call one of the last three functions in Table 16-2 to read a string from a group. The functions expect you to specify the group (RES_ID) and the string's location (IX_RES_ID) in that group. Use the RES_ID and TAGs you defined in your header file with the **MakeTag**() and **MakeListResID**() macros. See "Strings in resource files" on page 190 for details on strings in groups.

2 In most cases, avoid the load utilities **ResUtilLoadObject**() and **ResUtilLoadListString**() because these fuctions allocate their own memory.

3 Link RESFILE.LIB with your code if you use any of these functions.

Resource utility functions

TABLE 16-2

Function	*Description*
ResUtilLoadObject	Loads an object from **theProcessResList**.
ResUtilLoadString	Loads a string from **theProcessResList** into a buffer or a heap..
ResUtilGetString	Same as **ResUtilLoadString** except that you provide a buffer and its size.
ResUtilAllocString	Loads a string from **theProcessResList** into a heap you specify.
ResUtilLoadListString	Loads an item from a string list in **theProcessResList** into a buffer or a heap. You pass in the desired string's group and its index in that group.
ResUtilGetListString	Loads a string from a string array in **theProcessResList** into a buffer you specify. You pass in the desired string's group and index.
ResUtilAllocListString	Loads a string from a string array in **theProcessResList** into a heap you specify.

Example

Replace this code:

```
#define sampleResId    MakeWknResId(clsSample, 17)
readObj.resId = sampleResId;
readObj.mode = resReadObjectMany;
ObjCallRet(msgNewDefaults, clsObject, &readObj.objectNew, status);
status = ObjCallWarn(msgResReadObject, file, &readObj);
object = readObj.objectNew.uid;
```

with this code:

```
#define sampleResId    MakeWknResId(clsSample, 17)
status = ResUtilLoadObject(sampleResId, &object);
```

Related information

- "Moving strings to resource files" on page 216.
- "Updating your makefile" on page 224.

Using tags in source code

This procedure shows you two ways to use tags that you have defined in resource and header files in your code.

Prerequisite information

- "Resource files" on page 189.
- "Resource file structure" on page 191.
- "Tags in source code" on page 193.

Procedure

You can choose any one of these steps as needed:

1. Use the tag directly when the function or message expects a tag as a parameter.
2. Use a tag in place of a literal string with any UI component that inherits from **clsLabel**. Set the label style to **lsInfoStringId** to let the object know you are using a tag rather than a literal string.
3. Use ResUtil functions to read the required string out of the resource file. Pass the tag as a parameter to the function to let it know which string you want.

Examples

The three examples below show the different ways to use a tag in source code.

In toolkit tables

Code from \2_0\PENPOINT\SDK\SAMPLE\CNTRAPP\CNTRAPP.C uses tags to set up Counter Application's standard toolkit menu.

```
static const TK_TABLE_ENTRY CntrAppMenuBar[] = {
      {tagCntrMenu, 0, 0, 0, tkMenuPullDown | tkLabelStringId, clsMenuButton},
            {tagCntrDec, msgCntrAppChangeFormat, dec, 0, tkLabelStringId},
            {tagCntrOct, msgCntrAppChangeFormat, oct, 0, tkLabelStringId},
            {tagCntrHex, msgCntrAppChangeFormat, hex, 0, tkLabelStringId},
            {pNull},
      {pNull}
};
```

When you use tags instead of strings in a TK_TABLE_ENTRY, you must set the flag **tkLabelStringID** flag. Notice that the bitwise OR operator is used to add the flag to another flag, **tkMenuPullDown**.

In place of a literal string

The toolkit demo sample application in 2_0\PENPOINT\SDK\SAMPLES\ICONS.C shows the use of tags in place of literal strings. Note that **clsIcon** inherits from **clsLabel**.

```
ICON_NEW    in;
. . .
// Set the icon's label
in.label.style.infoType = lsInfoStringId;
in.label.pString = (P_CHAR)tagIconGoLogo;
```

Using resource utility functions

You can also use tags to fetch the required string out of your resource file.

```
size = sizeof(resStr) / sizeof(CHAR);
ResUtilGetListString (resStr, size, resGrpMisc, tagCntrMessage);
```

Note that one of the parameter **ResUtilGetListString**() expects is the group in which the string is defined; in this case, **resGrpMisc**.

Using ComposeText functions

This procedure uses **ComposeText** functions to compose strings while your application is running. These functions are described in CMPSTEXT.H. Remember to link SYSUTIL.LIB if you any of these functions.

Prerequisite information

- "Resource files" on page 189.
- "Strings in resource files" on page 190.
- "Composed strings" on page 204.

Procedure

1 Identify strings that you compose dynamically from variable values and pieces of text.

2 Unless these composed strings are never displayed to the user, move these strings to resource files (if you have not already done so).

3 Include **ComposeText** parameters in each string, making sure you place the parameter in the appropriate place.

4 Call **ComposeText** functions in your source code when you need to create this string.

Example

This code is from the sample Counter Application. First, here is the entry in the resource file:

```
// Message used to display counter value.  The '^1s' argument allows
// the code to fill in the appropriate value based on the user's menu
// choice.
      tagCntrMessage,  U_L("The counter value is: ^1s"),
```

Here is the code that retrieves the string along with formatting information from the resource file. The next block of lines composes the string.

```
P_CHAR      p;
U32         size;
CHAR        buffer[MINSTRLEN];
...
// Retrieve format string from resource file, and construct display
// string from format string and counter value.
size = MAXSTRLEN;
SComposeTextL(&p, &size, pNull, resGrpMisc, tagCntrMessage, buffer);
```

Related information

- "Using resource utility functions" on page 220.
- "Updating your makefile" on page 224.

Updating your makefile

This procedure shows you how to update your makefile to handle multiple resource files. This applies only if you are using the WATCOM WMAKE tool.

Prerequisite information

- "Managing your project" on page 205.
- "Makefiles" on page 206.

Procedure

1. Specify a locale in the command line. If you don't specify a locale, the standard makefile rules assume it is JPN.
2. Specify the resource files needed to build a localized version of your application by setting the RES_FILES, USA_RES_FILES, and JPN_RES_FILES variables in your makefile.
3. Add $(APP_DIR)\$(TARGET_RESFILE) to your "all" line.
4. Set the variable RES_STAMP to yes. This directs the makefile to use the application name and type defined in the resource file USA.RC or JPN.RC (depending on the value of LOCALE).

Example

From the command line, you can type either:

```
wmake LOCALE=usa
wmake LOCALE=jpn
```

to make the appropriate version of your application.

This sample makefile comes from NotePaper App, one of the sample applications included with the SDK. You can find the code in \2_0\PENPOINT\SDK\SAMPLE\NPAPP.

```
#  The .res files for your project.  If you have resources, add
#  $(APP_DIR)\$(TARGET_RESFILE) to the "all" target.
RES_FILES       = bitmap.res
USA_RES_FILES   = usa.res
JPN_RES_FILES   = jpn.res
# Targets
all: $(APP_DIR)\$(PROJ).exe $(APP_DIR)\$(TARGET_RESFILE) .SYMBOLIC
```

Related information

- "Using tags in source code" on page 222.

Writing locale-independent code

This procedure helps make your code general enough to behave appropriately for a given locale. The goal is to maintain a single code base for all local versions of your application.

Prerequisite information

- "Writing international software" on page 178.
- "Locale-independent code" on page 196.
- "Using the DOS utility INTLSCAN" on page 210.
- "Interpreting INTLSCAN messages" on page 211.

Procedure

1. Run INTLSCAN on your source files to identify code that may be locale-dependent by typing:
   ```
   intlscan -r -u PROJECT.C
   ```
   The flags -r and -u force INTLSCAN to suppress messages about Unicode and resource files. See "Using the DOS utility INTLSCAN" on page 210 for details on using the INTLSCAN.
2. Identify other functionality that your application performs that may vary between locales. See "Locale-independent code" on page 196 for a partial listing of functionality categories that tend to vary tremendously between locales.
3. Replace locale-dependent function calls with calls to PenPoint international functions. Table 16-3 lists all the functions INTLSCAN flags and suggests PenPoint replacements.
4. If the required function does not exist in the PenPoint international package, write your own locale-independent code. Usually this means your function or message accepts a locale (and, optionally, a style) as a parameter. If you think the function you are writing would be widely useful to PenPoint developers, contact GO Technical Services with your suggestion.

Example

Table 16-3 lists locale-dependent functions and their suggested replacements from the PenPoint international package. This table should orient you to the problem of locale-dependent code and suggest further areas of code that may be locale-dependent.

Ellipses (...) in the chart indicate that there are a number of related functions with similar names. For example, IntlFormat... means that there are a variety of functions like **IntlFormatS32()**, **IntlFormatDate()**, and so on whose names begin with IntlFormat.

Converting to international functions TABLE 16-3

If you are using ...	*Consider using*	*And #include this*
_asctime	IntlFormatDate/Time	ISR.H
_bprintf	ComposeText	CMPSTEXT.H
_ctime	IntlFormatDate/Time	ISR.H
_gmtime	IntlSecToTimeStruct	ISR.H
_localtime	IntlSecToTimeStruct	ISR.H
_vbprintf	ComposeText/IntlFormat...	CMPSTEXT.H / ISR.H
asctime	IntlFormatDate/Time	ISR.H
atof	IntlParseNumber	ISR.H
atoi	IntlParseS32	ISR.H
atol	IntlParseS32	ISR.H
bsearch	IntlCompare (for compare routine)	ISR.H
ctime	IntlFormatDate/Time	ISR.H
fprintf	ComposeText/IntlFormat...	CMPSTEXT.H / ISR.H
fscanf	IntlParse...	ISR.H
gcvt	IntlFormatNumber	ISR.H
gmtime	IntlSecToTimeStruct	ISR.H
isalnum	IntlCharIsAlphaNumeric	CHARTYPE.H
isalpha	IntlCharIsAlphabetic	CHARTYPE.H
iscntrl	IntlCharIsControl	CHARTYPE.H
isdigit	IntlCharIsDecimalDigit	CHARTYPE.H
isgraph	IntlCharIsGraphic	CHARTYPE.H
islower	IntlCharIsLower	CHARTYPE.H
isprint	IntlCharIsPrinting	CHARTYPE.H
ispunct	IntlCharIsPunctuation	CHARTYPE.H
isspace	IntlCharIsSpace	CHARTYPE.H
isupper	IntlCharIsUpper	CHARTYPE.H
isxdigit	IntlCharIsHexadecimalDigit	CHARTYPE.H
itoa	IntlFormatS32	ISR.H
lfind	IntlCompare (for compare routine)	ISR.H
localtime	IntlSecToTimeStruct	ISR.H
lsearch	IntlCompare (for compare routine)	ISR.H
ltoa	IntlFormatS32	ISR.H

Converting to international functions TABLE 16-3 (continued)

If you are using . . .	*Consider using*	*And #include this*
memicmp	IntlNStrCompare	ISR.H
mktime	IntlTimeStructToSec	ISR.H
printf	ComposeText/IntlFormat...	CMPSTEXT.H / ISR.H
qsort	IntlSort/Compare	ISR.H
scanf	IntlParse...	ISR.H
sprintf	ComposeText/IntlFormat...	ISR.H
sscanf	IntlParse...	ISR.H
strftime	IntlFormatTime	ISR.H
stricmp	IntlStrCompare	ISR.H
strlwr	IntlStrConvert	ISR.H
strnicmp	IntlNStrCompare	ISR.H
strtod	IntlParseS32	ISR.H
strtol	IntlParseS32	ISR.H
strtoul	IntlParseS32 (if possible)	ISR.H
strupr	IntlStrConvert	ISR.H
tolower	IntlStrConvert	ISR.H
toupper	IntlStrConvert	ISR.H
ultoa	IntlFormatS32 (if possible)	ISR.H
utoa	IntlFormatS32 (if possible)	ISR.H
vfprintf	ComposeText/IntlFormat...	CMPSTEXT.H / ISR.H
vfscanf	IntlParse...	ISR.H
vprintf	ComposeText/IntlFormat...	CMPSTEXT.H / ISR.H
vscanf	IntlParse...	ISR.H
vsprintf	ComposeText/IntlFormat...	CMPSTEXT.H / ISR.H
vsscanf	IntlParse...	ISR.H

Checking the system locale

This procedure shows you how to check **systemLocale** to control your application's behavior.

Prerequisite information

- "Locale-independent code" on page 196.
- "Locales" on page 199.
- "Predefined locale identifiers" on page 199.

Procedure

1 Compare the value of **systemLocale** with the locale you are interested in.

2 Write the code to perform the special function.

3 Use the comparison to control whether the code executes.

Example

The following code comes from the Clock sample application. The fragments come from \2_0\PENPOINT\SDK\SAMPLE\CLOCK\CLOCKAPP.C. Here are the relevant type and macro definitions:

```
#define bothUp(timeUp|dateUp)
#define oneRow 4
#define oneCol 8
#define sideBySide 32
#define alarmOnTime 64
#define timeAndDate (bothUp | oneRow | sideBySide | timeFirst)
#define dateOverTime(bothUp | oneCol | alarmOnTime)
...
#define     defaultFmtUSA     timeAndDate
#define     defaultFmtJPN     dateAndTime
#define filedData \
      U8          fmt;
      ...
      BOOLEAN     alarmSnoozeEnable;
typedef struct CLOCK_APP_DATA {
      filedData   // the filed portion of the instance data must come first
      ...
      OBJECT      self;
} CLOCK_APP_DATA, *P_CLOCK_APP_DATA;
```

The code that uses these macros and types checks the system locale and selects an appropriate date format.

```
#ifdef PP1_0
            // Assume a USA locale for PenPoint 1.0
            pInst->fmt       = defaultFmtUSA;
      #else
            {
            // Choose different defaults depending on locale
            SYS_LOCALE          currentLocale;
            currentLocale.pLocaleString=pNull;
            // get the current locale
            ObjCallWarn(msgSysGetLocale, theSystem, &currentLocale);
            if (currentLocale.localeId==locUSA)
                        pInst->fmt=defaultFmtUSA;
            else if (currentLocale.localeId==locJpn)
                        pInst->fmt=defaultFmtJPN;
            else
                        pInst->fmt=defaultFmt;
            }
      #endif
```

Like most of the sample applications, the Clock application has been written to be compiled under both PenPoint 1.0 and PenPoint 2.0 Japanese. The symbol PP1_0 is defined to mark code that is for PenPoint 1.0 only.

To maintain a single source code base, you must use the PenPoint bridging package included with the PenPoint SDK 2.0 Japanese. See "Single code base" on page 233 in Chapter 17 for more information.

Related procedures

- "Writing locale-independent code" on page 225.

Chapter 17 / Porting to PenPoint 2.0

This chapter discusses the changes you need to make if you are porting an existing PenPoint™ 1.0 application to PenPoint 2.0 Japanese. You must perform these four steps:

1 Make changes required by changed PenPoint APIs.

2 Update your gesture handling code.

3 Use bitmaps rather than fonts to display special characters.

4 If you have not already done so, file version information as part of your instance data.

You must also make the changes discussed in the first four chapters of this handbook. For example, your 8-bit character should now be 16-bit data, your strings should be in resource files, and your locale-dependent functions should have been replaced with locale-independent functions.

Changed APIs

This procedure shows you how to update your PenPoint 1.0 code to reflect the new PenPoint 2.0 APIs.

Prerequisite information

Various categories of API changes have been made. Many of the fundamental changes have been discussed in this handbook, such as 8-bit to 16-bit character data. The *PenPoint SDK 2.0 Release Notes* describes most of the general API changes, and the *Part 5: Architectural Reference Supplement* provides more message and structure-level details.

The DOS utility INTLSCAN flags lines of code that use PenPoint 1.0 APIs.

Procedure

1 Run the utility to identify the lines in your code that contain PenPoint 1.0 APIs. See "Using the DOS utility INTLSCAN" on page 210 for details.

2 Replace the PenPoint 1.0 APIs with their updated APIs.

Related information

- "Single code base" on page 233.

Gesture handling code

This procedure updates your PenPoint 1.0 gesture handling code so that it compiles under PenPoint 2.0.

Prerequisite information

In PenPoint 1.0, gestures were encoded as 32-bit numbers. Beginning in 2.0, gestures are encoded as Unicode characters.

The change is simple. The **msg** member of the GWIN_GESTURE date structure has been renamed to **gesture.** So the declaration used to look like:

```
typedef struct GWIN_GESTURE {
      MESSAGE     msg;            // gesture Id
      RECT32      bounds;         // bounding box in LWC
      XY32        hotPoint;       // gesture hot point
      OBJECT      uid;            // object in which the gesture was generated
      U32         reserved;       // reserved for future use
} GWIN_GESTURE, *P_GWIN_GESTURE;
```

and now looks like this:

```
typedef struct GWIN_GESTURE {
      CHAR        gesture;        // gesture Id (Unicode point)
      RECT32      bounds;         // bounding box in LWC
      XY32        hotPoint;       // gesture hot point
      OBJECT      uid;            // object in which the gesture was generated
      U32         reserved[2];    // reserved for future use
} GWIN_GESTURE, *P_GWIN_GESTURE;
```

Procedure

To change your code, simply:

1 Search your .C files for instances of the **msg** member of the GWIN_GESTURE structure.

2 Replace them with references to the gesture member.

3 Anywhere you have declared **msg** to be of type TAG, MESSAGE, or U32, make sure to change the field name to be gesture of type CHAR.

4 Replace the following obsolete code fragments with their newer counterparts. Unless you are maintaining a very old code base, you should not have to worry about this last step.

- Replace **MsgNew(pg->msg)** with **pg->gesture.**
- Replace **TagNum(xgsGestureName)** with *xgsGestureName.*

Examples

This sample comes from the Tic-Tac-Toe application. You can find the code listed here in \2_0\PENPOINT\SDL\SAMPLE\TTT\TTTVIEW.C.

This code is from the **TttViewGesture** message handler. The old version reads:

```
MsgHandlerWithTypes(TttViewGesture, P_GWIN_GESTURE, PP_TTT_VIEW_INST)
{
      STATUS      s;
//    OBJECT      owner;
```

```
    . . .
          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;
    . . .
```

The new code instead looks like this. Notice **pArgs->msg** is now **pArgs->gesture.**

```
    MsgHandlerWithTypes(TttViewGesture, P_GWIN_GESTURE, PP_TTT_VIEW_INST)
    {
          STATUS      s;
    //    OBJECT      owner;

    . . .
          switch(pArgs->gesture) {

                case xgs1Tap:
                      ObjCallJmp(msgTttViewToggleSel, self, pNull, s, Error);
                      break;

                case xgsCross:
                      StsJmp(TttViewGestureSetSquare(self, pArgs, tttX), s, Error);
                      break;

                case xgsCircle:
                      StsJmp(TttViewGestureSetSquare(self, pArgs, tttO), s, Error);
                      break;
    . . .
```

Special characters

PenPoint 2.0 no longer supports the 1.0 font editor. If you used the font editor to design special glyphs to display in your application's user interface, these glyphs will not display under 2.0.

Prerequisite information

You might have designed certain user interface elements with the font editor. For example, you might have designed a special interface that allows your application to control a CD-ROM player. Its buttons are the familiar buttons found on most CD players, and the icons representing play, skip track, and so on are actually special glyphs of a font.

If you need to design special screen elements, use the bitmap editor instead of the font editor.

If you have already created outline fonts with the font editor and need them in your PenPoint 2.0 Japanese applications, contact GO Technical Services to see if your fonts can be translated.

Also contact GO Technical Services if you need a particular Unicode glyph for your application that is currently unsupported. Given enough demand, it is possible that future releases of PenPoint will support the glyph you need.

Procedure

1 Install the bitmap editor as you would any other PenPoint application. It is available in \PENPOINT\APP\BITMAP.

2 Create the special symbols your application needs. Documentation on the bitmap editor is in Chapter 31, Bitmap Editor, of *Part 4: PenPoint Development Tools Supplement.* The bitmap editor saves your bitmap as a resource file with the extension .RES.

3 Use messages from **clsIcon** or **clsBitmap** to read the bitmap out of the resource file and display it on the screen.

Example

This code comes from the toolkit demo sample application. You can find this code in \2_0\PENPOINT\SDK\SAMPLE\TKDEMO\ICONS.C. The first thing to do is create an instance of **clsIcon.**

```
ObjCallRet(msgNewDefaults, clsIcon, &in, s);
in.control.client = app;
in.win.tag = tagIconResource;
in.label.style.infoType = lsInfoStringId;
in.label.pString = (P_CHAR)tagIconResource;
ObjCallRet(msgNew, clsIcon, &in, s);
in.win.parent = parent;
ObjCallRet(msgWinInsert, in.object.uid, &in.win, s);
```

Notice no bitmap is specified here. When it needs the bitmap, **clsIcon** sends the icon's client **msgIconProvideBitmap**. In this case, the client is the application itself. When the application receives this message, it responds by passing the message to its ancestor which provides the bitmap.

When you make TKDEMO, the resource compiler appends the ICON.RES file created by the bitmap editor into either USA.RES or JPN.RES (depending on which localization you are working on). The application class **clsApp** knows how to read the icon our of the compiled resource file, so it responds appropriately to the message **msgIconProvideBitmap**.

Notes

There are several reasons GO requires you to create special symbols with the bitmap editor rather than the font editor.

- Bitmaps can be local to an application, whereas fonts are a global resource available to all applications.
- You can manipulate gray pixels with the bitmap editor.

- **clsIcon** will scale bitmaps with respect to screen resolution and the window layout, while fonts scale mathematically without regard for the surrounding visual context. A 10-pt font scaled 120% is 12 points, regardless of whether this is visually appropriate.

File version data

Remember to file a version number with the instance data of your application. This will make it possible for future versions of your application to read documents created by previous versions of your application. One possible way to file version data is to set aside the first byte of your filed instance data for a version number.

In general, you cannot read documents created by PenPoint 1.0 applications with applications created for PenPoint 2.0 Japanese. This is because many PenPoint objects are filing different data than they did in PenPoint 1.0.

Single code base

GO provides a bridging package that allows you to maintain a single code base that compiles and runs under both PenPoint 1.0 and PenPoint 2.0 Japanese. Your code must be written in a special way and must make use of the header files, makefiles, and library files provided with the bridging package.

See the *PenPoint Bridging Handbook* included with the PenPoint SDK 2.0 Japanese for more details on how to use the bridging package. Most of all, the PenPoint sample applications are specially written to compile and run under both versions of PenPoint. Use these samples as templates for the applications and services you want to create to run under both versions of PenPoint.

Chapter 18 / Localization Guidelines

After you finish internationalizing your application, the only step remaining is to prepare your application for a specific locale.

Remember that your product is much more than code. The released product should include translated documentation, appropriate packaging, a support plan, and other marketing and sales preparation.

The goal of localization is to produce a software product that respects a particular culture's language, customs, and traditions. Though this may seem obvious, a localized software product should behave similarly to applications developed by people native to your target locale.

This handbook does not cover specific details on how to localize to a particular country. However, here are a few guidelines to consider as you begin the localization process:

- Does the application support the local writing system? Your application should read, write, render, process, and receive user input for all the characters needed for communicating in the local writing system. The PenPoint™ operating system provides much, if not all, of the required support. Make sure your application takes advantage of the provided support.
- Does the application respect local text formatting conventions? Numbers, times, dates, currencies, and other text should display as the local user expects.
- Does the application behave as expected? Localized applications, for instance, should sort and compare using locally accepted precedence rules, calculate mortgage and interest payments using local formulas, and select words, sentences, and paragraphs using local grammatical rules.
- Does the application support standards popular in the local computing environment? File and communication standards are particularly important.
- Does the application respect local customs, taboos, and traditions? For example, make sure that any gestures, icons, and strings the application uses are appropriate, meaningful, and nonoffensive.
- Is the user interface graphically pleasing? What one country considers attractive may not be attractive in another country. Japanese characters, for instance, usually require more space than Roman characters. Does your interface make more room elegantly?

- Is the documentation translated in a way local users find informative and appropriate? Japanese users, for instance, tend to read documentation from cover to cover rather than referring to the documentation only when needed. Is your translation appropriate for such reading?
- Is your packaging appropriate to the locale?
- Has your software and documentation been tested by quality assurance personnel as well as local users?

Chapter 19 / Additional Resources

This appendix contains references to resources you may find helpful as you prepare your code for an international market.

Texts

These books may be helpful to you as you plan and design your application. Some are general guidebooks; others provide specific information on particular countries.

PenPoint Application Writing Guide: Expanded Edition GO Corporation, 1992. An introduction to PenPoint programming updated from the original edition to discuss new sample code and other changes to the PenPoint SDK since PenPoint SDK 1.0.

Do's and Taboos Around the World, 2nd ed. Roger Axtell, John Wiley & Sons, 1990. A funny but informative guide to culturally acceptable and unacceptable behavior in various cultures.

Do's and Taboos Roger Axtell, John Wiley & Sons, 1989 Similar to *Do's and Taboos Around the World*, this book is aimed at small businesses. It includes discussions of planning for international markets, pricing, shipping, managing and motivating distributors, and communication. It also includes an entire chapter on Japan.

Symbol Sourcebook Henry Dreyfuss, Van Nostrand Reinhold, 1984. A collection of internationally recognized symbols and icons.

Hoover's Handbook of World Business 1992 The Reference Press, 1991. Includes statistical and descriptive profiles of major countries and companies around the world.

The Unicode Standard 1.0: Worldwide Character Encoding The Unicode Consortium, Addison-Wesley, 1991. The definitive, two-volume book on the Unicode standard, its history and design, implementation help, and common glyphs for all characters defined in Unicode 1.0.

Guide to Macintosh Software Localization Apple Computer, Inc., Addison-Wesley, 1992. Despite its title, this book contains general information that will help developers of any platform internationalize their software.

Digital Guide to Developing International Software Digital Press, 1991. Although aimed at DEC programmers, this practical book contains tables of sort orders, formatting conventions, and other specific data that will help developers localize their products to North American and European markets.

National Language Information and Design Guide, Volumes 1-4, 2nd ed., IBM Canada, 1990. Order nos. SE09-8001-01 through SE09-8004-01. A set of general guidelines and specific details on how to support national languages. Volume 1 is an overview, and volumes 2 through 4 cover technical details on implementing "left-to-right and double-byte character set languages" (vol. 2), Arabic scripts (vol. 3), and Hebrew (vol. 4).

Gestures Desmond Morris, Peter Collett, Peter Marsh, and Marie O'Shaughnessy. Scarborough House, Chelsea, Michigan. A vast collection of appropriate and inappropriate gestures by culture.

The Standard C Library P.J. Plauger, Prentice Hall, 1992. Although this book discusses the entire library, it also discusses the C library functions that deal with multibyte and wide character encoding. See "Large Character Sets for C" by P.J. Plauger in the August 1992 issue of *Dr. Dobb's Journal* for an overview.

Standards organizations

Contact these organizations for more information on their specific standards.

American National Standards Institute (ANSI)
1430 Broadway
New York, NY 10018

Japanese Industrial Standards Committee (JISC)
c/o Standards Department
Agency of Industrial Science and Technology
Ministry of International Trade and Industry
1-3-1, Kasumigaseki
Chiyoda-ku
Tokyo 100
Japan

Unicode Incorporated
c/o Metaphor Computer Systems
1965 Charleston Avenue
Mountain View, CA 94043
Fax: USA 415-71—3714

Part 3 / PenPoint Japanese Localization Handbook

Chapter 20 / Introduction

Japan is an exciting market for PenPoint™ applications. The Japanese localization of the PenPoint 2.0 Japanese operating system provides many building blocks you can use to create high-quality, innovative Japanese applications. These building blocks include:

- A highly accurate handwriting recognition engine.
- An innovative font rendering engine.
- Functions that provide high-level support for Japanese, such as sorting and date formatting and parsing.
- Support for various ways of accepting Japanese input.

This handbook introduces concepts that help you localize your application to Japan. It discusses the changes you may need to make to your code, the development environment, and other issues that may influence the design of your Japanese product.

Intended audience

This handbook assumes the following about its readers:

- You are a developer planning to localize your application or service to Japan.
- You are familiar with PenPoint programming. *Part 1: PenPoint Application Writing Guide* is the best place to start if you are new to PenPoint programming.
- You have code that is ready to localize. Specifically, this handbook assumes that you have applied the procedures described in the *Part 2: PenPoint Internationalization Handbook* to internationalize your code. For example, your application should support Unicode, use resource files to store strings, and contain locale-independent code.

Organization of this handbook

Chapter 20, Introduction, describes the organization of this handbook.

Chapter 21, Japanese Characters, describes the Japanese language from a developer's point of view. It describes the official Japanese character set and how PenPoint 2.0 Japanese represents the character set internally. This chapter includes a discussion of the popular Shift-JIS (Japanese Industrial Standards) character encoding standard and how it compares with Unicode.

Chapter 22, Processing Japanese Text, builds on the previous chapter on Japanese characters and discusses more global issues about processing Japanese text. Topics include formatting conventions, sorting, and other text-related issues.

Chapter 23, Development Environment, describes the PenPoint 2.0 Japanese development environment. It describes the tools, utilities, and sample files that are designed specifically to help you create Japanese applications and services.

Chapter 24, Procedures, gives step-by-step instructions on how to perform common tasks such as creating Shift-JIS strings, supporting kana-kanji conversion, and using Japanese fonts.

Chapter 25, Resources, lists some books that may help you design, translate, and market your Japanese application.

Chapter 26, Japanese Characters, contains a chart that shows the JIS character set and the Unicode values of each character.

Chapter 21 / Japanese Characters

This chapter and Chapter 22, Processing Japanese Text, explain concepts that you should understand when writing a Japanese application. This chapter discusses the Japanese language and how the PenPoint™ operating system encodes Japanese characters. Topics include:

- Overview of Japanese.
    - Kanji.
    - Kana.
    - Romaji.
- Character encoding.
    - The Japanese character set.
    - Half- and full-width variants.
    - Unicode.
- Fonts.
- JIS and Shift-JIS encoding.
    - JIS encoding details.
    - Shift-JIS encoding details.
    - Converting to and from Shift-JIS.
    - Gaiji.

The next chapter discusses more general issues about handling user text input and processing Japanese text. If you are new to the Japanese language and its encoding, we recommend you read these two chapters in order.

Overview of Japanese

The Japanese writing system is among the most complicated in the world. Where most writing systems use fewer than 255 symbols, Japanese uses over 6,000 symbols.

Fortunately, you do not need to write any code to support this complex language. Many PenPoint 2.0 Japanese classes and objects already support Japanese behavior. For example, **clsTextView** can manipulate and display Japanese text in a window.

Use PenPoint 2.0 Japanese classes and objects whenever possible to implement this behavior. See *Part 4: UI Toolkit* and *Part 5: Input and Handwriting Recognition* of the *PenPoint Architectural Reference* for details.

Furthermore, the PenPoint 2.0 Japanese operating system provides a large set of international functions that have been localized to manipulate Japanese characters. For example, the **IntlSort()** function can correctly sort Japanese characters.

Use these international functions whenever available to provide behavior Japanese users expect. See Chapter 22, Processing Japanese Text, for details.

Most languages are written with a single set of symbols. English, for example, uses a single set of characters from a 26-letter alphabet and a collection of numerals and punctuation marks. Japanese writing, in contrast, uses four different sets of symbols called kanji, hiragana, katakana, and romaji. Each of these sets is discussed below. Table 21-1 summarizes the discussion.

Kanji

Kanji is a collection of more than 6,000 characters derived from Chinese. Kanji is the core of Japanese, representing nouns, verbs, adverbs, and adjectives. When a PenPoint 2.0 Japanese term has a good kanji translation, the kanji is used in the user interface. An official list of 6,355 characters, representing more than 99 percent of kanji in common use, has been published by the Japanese Industrial Standards (JIS) organization. See "Character encoding" on page 247 for more details.

Kanji is the most complex of all scripts. Each character is composed of an average of eight strokes.

Document	書類
Cancel	取消
Print	印刷

Kana

Kana are two sets of symbols that represent syllables of spoken Japanese. These sets are called **syllabaries** because each symbol represents a syllable of spoken language. Each syllabary contains 46 basic characters. You can apply vocalization markings to these basic characters to represent a possible total of 104 syllables. These vocalization markings indicate how to pronounce a syllable. Not all of the possible characters are used in practice.

Hiragana is a set of 83 characters used mainly to write inflections. Both verbs and adjectives are inflected in Japanese. Pure hiragana words are rare in computer interfaces. Sometimes, though, you may see hiragana following kanji to form a complete word, as shown in the examples for "Apply" and "Close" in the margin.

Hiragana characters are rounded and composed of two or three strokes.

Apply	適用する
Close	閉じる
Yes	はい

Katakana is a set of 86 characters used mainly to write words borrowed from foreign languages. These borrowed terms are called **loanwords**. For example, the Japanese word for *truck* is written in katakana and pronounced *teruku*; similarly, the word for *baseball* is written and pronounced *besubaru*. A popular Japanese dictionary lists more than 13,000 loanwords. Katakana words, because of their foreign origin, are often used in computer interfaces. The katakana equivalents of PenPoint (*penpointo*), notebook (*noto*), and printer (*purinda*) are shown in the margin.

Katakana characters are more angular than hiragana.

PenPoint	ペンポイント
Notebook	ノート
Printer	プリンタ

Romaji

Romaji is the set of characters of the Latin alphabet. *Ji* means *character* in Japanese, so romaji is literally "roman character." Romaji includes both uppercase and lowercase letters, numerals, and English punctation marks. Japanese uses romaji to represent expressions without turning them into loanwords.

Examples include:
LPT1:
SDK
DOS

The Japanese localization of the PenPoint 2.0 Japanese operating system provides a great deal of support for Japanese language processing. For example, the operating

Japanese writing TABLE 21-1

Script	*Number of characters*	*Typical uses*	*Example*
Kanji	Roughly 6,400	Key concepts that translate well into Japanese	日本語　書類
Hiragana	83 commonly used	Articles Verb and adjective inflections	あなた　はい
Katakana	86 commonly used	Accepted loanwords Plant, animal names Onomatopoeia (bang, click) Telegrams	ペンポイント プリンタ
Romaji	52 letters, 10 numerals, 147 symbols	Foreign words Transliteration of Japanese	2.0 SDK, VGA, DOS

system provides an easy way for developers to encode, display, and recognize Japanese characters. The next few sections discuss character encoding, fonts, handwriting recognition, and conversion to and from existing Japanese encoding standards.

Character encoding

The 7-bit ASCII character encoding scheme is too small to accommodate the thousands of Japanese characters. The most popular encoding system commonly used to encode Japanese in personal computers is called Shift-JIS. See "Shift-JIS encoding details" on page 252 for details on this encoding system.

Because code that processes Shift-JIS text can be quite difficult to write, the PenPoint 2.0 Japanese operating system uses Unicode to encode Japanese characters. The following sections discuss Unicode and how it compares with Shift-JIS.

PenPoint 2.0 Japanese provides simple facilities to work with Japanese encoded characters in either Shift-JIS or Unicode, although your application must process Unicode characters internally.

The Japanese character set

PenPoint 2.0 Japanese supports a standard list of characters published by the JIS organization in 1990. The characters are listed in a document called JIS C 0208-1990 and include the following:

- 6,355 kanji in Level 1 and Level 2.
- 86 katakana characters.
- 83 hiragana characters.
- 10 numerals.
- 52 Roman characters.
- 147 symbols.
- 66 Cyrillic characters.
- 48 Greek characters.
- 32 line elements for making charts.

The kanji are divided into two levels. The Level 1 kanji contains 2,965 of the most commonly used kanji sorted by pronunciation. The Level 2 kanji includes 3,390 less-frequently used characters sorted by **radical**. A radical is the most important part of a kanji, somewhat analogous to a Latin or Greek root word in English. Within each radical, characters are sorted by the number of strokes required to write the character (excluding the radical).

These are examples of radicals.

亅 亠 丿

Together, these levels define 6,355 characters, or more than 99 percent of the kanji in common use.

The 1990 JIS standard derives from two previous JIS standards: one published in 1978 and the other in 1983. New kanji were added and existing characters rearranged in each edition, so that the standards are not strict supersets. Conversion between sets, however, is straightforward.

PenPoint 2.0 Japanese has glyphs for all of the characters in the 1990 character set. See "Fonts" on page 250 for more information.

The handwriting recognition engine that comes with the PenPoint SDK 2.0 Japanese can recognize a large fraction of the characters in the 1990 list. See "Handwriting recognition" on page 255 for details.

Supplemental characters

In 1990, JIS also published a supplemental character list in a document called JIS X 0212-1990. It specifies an additional 5,801 kanji, a collection of 245 Latin-based characters, and 21 miscellaneous symbols and diacritical marks. These characters are called the JIS Supplemental Characters. Because they are not part of the JIS Level 1 or 2 kanji, these characters are sometimes called **gaiji**, literally characters (*ji*) which are outside (*gai*) the standard.

These supplemental characters are rarely used variants of characters primarily used in proper names. The fonts shipped with PenPoint 2.0 Japanese do not contain glyphs for these supplemental characters, although Unicode does assign each character a code point. Thus you can represent any of these supplemental characters internally, but PenPoint 2.0 Japanese cannot display the appropriate glyph.

Because there was no standard way of encoding these characters prior to Unicode, PenPoint 2.0 Japanese files containing these supplemental characters are incompatible. See "Gaiji" on page 254 for information on how PenPoint imports and exports files containing these characters.

Half- and full-width variants

Any katakana character may be half- or full-width. In Japanese, this is translated as **hankaku** (half-width) or **zenkaku** (full-width). PenPoint 2.0 Japanese can represent and display these half- and full-width variants.

Zenkaku (full-width)

ペンポイン
ガガガガガ

Hankaku (half-width)

ﾍﾟﾝﾎﾟｲﾝ
ｶﾞｶﾞｶﾞｶﾞｶﾞ

This width distinction is not an inherent part of the language. Rather, it is a historical convention from the JIS standard. To allow more characters to fit per line, the original JIS standard allowed a variant of the katakana characters to be as wide as a monospaced Roman letter. Because kanji were twice as wide as Roman characters, these katakana variants were called half-width characters.

In PenPoint 2.0 Japanese, a normal Roman character remains roughly half the width of a kanji character. Because Roman characters are often proportional while Japanese kana and kanji are always fixed-width, the comparison is a rough estimate. In addition to these normal-width (hankaku) ASCII characters, PenPoint can also represent and display double-width (zenkaku) ASCII characters.

The double-width Roman characters are monospaced, so they line up evenly with kanji characters. You might use these double-width characters in a title or table that contains mixed kanji and roman characters.

To see these zenkaku and hankaku variants, select some text in a MiniText document and select To Zenkaku or To Hankaku from the Convert menu.

PenPoint 2.0 Japanese provides a function called **IntlStrConvert()** that can convert between the half- and full-width characters. Remember that only katakana and ASCII characters have these half-and full-width variants. See "Converting between character variants" on page 287 for more information.

Unicode

PenPoint 2.0 Japanese uses the 16-bit Unicode encoding standard to represent Japanese characters. Your source code should already support 16-bit Unicode characters. If it does not, see *Part 2: PenPoint Internationalization Handbook* for details on how to support Unicode.

For more information on the Unicode standard, consult the two-volume *Unicode Standard: Version 1.0* and *Part 2: PenPoint Internationalization Handbook.* Unicode encodes over 28,000 characters from the world's scripts.

The Unicode standard assigns all the characters discussed above a unique 16-bit number, sometimes called a **code point**. All the characters specified in the most current 1990 list, the 5,801 supplemental kanji characters, as well as the half- and full-width versions of katakana and Roman alphanumerics are assigned unique Unicode code points.

Thus, your application can represent and manipulate any of these characters internally.

Table 21-2 shows how various Japanese characters are encoded in Unicode. One of the design goals of Unicode was to eliminate redundant coding of characters common to Chinese, Japanese, and Korean (CJK). If all three languages use the same character, that character is assigned a single Unicode value.

The space allotted to these unified characters is labelled **CJK ideographs**. All of the JIS kanji fall into this range. Also, because Chinese, Japanese, and Korean share many punctuation marks, many of the Japanese punctuation marks are encoded as ideographic punctuation.

Unicode encoding of Japanese characters **TABLE 21-2**

Description	*Unicode values (hex)*	*Width*
Romaji (ASCII, Extended Latin)	U+0000→U+03FF	Half, proportional
Ideographic punctuation	U+3000→U+303F	Full, monospaced
Hiragana	U+3040→U+309F	Full, monospaced
Regular-width katakana (zenkaku)	U+30A0→U+30FF	Full, monospaced
CJK ideographs (kanji)	U+4E00→U+9FFF	Full, monospaced
Half-width katakana (hankaku)	U+FF60→U+FF9F	Half, monospaced
Double-width ASCII	U+FF00→U+FF5F	Full, monospaced
Compatibility Zone	U+FE00→U+FFEF	Not applicable
Private Use Zone	U+E000→U+F7FF	Not applicable

Unicode encodes half-width katkana and double-width ASCII in an area called the Compatibility zone. It is called the Compatibility zone because the characters in this zone exist in Unicode solely to be compatible with other character sets like Shift-JIS. Remember that the half- and full-width distinction for katakana is not inherent in Japanese, so these characters would not need code points if they did not exist in the JIS standard.

Because files created on Japanese computers may contain characters outside of the official JIS list, PenPoint 2.0 Japanese must map them to some location in the Unicode code space. The Unicode Private Use Zone is used for this purpose. See "Gaiji" on page 254 for details.

Fonts

The PenPoint operating system 2.0 Japanese currently provides two Japanese fonts, Heisei Mincho and Heisei Gothic. Use the Mincho font in roughly the same way you use a Roman serif font, and use Gothic as you would a Roman sans-serif font. Note that all kanji and kana are monospaced.

Mincho
明朝 日本語

Gothic
ゴシック 日本語

The Mincho and Heisei fonts contain glyphs for JIS levels 1 and 2 kanji as well as all of the other JIS C 0208-1990 characters. This includes the hankaku and zenkaku versions of ASCII and katakana characters, but does not include the supplemental characters.

The default system font, used by the system and text applications, is 12-point Mincho. The default user font used in fields is 12-point Gothic.

Users can set either of these defaults to Gothic, Roman, Sans Serif, or Mincho in the Preferences section of the Settings notebook. If Roman is the chosen default font, Japanese characters appear in Mincho. If Sans Serif is chosen, Japanese characters appear in Gothic.

Again, the standard fonts do not contain glyphs for any of the 5,801 supplemental kanji. So while your application can represent internally any Unicode character, the only kanji that appear on the screen are JIS levels 1 and 2 characters.

If your application tries to display one of the 5,801 JIS Supplemental Characters, it will appear as a **hex quad**. A hex quad is a collection of four hex numbers that represent a single 16-bit code. The first (high) byte is on top, and the second (low) byte is on the bottom. The first example in the margin represents the hexadecimal number 0x001B.

Hex quads

```
00  F1  00
1B  F2  12
```

The fonts are divided into several files, as shown in Table 21-3.

Japanese font files

TABLE 21-3

Font file	*Size in kilobytes*	*Contents*
MC55.FDB	873	Mincho, JIS Level 1
MC80.FDB	1,101	Mincho, JIS Level 2
MC81.FDB	10	Mincho, half-width (hankaku)
GT55.FDB	712	Gothic, JIS Level 1
GT80.FDB	878	Gothic, JIS Level 2
GT81.FDB	7	Gothic, half-width (hankaku)

JIS and Shift-JIS encoding

JIS and Shift-JIS are two popular character encoding schemes used by current Japanese computer systems. Think of JIS encoding as the standard on larger computers and Shift-JIS as the personal computer standard. For example, IBM DOS J5.0/V and KanjiTalk, the Japanese version of the Macintosh operating system, use the Shift-JIS encoding standard.

Do not confuse the JIS encoding standard with the JIS character list. The JIS encoding standard maps characters in the JIS character list to a particular code point.

Both JIS and Shift-JIS are multibyte encoding systems. That is, both use two bytes to represent Japanese characters. The only exception is a hankaku character, the half-width version of katakana. Each hankaku character requires one byte.

Both schemes also use a single byte to represent ASCII characters. This allows a text file to mix ASCII and Japanese characters.

JIS encoding details

JIS encoding overlaps with the printable ASCII characters; that is, its codes fall between decimal 33 and 126 (hex 21 through 5F). ASCII codes still represent ASCII characters, and each Japanese character is represented as a sequence of 2 byte-long ASCII codes. Hankaku characters are represented by a single byte between 0xA1 and 0xDF.

To distinguish a single-byte ASCII character from a double-byte Japanese character, applications must search for a **shift state**. The shift state indicates whether a given text stream is in one-byte-per-character mode (ASCII) or two-bytes-per-character mode (Japanese).

A shift is indicated by a particular escape sequence like ESC $ @ (hex 1B 24 40). "Shift out" marks the beginning of a series of double-byte JIS characters, while "shift in" marks the return to single-byte ASCII characters. Different shift states are used for each different character set (1978, 1983, 1990).

The shift state can make text-processing code quite complex. If the application needs to process text in the middle of a sentence or page, for example, the code may be required to read backwards to determine the state.

Because the JIS encoding system is not widely used by Japanese personal computers, PenPoint 2.0 Japanese does not provide any support of the JIS encoding. If necessary, convert any JIS files to Shift-JIS before importing them into PenPoint 2.0 Japanese.

Shift-JIS encoding details

Shift-JIS, sometimes abbreviated XJIS, is a variation of JIS encoding used widely by Japanese personal computers. It eliminates state information by shifting the code of the first byte of a Japanese character to above hex 80.

The second byte falls between decimal 64 and 126 (hex 40 and 7E). This range contains both printing and nonprinting ASCII characters. So while the first byte of a Shift-JIS character cannot be confused with a standard 7-bit ASCII character, the second byte can be. As in the JIS encoding, hankaku characters are represented by a single byte between 0xA1 and 0xDF.

Although the ASCII standard itself is only 7 bits, most vendors use the high ASCII characters above hex 80 for special characters. For example, IBM uses codes above hex 80 for line drawing elements, European alphabets, and other glyphs. Thus even the first byte of a Shift-JIS character overlaps with codes that are previously assigned code points.

This overlap makes processing text difficult even without explicit state information embedded in the text stream.

For example, say your code encounters a character with code value below hex 80. It might be an ASCII character, but it might also be the second byte of a Japanese character. You can check the code of the previous character, but this check does not always resolve the ambiguity.

If the previous character is above hex 80, it can still be the first or second byte of a Japanese character. To determine the state of the current character, your code must scan through the stream backwards until two sequential ASCII characters appear. This algorithm is complex, error-prone, and computationally expensive.

Character set code spaces

Figure 21-1 shows what codes the different character encoding systems occupy. Each of the two-dimensional charts shows the high byte along the left edge and the low byte along the top edge. Notice that the original JIS encoding completely overlaps with 7-bit ASCII; all bytes fall between hex 20 and 80.

Although Shift-JIS solves this overlap problem for 7-bit ASCII, most 8-bit ASCII code points still overlap with Shift-JIS code points.

Unicode code points are shown on the left side. The four labelled zones contain the following characters:

Alphabets contains alphabets, syllabaries, and symbols.

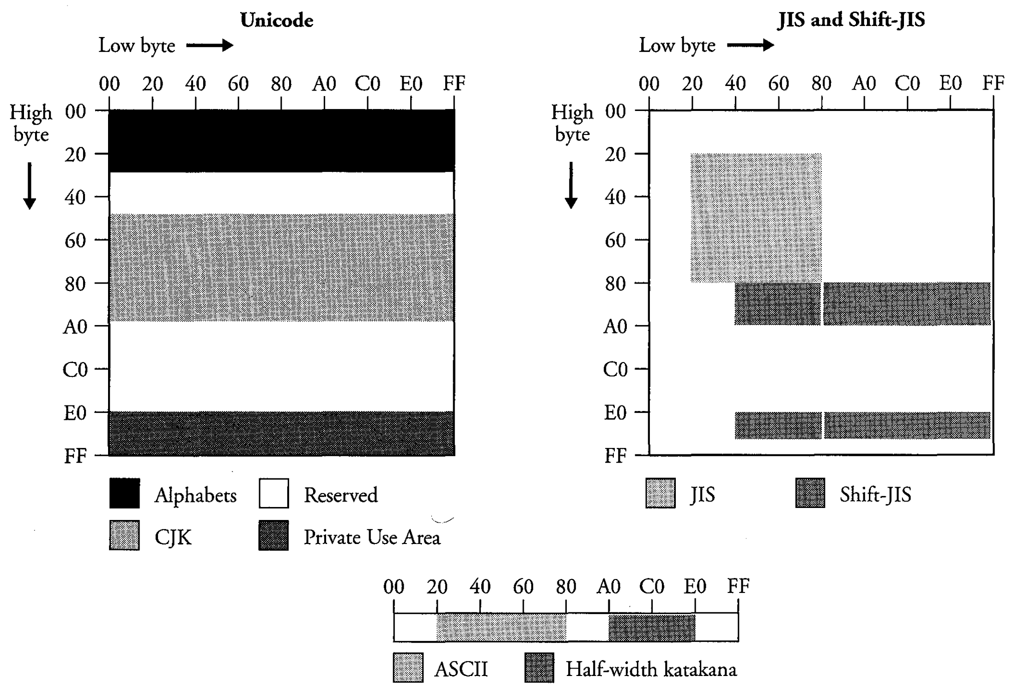
CJK contains Chinese, Japanese, and Korean characters, including all the JIS Level 1, Level 2, and supplemental kanji.

Private Use area contains compatibility zone characters and characters for private, corporate use. GO's gesture glyphs are in the corporate use zone. The hankaku, katakana, and zenkaku ASCII characters are the in compatibility zone.

Reserved area is reserved by the Unicode Consortium for future use.

Character code spaces

FIGURE 21-1



Shift-JIS
High byte: 81-9F, E0-EF
Low byte: 40-7E, 80-FC
Zenkaku: A0-E0

JIS
High byte: 21-7E
Low byte: 21-7E
Zenkaku: A0-E0

Unicode
See Table 21-2 for the ranges of Unicode code points.

Converting to and from Shift-JIS

PenPoint 2.0 Japanese provides various conversion facilities between Shift-JIS and Unicode:

- Convert Shift-JIS files to and from Unicode with the DOS utility UCONVERT. This utility is included with the SDK in \2_0\SDK\UTIL\DOS\UCONVERT. See "Converting Unicode and Shift-JIS files" on page 285 for details on using the utility.
- Directly import Shift-JIS files into MiniText. See "Working with Shift-JIS in text files" on page 283 for details.
- Use the functions **IntlMBToUnicode**() and **IntlUnicodeToMB**() to translate text programmatically. The default behavior of this function in PenPoint 2.0 Japanese converts between Unicode and the 1990 Shift-JIS encoding.
- Use **clsText** messages **msgTextRead** and **msgTextWrite** to read and write Shift-JIS strings. These messages are documented in TXTDATA.H. Specify **fileTypeASCII** as the **format**. Because Shift-JIS uses 8-bit characters, **file-TypeASCII** works for both Shift-JIS and 8-bit extensions to ASCII. File types are defined in FILETYPE.H.

Gaiji

Gaiji literally means characters (*ji*) that are outside (*gai*) of the standard. There are thousands of characters that are not included in JIS levels 1 or 2, many of which are rarely used characters or rare forms of characters used in proper names.

Many of these characters have been defined as part of the 5,801 supplementary kanji added to JIS in 1990. Unicode assigns each of these characters a unique 16-bit code.

Before Unicode, however, implementation of these gaiji varied tremendously. Consequently, files are often incompatible between applications and computer systems. For example, the AX Consortium, NEC, and Fujitsu each support mutually incompatible gaiji encoding schemes.

When you import a file containing gaiji encoded by one of these three schemes, PenPoint 2.0 Japanese automatically maps the characters into parts of the Unicode Private Use Area. The characters are displayed as hex quads because the fonts shipped with PenPoint 2.0 Japanese do not contain glyphs for gaiji. When you export the documents, all the gaiji characters are mapped to their original values.

Unicode sets aside an area called the Private Use Area to use as a repository for private codes. The area lies between U+E000 and U+F7FF. See Unicode Version 1.0, Volume 2 for details.

Note that if the computer from which you are importing does not use the same gaiji mapping as the computer to which you are exporting, the gaiji are not mapped correctly. In other words, PenPoint 2.0 Japanese does not translate between different gaiji encodings.

Chapter 22 / Processing Japanese Text

This chapter discusses how Japanese text is typically processed and how your application can use the PenPoint™ operating system's support for high-level text processing. Topics include:

- Japanese text entry.
  - Handwriting recognition.
  - Kana-kanji conversion.
  - Romaji-kanji conversion.
  - Supporting KKC and RKC.
  - Using keyboards.
- Handling Japanese text.
  - Delimiting words.
  - Delimiting sentences.
  - Comparing and sorting.
  - Converting between Shift-JIS and Unicode.
  - Compressing Unicode.
- Formatting Japanese text.
  - Line breaks.
  - Dates.
  - Times.
  - Numbers.

Japanese text entry

Using a keyboard to enter Japanese kanji is a cumbersome and time-consuming process. One of the most exciting features of PenPoint 2.0 Japanese is Japanese handwriting recognition.

With PenPoint 2.0 Japanese, users can simply write Japanese characters on their PenPoint machine and the handwriting recognition engine translates the characters into a machine-readable form. You do not have to write any code to support this feature.

Handwriting recognition

The handwriting engine shipped with PenPoint 2.0 Japanese recognizes all of the JIS kana, romaji, and almost all of the JIS levels 1 and 2 kanji. See "Character recognition" on page 256 for more details on which characters the handwriting engine recognizes.

Here are a few tips that help the handwriting recognizer achieve higher accuracy. You might mention these tips in your user documentation:

- Use the correct stroke order. Each Japanese character has a standard stroke order. Although the engine recognizes popular variations on the stroke order, recognition is better with the standard stroke order.
- Print neatly. Highly curved and joined strokes take more time to recognize.
- Keep radicals separate. Many Japanese characters are composed of two or more radicals. Do not overlap them when writing.
- Do not add extra strokes. The engine tolerates missing strokes but not additional strokes.
- Experiment with simplified forms of a character. The engine recognizes traditional as well as some of the simplified forms of a character.

Character recognition

The handwriting engine recognizes all of the JIS Level 1 kanji (2,965) and roughly 2,900 of the 3,390 Level 2 kanji. The unrecognized characters fall into three categories:

- Radicals that are not complete characters in themselves, such as 亅 亠 丿.
- Rarely used characters, such as 鮖 鯒 鯱.
- Rare variants of a character whose common style is recognized, such as 鵞 剱.

Users can enter characters not recognized by the handwriting engine in one of three ways:

- Kana-kanji conversion (KKC) allows users to "spell" the kanji character in either hiragana or katakana. The user then converts the kana sequence into a kanji character. See "Kana-kanji conversion" on page 257 for more details on KKC.
- Romaji-kanji conversion (RKC) allows users to type in the English romanization for a kanji character. PenPoint 2.0 Japanese uses the Hepburn system of romanization. See "Romaji-kanji conversion" on page 258 for more information.
- The Unicode Browser, a PenPoint accessory, allows users to enter these characters from a collection of pop-up lists. See "Unicode Browser" on page 279 for details. The document *New UI Features in PenPoint 2.0* shows you how to use the Unicode Browser.

Because the fonts shipped with PenPoint 2.0 Japanese contain glyphs for all Level 1 and 2 characters, your application can display these characters even though the handwriting recognition engine cannot recognize them. The limitation discussed here applies only to the character recognition engine.

Punctuation recognition

The handwriting engine recognizes the following Japanese punctuation marks and symbols. ASCII punctuation marks are used primarily with romaji, although there is some overlap. Japanese, for example, uses the English question mark.

See the *Unicode Standard, Volume 1,* pages 332 through 338, for representative glyphs.

Japanese punctuation marks

TABLE 22-1

Unicode value	Unicode name	Use
U+3002	Ideographic period	Denotes end of sentence.
U+3001	Ideographic comma	Indicates pause, clarifies sentence structure.
U+30FB	Katakana middle dot	Separates loanwords that may be unfamiliar to the reader.
U+30FD	Katakana iteration mark	Indicates that the previous katakana character should be repeated.
U+30FE	Katakana voiced interaction mark	Indicates that the previous katakana character should be repeated as a voiced character.
U+309D	Hiragana iteration mark	Indicates that the previous hiragana character should be repeated.
U+309E	Hiragana voiced interation mark	Indicates that the previous hiragana character should be repeated as a voiced character.
U+3003	Ditto mark	Indicates above line should be repeated.
U+3004	Ideographic ditto mark	Used like a ditto mark to indicate the line above should be repeated.
U+3005	Ideographic iteration mark	Indicates previous kanji should be repeated.
U+3006	Ideographic closing mark	Indicates a deadline (for example, to mail in tax forms).
U+3007	Ideographic number 0	Denotes the number 0, commonly seen on business cards.
U+30FC	Katakana-hiragana prolonged sound mark	Used to indicate that the previous kana sound should be elongated.
U+300C	Opening corner bracket	Used to start a quotation.
U+300D	Closing corner bracket	Used to end a quotation.
U+3012	Postal mark	Indicates Japanese postal code, analogous to U.S. zip codes.

Kana-kanji conversion

The typical method of entering Japanese with a personal computer is called **kana-kanji conversion** (KKC). The approach is as follows.

The user types kana with a Japanese keyboard. The user then presses a special key to convert a sequence of kana to a single kanji character.

Japanese has many homophones, words that sound alike. Consequently, a single sequence of kana specifies a number of possible kanji. After the user presses the convert key, a list of possible matches appears, and the user then selects the desired character.

PenPoint 2.0 Japanese supports this method of entering kanji in addition to the direct handwriting recognition discussed above. Users can type or write kana characters, and then initiate KKC by pressing a special key (the space bar on American keyboards and a dedicated KKC key on Japanese keyboards) or by using the right up ⅃ gesture.

The easiest way to provide KKC support in your application is to use a PenPoint 2.0 Japanese object that implements the behavior. Instances of **clsIP** or **clsField** automatically support KKC without any additional code.

To provide KKC support with your own custom objects, read the protocol described below in "Supporting KKC and RKC." Also see "Supporting kana-kanji conversion" on page 296 for a code sample.

Romaji-kanji conversion

A process similar to KKC called **romaji-kanji conversion** (RKC) allows users to enter Japanese characters by typing English letters. The letters are first translated into kana, which then undergo KKC to specify a list of possible kanji.

For example, if the user types the word *nihongo* and hits the convert key, the insertion pad replaces *nihongo* with Japanese characters.

nihongo= 日本語

Users can use an attached keyboard or PenPoint's virtual keyboard to type Japanese characters. The space bar initiates RKC on the English keyboard. The Japanese keyboard has a dedicated conversion key, as well as extra keys for scrolling through alternatives and reversing the conversion.

See "Using keyboards" on page 261 for tips on using the keyboards to type Japanese.

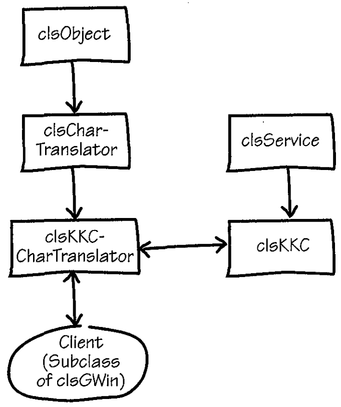
Supporting KKC and RKC

The easiest way to support KKC and RKC is to create an instance of **clsIP** or **clsField** because these objects already support both character conversions. In general, only sophisticated text-processing applications, such as word processors, need create their own classes to handle KKC and RKC.

If you create your own class to support KKC and RKC, it should follow the protocol described below. Before we describe the protocol, you should know about three new PenPoint 2.0 Japanese classes.

The first new class, called **clsCharTranslator**, is an intermediary between clients that want to support character translation and services that provide character translation functionality. Because **clsCharTranslator** is an abstract class, its descendant **clsKKCT** serves as the actual intermediary.

Both these classes receive messages from the client (often via **clsGwin**, as described below), and then request services from **clsKKC**. Because **clsKKC** is a descendant of **clsService**, it provides APIs for requesting services to perform actual character translations. This architecture permits you to replace the translation engine provided with PenPoint 2.0 Japanese with your own engine.



Because the character translator requests gesture information, its clients are almost always subclasses of **clsGWin**. Every instance of **clsGWin** creates a character translator (during **msgInit**) to which it sends translation requests.

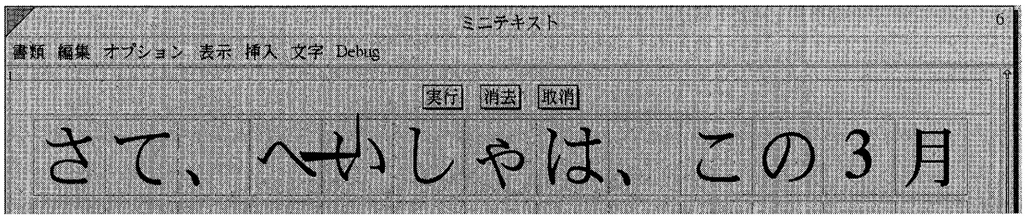
You can specify which translator **clsGWin** sends the message to by filling in the LOCALE_ID field of GWIN_NEW_ONLY. If you do not specify a translator, **clsGWin** creates a translator appropriate to the system locale. The default translator for Japan (**locJpn**) is an instance of **clsKKCT**.

Here is an example of the protocol in action, described as **clsIP** implements it:

1 The user writes a few kana characters in an insertion pad, then requests KKC with the right up ⅃ gesture, as shown in Figure 22-1. When the pad receives a gesture, it self-sends the message **msgCharTransGesture**.

Handling the KKC gesture

FIGURE 22-1



Send **msgCharTrans-Gesture** when the user makes a gesture.

Respond to **msgCharTrans-GetClientBuffer** by sending the requested portions of your text buffer.

2 Rather than handling the message itself, **clsIP** allows the message to be handled by **clsGWin**. In turn, **clsGWin** sends the message to the character translator it created as part of its response to **msgInit**. Again, for PenPoint 2.0 Japanese, the default translator is an instance of **clsKKCT**.

3 When the character translator (an instance of **clsKKCT**) receives the gesture information it determines if the gesture is relevant to character translation. Since the right-up gesture explicitly requests KKC, it sends the **msgCharTrans-GetClientBuffer** to the client (**clsIP**) requesting a portion of its buffer.

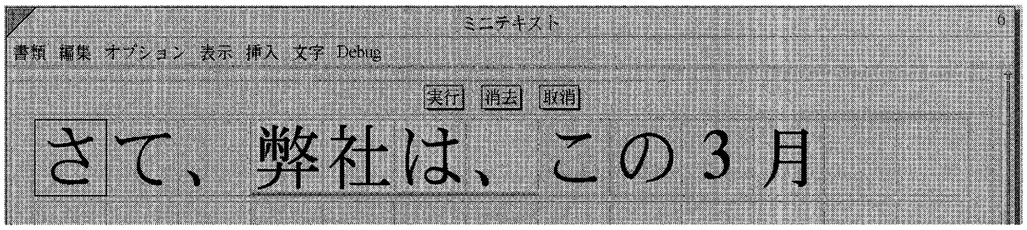
4 The client sends the requested characters in response to **msgCharTransGet-ClientBuffer.**

5 The translator communicates with **clsKKC**, the front-end to the actual service that provides KKC. In this case, a translation is needed, so the translator sends **msgCharTransModifyBuffer** with the translation to the client.

6 Using information sent with **msgCharTransModifyBuffer**, the insertion pad updates its internal buffer and user interface to display the translated character. Note that in the result, shown in Figure 22-2, the translated characters are highlighted. The arguments sent with **msgCharTransModifyBuffer** contain information on which characters to highlight. See *Part 6: PenPoint User Interface Design Reference Supplement* for details on how character highlighting should behave during KKC.

Displaying the translated characters

FIGURE 22-2



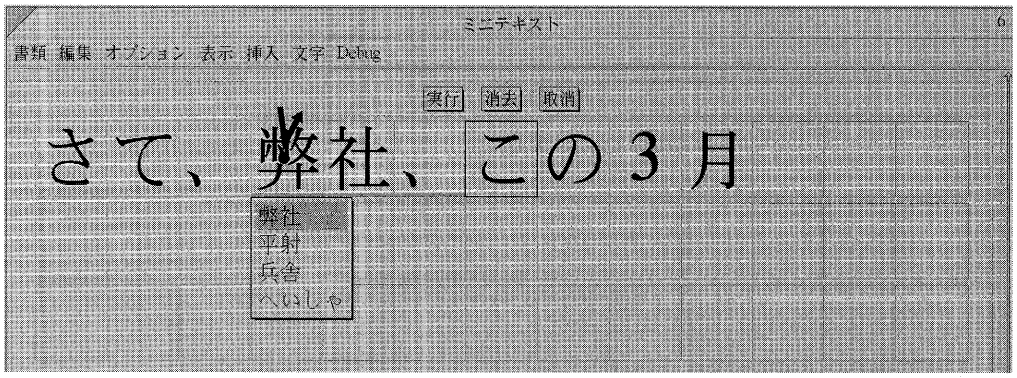
Handle **msgCharTrans-ModifyBuffer** to display the result of character translation.

7 The user then requests a list of alternatives by tapping on the highlighted character. The insertion pad self-sends **msgCharTransGesture**, again allowing the message to be handled by **clsGWin**.

8 The translator receives the message from **clsGWin** and queries **clsKKC** for character alternatives. It also asks the client where the character alternatives pop-up box should be placed by sending **msgCharTransProvideListXY**. The insertion pad calculates the coordinates of the upper-left corner of the pop-up box. The pop-up box should appear directly below the original character.

Handling a character alternatives request

FIGURE 22-3



Self-send **msgCharTransGesture** to notify the character translator of the user's tap.

Handle **msgCharTransProvideListXY** to let the character translator calculate where to place the character alternatives list.

Handle **msgCharTransModifyBuffer** to update your buffer with the user's choice.

9 If the user selects an alternative from the pop-up box, the translator sends **msgCharTransModifyBuffer** to the insertion pad. The insertion pad then updates its buffer and user interface.

10 When the user taps OK to dismiss insertion pad, the pad self-sends **msgCharTransGoQuiescent** to reset the translator in preparation for the next character translation request.

The description above does not exhaust the messages involved in the character translation protocol. For example, it did not mention any of the messages involved for supporting keyboard input. The following paragraphs describe the most important messages involved in the protocol.

The client should self-send the following four messages when appropriate. However, the client should not define a method to handle the message. Rather, the client should allow the message to be passed up to **clsGWin**.

1 Self-send **msgCharTransKey** each time the user presses a key.

2 Self-send **msgCharTransChar** each time the user edits an existing buffer (for example, when the user inserts or deletes a character). Normally, you need not send this message as the user writes a new character. See step 4 below for handling this case.

3 Self-send **msgCharTransGoQuiescent** to cancel the current translation. When the user taps outside an insertion pad, for example, **clsIP** self-sends **msgCharTransGoQuiescent**.

4 Self-send **msgCharTransGesture** each time the user makes a gesture on your text.

The client should respond to the following messages sent by the character translator:

msgCharTransModifyBuffer, which contains information on how to translate characters. The client should respond by updating its text buffer and user interface, including updating strong and weak highlighting. The character translator sends the client a CHAR_TRANS_MODIFY structure containing all the relevant information.

msgCharTransGetClientBuffer, which asks the client for some text from its buffer. Pass the requested text to the character translator as part of a CHAR_TRANS_GET_BUF structure.

msgCharTransProvideListXY, which asks the client where to put the character alternative list. The client should compute root window coordinates so that the pop-up box appears below the original character.

See "Supporting kana-kanji conversion" on page 296 for more details and a code sample.

Using keyboards

The PenPoint operating system 2.0 Japanese supports a number of keyboards including:

- IBM Japanese A01.
- IBM U.S. keyboard (IBM AT).
- Toshiba laptop keyboards (Toshiba Dynabook 386/20).
- Toshiba desktop keyboards (Toshiba J3100ZS).
- AX Consortium keyboard (Okidata 486 VX530)

Set the **Keyboard** variable in MIL.INI to identify your keyboard. Valid values are shown in MIL.INI.

Here are some tips when using the American keyboard:

- The keyboard has two modes: One lets you type English characters, the other Japanese characters. Press Ctrl-Shift-L to toggle between modes.
- If you are having problems toggling modes, cancel the insertion pad, press Ctrl-Shift-L, and then open another pad.
- Press the space bar to initiate KKC or RKC.
- In Japanese mode, alphabetic keys map to hiragana. Hold down the Shift key to enter katakana.
- Use the up and down arrows to scroll through the character alternatives pop-up box.

The Japanese keyboard has dedicated keys to initiate character conversion, scroll through character alternatives, and adjust the current selection.

The virtual keyboard included as a PenPoint 2.0 Japanese accessory emulates both American and Japanese keyboards. Bring up the keyboard from the Accessories notebook and make the check gesture ✓ on the keyboard title bar to select an emulation mode. See "Japanese virtual keyboard" on page 279 for more information.

Handling Japanese text

PenPoint 2.0 Japanese provides a collection of international functions to perform tasks like formatting dates and times, sorting, and word and paragraph selection. Because the desired behavior of these functions varies widely between locales, the international functions accept an argument that identifies a locale. The value of this argument determines the function's behavior.

Remember that in this context, a locale identifies a country, a language, and an optional dialect. The default locale in PenPoint 2.0 Japanese is Japan, which is defined as the 32-bit locale identifier **locJpn** in GOLOCALE.H.

See Table 22-2 for a summary of the default behavior of the most important international functions for Japan. The rest of this chapter provides more details by describing how Japanese is typically processed. Topics include line breaking, selecting words, sorting, and more.

See *Part 2: PenPoint Internationalization Handbook* for general information on these international functions and locales. Most of the international functions are defined in \2_0\PENPOINT\SDK\INC\ISR.H.

Chapter 24, Procedures, describes how to use PenPoint's international functions to give your applications the behavior described here.

If your application needs to provide appropriate behavior in just the default locale **locJpn**, use the **Loc...()** macros rather than the **Intl...()** functions. For example, here is the definition of **LocDelimitWord()** from ISR.H. Calling **LocDelimitWord()** in PenPoint 2.0 Japanese delimits the Japanese equivalent of a word.

```
#define LocDelimitWord(tx,s,st)     IntlDelimitWord(tx,s,intlDefaultLocale,st)
```

Notice that it calls the equivalent international function, sending **intlDefaultLocale** as an argument.

Japanese behavior of international functions

TABLE 22-2

Function	Default behavior
IntlDelimitWord()	Delimits a bunsetsu.
IntlDelimitSentence()	Delimits a sentence ended by an ideographic period or other punctuation mark.
IntlBreakLine()	Prevents taboo characters from beginning or ending a line.
IntlSecToTimeStruct()	Converts time since 1970 from seconds to the Imperial calendar system.
IntlTimeStructToSec()	Converts from the Imperial calendar system to seconds since 1970.
IntlFormatS32()	Adds thousands separators and a minus sign, as in –1,234,567.
IntlFormatNumber()	Same as **IntlFormatS32()**, only adds decimal points as needed.
IntlFormatDate()	Displays kanji to separate era, day, month, and year.
IntlFormatTime()	Displays A.M./P.M., hours, and minutes with kanji separators.

Japanese behavior of international functions

TABLE 22-2 (continued)

Function	*Default behavior*
IntlParseS32()	Parses signed integers with thousands separators, decimal point, minus signs.
IntlParseNumber()	Same as **IntlParseS32**(), only parses floating-point numbers.
IntlParseDate()	Parses calendar format with kanji to indicate day, month, year.
IntlParseTime()	Parses A.M./P.M., hours, minutes, with kanji separators.
IntlCompare()	Compares Unicode values of two characters.
IntlSort()	Sorts characters by Unicode value.
IntlMBToUnicode()	Converts latest Shift-JIS encoding (1990) to Unicode.
IntlUnicodeToMB()	Converts Unicode to latest Shift-JIS encoding (1990) to Unicode.

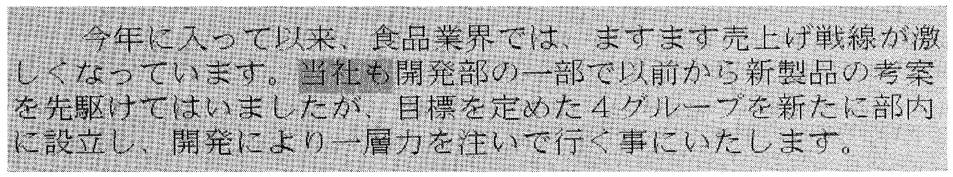
Many of these functions are discussed in detail in the rest of this chapter.

Delimiting words

The Japanese equivalent of an English word is called a **bunsetsu**, which literally means a phrase.

Text with selected bunsetsu

FIGURE 22-4



　今年に入って以来、食品業界では、ますます売上げ戦線が激しくなっています。当社も開発部の一部で以前から新製品の考案を先駆けてはいましたが、目標を定めた４グループを新たに部内に設立し、開発により一層力を注いで行く事にいたします。

The rules for delimiting an English word are relatively straightforward because English uses spaces and punctuation to separate words. Japanese does not use spaces, so the rules for locating a bunsetsu are quite complicated.

Call the PenPoint 2.0 Japanese function **LocDelimitWord**() to locate a bunsetsu. The prototype for the international function follows. Remember that the **Loc...**() macro calls the **Intl...**() function, passing **intlDefaultLocale** as the LOCALE_ID.

```
S32 EXPORTED IntlDelimitWord(
     P_CHAR      pString,     // Beginning of text region
     P_U32       pStart,      // In/Out: seed position/start of word
     LOCALE_ID   locale,      // Locale to use -- from golocale.h
     U32         style        // Delimit style -- from isrstyle.h
);
```

This function and the **IntlDelimitSentence**() function both take a start position and return the start and length of the requested item (a word or sentence). The length is returned by the function, and the start position is returned as one of its out parameters **pStart**.

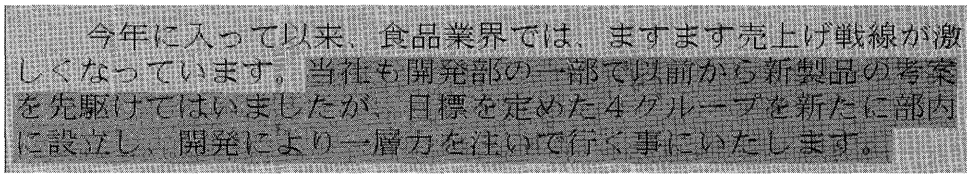
Use the **intlDelimitExpandLeft** or **intlDelimitExpandRight** flags to extend the selection in a single direction one bunsetsu at a time. See the file ISRSTYLE.H for more details and other valid styles.

See "Delimiting words" on page 289 for more details on how to locate bunsetsu in your application.

Delimiting sentences

Japanese uses a mark called a **maru** to end a sentence. It works similarly to the English period. Unicode calls the symbol the ideographic period (U+3002) because it is common to Chinese, Japanese, and Korean.

Text with sentence selected FIGURE 22-5



　今年に入って以来、食品業界では、ますます売上げ戦線が激しくなっています。当社も開発部の一部で以前から新製品の考案を先駆けてはいましたが、目標を定めた4グループを新たに部内に設立し、開発により一層力を注いで行く事にいたします。

Use the **LocDelimitSentence**() macro to find a sentence in a text stream. Here is the prototype:

```
S32 EXPORTED IntlDelimitSentence(
      P_CHAR     pString,    // Beginning of text region
      P_U32      pStart,     // In/Out: seed position/start of sentence
      LOCALE_ID  locale,     // Locale to use -- from golocale.h
      U32        style       // Delimit style -- from isrstyle.h
);
```

See "Delimiting sentences" on page 290 for details on how to locate sentences in your application.

Comparing and sorting

There is a well-established ordering for the kana characters. The characters are arranged according to the sounds of the "Fifty Sounds Table." You can find the table in any Japanese dictionary or introduction to Japanese writing. See Chapter 25, Resources, for references to some of these texts.

The kanji characters, however, are more difficult to order. Popular dictionaries sort characters by radical. Within radicals, they sort characters by the number of additional strokes, not including the radical, it takes to write the character.

The JIS character list, unfortunately, is not uniformly ordered this way. The Level 1 kanji are ordered phonetically (that is, by their kana equivalents), while the Level 2 kanji are ordered by the radical-stroke scheme.

In a Shift-JIS text that contains both Level 1 and Level 2 kanji, sorting characters is quite a challenge. Fortunately, the Unicode encoding already puts Japanese characters in sorted order. Thus PenPoint 2.0 Japanese can sort Japanese characters simply by their Unicode value. Specify the **intlSortStyleDictionary** style when you call **IntlCompare**() or **IntlSort**() to sort by radicals and number of strokes.

For more information on how Unicode orders Japanese characters, see *The Unicode Standard: Version 1.0, Volume 1.*

The other available sort and compare style is **intlSortStylePhoneBook**. If you specify this style, the sort and compare functions use the JIS ordering for Level 1 kanji; that is, comparing and sorting is done phonetically. There are various complicated comparison rules for characters outside of the Level 1 kanji.

Here is the prototype for the **IntlSort**() function:

```
STATUS EXPORTED   IntlSort(
     PP_CHAR      ppString,    // list of strings to sort
     U32          count,       // number of strings in list
     LOCALE_ID    locale,      // Locale to use -- from golocale.h
     U32          style        // Collation style -- from isrstyle.h
);
```

See "Comparing strings" on page 291 and "Sorting strings" on page 292 for details on how to give your application comparison and sort capabilities.

Converting between character variants

There are four typical character conversions you may want to support:

- Katakana to hiragana.
- Hiragana to katakana.
- Zenkaku (full-width) to hankaku (half-width).
- Hankaku to zenkaku.

The width conversion functions work with the ASCII and katakana characters. The normal size for katakana is full-width (zenkaku), and the normal size for alphanumerics is half-width (hankaku).

You can convert individual characters or strings. Functions that work on individual characters are in CHARTYPE.H, and have names that begin with **IntlChar...**(), as in **IntlCharToUpper**(). The string conversion functions, defined in ISR.H. are **IntlStrConvert**() and **IntlNStrConvert**().

See the header file CHARTYPE.H for more information about how the character conversion functions work. Some functions provide only an approximation of the desired conversion.

All of these functions convert a Unicode character or string to another Unicode character or string. They do not convert between character sets. For more information on conversions between character sets, see the next section, "Converting between Shift-JIS and Unicode."

The Unicode representation of zenkaku and hankaku are in a special area called the **Unicode Compatibility Zone**, which extends from U+FE00 to U+FFEF. The zone contains character variants that exist in Unicode solely to be compatible with other characters sets like Shift-JIS.

The string conversion functions also support conversions to and from the Compatibility Zone. Your application might, for example, import a Shift-JIS text, convert it to Unicode, and then convert all the characters in the Compatibility Zone to their equivalents outside of the Compatibility Zone. This would convert any half-width katakana characters to full-width katakana. It would also convert any full-width alphanumerics to half-width. Think of conversions out of the Compatibility Zone as converting characters to their most typical form.

The string and character conversion functions also handle conversions between upper and lowercase and between composed characters and their base character plus diacritical mark equivalent.

See "Converting between character variants" on page 287 for details on how to provide character conversion support in your application.

Converting between Shift-JIS and Unicode

Many existing Japanese files are in Shift-JIS format. Therefore, your application may want to provide import capabilities for Shift-JIS files. PenPoint 2.0 Japanese provides functions named **IntlMBToUnicode()** and **IntlUnicodeToMB()** to convert between Shift-JIS to Unicode strings. The default translation converts to and from the latest (1990) Shift-JIS encoding.

See "Converting Unicode and Shift-JIS strings" on page 286 for details on how to use these functions.

If you are converting a string that contains a filename, set the **intlCharSetFileNameMapping** flag. Because operating systems use different characters to represent path and file names, the string conversion function must know whether the string to be converted is (or contains) a filename. For example, most Japanese versions of DOS use the yen (¥) character to separate path names, while most U.S. English versions of DOS use the backslash (\) character.

To convert entire files between different character sets, use the DOS utility UCONVERT. See "Converting Unicode and Shift-JIS files" on page 285 for details.

Compressing Unicode

Unicode can be efficiently compressed when written to a file, especially if all the characters in the text stream are from the same character set (for example, all ASCII text).

All Unicode characters are 16-bits long. Shift-JIS, on the other hand, uses a single byte to encode hankaku, katakana, and ASCII characters, and two bytes to encode a all other Japanese characters. Thus, the two character encodings require roughly the same amount of memory with mostly Japanese text.

When filed, however, Unicode data can be compressed. PenPoint 2.0 Japanese provides functions that allow you to compress Unicode strings before filing them. Typically, these compressed Unicode files store Japanese text using less space than the identical Shift-JIS file.

Call **IntlCompressUnicode()** and **IntlUncompressUnicode()** to compress and decompress Unicode strings. See the header file ISR.H for more information.

Formatting Japanese text

The following sections describe Japanese text formatting conventions. Table 22-3 shows some of these conventions.

PenPoint 2.0 Japanese provides many formatting functions that provide appropriate formatting behavior for Japanese text. Your application should simply call these functions whenever they are available.

The only formatting convention shown in Table 22-3 that does not have native PenPoint 2.0 Japanese support is phone number formatting. Your application should provide its own formatting functions to handle phone numbers. Note that the number of digits in a Japanese area code varies with geographical location.

Default Japanese Formatting TABLE 22-3

Formatting area	American English formatting	Default Japanese formatting
Date Formatting	3/31/92	1992年3月31日
Time Formatting	11:45 P.M.	午後3時51分
Number Formatting	1,234,567.89	1,234,567.89
Currency Formatting	$1995.95	¥199,500
Phone Numbers	(415) 358-2000	(045) 472-6000
Paper Sizes	Letter, 8.5 in. x 11 in.	A4, 210 cm x 297 cm

Table 22-3 shows the default format for a Western-style (Gregorian) date. See Table 22-5 for the default formatting of an Imperial calendar date.

Line breaks

Japanese, like most other languages, does not permit certain characters to appear at the beginning or end of a line. For example, in both English and Japanese, you cannot begin a line with a close parenthesis or end a line with an open parenthesis.

Japanese characters do not use hyphens when they break across lines. Either a break is permitted and the subsequent characters continue onto the next line, or no break is permitted.

When romaji appears in text, Japanese uses the same rules as English for line breaks.

Call **IntlBreakLine**() to ensure your text breaks correctly. The function uses an INTL_LINE_BREAK structure to contain information about how to break a line. Here is the structure, defined in ISR.H:

```
typedef struct INTL_BREAK_LINE {
      U32  breakAt;          // position of line break
      U32  deleteThis;       // chars to delete from end of this line
      CHAR insertThis[intlBreakLineMaxInsert];
                             // chars to insert at end of this line
      U32  deleteNext;       // chars to delete from start of next line
      CHAR insertNext[intlBreakLineMaxInsert];
                             // chars to insert at start of next line
} INTL_BREAK_LINE, *P_INTL_BREAK_LINE;
```

Because Japanese does not need hyphens to indicate a line break, you do not need to use the fields when dealing with Japanese characters. However, because Japanese follows the same rules as English when text contains romaji, your code should be prepared to handle these fields. Here is the prototype for **IntlBreakLine**() itself:

The line break function does not currently support hyphenation, so the various insert and delete fields in **INTL_BREAK_LINE** are empty. Hyphenation support is planned for future releases of PenPoint.

```
S32 EXPORTED      IntlBreakLine(
      P_CHAR            pString,      // Line to break
      U32               pos,          // 1st char that won't fit
      P_INTL_BREAK_LINE pBreak,       // Out: how to break it
      LOCALE_ID         locale,       // Locale to use
      U32               style         // breaking style
);
```

See “Delimiting words” on page 289 for details.

Dates

Japanese uses two different date formats. One is based on the Western-style Gregorian calendar, the other on the Japanese imperial calendar. In the Japanese imperial calendar, the year 1992 is called Heisei 4, the fourth year of the reign of the current emperor. Otherwise, the two calendar systems are identical.

The international functions use the structure INTL_TIME, defined in ISR.H, to represent the current time. The INTL_TIME structure contains a field to represent the era. Use macros defined in GOLOCALE.H to fill in this field if you use the era field to represent, for example, a Japanese imperial date.

Table 22-4 shows the four Japanese eras that PenPoint 2.0 Japanese supports, along with the macro that represents the era.

Supported Japanese eras

TABLE 22-4

Era name	Macro in GOLOCALE.H	Years
Meiji	itcEraMeiji	1868–1912
Taisho	itcEraTaisho	1912–1926
Showa	itcEraShowa	1926–1989
Heisei	itcEraHeisei	1989–present

Call **IntlFormatDate**() to get a formatted date string from an INTL_TIME structure. The functions accept a number of style flags that can present dates in various formats, examples of which are shown in Table 22-5.

Date formats

TABLE 22-5

Date	Locale and style
1992年3月31日	locJpn, intlFmtDateStyleFull
平成4年3月31日	locJpn, intlFmtDateStyleFull; intlSecToTimeStructStyleJapanese
1990.1.15	locJpn, intlFmtDateStyleAbbrv
90/1/15	locJpn, intlFmtDateStyleNumeric
January 15, 1990	locUSA, intlFmtDateStyleFull
Jan. 15, 1990	locUSA, intlFmtDateStyleShort
1/15/90	locUSA, intlFmtDateStyleNumeric
15-Jan-90	locUSA, intlFmtDateStyleAbbrv

You use the **intlSecToTimeStructStyleJapanese** style with the **IntlSecToTime-Struct**() function. All the other styles shown work with **IntlFormatDate**().

If you cannot create the date string you want, **IntlFormatDate**() also accepts an explicit format string. The string represents a date string constructed from its constituent parts. PenPoint 2.0 Japanese allows you to construct a date string using any of the following parts: day, month, year, day of the week, day of the year, and an era. See the header file ISRSTYLE.H for more information.

You can also format a date according to user-specified system preferences. The function **PrefsIntlDateToString()** returns a pointer to the string containing a formatted date when you pass it a P_INTL_TIME structure. The function is defined in PREFS.H.

See Chapter 107 in the *PenPoint Architectural Reference* for more general information on how to observe system preferences.

Times

Japanese uses almost the same time formats as American English. The only difference is that kanji characters are used to distinguish hours, minutes, seconds, and whether the time is A.M. or P.M. Table 22-6 shows some of the time formats you can create by specifying the appropriate styles when calling **IntlFormatTime()**. All of the examples below assume the locale is **locJpn**.

Time Formats TABLE 22-6

Time	*Locale and styles*
15時51分	**intlFmtTimeStyleLocal**
15時51分34秒	**intlFmtTimeStyleLocal, intlFmtTimeDispSeconds**
午後3時51分	**intlFmtTimeStyleLocal**
午後3時51分34秒	**intlFmtTimeStyleLocal, intlFmtTimeDispSeconds**
3:51午後	**intlFmtTimeStyleStandard**
3:51:34午後	**intlFmtTimeStyleStandard, intlFmtTimeDispSeconds**
13:51:34	**intlFmtTimeStyleStandard, intlFmtTimeForce24Hour, intlFmtTimeDispSeconds**
13:51	**intlFmtTimeStyleStandard, intlFmtTimeForce24Hour**

Numbers

Japanese uses Arabic numerals to represent numbers for most purposes. In more formal settings, however, Japanese text uses kanji to represent numbers. PenPoint 2.0 Japanese currently supports only Arabic numerals, although ISRSTYLE.H defines a style **intlFmtNumStyleKanji** for future use.

Numbers like 1,234,567 are split every thousand with commas as they are in English. Specify the default style **intlStyleDefault** when you call one of the number formatting functions to format numbers this way.

Japanese occasionally uses an older style of formatting that puts a comma after every ten thousand, as in 12,3456. You must provide your own formatting function if you want to support the older style.

Remember that Japanese currency amounts can get quite large. Billions of yen are not uncommon in typical texts. Remember to set aside screen space to display all the necessary digits.

Call **IntlFormatS32()** or **IntlNFormatS32()** to format a signed integer. The equivalent functions **IntlNFormatNumber()** and **IntlFormatNumber()** work on floating point numbers.

You can specify many styles that control how numbers are formatted. The following listing comes from ISRSTYLE.H.

```
/*
 The style flags for number formatting give you extensive control of
 how the number is formatted.  They work for both the FormatS32 and
 the FormatNumber (double) functions.
      intlFmtNumLeftJustify: Add padding spaces on the left so that the
            decimal points align.  This is based on the number of characters not
            their widths, so it only works with fixed width fonts.
      intlFmtNumRightJustify: Add padding spaces on the right so that the
            decimal points align.  This is based on the number of characters not
            their widths, so it only works with fixed width fonts.
      intlFmtNumDropTrailZeros: Drop trailing zeros after the decimal point.
            E.g. 23.020 would become 23.02 with this set.
      intlFmtNumScale: Move the decimal place to the left by the
            number of digits specified by the 'scale' parameter.  E.g. a scale of
            two would cause 1234. to come out as 12.34 when this flag is set.
      intlFmtNumSpaceFill: Force the fill character to be a space. So
            if the results of a format would have been "***23.4" it would instead
            be "   23.4".
      intlFmtNumZeroFill: Force the fill character to be a zero. So
            if the results of a format would have been "***23.4" it would instead
            be "00023.4".
      intlFmtNumForceDecimal: Force a decimal point to be displayed
            even if it would not normally be shown. E.g. "123" would become
            "123." with this set.  This is usually used with a scale of 0 or if
            intlFmtNumDropTrailZeros is set.
      intlFmtNumDisplayPositive: Force the display of the sign on
            positive numbers.  E.g. "123" would become "+123" with this set.
*/
#defineintlFmtNumLeftJustifyflag16        // Pad to align on left side
#defineintlFmtNumRightJustifyflag17       // Pad to align on right side
#defineintlFmtNumDropTrailZerosflag18     // Drop trailing zeros in fraction
#defineintlFmtNumScale       flag19       // Move decimal by scale
#define intlFmtNumSpaceFillflag20         // Use space character for fill
#define intlFmtNumZeroFillflag21          // Use 0 digit for fill
#define intlFmtNumForceDecimalflag22      // Use decimal even if not needed
#define intlFmtNumDisplayPositiveflag23   // Sign on positive num. (e.g. +5)
/*
```

Each style specifies a general way of formatting a number. The details depend on the locale and the style flags you give. Also some of the styles are specific to some regions of the world, and do not make sense everywhere.

```
      intlFmtNumStylePlain: The simplest format for the locale.  No
            thousands separators or other fancy stuff.  In USA & Japan you get
            results like "1000.0" and "-1000.0" with this.
      intlFmtNumStyleSimple: Default] This is the standard format used
            in the locale.  It normally includes the thousands separators.  In
            USA & Japan you get results like "1,000.0" and "-1,000.0" with this.
      intlFmtNumStyleAccounting: This is the typical style of numbers used
            by accountants and such for the locale.  In USA & Japan you get
            results like "1,000.0" and "(1,000.0)" with this.  This format always
            uses some non-blank form of fill by default.  For example "**3.45" is
            used in USA and Japan.
      intlFmtNumStyleFillSign: A common style in some places is to put the
            space fill between the sign and the number.  This style is only
            defined for locales where this makes sense.  In USA & Japan you get
            results like "-  1,000.0" with this.
      intlFmtNumStyleKanji: <<Not implemented>> This style indicates you
            want Kanji digits instead of the normal 0-9.
*/
// International styles
#define intlFmtNumStylePlain0x0001              // e.g. 1000.0 & -1000.0
#define intlFmtNumStyleSimple0x0002             // e.g. 1,000.0 & -1,000.0
#define intlFmtNumStyleAccounting0x0003         // e.g. 1,000.0 & (1,000.0)
// Common European/North American styles
#define intlFmtNumStyleFillSign0x0004           // e.g. "-   1,000.0"
// Japanese Number Format styles, NOT supported at this time
#define intlFmtNumStyleKanji0x0005              // Use Kanji digits
```

Chapter 23 / Development Environment

The Japanese localization of the PenPoint™ 2.0 Japanese operating system development environment contains many tools, utilities, and sample files that help you edit, compile, link, and debug Japanese applications. This chapter highlights the available tools, but does not discuss them in detail. More detailed information can be found in *PenPoint Development Tools* and *Part 4: PenPoint Development Tools Supplement*, in this book.

This chapter assumes that you are familiar with the process of creating PenPoint applications. For more information on these topics, consult the *PenPoint Application Writing Guide, Expanded Edition* and the two manuals mentioned above. Chapter 28 of *Part 4: PenPoint Development Tools Supplement* contains a visual overview of the entire process of creating PenPoint 2.0 Japanese applications and services.

Development tools

This section describes the tools you should use to edit, compile, and make applications.

Text editors

Your source code consists mostly of ASCII files since it is mostly C code.

Sometimes, though, your code contains literal Japanese strings. For example, the Japanese version of your application resource file, JPN.RC, must contain Japanese strings encoded as a combination of ASCII and Shift-JIS. Your application uses the Japanese strings in JPN.RC in its user interface.

The easiest way to work with Shift-JIS files is with a Shift-JIS editor. Most editors popular in the U.S. have Japanese versions that allow you to edit Japanese text.

You can use MiniText as a Shift-JIS and Unicode editor. First, make sure the PenPoint system locale is JPN by specifying it when you run the GO batch file:

```
    go jpn
```

When MiniText imports a DOS file, it assumes high ASCII characters are part of a Japanese character; that is, it assumes the file contains Shift-JIS data. See "Working with Shift-JIS in text files" on page 283 for details.

Keep the number of files that contain Shift-JIS characters at a minimum. This will make your project easier to maintain because all your Japanese strings are in one place. In the best case, only your application resource file JPN.RC will contain Shift-JIS characters.

Compilers

Make sure your compiler can compile code containing 16-bit characters. You must set the compiler flag that enables this feature when compiling code that contains 16-bit Unicode or multibyte Shift-JIS strings.

For example, if you are using the WATCOM C compiler, you must set the compiler flag /ZK0U. The standard makefile rules provided with the sample applications as SDEFINES.MIF automatically set this flag.

The PenPoint 2.0 Japanese resource compiler RC.EXE also uses this flag because your resource files often contain Shift-JIS characters.

Debuggers

PenPoint 2.0 Japanese allows you to display Japanese strings in the debugger stream. You can specify which character set you want to display using the **DebugCharSet** variable in ENVIRON.INI discussed in the next section.

You can view the debugger stream on a second monitor only if your debugger stream contains ASCII characters.

To view kanji in the debugger stream, use the System Log application or save the debugger stream to a file. See Chapter 10 of *PenPoint Development Tools* and Chapter 30 of *PenPoint Development Tools Supplement* for information on saving the debugger stream to a file.

The value of **DebugCharSet** also controls the interpretation of the mini-debugger memory dump commands (**d, da, db, dd**, and **dw**). See Chapter 30 of *PenPoint Development Tools Supplement* for details on debugging.

DebugCharSet

The **DebugCharSet** variable in ENVIRON.INI controls the character set of your debugging output. Table 23-1 shows the currently permissible values.

Debug CharSet variable values TABLE 23-1

Value	*Description*
ASCII	Standard 7-bit ASCII
XJIS	1990 Shift-JIS character set
437	IBM Code Page 437 used in U.S. IBM PCs
850	IBM Code Page 850 used in European IBM PCs

If you are sending debugging information to your PenPoint monitor or a second debugging monitor, make sure it can display characters in the specified **DebugCharSet.** GO does not support using Shift-JIS monitors as second debugging monitors. See Chapter 30, Debugging, of *Part 4: PenPoint Development Tools Supplement* for information about how to see Shift-JIS in your files.

Literal strings in **Debugf()** and **DPrintf()** appear in the specified character set. Unsupported Unicode characters display as hex quads in PenPoint 2.0 Japanese. On your monitor, they display as \x*nnnn*, where *nnnn* is a four-digit hex number.

The default value of **DebugCharSet** depends on the value of LOCALE, another ENVIRON.INI variable. If LOCALE=JPN, the default is Shift-JIS. The default is ASCII if LOCALE=USA.

If **DebugCharSet** is set to an invalid value, the default character set is assumed.

Makefiles

The standard makefile rules provided with the sample applications help you make different localized versions of your application. If you write your makefile by tailoring a makefile from a sample application, you can add a LOCALE argument to the command line to make a particular localized version of your application. For instance, type:

```
wmake LOCALE=jpn
wmake LOCALE=usa
```

to create the Japanese and American versions of your application, respectively. If you do not supply a LOCALE argument, JPN is the default locale.

You must create a file called JPN.RC to contain your application's Japanese strings. The file should at least contain strings for the **tagAppMgrAppFilename** and **tagAppMgrAppClassName**. The standard makefile rules stamp the application directory with the strings associated with these tags.

In your makefile, you can use three new variables to identify which resource files to compile and copy into the application directory with the executable image.

GO's sample makefile variables TABLE 23-2

Variable	*Use*
RES_FILES	Resource files to be included with all versions of your application.
USA_RES_FILES	Resource files to be included with only the American version.
JPN_RES_FILES	Resource files to be included only with the Japanese version of your application.

See Chapter 29 of *PenPoint Development Tools* for details on creating PenPoint applications and services.

DOS utilities

PenPoint 2.0 Japanese provides a collection of DOS utilities that help you work with resource files, PenPoint file names, and international character sets. See Chapter 14 of *PenPoint Development Tools* and Chapter 31 of *Part 4: PenPoint Development Tools Supplement* for detailed information on how to use the utilities.

The following table briefly summarizes the purpose of each utility.

DOS utilities TABLE 23-3

Name	*Purpose*
PSTAMP.EXE	Adds special PenPoint information to a DOS file or directory. Replaces STAMP from PenPoint 1.0.
PDEL.EXE	Deletes specific directory entries from PENPOINT.DIR files.
PCOPY.EXE	Recursively copies files and directories to other PenPoint directories.
PDIR.EXE	Lists the PenPoint names and file systems attributes for all the files and directories in a DOS directory. Replaces GDIR from the utilities included with PenPoint 1.0.
PSYNC.EXE	Scans the current directory and removes any entries from PENPOINT.DIR for which there are no corresponding files.
RC.EXE	Compiles resource files.
RESAPPND.EXE	Appends resources from one resource file into another.
RESDUMP.EXE	Shows the contents of a compiled resource file.
RESDEL.EXE	Deletes specified resources from a compiler file.
UCONVERT.EXE	Converts files between character sets, for example from Shift-JIS to Unicode.
CONTEXT.BAT	A DOS batch file that sets the required DOS environment variables PenPoint requires. Takes an argument to indicate which version of PenPoint (1.0 or 2.0).
GO.BAT	Boots PenPoint on your development machine, allowing choice of the system and user locales.
LOCALE.BAT	Switches the system and user locales that PenPoint uses.

In PenPoint 2.0 Japanese, the PENPOINT.DIR file is in Unicode format, although the utilities that deal with PenPoint information can still read ASCII files. For additional information on each utility, type -? or /? after most of these commands to see a help message. For example, type **PDIR /?** for help on the PDIR utility.

See *PenPoint Development Tools* and its supplement for more information.

You can set two DOS environment variables to notify the utilities which character set or locale you typically work with.

CHARSET can be one of ASCII, 437, LATIN1, or 850 to denote a character set.

LOCALE can be either USA or JPN.

For example, if you specify a LOCALE of JPN, then the DOS utility PDIR will interpret your PenPoint names as a Shift-JIS string.

Do not confuse the DOS environment variable with the LOCALE in ENVIRON.INI. Only the DOS utilities are sensitive to the DOS environment variable. PenPoint itself is sensitive to the LOCALE in ENVIRON.INI.

If you want, set these environment variables in your AUTOEXEC.BAT with the DOS command SET. Other character sets and locales are supported, but the ones listed here are the relevant values for Japan.

Running PenPoint

You must remove any terminate-and-stay-resident (TSR) programs before booting PenPoint 2.0 Japanese. Some TSRs use the same interrupts as PenPoint. This conflict causes boot error 106 (unknown boot error).

The easiest way to remove TSRs is to remove the programs in GO.BAT and reinstall them, if necessary, after PenPoint exits. Comments in GO.BAT indicate where you should remove and reload your TSRs.

The GO.BAT batch file now takes two optional parameters to specify the locales to boot with:

```
go system_locale user_locale
```

When you specify a system locale, PenPoint's behavior and user interface are changed to be appropriate to the specified locale (U.S. or Japanese).

When you specify both a system and user locale, the batch file directs PenPoint to change its behavior to match the system locale, but to change its user interface strings to match the target locale.

When you type GO with no parameters, PenPoint boots in the same state as it was last booted. If you type GO with no parameters and you are in **DebugTablet** mode, PenPoint warm boots. See Chapter 30 of *Part 4: PenPoint Development Tools Supplement* for more information about debugging modes and warm booting. For example:

- To boot with Japanese behavior and strings, type
  ```
  go jpn
  ```
- To boot with Japanese behavior, but English strings, type
  ```
  go jpn usa
  ```

Because the batch file only controls the resource files PenPoint loads, the stamped application and service names appear in the system locale language.

GO.BAT relies on LOCALE.BAT to do the locale switch. Make sure \2_0\PENPOINT\SDK\UTIL\DOS is in your DOS PATH. Both GO.BAT and LOCALE.BAT require utilities in that directory to switch locales.

When you specify a locale with GO.BAT (or LOCALE.BAT) the batch file recursively deletes your \PENPOINT\SS directory. This deletes any documents that you had saved in your PenPoint 2.0 Japanese file system. Make sure to save the files to your hard drive if you need them.

Warning The GO.BAT and LOCALE.BAT batch files delete your PenPoint files when you specify locales.

Currently, only two locales are supported: JPN and USA. See page 45 of *PenPoint Development Tools* for more information about the GO batch file. The manual describes the PenPoint boot process, including the order in which files are read and the actions that are taken as a result.

See Chapter 31 of *Part 4: PenPoint Development Tools Supplement* for details on how the batch file coordinates the locale switching.

PenPoint environment

This section describes the PenPoint 2.0 Japanese environment variables relevant for developing Japanese applications.

ENVIRON.INI

There are two important variables new to PenPoint 2.0 Japanese that you should set when running Japanese applications.

- **Locale** can be set to USA or JPN. Its value controls PenPoint's behavior and appearance. Different locales use different fonts, dynamic link libraries, applications, and services. See Chapter 32 of *Part 4: PenPoint Development Tools Supplement* for details.
- **DebugCharSet** can be set to ASCII, XJIS, 437, or 850, controls the interpretation of characters you send to the debugger stream. See Chapter 4 of *PenPoint Development Tools Supplement* for details.

Remember to set your PenPointPath to \2_0 if you are working with PenPoint 2.0 Japanese development.

MIL.INI

PenPoint supports many different U.S. and Japanese keyboard models. Set your **Keyboard** variable in MIL.INI to identify your keyboard.

The value of **Keyboard** determines how the keyboard behaves throughout PenPoint. For example, **clsField** and **clsKKCT** observe this variable to determine how it should handle character input. To change keyboards, you must warm or cold boot. Swap booting does not change keyboard behavior.

See "Using keyboards" on page 261 for tips on using your keyboard to type Japanese and English characters.

Initialization files

PenPoint 2.0 Japanese uses a collection of control files to set up its environment. Since these files can sometimes contain Japanese filenames, some of these control files can contain Shift-JIS or Unicode characters. The following table shows which combinations are permitted.

Character sets in control files TABLE 23-4

Filename	Permissible character sets
MIL.INI	ASCII only
ENVIRON.INI	ASCII only
BOOT.DLC	ASCII, Unicode
CONSOLE.DLC	ASCII, Unicode
APP.INI	ASCII, Shift-JIS
SERVICE.INI	ASCII, Shift-JIS
SYSAPP.INI	ASCII, Shift-JIS
SYSCOPY.INI	ASCII, Shift-JIS

PenPoint tools

The SDK includes some PenPoint 2.0 Japanese applications and accessories that can help you write Japanese applications.

MiniText

You can use MiniText as a Shift-JIS and Unicode editor. It supports Japanese handwriting recognition, KKC, and RKC. Although insertion pads only let you enter hankaku, you can convert between hankaku and zenkaku by selecting the To Hankaku or To Zenkaku commands under the Convert menu.

MiniNote assumes any imported text file contain Shift-JIS when the Locale variable in ENVIRON.INI is set to JPN. Your Shift-JIS file can also contain RTF keywords. Before you import an RTF file, run the file through the DOS utility RTFTRIM before importing it. RTFTRIM removes RTF keywords that PenPoint's text component does not use from an RTF file. See Chapter 31 of *Part 4: PenPoint Development Tools Supplement* for more information on RTFTRIM.

MiniText assumes that any file with a .UNC extension imported into PenPoint is a Unicode file. Be sure your Unicode files have the .UNC extension before you import them into your PenPoint notebook.

See "Working with Shift-JIS in text files" on page 283 for details on how to create, import, and export Shift-JIS files between PenPoint 2.0 Japanese and your development machine.

Unicode Browser

The Unicode Browser is a PenPoint 2.0 Japanese accessory that allows users to send characters to the text stream by tapping on them in a table of possible characters. See *Using PenPoint* for instructions on using the Unicode Browser.

Japanese virtual keyboard

The virtual keyboard is another PenPoint 2.0 accessory that allows you to send characters to the text stream. It offers various emulations, including American and Japanese IBMJ-A01 keyboard modes.

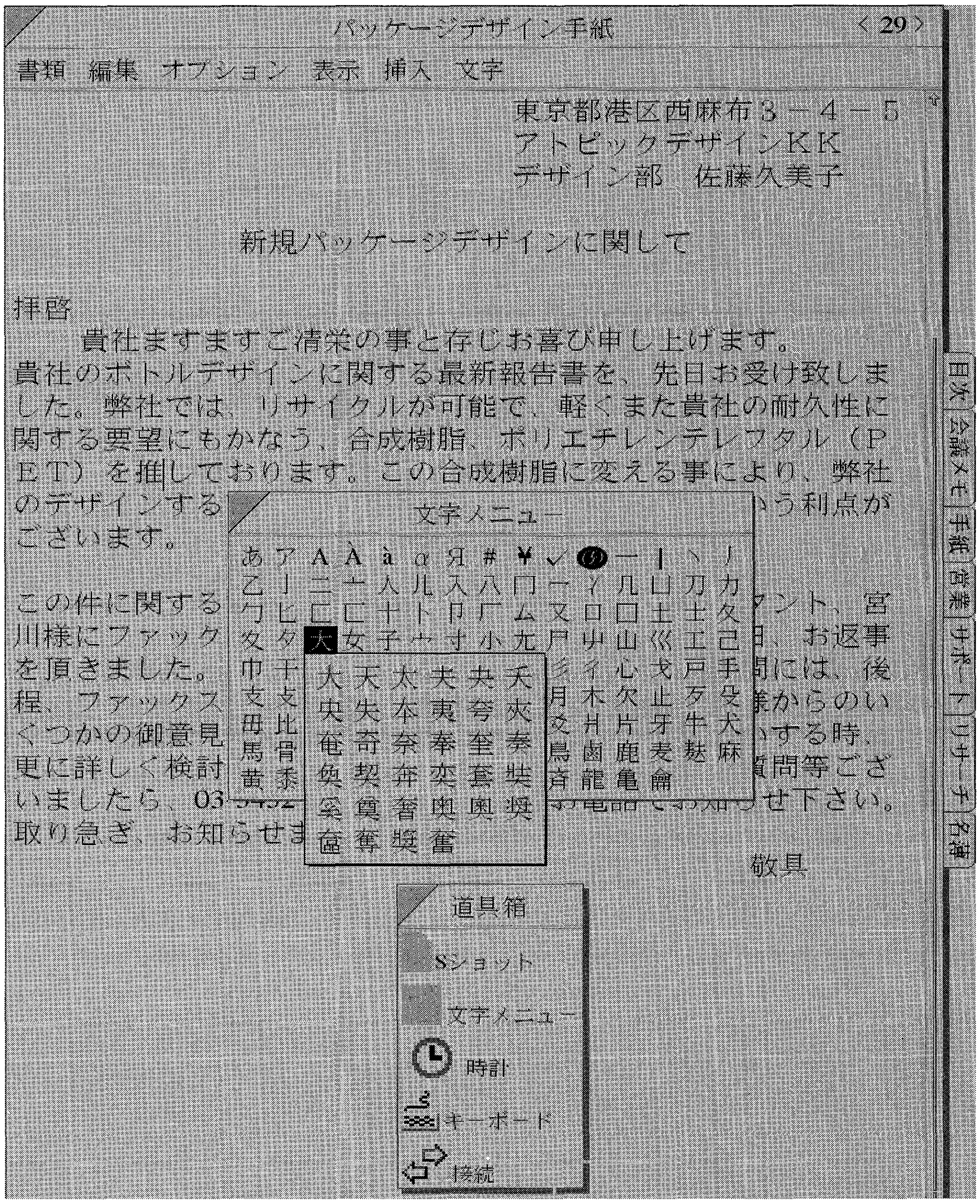
Bring up the keyboard by tapping on its icon in the Accessories notebook. Change modes by making the check ✓ gesture over the title bar to switch modes.

With the Japanese keyboard, you can type romaji, hiragana, or katakana. There are keys that toggle the keyboard between the character sets.

Sample code

The sample code included with the SDK is a good starting point for your own applications. Here are a few details to note about the sample code included with the 2.0J SDK in \2_0\PENPOINT\SDK\SAMPLE.

Unicode Browser FIGURE 23-1



東京都港区西麻布３－４－５
アトピックデザインＫＫ
デザイン部　佐藤久美子

新規パッケージデザインに関して

拝啓
　貴社ますますご清栄の事と存じお喜び申し上げます。
貴社のボトルデザインに関する最新報告書を、先日お受け致しました。弊社では、リサイクルが可能で、軽くまた貴社の耐久性に関する要望にもかなう、合成樹脂、ポリエチレンテレフタル（ＰＥＴ）を推しております。この合成樹脂に変える事により、弊社のデザインする　いう利点がございます。

この件に関する　ント、宮川様にファック　日、お返事を頂きました。　月には、後程、ファックス　様からのいくつかの御意見　する時、更に詳しく検討　質問等ございましたら、03　せ下さい。取り急ぎ、お知らせま

敬具

Use the Unicode Browser to enter hard-to-write characters or characters that are not recognized by the handwriting recognition engine.

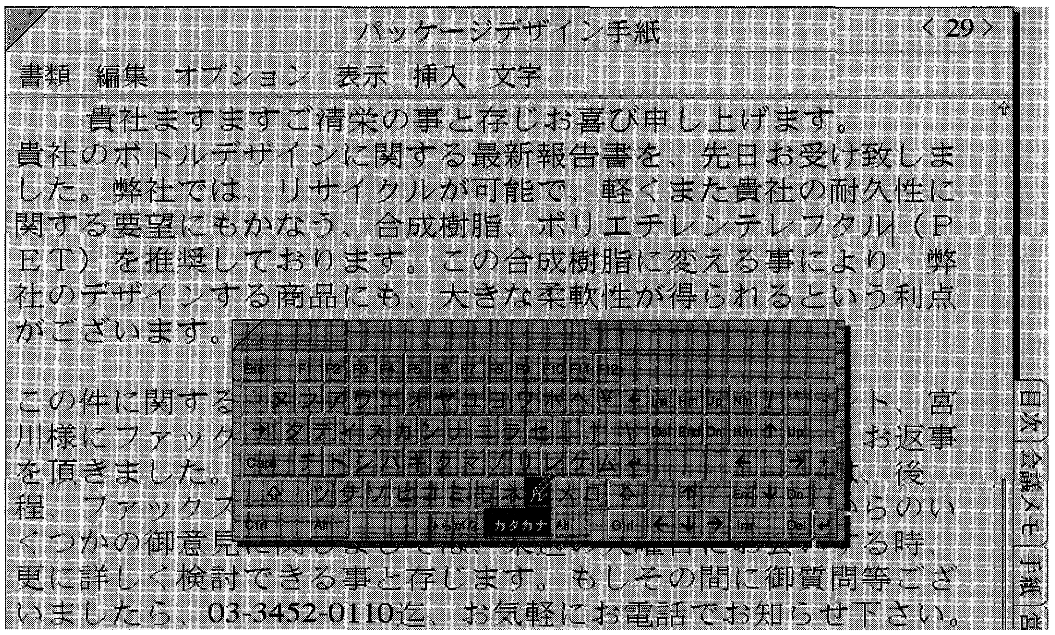
Japanese versions of sample code

Most of the sample applications have two resource files, USA.RC and JPN.RC. As their names suggest, these files contain U.S. English and Japanese strings. Use these files to help write your own resource files.

All of the sample applications except the Keisen Table application make use of the Bridging Package. This package allows you to maintain a single code base that compiles under both PenPoint 1.0 and 2.0. See the *PenPoint Bridging Handbook* included with the 2.0 SDK for details on how to do this.

Japanese virtual keyboard

FIGURE 23-2


パッケージデザイン手紙
< 29 >
書類 編集 オプション 表示 挿入 文字
Esc F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12
Caps
Ctrl
Alt
ひらがな
カタカナ
目次
会議メモ
手紙

　貴社ますますご清栄の事と存じお喜び申し上げます。
貴社のボトルデザインに関する最新報告書を、先日お受け致しました。弊社では、リサイクルが可能で、軽くまた貴社の耐久性に関する要望にもかなう、合成樹脂、ポリエチレンテレフタル（PET）を推奨しております。この合成樹脂に変える事により、弊社のデザインする商品にも、大きな柔軟性が得られるという利点がございます。

この件に関する … ト、宮
川様にファック … お返事
を頂きました。 … 、後
程、ファックス … らのい
くつかの御意見 … る時、
更に詳しく検討できる事と存じます。もしその間に御質問等ございましたら、03-3452-0110迄、お気軽にお電話でお知らせ下さい。

Use the virtual keyboard to enter characters into the text stream. You can simulate both American and Japanese keyboards.

Keisen Table application

The Keisen Table sample application uses hard-coded Japanese strings because the application is designed exclusively for Japan. It shows how to use toolkit tables to create a complex Keisen Table, a popular way of gathering data in Japan.

All the hard-coded strings are in Shift-JIS format.

Chapter 24 / Procedures

This chapter contains step-by-step instructions on how to take advantage of the PenPoint™ operating system's support for Japanese applications. It describes in detail how to perform several of the common procedures that developers use to write Japanese applications.

Working with Shift-JIS in text files

This procedure shows you one way of creating and editing Shift-JIS strings in a text file.

Prerequisite information

The easiest way to work with Shift-JIS is to edit it with a Shift-JIS editor. This procedure shows you how to use MiniText as a Shift-JIS editor.

Shift-JIS strings are most commonly used in control files like APP.INI and the Japanese version of your resource file, JPN.RC.

- "Character encoding" on page 247.
- "Shift-JIS encoding details" on page 252.
- "Text editors" on page 273.
- "Initialization files" on page 278.

Procedure

1. Set the **B800** debugging flag so that you can access your hard drive with the Connections notebook. You can do this one of two ways:
   - Add **/B800** to the **DebugSet** line in ENVIRON.INI.
   - While in PenPoint, press Break to drop into the mini-debugger. Type **fs B +800** to set the flag, and then **g** to resume PenPoint.
2. Create a new MiniText document or import an existing document. You can import by opening the Connections notebook and choosing Directory under the View menu. Then browse through your disk and copy a Shift-JIS file to your PenPoint notebook. Import the file as a MiniText document.
3. Turn to your new or imported MiniText document to edit it.
4. When you are done editing the file, turn back to your table of contents.
5. Open the Connections notebook and choose Directory under the View menu.
6. Move or copy the file to your hard drive.
7. Select Text File as the export type.

Related information

- "Working with Unicode in source code" on page 284.
- "Converting Unicode and Shift-JIS files" on page 285.
- "Converting Unicode and Shift-JIS strings" on page 286.

Working with Unicode in source code

This procedure shows you how to create Unicode strings in your source code.

Prerequisite information

- Chapter 15, *Part 2: PenPoint Internationalization Handbook.*
- "Character encoding" on page 247.
- "Unicode" on page 249.

Procedure

1. Declare your character or pointer to characters as a 16-bit type. Use CHAR16 for data that is always 16 bits and CHAR for data that will be 8-bits long in PenPoint 1.0 and 16 bits long in 2.0 and later releases.
2. Wrap the **U_L()** macro around literal characters and strings. Use the **L" "** modifier if you do not need your code to compile under PenPoint 1.0.
3. Type ASCII characters between the quotation marks.
4. To specify special Unicode characters, use a **\x** followed by a Unicode code point, which has 4 hexadecimal digits.

Examples

The following code uses the U_L() macro to indicate that the declared character or strings are 8 bits long in PenPoint 1.0 and 16 bits long in PenPoint 2.0 Japanese. The second example declares character data that is always 8 bits long.

```
Uprintf(U_L("I am 8 bits long in PenPoint 1.0; 16 bits in PenPoint 2.0");
P_CHAR8 pTheString = L"I am always a 16-bit string.";
static RC_TAGGED_STRING qHelpStrings[] = {
     tagTextView,      U_L("\xF61F \\tab Pigtail.  Delete a character.\\par "),
     Nil(TAG)
};
```

This last example specifies Unicode values directly because they cannot be typed with the keyboard.

Related information

- "Working with Shift-JIS in text files" on page 283.
- "Converting Unicode and Shift-JIS files" on page 285.
- "Converting Unicode and Shift-JIS strings" on page 286.

Converting Unicode and Shift-JIS files

This procedure converts files between Unicode and Shift-JIS formats.

Prerequisite information

- "Character encoding" on page 247.
- "Unicode" on page 249.
- "Shift-JIS encoding details" on page 252.
- "Converting to and from Shift-JIS" on page 254.

Procedure

1 If necessary, run CONTEXT.BAT to put your system in the 2_0 context.
The batch file adds \2_0\SDK\UTIL\DOS to the beginning of your PATH.

2 Run UCONVERT.EXE on the file to be converted. The syntax for this DOS utility is:

```
UCONVERT s[-d] [-m] ource-file dest-file [source CharSet] [dest CharSet]
```

You can specify a character set as either a code page or a locale as follows:

- Specify ASCII with one of the following: **ASCII**, **437**, or **USA**.
- Specify Shift-JIS with **XJIS** or **JPN**.
- Specify Unicode with **UNI**.

Examples

Table 24-1 shows sample runs of the UCONVERT utility.

Using UCONVERT TABLE 24-1

Command	*Description*
uconvert mytext.doc mytext.unc	Puts a Unicode copy of ASCII document MYTEXT.DOC in the file MYTEXT.UNC. ASCII-to-Unicode is the default conversion.
uconvert mytext.unc mytext.jis uni xjis	Puts a Shift-JIS version of the Unicode document MYTEXT.UNC in the file MYTEXT.JIS
uconvert -d myfiles.doc myfiles.jis xjis uni	Puts a Shift-JIS version of the file MYFILES.TXT containing filenames in the file MYFILES.JIS. The -d flag is necessary when the input Shift-JIS file contains filenames.
uconvert letter.jis letter.unc jpn uni	Puts a Unicode copy of the Shift-JIS file LETTER.JIS in the file LETTER.UNC.
uconvert -m longfile.437 longfile.unc	Puts a copy of the extended ASCII file LONGFILE.437 in the Unicode file LONGFILE.UNC, converting all CR/LF combinations to the Unicode line separator character (U+2028).

Related information

- "Working with Shift-JIS in text files" on page 283.
- "Working with Unicode in source code" on page 284.
- "Converting Unicode and Shift-JIS strings" on page 286.

Converting Unicode and Shift-JIS strings

This procedure allows your code to convert between Unicode and Shift-JIS strings.

Prerequisite information

- "Unicode" on page 249.
- "The Japanese character set" on page 247.
- "Shift-JIS encoding details" on page 252.
- "Converting to and from Shift-JIS" on page 254.

Procedure

1. Include ISR.H in your source file. Link INTL.LIB with your code by listing it in your makefile.
2. Call **IntlNUnicodeToMB()** to convert a Unicode string to a Shift-JIS string. Use one of these styles to indicate which JIS character set to convert to:
   - **intlCharSetStyleXJIS** maps to the most recent character set (currently JIS X0208-1990)
   - **intlCharSetStyleXJIS1978** for JIS C6226-1978
   - **intlCharSetStyleXJIS1983** for JIS X0208-1983
   - **intlCharSetStyleXJIS1990** for JIS X0208-1990
3. Call **IntlMBToUnicode()** to convert a Shift-JIS string to a Unicode string. Use the same styles to indicate which character set you are converting from. The default style uses the most current (1990) Shift-JIS standard.
4. Specify the style **intlCharSetFileNameMapping** if the string you want to convert contains a filename.

Example

This code fragment converts the multibyte string **pStr8** to the Unicode string **pStr**.

```
MsgHandlerArgType(MyHandler, P_MY_ARGS)
{
      STATUS      s;
      U32         oLength, length;
      P_CHAR      pStr;
      P_CHAR8     pStr8;
      length = pArgs->len;
      pStr8 = (P_CHAR8) pArgs->pData;
      oLength = length;
      if (SizeOf(CHAR) > 1)
      {
            StsWarn(OSHeapBlockAlloc(osProcessHeapId,
                  length*sizeof(CHAR), &pStr));
            StsWarn(oLength=IntlNMBToUnicode(pNull, 0, pStr8, &length,
                  intlStyleDefault));
            StsWarn(length=IntlNMBToUnicode(pStr, oLength, pStr8, &length,
                  intlStyleDefault));
      }
```

Related information

- "Working with Shift-JIS in text files" on page 283.
- "Working with Unicode in source code" on page 284.
- *Part 2: PenPoint Internationalization Handbook*, "Locale-Independent Code," in Chapter 15.

Converting between character variants

This procedure converts between various character sets, such as from zenkaku (full-width) to hankaku (half-width), and from katakana to hiragana.

Prerequisite information

- "Kana" on page 246.
- "Half- and full-width variants" on page 249.
- "Converting between character variants" on page 265.

Procedure

1. Include ISR.H in your source file. Link INTL.LIB with your code by listing it in your makefile.
2. Allow the user to specify a string to be converted. Collect the string in a buffer with a terminating null.
3. Call the **IntlStrConvert**() function.
4. Update your memory and user interface.

Example

This code sample uses **clsTextView** to support string conversion requested by the user. The functions convert the selected text to all upper-case, all lower-case, or initial capitals.

```
OBJECT          myTextObject;
int             desiredState;
U32             attrLimit, startLen, amtRemain, newLen, style;
TEXT_BUFFER     textBuffer;
P_TV_SELECT     pTarget;
...
switch (desiredState) {
            case 1:
                  style = intlStrConvertStyleToUpper;
                  break;
            case 2:
                  style = intlStrConvertStyleToProper;
                  break;
            case 3:
                  style = intlStrConvertStyleToLower;
                  break;
      }
...
```

```
while (pTarget->length)
{
      // ensure buffer size is less then maxbufferlen
      if (pTarget->length> MAXBUFFERLEN)
      {
            amtRemain = pTarget->length - MAXBUFFERLEN;
            pTarget->length = MAXBUFFERLEN;
      }
      else
            amtRemain = 0;

// Get selected chars into buffer
      textBuffer.first = pTarget->first;
      textBuffer.length = pTarget->length;
      textBuffer.bufUsed = 0;
      textBuffer.buf = pSrc;
      textBuffer.bufLen = pTarget->length;
      ObjCallWarn(msgTextGetBuffer, myTextObject, &textBuffer);
      startLen = pTarget->length;
      if (pTarget->length)
      {
            if (amtRemain)
                  style |= intlStrConvertMoreText;
            else
                  style &= ~intlStrConvertMoreText;
      // Do conversion with result in pDest
            newLen = IntlNStrConvert(pDest, MAXBUFFERLEN * 2, pSrc,
                  &(pTarget->length),&ctx, intlDefaultLocale, style);
```

Notes

The function prototype for **IntlStrConvert**() is as follows:

```
S32 EXPORTEDINTLStrConvert(
      P_CHARpDest,      // Out: converted string
      U32   destLen,    // Max space available in pDest
      P_CHARpSrc,       // Null-terminted string to be converted.
      LOCALE_IDlocale,  // Locale to use -- from golocale.h
      U32   style       // Conversion style -- from isrstyle.h
);
```

The relevant styles are:

```
// Flags used with string conversion styles.
#define intlStrConvertMoreText     flag16     // More text than was passed.
// String Conversion styles
#define intlStrConvertStyleToUpper       0x0001      // All characters
#define intlStrConvertStyleToProper      0x0002      // 1st letter of words only
#define intlStrConvertStyleToLower       0x0003      // All characters
#define intlStrConvertStyleToHiragana    0x0004      // from katakana, not kanji
#define intlStrConvertStyleToKatakana    0x0005      // from hiragana, not kanji
#define intlStrConvertStyleToComposed    0x0006      // minimize floating forms
#define intlStrConvertStyleToClean       0x0007      // maximize floating forms
#define intlStrConvertStyleToCompatibility     0x0008      // Map to C-Zone
#define intlStrConvertStyleFromCompatibility   0x0009      // Map from C-Zone
#define intlStrConvertStyleToHankaku     0x000A      // Map to half-width chars
#define intlStrConvertStyleToZenkaku     0x000B      // Map to full-width chars
```

Related information

See ISR.H for more information about how to convert large chunks of text extending over multiple buffers (such as converting an entire file).

Delimiting words

This procedure locates a bunsetsu, the Japanese equivalent of an English word or phrase, in a text stream.

Prerequisite information

"Delimiting words" on page 263.

Procedure

1 Include ISR.H in your source file. Link INTL.LIB with your code by listing it in your makefile.

2 Locate where the user has requested a phrase selection.

3 Call **LocDelimitWord**() or **LocNDelimitWord**().

Example

The following code demonstrates the query capabilities of the delimit word and sentence functions. The code queries a function by calling it with **pNull** where it expects a buffer. The function responds to the query by returning the size of the buffer that the code needs to send to the function. The returned size is used in the **GetSpanBuf**() call that fills the buffer with **nCharToCopy** characters.

```
#define atomSentence 4
...
typedef struct SPAN_BUF {
      P_CHAR      buf;
      TEXT_INDEX  len;
      U32         pos;
      BOOLEAN     freeBuf;
} SPAN_BUF, *P_SPAN_BUF;
...
SPAN_BUF    spanBuf
TEXT_INDEX  oldPos, first, baLen;
TEXT_SPAN   span, savNChToCopy, nCharToCopy;
STATUS      s;
S32         style;
if (span.type == atomSentence)
      savNChToCopy = nCharToCopy = LocDelimitSentence(pNull, pNull, style);
else
      savNChToCopy = nCharToCopy = LocDelimitWord(pNull, pNull, style);
```

```
while (TRUE) {
      spanBuf.pos = first;
      s = GetSpanBuf(pB, &spanBuf, nCharToCopy);
      if (s < stsOK) goto CleanUp;
      oldPos = spanBuf.pos;
      style = (first - spanBuf.pos) > 0 ?
            FlagSet(intlDelimitMoreLeft, style) :
            FlagClr(intlDelimitMoreLeft, style);
      style = (first + nCharToCopy) < baLen ?
            FlagSet(intlDelimitMoreRight, style) :
            FlagClr(intlDelimitMoreRight, style);
      if (span.type == atomSentence)
            s = LocNDelimitSentence(spanBuf.buf, spanBuf.len,
                  &spanBuf.pos, style);
      else
            s = LocNDelimitWord(spanBuf.buf, spanBuf.len, &spanBuf.pos, style);
```

The code uses two class manager macros **FlagSet**() and **FlagClr**() to set and clear style flags. The macros are defined as follows in CLSMGR.H:

```
#define FlagSet(f,v) ((v) | (f))
#define FlagClr(f,v) ((v) & (~f))
```

Notes

The function prototype looks like this:

```
S32 EXPORTED IntlNDelimitWord(
      P_CHAR      pString,    // Beginning of text region
      U32         length,     // Length of text region.
      P_U32       pStart,     // In/Out: seed position/start of word
      LOCALE_ID   locale,     // Locale to use -- from golocale.h
      U32         style       // Delimit style -- from isrstyle.h
);
```

The function expects a counted string, a locale, and a style. Remember that calling **LocDelimitWord**() sends **intlDefaultLocale** and **intlDefaultStyle** as parameters.

Pass in a position you want to search from as **pStart**. When the function returns, **pStart** contains the start of the bunsetsu, and the function itself returns the length of the bunsetsu.

Related information

- "Delimiting sentences" on page 290.
- The header files ISR.H and ISRSTYLE.H contain more information about different ways to call the delimit word and sentence functions.

Delimiting sentences

This procedure locates a sentence in a text stream.

Prerequisite information

"Delimiting sentences" on page 264.

Procedure

1. Include ISR.H in your source file. Link INTL.LIB with your code by listing it in your makefile.
2. Locate the position in your text stream where the user requested a sentence selection.
3. Call **LocNDelimitSentence**() or **LocDelimitSentence**().

Example

See example for "Delimiting words" on page 289.

Notes

Here is the function prototype:

```
S32 EXPORTED IntlDelimitSentence(
      P_CHAR      pString,     // Beginning of text region
      P_U32       pStart,      // In/Out: seed position/start of sentence
      LOCALE_ID   locale,      // Locale to use -- from golocale.h
      U32         style        // Delimit style -- from isrstyle.h
);
```

Specify **intlDlmtSntcStyleSentence** as a style to select a sentence without any punctuation.

Related information

"Delimiting words" on page 289.

Comparing strings

This procedure compares two null-terminated strings and returns their sort order.

Prerequisite information

"Comparing and sorting" on page 264.

Procedure

1. Find two null-terminated strings you want to compare.
2. Send the characters to **IntlCompare**().

Example

This code compares two literal strings. It is intended as an example of how to call **IntlCompare**() rather than as good coding practice. Do not use literal strings in your code unless absolutely necessary.

```
P_CHAR      firstString = L"First string";
P_CHAR      secondString = L"Second string";
LocCompare(firstString, secondString, intlSortStyleDictionary);
```

Notes

The function prototype follows:

```
S32 EXPORTED IntlCompare(
      P_CHAR      pLeft,       // left string of comparison
      P_CHAR      pRight,      // right string of comparison
      LOCALE_ID   locale,      // Locale to use -- from golocale.h
      U32         style        // Collation style -- from isrstyle.h
);
```

The function returns:

- –1 when **pLeft** precedes **pRight** (left < right)
- 0 when **pLeft** is the same as **pRight** (left == right)
- 1 when **pLeft** follows **pRight** (left > right)
- **stsRequestNotSupported** if the locale or style is unsupported.

The following styles apply with sorting and comparing.

```
#define intlSortIgnoreCase          flag16      // (*) Ignore case
#define intlSortStyleDictionary     0x0001      // e.g. treat space as first
                                                   character
#define intlSortStylePhoneBook      0x0002      // e.g. ignore spaces altogether
```

Remember that **intlSortStyleDictionary** uses the JIS order for Level 1 kanji and various rules for other characters, while **intlSortStylePhoneBook** sorts in Unicode order, which is a good approximation of the radical and number of stroke sort orders used in Japanese dictionaries.

Related information

"Sorting strings" on page 292.

Sorting strings

This procedure sorts an array of null-terminated strings.

Prerequisite information

"Comparing and sorting" on page 264.

Procedure

1 Include ISR.H in your source file. Link INTL.LIB with your code by listing it in your makefile.

2 Encode the strings you want sorted as an array of null-terminated strings.

3 Pass in the array as a pointer to a string (type PP_CHAR) as a parameter to **Intl-Sort**().

Example

No example available.

Notes

The function prototype follows:

```
STATUS EXPORTED  IntlSort(
      PP_CHAR    ppString,    // list of strings to sort
      U32        count,       // number of strings in list
      LOCALE_ID  locale,      // Locale to use -- from golocale.h
      U32        style        // Collation style -- from isrstyle.h
);
```

See the Notes under "Comparing strings" on page 291 for details on the valid styles.

Related information

"Comparing strings" on page 291.

Handling line breaks

This procedure breaks lines of text, ensuring that no character that is not allowed to begin or end a line does so.

Prerequisite information

"Delimiting words" on page 289.

Procedure

1 Include ISR.H in your source file. Link INTL.LIB with your code by listing it in your makefile.

2 When displaying text that wraps, send the text stream to **IntlBreakLine**(). The result is returned in an INTL_BREAK_LINE structure.

3 Check the **breakAt** field of INTL_BREAK_LINE for the position of the line break.

4 If the position is at or before the start of a line, the function could not find an appropriate break point. You should provide a default method to handle this case. In most cases, you can just include all the characters that will fit on the line and break when necessary.

Example

The following code checks to see if the text in **pMetrics** fits on the current line. If the text does not fit, **LocNBreakLine**() is called to find an appropriate place to break the line. If the function returns a break position at the beginning of the line, no appropriate place was found to break the line, and hence the line need not be remeasured. Otherwise, the line is remeasured and the buffer updated with the correct line break information.

```
P_TEXT_LINE      pMetrics;
P_POSSIBLE_LINE  maybeMetrics;
TEXT_INDEX       savePos, pos, posInBuf;
CHAR             charBufMem[MAX_BUF_SIZE];
CHAR             *charBuf;
BOOLEAN          wordWrap;
```

```
INTL_BREAK_LINE   breakLine;
U32               style = intlStyleDefault;
. . .
if ((!TextFits(pMetrics, &maybeMetrics)) && wordWrap)
{
      savePos = pos;
      posInBuf = charBuf-charBufMem;
      LocNBreakLine(charBufMem, MAX_BUF_SIZE, posInBuf, &breakLine, style);
      if (posInBuf == breakLine.breakAt || breakLine.breakAt == 0)
            goto NoReMeasure;
      newBreakPos = pos - (posInBuf - breakLine.breakAt);
            goto Remeasure;
}
```

Notes

The **IntlBreakLine**() function requires a special structure as a parameter. When the function returns, the information on how to break the line is passed out in this structure. The following structure definition is in ISR.H:

```
typedef struct INTL_BREAK_LINE {
      U32  breakAt;     // position of line break
      U32  deleteThis;  // chars to delete from end of this line
      CHAR insertThis[intlBreakLineMaxInsert];
                        // chars to insert at end of this line
      U32  deleteNext;  // chars to delete from start of next line
      CHAR insertNext[intlBreakLineMaxInsert];
                        // chars to insert at start of next line
} INTL_BREAK_LINE, *P_INTL_BREAK_LINE;
```

The constant **intlBreakLineMaxInsert** is also defined in ISR.H. Its current value is 8.

Because Japanese simply breaks lines with no changes to the text stream, the fields **deleteThis**, **insertThis**, **deleteNext** and **deleteNext** are typically empty.

The current version of this function does not support hyphenation, although such support is planned. When hyphenation support is provided, and you use this function to check line breaks for romaji, the fields **deleteThis** and **deleteNext** are typically empty, while **insertThis** contains a hyphen.

The prototype for **IntlBreakLine**() is as follows:

```
S32 EXPORTED     IntlBreakLine(
      P_CHAR                 pString,    // Line to break
      U32                    pos,        // 1st char that won't fit
      P_INTL_BREAK_LINE      pBreak,     // Out: how to break it
      LOCALE_ID              locale,     // Locale to use -- from golocale.h
      U32                    style       // Break style -- from isrstyle.h
);
```

Related information

- "Delimiting words" on page 289.
- "Delimiting sentences" on page 290.

Using Japanese fonts

This procedure describes various methods you can use to specify a particular Japanese font.

Prerequisite information

- "Fonts" on page 250
- *PenPoint Architectural Reference,* Part 3, Chapters 25–26.

Procedure

There are a variety of ways your application can work with fonts.

- Use the default system fonts. Set the **group** field of SYSDC_FONT_ATTR structure to **sysDcGroupDefault** or **sysDcGroupUserInput**. The default fonts are Mincho for the system and Gothic for the user.
- Use **clsPopUpChoice** to display currently installed fonts in a scrolling window from which the user may select a font. See the example below for sample code.
- Set the drawing context with the desired font. The short font string for Mincho is MC55; for Gothic, the string is GT55. You can convert the string to a 16-bit font identifier with the **SysDcFontID**() function. Note that if you specify **sysDcGroupTransitional**, the group for Roman fonts, the system displays Japanese characters in the Mincho font. Similarly, the system displays Gothic characters when you specify the group as **sysDcGroupSansSerif**. See Chapter 26 of the *Architectural Reference* for details.

Examples

The first example is from the Hello World application. It sets the font to be the default user font by creating a drawing context in which the font group is **sysDcGroupUserInput**.

```
// Create a dc.
     ObjCallRet(msgNewWithDefaults, clsSysDrwCtx, &dn, s);
     data.dc = dn.object.uid;
. . .
// 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);
```

You can find this code in \2_0\PENPOINT\SDK\SAMPLE\HELLO\HELLOWIN.C.

The second example comes from the Clock Application. You can find the code in \2_0\PENPOINT\SDK\CLOCK\CLOCKAPP.C.

You can set up a TK_TABLE that allows the user select from the available fonts. To do so, include **tkPopupChoiceFont** as part of the flags field of a **clsPopupChoice.** This notifies the popup filed to get the list of available fonts from the system.

```
static const TK_TABLE_ENTRY clockDisplayCardEntries[] = {
. . .
      {hlpClkAppDisplayFont, 0, 0, 0, tkLabelStringId, 0, hlpClkAppDisplayFont},
      {fontPrune, 1, 0, tagFont, tkNoClient | tkPopupChoiceFont, clsPopupChoice,
            hlpClkAppDisplayFont},
. . .
      {pNull}
};
```

When the user taps Apply and this control is dirty, the Clock application must rewrite each of its labels in the new chosen font.

```
StsRetNoWarn(ReadControl(pArgs->win, tagFont, &value, 0, 0, pNull, false), s);
if (s == stsDirtyControl) {
      pInst->fontId = (U16) value;
      SysDcFontString( (U16) value, fontName);
      Dbg(Debugf(U_L("ClockApp: new font id is 0x%lx, \"%s\""), value,
fontName);)
      SetLabelFont(pInst->timeWin, pInst->fontId);
      SetLabelFont(pInst->amPmWin, pInst->fontId);
      SetLabelFont(pInst->dateWin, pInst->fontId);
      SetLabelFont(pInst->alarmWin, pInst->fontId);
      *pAppLayout = true;
```

SetLabelFont() is an internal function that updates the current font specs with the new font ID.

```
STATUS SetLabelFont(OBJECT win, U16 fontId) {
      SYSDC_FONT_SPEC spec;
      STATUS s;
      if (win) {
            ObjCallRet(msgLabelGetFontSpec, win, &spec, s);
            spec.id = fontId;
            ObjCallRet(msgLabelSetFontSpec, win, &spec, s);
      }
}  // SetLabelFont
```

Related information

- "Working with Shift-JIS in text files" on page 283.
- "Working with Unicode in source code" on page 284.

Supporting kana-kanji conversion

The easiest way to support KKC or RKC in your application is to create an instance of **clsIP** or **clsField**, because these classes already support character conversion.

This procedure describes how to make your own class the client of **clsCharTrans.** The easiest way to do this is to make your class a subclass of **clsGWin** or one of its descendants.

Prerequisite information

- "Kana-kanji conversion" on page 257.
- *PenPoint Architectural Reference*, Part 4, Chapter 32.

Procedure

1. Subclass **clsGWin** or one of its descendants.
2. Create your window as an instance of this subclass.
3. When appropriate, self-send the following messages to your window instance. Do not handle the messages. Rather, allow them to pass up to **clsGWin**, which sends the messages to its associated character translator. In PenPoint 2.0 Japanese, this translator is **clsKKCT**.

   Send **msgCharTransKeyEvent** whenever the user presses a key.

   Send **msgCharTransChar** whenever the user changes the buffer (for example, when the user inserts or deletes a character).

   Send **msgCharTransGesture** when the user makes a gesture. If the gesture is relevant to character translator, be prepared to handle **msgCharTransGet-ClientBuffer** (described below).

   Send **msgCharTransGoQuiescent** to abort any current translations.
4. Your class should respond to the following messages sent by the character translator.

   msgCharTransModifyBuffer, which contains information on how to update your buffer with the newly translated characters. Respond by updating your text buffer and user interface, including updating strong and weak highlighting. The character translator passes you a CHAR_TRANS_MODIFY structure.

   msgCharTransGetClientBuffer, which asks your window instance for some text. Pass the requested text to the character translator as part of a CHAR_TRANS_GET_BUF structure.

   msgCharTransProvideListXY, which asks your class where to put the character alternative list. Compute the coordinates so that the list pops up below the character.

   msgCharTransSetMark, which notifies your class that the translator is collecting characters. This message is sent for historical reasons. You can largely ignore it.

Notes

The following structures are used or required by the messages that the character translator sends your class.

The character translator sends the CHAR_TRANS_MODIFY structure to let the client know how to modify its buffer. The structure is sent with **msgCharTransModifyBuffer**. Note that the **markRelative**, **setActiveLenTo0**, **popupEvent**, and **user** fields are used internally and you generally do not need to worry about them.

```
typedef struct CHAR_TRANS_MODIFY {
      CHAR_TRANSLATOR        ct;                 // in: originating translator
      S32                    first;              // in: 1st char to modify
      S32                    length;             // in: # of chars to replace
      S32                    bufLen;             // in: # of chars in buf
      P_CHAR                 buf;                // in: chars to replace with
      CHAR_TRANS_HIGHLIGHT   highlight;
      U32                    markRelative:1,
                             setActiveLenTo0:1,
                             popupEvent:1,
                             reserved:29;        // unused (reserved)
      U32                    user;
      U32                    spare1;             // unused (reserved)
} CHAR_TRANS_MODIFY, *P_CHAR_TRANS_MODIFY;
```

The CHAR_TRANS_HIGHLIGHT structure contains information on how to highlight characters in the current buffer. The character translator sends you this structure as part of the P_ARGS for **msgCharTransModifyBuffer**.

```
typedef struct CHAR_TRANS_HIGHLIGHT {
      S32         weakStart;
      S32         weakLen;
      S32         strongStart;
      S32         strongLen;
      S32         oldWeakLen;
      S32         oldStrongStart;
      S32         oldStrongLen;
} CHAR_TRANS_HIGHLIGHT, *P_CHAR_TRANS_HIGHLIGHT;
```

The character translator requests part of its client's buffer with **msgCharTransGetClientBuffer**. The CHAR_TRANS_GET_BUF structure describes what portion of the buffer the character translator requires.

```
typedef struct CHAR_TRANS_GET_BUF   {
      P_CHAR      buf;
      S32         startPosition;
      S32         length;
      U32         reserved;
} CHAR_TRANS_GET_BUF, *P_CHAR_TRANS_GET_BUF;
```

When the user requests an alternative to the current translation, the translator requests the client to provide the location for the pop-up box by sending **msgCharTransProvideListXY**. The client fills in the requested information as part of a CHAR_TRANS_LIST_XY structure.

```
typedef struct CHAR_TRANS_LIST_XY   {
      S32         charPosition;     //    character position in client buffer
      XY32        xy;               //    root window coordinates for list
      U32         reserved1;
      U32         reserved2;
} CHAR_TRANS_LIST_XY, *P_CHAR_TRANS_LIST_XY;
```

Examples

The following code fragments illustrate different parts of the character translation protocol. The first fragment shows a typical response to the user pressing a key.

```
const P_INPUT_EVENT     pEvent,
const OBJECT            self,
P_KEY_DATA              pKeyData;
P_MY_TEXT_STRUCTURE     pText;
U16                     key;
CHAR_TRANS_CHAR         ctChar;
. . .
switch MsgNum(pEvent->devCode) {
      case MsgNum(msgKeyChar):
            pKeyData = (P_KEY_DATA)(pEvent->eventData);
            key = pKeyData->keyCode;
            ctKeyEvent.keyEvent = msgKeyChar;
            ctKeyEvent.keyCode = key;
            ctKeyEvent.scanCode = pKeyData->scanCode;
            ctKeyEvent.shiftState = pKeyData->shiftState;
            s = ObjectCall(msgCharTransKeyEvent,self,&ctKeyEvent);
            if ( s < stsOK)
                  s = HandleAnyKey(self, pText, pKeyData->shiftState,
                        key, pKeyData->repeatCount);
            }
            break;
```

The client self-sends **msgCharTransKeyEvent** each time the user presses a key. If the translator does not use the key, the message returns a status less than **stsOK**. In this case, the client responds by sending the key event to the internal function **HandleAnyKey**().

The second fragment is part of the **HandleAnyKey**() function. It shows a typical instance of sending **msgCharTransChar**. Remember that you send **msgCharTransChar** to **self** when a character in your buffer changes (for example, when the user inserts or deletes a character). This particular code responds to the user pressing the backspace key.

```
CHAR_TRANS_CHAR   ctChar;
U16               repeatCount;
switch(key)
{
...
      case uKeyBackSpace:
            ctChar.c = (CHAR)key;
            ctChar.position = r.first-1;
            ctChar.operation = ctDeleteChar;
            for (i = 0, sts==OK && i < repeatCount && ctChar.position >= 0; i++)
            {
                  ObjCallWarn(msgCharTransChar,self,&ctChar);
                  ctChar.position--;
...
}
```

The third fragment shows the entire handler for **msgTransCharGetClientBuf**. The character translator sends you this message to request a part of your buffer.

```
MsgHandlerArgType(SampleTextCharTransGetClientBuf, P_CHAR_TRANS_GET_BUF)
{
      const P_MY_TEXT_STRUCTURE pText = IDataDeref(pData, P_MY_TEXT_STRUCTURE);
      TEXT_BUFFER myText;
      myText.buf = pArgs->buf,
      myText.first = pArgs->startPosition;
      myText.length = text.bufLen = pArgs->length;
      ObjCallWarn(msgTextGetBuffer, pText->tb, &myText);
      return(stsOK);
      MsgHandlerParametersNoWarning;
}
```

Chapter 25 / Resources

Here are some texts that may help you during your localization process. Though not listed here, there are also consulting, translation, and marketing companies that can help you design, test, and translate your Japanese application and documentation.

Do's and Taboos Around the World, 2nd ed. Roger Axtell, John Wiley & Sons, 1990. A funny but informative guide to culturally acceptable and unacceptable behavior in various cultures.

Do's and Taboos Roger Axtell. John Wiley & Sons, 1989. Similar to *Do's and Taboos Around the World,* this book is aimed towards small businesses. Includes discussion of planning for international markets, pricing, shipping, managing and motivating distributors, and communication. Also devotes an entire chapter to Japan.

Electronic Handling of Japanese Text Ken Lunde. Describes how Japanese text is handled electronically. Includes a superb history of Japanese character encoding. Available through the Internet via anonymous FTP at MSI.UMN.EDU (128.101.24.1). The files, which include various utility programs, are in the /PUB/LUNDE directory.

Localization for Japan Apple Computer, Inc. Apple Developer Technical Publications, 1992. Contains a general overview of the Japanese computer market. Aimed at the non-programmer.

Kanji and Kana Wolfgang Hadamitzky and Mark Spahn. Charles E. Tuttle Company, 1981. A concise introduction to the Japanese writing system.

Soft Landing in Japan: A Market Entry Handbook for Software Companies American Electronics Association, 1990. Contact the AEA at 408-987-4200 for more information.

The Unicode Standard: Version 1.0, Volume 1 The Unicode Consortium. Addison-Wesley, 1991. Introduces the Unicode character encoding system.

The Unicode Standard: Version 1.0, Volume 2 The Unicode Consortium. Addison-Wesley, 1992. Shows glyphs for Chinese, Japanese, and Korean ideographs.

Chapter 26 / Japanese Character Set

The following pages list all the kanji defined by the 1990 JIS character set listed in Shift-JIS order. The Unicode value for each character is listed underneath each character as a 4-digit hexadecimal number.

We used PenPoint 2.0 Japanese to print this list with a standard 300 dots per inch (dpi) laser printer.

The fonts shipped with PenPoint 2.0 Japanese contain glyphs for all the characters listed. The characters that the handwriting recognition engine cannot recognize are marked with an asterisk(*).

Shift-JIS is ordered by a system called **ku-ten**. Most Japanese characters require two bytes of memory (half-width katakana characters, which require a single byte, are the exception).

Shift-JIS identifies the first byte with a string between ku 1 and ku 94, and the second byte with a string between ten 1 and ten 94. The kanji begin with ku 16 (hexadecimal 0x81).

Each ku is printed on a separate page that contains characters running from ten 1 to ten 94 for a given ku.

How the list was created

The list was created as follows:

1. A C program generated an RTF file containing the characters in the proper order and with the Unicode values.
2. The RTF file was passed into the DOS utility RTFTRIM. The result is a legal RTF file stripped of the RTF keywords that PenPoint's text component does not use.
3. The trimmed file was imported as a MiniText document and printed to a spool file. See Chapter 32 of *Part 4: PenPoint Development Tools Supplement* for information on printing to a spool file.
4. The spooler output was copied to a laser printer.

KU 16		1	2	3	4	5	6	7	8	9
		亜 4e9c	唖 5516	娃 5a03	阿 963f	哀 54c0	愛 611b	挨 6328	姶 59f6	逢 9022
10	葵 8475	茜 831c	穐 7a50	悪 60aa	握 63e1	渥 6e25	旭 65ed	葦 8466	芦 82a6	鯵 9bf5
20	梓 6893	圧 5727	斡 65a1	扱 6271	宛 5b9b	姐 59d0	虻 867b	飴 98f4	絢 7d62	綾 7dbe
30	鮎 9b8e	或 6216	粟 7c9f	袷 88b7	安 5b89	庵 5eb5	按 6309	暗 6697	案 6848	闇 95c7
40	鞍 978d	杏 674f	以 4ee5	伊 4f0a	位 4f4d	依 4f9d	偉 5049	囲 56f2	夷 5937	委 59d4
50	威 5a01	尉 5c09	惟 60df	意 610f	慰 6170	易 6613	椅 6905	為 70ba	畏 754f	異 7570
60	移 79fb	維 7dad	緯 7def	胃 80c3	萎 840e	衣 8863	謂 8b02	違 9055	遺 907a	医 533b
70	井 4e95	域 4ea5	域 57df	育 80b2	郁 90c1	磯 78ef	一 4e00	壱 58f1	溢 6ea2	逸 9038
80	稲 7a32	茨 8328	芋 828b	鰯 9c2f	員 5141	印 5370	咽 54bd	員 54e1	因 56e0	姻 59fb
90	引 5f15	飲 98f2	淫 6deb	胤 80e4	蔭 852d					

KU 17		1	2	3	4	5	6	7	8	9
		院 9662	陰 9670	隠 96a0	韻 97fb	吋 540b	右 53f3	宇 5b87	烏 70cf	羽 7fbd
10	迂 8fc2	雨 96e8	卯 536f	鵜 9d5c	窺 7aba	丑 4e11	碓 7893	臼 81fc	渦 6e26	嘘 5618
20	唄 5504	欝 6b1d	蔚 851a	鰻 9c3b	姥 59e5	厩 53a9	浦 6d66	瓜 74dc	閏 958f	噂 5642
30	云 4e91	運 904b	雲 96f2	荏 834f	餌 990c	叡 53e1	営 55b6	嬰 5b30	影 5f71	洩 6620
40	曳 66f3	栄 6804	永 6c38	泳 6cf3	洩 6d29	瑛 745b	盈 76c8	穎 7a4e	頴 9834	英 82f1
50	衛 885b	詠 8a60	鋭 92ed	液 6db2	疫 75ab	益 76ca	駅 99c5	悦 60a6	謁 8b01	越 8d8a
60	閲 95b2	榎 698e	厭 53ad	円 5186	園 5712	堰 5830	奄 5944	宴 5bb4	沿 5ef6	怨 6028
70	掩 63a9	援 63f4	沿 6cbf	演 6f14	炎 708e	艶 7114	煙 7159	燕 71d5	猿 733f	縁 7e01
80	艶 8276	苑 82d1	汚 8597	遠 9060	鉛 925b	鴛 9d1b	塩 5869	於 65bc	汚 6c5a	甥 7525
90	凹 51f9	央 592e	奥 5965	往 5f80	応 5fdc					

KU 18		1	2	3	4	5	6	7	8	9
		押 62bc	旺 65fa	横 6a2a	欧 6b27	殴 6bb4	王 738b	翁 7fc1	襖 8956	鴬 9d2c
10	鴎 9d0e	黄 9ec4	岡 5ca1	沖 6c96	荻 837b	億 5104	屋 5c4b	憶 61b6	臆 81c6	桶 6876
20	牡 7261	乙 4e59	俺 4ffa	卸 5378	恩 6069	温 6e29	穏 7a4f	音 97f3	下 4e0b	化 5316
30	仮 4eee	何 4f55	伽 4f3d	価 4fa1	佳 4f73	加 52a0	可 53ef	嘉 5609	夏 590f	嫁 5ac1
40	家 5bb6	寡 5be1	科 79d1	暇 6687	果 679c	架 67b6	歌 6b4c	河 6cb3	火 706b	珂 73c2
50	禍 798d	禾 79be	稼 7a3c	箇 7b87	花 82b1	苛 82db	茄 8304	荷 8377	華 83ef	菓 83d3
60	蝦 8766	課 8ab2	嘩 5629	貨 8ca8	迦 8fe6	過 904e	霞 971e	蚊 868a	俄 4fc4	峨 5ce8
70	我 6211	牙 7259	画 753b	臥 81e5	芽 82bd	蛾 86fe	賀 8cc0	雅 96c5	餓 9913	駕 99d5
80	介 4ecb	会 4f1a	解 89e3	回 56de	塊 584a	壊 58ca	廻 5efb	快 5feb	怪 602a	悔 6094
90	恢 6062	懐 61d0	戒 6212	拐 62d0	改 6539					

KU 19

		1	2	3	4	5	6	7	8	9
		魁 9b41	晦 6666	械 68b0	海 6d77	灰 7070	界 754c	皆 7686	絵 7d75	芥 82a5
10	蟹 87f9	開 958b	階 968e	貝 8c9d	凱 51f1	劾 52be	外 5916	咳 54b3	害 5bb3	崖 5d16
20	慨 6168	概 6982	涯 6daf	碍 788d	蓋 84cb	街 8857	該 8a72	鎧 93a7	骸 9ab8	浬 6d6c
30	馨 99a8	蛙 86d9	垣 57a3	柿 67ff	蛎 86ce	鈎 920e	劃 5283	嚇 5687	各 5404	廓 5ed3
40	拡 62e1	撹 64b9	格 683c	核 6838	殻 6bbb	獲 7372	確 78ba	穫 7a6b	覚 899a	角 89d2
50	赫 8d6b	較 8f03	郭 90ed	閣 95a3	隔 9694	革 9769	学 5b66	岳 5cb3	楽 697d	額 984d
60	顎 984e	掛 639b	笠 7b20	樫 6a2b	橿 6a7f	梶 68b6	鰍 9c0d	潟 6f5f	割 5272	喝 559d
70	恰 6070	括 62ec	活 6d3b	渇 6e07	滑 6ed1	葛 845b	褐 8910	轄 8f44	且 4e14	鰹 9c39
80	叶 53f6	椛 691b	樺 6a3a	鞄 9784	株 682a	兜 515c	竃 7ac3	蒲 84b2	釜 91dc	鎌 938c
90	噛 565b	鴨 9d28	栢 6822	茅 8305	萱 8431					

KU 20

		1	2	3	4	5	6	7	8	9
		粥 7ca5	刈 5208	苅 82c5	瓦 74e6	乾 4e7e	侃 4f83	冠 51a0	寒 5bd2	刊 520a
10	勘 52d8	勧 52e7	巻 5dfb	喚 559a	堪 582a	姦 59e6	完 5b8c	官 5b98	寛 5bdb	干 5e72
20	幹 5e79	患 60a3	感 611f	慣 6163	憾 61be	換 63db	敢 6562	柑 67d1	桓 6853	棺 68fa
30	款 6b3e	歓 6b53	汗 6c57	漢 6f22	澗 6f97	潅 6f45	環 74b0	甘 7518	監 76e3	看 770b
40	竿 7aff	管 7ba1	簡 7c21	緩 7de9	缶 7f36	翰 7ff0	肝 809d	艦 8266	莞 839e	観 89b3
50	諌 8acc	貫 8cab	還 9084	鑑 9451	間 9593	閑 9591	関 95a2	陥 9665	韓 97d3	館 9928
60	舘 8218	丸 4e38	含 542b	岸 5cb8	巌 5dcc	玩 73a9	癌 764c	眼 773c	岩 5ca9	翫 7feb
70	贋 8d0b	雁 96c1	頑 9811	顔 9854	願 9858	企 4f01	伎 4f0e	危 5371	喜 559c	器 5668
80	基 57fa	奇 5947	嬉 5b09	寄 5bc4	岐 5c90	希 5e0c	幾 5e7e	忌 5fcc	揮 63ee	机 673a
90	旗 65d7	既 65e2	期 671f	棋 68cb	棄 68c4					

KU 21

		1	2	3	4	5	6	7	8	9
		機 6a5f	帰 5e30	毅 6bc5	気 6c17	汽 6c7d	畿 757f	祈 7948	季 5b63	稀 7a00
10	紀 7d00	徽 5fbd	規 898f	記 8a18	貴 8cb4	起 8d77	軌 8ecc	輝 8f1d	飢 98e2	騎 9a0e
20	鬼 9b3c	亀 4e80	偽 507d	儀 5100	妓 5993	宜 5b9c	戯 622f	技 6280	擬 64ec	欺 6b3a
30	犠 72a0	疑 7591	祇 7947	義 7fa9	蟻 87fb	誼 8abc	議 8b70	掬 63ac	菊 83ca	鞠 97a0
40	吉 5409	吃 5403	喫 55ab	桔 6854	橘 6a58	詰 8a70	砧 7827	杵 6775	黍 9ecd	却 5374
50	客 5ba2	脚 811a	虐 8650	逆 9006	丘 4e18	久 4e45	仇 4ec7	休 4f11	及 53ca	吸 5438
60	宮 5bae	弓 5f13	急 6025	救 6551	朽 673d	求 6c42	汲 6c72	泣 6ce3	灸 7078	球 7403
70	究 7a76	窮 7aae	笈 7b08	級 7d1a	糾 7cfe	給 7d66	旧 65e7	牛 725b	去 53bb	居 5c45
80	巨 5de8	拒 62d2	拠 62e0	挙 6319	渠 6c20	虚 865a	許 8a31	距 8ddd	鋸 92f8	漁 6f01
90	禦 79a6	魚 9b5a	亨 4ca8	享 4eab	京 4eac					

KU 22		1	2	3	4	5	6	7	8	9
		供 4f9b	侠 4fa0	僑 50d1	兇 5147	競 7af6	共 5171	凶 51f6	協 5354	匡 5321
10	卿 537f	叫 53eb	喬 55ac	境 5883	峡 5ce1	強 5f37	彊 5f4a	怯 602f	恐 6050	恭 606d
20	挟 631f	教 6559	橋 6a4b	況 6cc1	狂 72c2	狭 72ed	矯 77ef	胸 80f8	脅 8105	興 8208
30	蕎 854e	郷 90f7	鏡 93e1	響 97ff	饗 9957	驚 9a5a	仰 4ef0	凝 51dd	尭 5c2d	暁 6681
40	業 696d	局 5c40	曲 66f2	極 6975	玉 7389	桐 6850	粁 7c81	僅 50c5	勤 52e4	均 5747
50	巾 5dfe	錦 9326	斤 65a4	欣 6b23	欽 6b3d	琴 7434	禁 7981	禽 79bd	筋 7b4b	緊 7dca
60	芹 82b9	菌 83cc	衿 887f	襟 895f	謹 8b39	近 8fd1	金 91d1	吟 541f	銀 9280	九 4e5d
70	倶 5036	句 53e5	区 533a	狗 72d7	玖 7396	矩 77e9	苦 82e6	躯 8eaf	駆 99c6	駈 99c8
80	駒 99d2	具 5177	愚 611a	虞 865e	喰 55b0	空 7a7a	偶 5076	寓 5bd3	遇 9047	隅 9685
90	串 4e32	櫛 6adb	釧 91e7	屑 5c51	屈 5c48					

KU 23

		1	2	3	4	5	6	7	8	9
		掘 6398	窟 7a9f	沓 6c93	靴 9774	轡 8f61	窪 7aaa	熊 718a	隈 9688	粂 7c82
10	栗 6817	繰 7e70	桑 6851	鍬 936c	勲 52f2	君 541b	薫 85ab	訓 8a13	群 7fa4	軍 8ecd
20	郡 90e1	卦 5366	袈 8888	祁 7941	係 4fc2	傾 50be	刑 5211	兄 5144	啓 5553	圭 572d
30	珪 73ea	型 578b	契 5951	形 5f62	径 5f84	恵 6075	慶 6176	慧 6167	憩 61a9	掲 63b2
40	携 643a	敬 656c	景 666f	桂 6842	渓 6e13	畦 7566	稽 7a3d	系 7cfb	経 7d4c	継 7d99
50	繋 7e4b	罫 7f6b	茎 830e	荊 834a	蛍 86cd	計 8a08	詣 8a63	警 8b66	軽 8efd	頚 981a
60	鶏 9d8f	芸 82b8	迎 8fce	鯨 9be8	劇 5287	戟 621f	撃 6483	激 6fc0	隙 9699	桁 6841
70	傑 5091	欠 6b20	決 6c7a	潔 6f54	穴 7a74	結 7d50	血 8840	訣 8a23	月 6708	件 4ef6
80	倹 5039	倦 5026	健 5065	兼 517c	券 5238	剣 5263	喧 55a7	圏 570f	堅 5805	嫌 5acc
90	建 5efa	憲 61b2	懸 61f8	拳 62f3	捲 6372					

KU 24

		1	2	3	4	5	6	7	8	9
		検 691c	権 6a29	牽 727d	犬 72ac	献 732e	研 7814	硯 786f	絹 7d79	県 770c
10	肩 80a9	見 898b	謙 8b19	賢 8ce2	軒 8ed2	遣 9063	鍵 9375	険 967a	顕 9855	験 9a13
20	鹸 9e78	元 5143	原 539f	厳 53b3	幻 5e7b	弦 5f26	減 6e1b	源 6e90	玄 7384	現 73fe
30	絃 7d43	舷 8237	言 8a00	諺 8afa	限 9650	乎 4e4e	個 500b	古 53e4	呼 547c	固 56fa
40	姑 59d1	孤 5b64	己 5df1	庫 5eab	弧 5f27	戸 6238	故 6545	枯 67af	湖 6e56	狐 72d0
50	糊 7cca	袴 88b4	股 80a1	胡 80e1	菰 83f0	虎 864e	誇 8a87	跨 8de8	鈷 9237	雇 96c7
60	顧 9867	鼓 9f13	五 4e94	互 4e92	伍 4f0d	午 5348	呉 5449	吾 543e	娯 5a2f	後 5f8c
70	御 5fa1	悟 609f	梧 68a7	檎 6a8e	瑚 745a	碁 7881	語 8a9e	誤 8aa4	護 8b77	醐 9190
80	乞 4e5e	鯉 9bc9	交 4ea4	佼 4f7c	侯 4faf	候 5019	倖 5016	光 5149	公 516c	功 529f
90	効 52b9	勾 52fe	厚 539a	口 53e3	向 5411					

KU 25		1	2	3	4	5	6	7	8	9
		后	喉	坑	垢	好	孔	孝	宏	工
		540e	5589	5751	57a2	597d	5b54	5b5d	5b8f	5de5
10	巧	巷	幸	広	庚	康	弘	恒	慌	抗
	5de7	5df7	5e78	5e83	5e9a	5eb7	5f18	6052	614c	6297
20	拘	控	攻	昂	晃	更	杭	校	梗	構
	62d8	63a7	653b	6602	6643	66f4	676d	6821	6897	69cb
30	江	洪	浩	港	溝	甲	皇	硬	稿	糠
	6c5f	6d2a	6d69	6e2f	6e9d	7532	7687	786c	7a3f	7ce0
40	紅	紘	絞	綱	耕	考	肯	肱	腔	膏
	7d05	7d18	7d5e	7db1	8015	8003	80af	80b1	8154	818f
50	航	荒	行	衡	講	貢	購	郊	酵	鉱
	822a	8352	884c	8861	8b1b	8ca2	8cfc	90ca	9175	9271
60	砿	鋼	閤	降	項	香	高	鴻	剛	劫
	783f	92fc	95a4	964d	9805	9999	9ad8	9d3b	525b	52ab
70	号	合	壕	拷	濠	豪	轟	麹	克	刻
	53f7	5408	58d5	62f7	6fe0	8c6a	8f5f	9eb9	514b	523b
80	告	国	穀	酷	鵠	黒	獄	漉	腰	甑
	544a	56fd	7a40	9177	9d60	9ed2	7344	6f09	8170	7511
90	忽	惚	骨	狛	込					
	5ffd	60da	9aa8	72db	8fbc					

KU 26		1	2	3	4	5	6	7	8	9
		此 6b64	頃 9803	今 4eca	困 56f0	坤 5764	墾 58be	婚 5a5a	恨 6068	懇 61c7
10	昏 660f	昆 6606	根 6839	梱 68b1	混 6df7	痕 75d5	紺 7d3a	艮 826e	魂 9b42	些 4e9b
20	佐 4f50	叉 53c9	唆 5506	嵯 5d6f	左 5de6	差 5dee	査 67fb	沙 6c99	瑳 7473	砂 7802
30	詐 8a50	鎖 9396	裟 88df	坐 5750	座 5ea7	挫 632b	債 50b5	催 50ac	再 518d	最 6700
40	哉 54c9	塞 585e	妻 59bb	宰 5bb0	彩 5f69	才 624d	採 63a1	栽 683d	歳 6b73	済 6e08
50	災 707d	采 91c7	犀 7280	砕 7815	砦 7826	祭 796d	斎 658e	細 7d30	菜 83dc	裁 88c1
60	載 8f09	際 969b	剤 5264	在 5728	材 6750	罪 7f6a	財 8ca1	冴 51b4	坂 5742	阪 962a
70	堺 583a	榊 698a	肴 80b4	咲 54b2	崎 5d0e	埼 57fc	碕 7895	鷺 9dfa	作 4f5c	削 524a
80	咋 548b	搾 643e	昨 6628	朔 6714	柵 67f5	窄 7a84	策 7b56	索 7d22	錯 932f	桜 685c
90	鮭 9bad	笹 7b39	匙 5319	冊 518a	刷 5237					

KU 27		1	2	3	4	5	6	7	8	9
		察 5bdf	拶 62f6	撮 64ae	擦 64e6	札 672d	殺 6bba	薩 85a9	雑 96d1	皐 7690
10	鯖 9bd6	捌 634c	錆 9306	鮫 9bab	皿 76bf	晒 6652	三 4e09	傘 5098	参 53c2	山 5c71
20	惨 60e8	撒 6492	散 6563	桟 685f	燦 71e6	珊 73ca	産 7523	算 7b97	纂 7e82	蚕 8695
30	讃 8b83	賛 8cdb	酸 9178	餐 9910	斬 65ac	暫 66ab	残 6b8b	仕 4ed5	仔 4ed4	伺 4f3a
40	使 4f7f	刺 523a	司 53f8	史 53f2	嗣 55e3	四 56db	士 58eb	始 59cb	姉 59c9	姿 59ff
50	子 5b50	屍 5c4d	市 5e02	師 5e2b	志 5fd7	思 601d	指 6307	支 652f	孜 5b5c	斯 65af
60	施 65bd	旨 65e8	枝 679d	止 6b62	死 6b7b	氏 6c0f	獅 7345	祉 7949	私 79c1	糸 7cf8
70	紙 7d19	紫 7d2b	肢 80a2	脂 8102	至 81f3	視 8996	詞 8a5e	詩 8a69	試 8a66	誌 8a8c
80	諮 8aee	資 8cc7	賜 8cdc	雌 96cc	飼 98fc	歯 6b6f	事 4e8b	似 4f3c	侍 4f8d	児 5150
90	字 5b57	寺 5bfa	慈 6148	持 6301	時 6642					

KU 28

		1	2	3	4	5	6	7	8	9
		次 6b21	滋 6ecb	治 6cbb	爾 723e	璽 74bd	痔 75d4	磁 78c1	示 793a	而 800c
10	耳 8033	自 81ea	蒔 8494	辞 8f9e	汐 6c50	鹿 9e7f	式 5f0f	識 8b58	鴫 9d2b	竺 7afa
20	軸 8ef8	宍 5b8d	雫 96eb	七 4e03	叱 53f1	執 57f7	失 5931	嫉 5ac9	室 5ba4	悉 6089
30	湿 6e7f	漆 6f06	疾 75be	質 8cea	実 5b9f	蔀 8500	篠 7be0	偲 5072	柴 67f4	芝 829d
40	屡 5c61	蕊 854a	縞 7e1e	舎 820e	写 5199	射 5c04	捨 6368	赦 8d66	斜 659c	煮 716e
50	社 793e	紗 7d17	者 8005	謝 8b1d	車 8eca	遮 906e	蛇 86c7	邪 90aa	借 501f	勺 52fa
60	尺 5c3a	杓 6753	灼 707c	爵 7235	酌 914c	釈 91c8	錫 932b	若 82e5	寂 5bc2	弱 5f31
70	惹 60f9	主 4e3b	取 53d6	守 5b88	手 624b	朱 6731	殊 6b8a	狩 72e9	珠 73e0	種 7a2e
80	腫 816b	趣 8da3	酒 9152	首 9996	儒 5112	受 53d7	呪 546a	寿 5bff	授 6388	樹 6a39
90	綬 7dac	需 9700	囚 56da	収 53ce	周 5468					

KU 29

		1	2	3	4	5	6	7	8	9
		宗 5b97	就 5c31	州 5dde	修 4fee	愁 6101	拾 62fe	洲 6d32	秀 79c0	秋 79cb
10	終 7d42	繍 7e4d	習 7fd2	臭 81ed	舟 821f	蒐 8490	衆 8846	襲 8972	讐 8b90	蹴 8e74
20	輯 8f2f	週 9031	酋 914b	酬 916c	集 96c6	醜 919c	什 4ec0	住 4f4f	充 5145	十 5341
30	従 5f93	戎 620e	柔 67d4	汁 6c41	渋 6e0b	獣 7363	縦 7e26	重 91cd	銃 9283	叔 53d4
40	夙 5919	宿 5bbf	淑 6dd1	祝 795d	縮 7e2e	粛 7c9b	塾 587e	熟 719f	出 51fa	術 8853
50	述 8ff0	俊 4fca	峻 5cfb	春 6625	瞬 77ac	竣 7ae3	舜 821c	駿 99ff	准 51c6	循 5faa
60	旬 65ec	楯 696f	殉 6b89	淳 6df3	準 6e96	潤 6f64	盾 76fe	純 7d14	巡 5de1	遵 9075
70	醇 9187	順 9806	処 51e6	初 521d	所 6240	暑 6691	曙 66d9	渚 6e1a	庶 5eb6	緒 7dd2
80	署 7f72	書 66f8	薯 85af	藷 85f7	諸 8af8	助 52a9	叙 53d9	女 5973	序 5e8f	徐 5f90
90	恕 6055	鋤 92e4	除 9664	傷 50b7	償 511f					

KU 30		1	2	3	4	5	6	7	8	9
		勝 52dd	匠 5320	升 5347	召 53ec	哨 54e8	商 5546	唱 5531	嘗 5617	奨 5968
10	妾 59be	娼 5a3c	宵 5bb5	将 5c06	小 5c0f	少 5c11	尚 5c1a	庄 5e84	床 5e8a	廠 5ee0
20	彰 5f70	承 627f	抄 6284	招 62db	掌 638c	捷 6377	昇 6607	昌 660c	昭 662d	晶 6676
30	松 677e	梢 68a2	樟 6a1f	樵 6a35	沼 6cbc	消 6d88	渉 6e09	湘 6e58	焼 713c	焦 7126
40	照 7167	症 75c7	省 7701	硝 785d	礁 7901	祥 7965	称 79f0	章 7ae0	笑 7b11	粧 7ca7
50	紹 7d39	肖 8096	菖 83d6	蒋 848b	蕉 8549	衝 885d	裳 88f3	訟 8a1f	証 8a3c	詔 8a54
60	詳 8a73	象 8c61	賞 8cde	醤 91a4	鉦 9266	鍾 937e	鐘 9418	障 969c	鞘 9798	上 4e0a
70	丈 4e08	丞 4e1e	乗 4e57	冗 5197	剰 5270	城 57ce	場 5834	壌 58cc	嬢 5b22	常 5e38
80	情 60c5	擾 64fe	条 6761	杖 6756	浄 6d44	状 72b6	畳 7573	穣 7a63	蒸 84b8	譲 8b72
90	醸 91b8	錠 9320	嘱 5631	埴 57f4	飾 98fe					

KU 31

		1	2	3	4	5	6	7	8	9
		拭 62ed	植 690d	殖 6b96	燭 71ed	織 7e54	職 8077	色 8272	触 89e6	食 98df
10	蝕 8755	辱 8fb1	尻 5c3b	伸 4f38	信 4fe1	侵 4fb5	唇 5507	娠 5a20	寝 5bdd	審 5be9
20	心 5fc3	慎 614e	振 632f	新 65b0	晋 664b	森 68ee	榛 699b	浸 6d78	深 6df1	申 7533
30	疹 75b9	真 771f	神 795e	秦 79e6	紳 7d33	臣 81e3	芯 82af	薪 85aa	親 89aa	診 8a3a
40	身 8eab	辛 8f9b	進 9032	針 91dd	震 9707	人 4eba	仁 4ec1	刃 5203	塵 5875	壬 58ec
50	尋 5c0b	甚 751a	尽 5c3d	腎 814e	訊 8a0a	迅 8fc5	陣 9663	靭 976d	笥 7b25	諏 8acf
60	須 9808	酢 9162	図 56f3	厨 53a8	逗 9017	吹 5439	垂 5782	帥 5e25	推 63a8	水 6c34
70	炊 708a	睡 7761	粋 7c8b	翠 7fe0	衰 8870	遂 9042	酔 9154	錐 9310	錘 9318	随 968f
80	瑞 745e	髄 9ac4	崇 5d07	嵩 5d69	数 6570	枢 67a2	趨 8da8	雛 96db	据 636e	杉 6749
90	椙 6919	菅 83c5	頗 9817	雀 96c0	裾 88fe					

KU 32		1	2	3	4	5	6	7	8	9
		澄 6f84	摺 647a	寸 5bf8	世 4e16	瀬 702c	畝 755d	是 662f	凄 51c4	制 5236
10	勢 52e2	姓 59d3	征 5f81	性 6027	成 6210	政 653f	整 6574	星 661f	晴 6674	棲 68f2
20	栖 6816	正 6b63	清 6e05	牲 7272	生 751f	盛 76db	精 7cbe	聖 8056	声 58f0	製 88fd
30	西 897f	誠 8aa0	誓 8a93	請 8acb	逝 901d	醒 9192	青 9752	静 9759	斉 6589	税 7a0e
40	脆 8106	隻 96bb	席 5e2d	惜 60dc	戚 621a	斥 65a5	昔 6614	析 6790	石 77f3	積 7a4d
50	籍 7c4d	績 7e3e	脊 810a	責 8cac	赤 8d64	跡 8de1	蹟 8e5f	碩 78a9	切 5207	拙 62d9
60	接 63a5	摂 6442	折 6298	設 8a2d	窃 7a83	節 7bc0	説 8aac	雪 96ea	絶 7d76	舌 820c
70	蝉 8749	仙 4ed9	先 5148	千 5343	占 5360	宣 5ba3	専 5c02	尖 5c16	川 5ddd	戦 6226
80	扇 6247	撰 64b0	栓 6813	栴 6834	泉 6cc9	浅 6d45	洗 6d17	染 67d3	潜 6f5c	煎 714e
90	煽 717d	旋 65cb	穿 7a7f	箭 7bad	線 7dda					

KU 33

	0	1	2	3	4	5	6	7	8	9
		纖 7e4a	羨 7fa8	腺 817a	舛 821b	船 8239	薦 85a6	詮 8a6e	賎 8cce	践 8df5
10	選 9078	遷 9077	銭 92ad	銑 9291	閃 9583	鮮 9bae	前 524d	善 5584	漸 6f38	然 7136
20	全 5168	禅 7985	繕 7e55	膳 81b3	糎 7cce	噌 564c	塑 5851	岨 5ca8	措 63aa	曾 66fe
30	曽 66fd	楚 695a	狙 72d9	疏 758f	疎 758e	礎 790e	祖 7956	租 79df	粗 7c97	素 7d20
40	組 7d44	蘇 8607	訴 8a34	阻 963b	遡 9061	鼠 9f20	僧 50e7	創 5275	双 53cc	叢 53e2
50	倉 5009	喪 55aa	壮 58ee	奏 594f	爽 723d	宋 5b8b	層 5c64	匝 531d	惣 60e3	想 60f3
60	捜 635c	掃 6383	挿 633f	掻 63bb	操 64cd	早 65e9	曹 66f9	巣 5de3	槍 69cd	槽 69fd
70	漕 6f15	燥 71e5	争 4e89	痩 75e9	相 76f8	窓 7a93	糟 7cdf	総 7dcf	綜 7d9c	聡 8061
80	草 8349	荘 8358	葬 846c	蒼 84bc	藻 85fb	装 88c5	走 8d70	送 9001	遭 906d	鎗 9397
90	霜 971c	騒 9a12	像 50cf	増 5897	憎 618e					

KU 34		1	2	3	4	5	6	7	8	9
		臓 81d3	蔵 8535	贈 8d08	造 9020	促 4fc3	側 5074	則 5247	即 5373	息 606f
10	捉 6349	束 675f	測 6e2c	足 8db3	速 901f	俗 4fd7	属 5c5e	賊 8cca	族 65cf	続 7d9a
20	卒 5352	袖 8896	其 5176	揃 63c3	存 5b58	孫 5b6b	尊 5c0a	損 640d	村 6751	遜 905c
30	他 4ed6	多 591a	太 592a	汰 6c70	詑 8a51	唾 553e	堕 5815	妥 59a5	惰 60f0	打 6253
40	柁 67c1	舵 8235	楕 6955	陀 9640	駄 99c4	騨 9a28	体 4f53	堆 5806	対 5bfe	耐 8010
50	岱 5cb1	帯 5e2f	待 5f85	怠 6020	態 614b	戴 6234	替 66ff	泰 6cf0	滞 6ede	胎 80ce
60	腿 817f	苔 82d4	袋 888b	貸 8cb8	退 9000	逮 902e	隊 968a	黛 9edb	鯛 9bdb	代 4ee3
70	台 53f0	大 5927	第 7b2c	醍 918d	題 984c	鷹 9df9	滝 6edd	瀧 7027	卓 5353	啄 5544
80	宅 5b85	托 6258	択 629e	拓 62d3	沢 6ca2	濯 6fef	琢 7422	託 8a17	鐸 9438	濁 6fc1
90	諾 8afe	茸 8338	凧 51c7	蛸 86f8	只 53ca					

KU 35

		1	2	3	4	5	6	7	8	9
		叩 53e9	但 4f46	達 9054	辰 8fb0	奪 596a	脱 8131	巽 5dfd	竪 7aea	辿 8fbf
10	棚 68da	谷 8c37	狸 72f8	鱈 9c48	樽 6a3d	誰 8ab0	丹 4e39	単 5358	嘆 5606	坦 5766
20	担 62c5	探 63a2	旦 65e6	歎 6b4e	淡 6de1	湛 6e5b	炭 70ad	短 77ed	端 7aef	箪 7baa
30	綻 7dbb	耽 803d	胆 80c6	蛋 86cb	誕 8a95	鍛 935b	団 56e3	壇 58c7	弾 5f3e	断 65ad
40	暖 6696	檀 6a80	段 6bb5	男 7537	談 8ac7	値 5024	知 77e5	地 5730	弛 5f1b	恥 6065
50	智 667a	池 6c60	痴 75f4	稚 7a1a	置 7f6e	致 81f4	蜘 8718	遅 9045	馳 99b3	築 7bc9
60	畜 755c	竹 7af9	筑 7b51	蓄 84c4	逐 9010	秩 79e9	窒 7a92	茶 8336	嫡 5ae1	着 7740
70	中 4e2d	仲 4ef2	宙 5b99	忠 5fe0	抽 62bd	昼 663c	柱 67f1	注 6ce8	虫 866b	衷 8877
80	註 8a3b	酎 914e	鋳 92f3	駐 99d0	樗 6a17	瀦 7026	猪 732a	苧 82e7	貯 8457	貯 8caf
90	丁 4e01	兆 5146	凋 51cb	喋 558b	寵 5bf5					

KU 36

		1	2	3	4	5	6	7	8	9
		帖 5e16	帳 5e33	庁 5e81	弔 5f14	張 5f35	5f6b	徴 5fb4	懲 61f2	挑 6311
10	暢 66a2	朝 671d	潮 6f6e	牒 7252	町 753a	眺 773a	聴 8074	脹 8139	腸 8178	蝶 8776
20	調 8abf	諜 8adc	超 8d85	跳 8df3	銚 929a	長 9577	頂 9802	鳥 9ce5	勅 52c5	捗 6357
30	直 76f4	朕 6715	沈 6c88	珍 73cd	賃 8cc3	鎮 93ae	陳 9673	津 6d25	墜 589c	椎 690e
40	槌 69cc	追 8ffd	鎚 939a	痛 75db	通 901a	塚 585a	栂 6802	掴 63b4	槻 69fb	佃 4f43
50	漬 6f2c	柘 67d8	辻 8fbb	蔦 8526	綴 7db4	鍔 9354	椿 693f	潰 6f70	坪 576a	壷 58f7
60	嬬 5b2c	紬 7d2c	爪 722a	吊 540a	釣 91e3	鶴 9db4	亭 4ead	低 4f4e	停 505c	偵 5075
70	剃 5243	貞 8c9e	呈 5448	堤 5824	定 5b9a	帝 5e1d	底 5e95	庭 5ead	廷 5ef7	弟 5f1f
80	悌 608c	抵 62b5	挺 633a	提 63d0	梯 68af	汀 6c40	碇 7887	禎 798e	程 7a0b	締 7de0
90	艇 8247	訂 8a02	諦 8ae6	蹄 8e44	逓 9013					

KU 37

	0	1	2	3	4	5	6	7	8	9
		邸 90b8	鄭 912d	釘 91d8	鼎 9f0e	泥 6ce5	摘 6458	擢 64e2	敵 6575	滴 6ef4
10	的 7684	笛 7b1b	適 9069	鏑 93d1	溺 6eba	哲 54f2	徹 5fb9	撤 64a4	轍 8f4d	迭 8fed
20	鉄 9244	典 5178	填 586b	天 5929	展 5c55	店 5e97	6dfb	纏 7e8f	甜 751c	貼 8cbc
30	転 8ee2	顛 985b	点 70b9	伝 4f1d	殿 6bbf	澱 6fb1	田 7530	電 96fb	兎 514e	吐 5410
40	堵 5835	塗 5857	妬 59ac	屠 5c60	徒 5f92	斗 6597	杜 675c	渡 6e21	登 767b	菟 83df
50	賭 8ced	途 9014	都 90fd	鍍 934d	砥 7825	砺 783a	努 52aa	度 5ea6	土 571f	奴 5974
60	怒 6012	倒 5012	党 515a	冬 51ac	凍 51cd	冬 5200	唐 5510	塔 5854	塘 5858	套 5957
70	宕 5b95	島 5cf6	嶋 5d8b	悼 60bc	投 6295	搭 642d	東 6771	桃 6843	梼 68bc	棟 68df
80	盗 76d7	淘 6dd8	湯 6e6f	涛 6d9b	灯 706f	燈 71c8	当 5f53	痘 75d8	祷 7977	等 7b49
90	答 7b54	筒 7b52	糖 7cd6	統 7d71	到 5230					

KU 38		1	2	3	4	5	6	7	8	9
		董	蕩	藤	討	謄	豆	踏	逃	踏
		8463	8569	85e4	8a0e	8b04	8c46	8e0f	9003	900f
10	鐙	陶	頭	騰	闘	働	動	同	堂	導
	9419	9676	982d	9a30	95d8	50cd	52d5	540c	5802	5c0e
20	憧	撞	洞	瞳	童	胴	萄	道	銅	峠
	61a7	649e	6d1e	77b3	7ae5	80f4	8404	9053	9285	5ce0
30	鴇	匿	得	徳	涜	特	督	禿	篤	毒
	9d07	533f	5f97	5fb3	6d9c	7279	7763	79bf	7be4	6bd2
40	独	読	栃	橡	凸	突	椴	届	鳶	苫
	72ec	8aad	6803	6a61	51f8	7a81	6934	5c4a	9cf6	82eb
50	寅	酉	瀞	噸	屯	惇	敦	沌	豚	遁
	5bc5	9149	701e	5678	5c6f	60c7	6566	6c8c	8c5a	9041
60	頓	呑	曇	鈍	奈	那	内	乍	凪	薙
	9813	5451	66c7	920d	5948	90a3	5185	4e4d	51ea	8599
70	謎	灘	捺	鍋	楢	馴	縄	畷	南	楠
	8b0e	7058	637a	934b	6962	99b4	7e04	7577	5357	6960
80	軟	難	汝	二	尼	弐	迩	匂	賑	肉
	8edf	96e3	6c5d	4e8c	5c3c	5f10	8fe9	5302	8cd1	8089
90	虹	廿	日	乳	入					
	8679	5eff	65e5	4e73	5165					

KU 39

		1	2	3	4	5	6	7	8	9
		如 5982	尿 5c3f	韮 97ee	任 4efb	妊 598a	忍 5fcd	認 8a8d	濡 6fe1	禰 79b0
10	祢 7962	寧 5be7	葱 8471	猫 732b	熱 71b1	年 5e74	念 5ff5	捻 637b	撚 649a	燃 71c3
20	粘 7c98	乃 4e43	廼 5efc	之 4e4b	埜 57dc	嚢 56a2	悩 60a9	濃 6fc3	納 7d0d	能 80fd
30	脳 8133	納 81bf	農 8fb2	覗 8997	蚤 86a4	巴 5df4	把 628a	播 64ad	覇 8987	杷 6777
40	波 6ce2	派 6d3e	琶 7436	派 7834	婆 5a46	罵 7f75	芭 82ad	馬 99ac	俳 4ff3	廃 5ec3
50	拝 62dd	排 6392	敗 6557	杯 676f	盃 76c3	牌 724c	背 80cc	肺 80ba	輩 8f29	配 914d
60	倍 500d	培 57f9	媒 5a92	梅 6885	楳 6973	煤 7164	狽 72fd	買 8cb7	売 58f2	賠 8ce0
70	陪 966a	這 9019	蝿 877f	秤 79e4	77e7	萩 8429	伯 4f2f	剥 5265	博 535a	拍 62cd
80	柏 67cf	泊 6cca	白 767d	箔 7b94	粕 7c95	舶 8236	薄 8584	迫 8feb	曝 66dd	漠 6f20
90	爆 7206	縛 7e1b	莫 83ab	駁 99c1	麦 9ea6					

KU 40		1	2	3	4	5	6	7	8	9
		函 51fd	箱 7bb1	硲 7872	箸 7bb8	肇 8087	筈 7b48	櫨 6ae8	幡 5e61	肌 808c
10	畑 7551	畠 7560	八 516b	鉢 9262	溌 6e8c	発 767a	醗 9197	髪 9aea	伐 4f10	罰 7f70
20	抜 629c	筏 7b4f	閥 95a5	鳩 9ce9	噺 567a	塙 5859	蛤 86e4	隼 96bc	伴 4f34	判 5224
30	半 534a	反 53cd	叛 53db	帆 5e06	搬 642c	斑 6591	板 677f	氾 6c3e	汎 6c4e	版 7248
40	犯 72af	班 73ed	畔 7554	繁 7e41	般 822c	藩 85e9	販 8ca9	範 7bc4	釆 91c6	煩 7169
50	頒 9812	飯 98ef	挽 633d	晩 6669	番 756a	盤 76e4	磐 78d0	蕃 8543	蛮 86ee	匪 532a
60	卑 5351	否 5426	妃 5983	庇 5e87	彼 5f7c	悲 60b2	扉 6249	批 6279	披 62ab	斐 6590
70	比 6bd4	泌 6ccc	諜 75b2	皮 76ae	碑 7891	長 79d8	緋 7dcb	罷 7f77	肥 80a5	被 88ab
80	誹 8ab9	費 8cbb	避 907f	非 975e	飛 98db	樋 6a0b	簸 7c38	備 5099	尾 5c3e	墜 5fae
90	枇 6787	毘 6bd8	琵 7435	眉 7709	美 7f8e					

KU 41		1	2	3	4	5	6	7	8	9
		鼻 9f3b	柊 67ca	稗 7a17	匹 5339	疋 758b	髭 9aed	彦 5f66	膝 819d	菱 83f1
10	肘 8098	弼 5f3c	必 5fc5	畢 7562	筆 7b46	逼 903c	桧 6867	姫 59eb	媛 5a9b	紐 7d10
20	百 767e	謬 8b2c	俵 4ff5	彪 5f6a	標 6a19	氷 6c37	漂 6f02	瓢 74e2	票 7968	表 8868
30	評 8a55	豹 8c79	廟 5edf	描 63cf	病 75c5	秒 79d2	苗 82d7	錨 9328	鋲 92f2	蒜 849c
40	蛭 86ed	鰭 9c2d	品 54c1	彬 5f6c	斌 658c	浜 6d5c	瀕 7015	貧 8ca7	賓 8cd3	頻 983b
50	敏 654f	瓶 74f6	不 4e0d	付 4ed8	埠 57e0	夫 592b	婦 5a66	富 5bcc	冨 51a8	布 5e03
60	府 5e9c	怖 6016	扶 6276	敷 6577	斧 65a7	普 666e	浮 6d6e	父 7236	符 7b26	腐 8150
70	膚 819a	芙 8299	譜 8b5c	負 8ca0	賦 8ce6	赴 8d74	阜 961c	附 9644	侮 4fae	撫 64ab
80	武 6b66	舞 821e	葡 8461	蕪 856a	部 90e8	封 5c01	楓 6953	風 98a8	葺 847a	蕗 8557
90	伏 4f0f	副 526f	復 5fa9	幅 5e45	服 670d					

KU 42		1	2	3	4	5	6	7	8	9
		福 798f	腹 8179	複 8907	覆 8986	淵 6df5	弗 5f17	払 6255	沸 6cb8	仏 4ecf
10	物 7269	鮒 9b92	分 5206	吻 543b	噴 5674	墳 58b3	憤 61a4	扮 626e	焚 711a	奮 596e
20	粉 7c89	糞 7cde	紛 7d1b	雰 96f0	文 6587	聞 805e	丙 4e19	併 4f75	兵 5175	塀 5840
30	幣 5e63	平 5e73	弊 5f0a	柄 67c4	並 4e26	蔽 853d	閉 9589	陛 965b	米 7c73	頁 9801
40	僻 50fb	壁 58c1	癖 7656	碧 78a7	別 5225	瞥 77a5	蔑 8511	箆 7b86	偏 504f	変 5909
50	片 7247	篇 7bc7	編 7de8	辺 8fba	返 8fd4	遍 904d	便 4fbf	勉 52c9	娩 5a29	弁 5f01
60	鞭 97ad	保 4fdd	舗 8217	鋪 92ea	圃 5703	捕 6355	歩 6b69	甫 752b	補 88dc	輔 8f14
70	穂 7a42	募 52df	墓 5893	慕 6155	戊 620a	暮 66ae	母 6bcd	簿 7c3f	菩 83e9	倣 5023
80	俸 4ff8	包 5305	呆 5446	報 5831	奉 5949	宝 5b9d	峰 5cf0	峯 5cef	崩 5d29	庖 5e96
90	抱 62b1	捧 6367	放 653e	方 65b9	朋 670b					

KU 43		1	2	3	4	5	6	7	8	9
		法 6cd5	泡 6ce1	烹 70f9	砲 7832	縫 7e2b	胞 80de	芳 82b3	萌 840c	蓬 84ec
10	蜂 8702	褒 8912	訪 8a2a	豊 8c4a	邦 90a6	鋒 92d2	飽 98fd	鳳 9cf3	鵬 9d6c	乏 4e4f
20	亡 4ea1	傍 508d	剖 5256	坊 574a	妨 59a8	帽 5e3d	忘 5fd8	忙 5fd9	房 623f	暴 66b4
30	望 671b	某 67d0	棒 68d2	冒 5192	紡 7d21	肪 80aa	膨 81a8	謀 8b00	貌 8c8c	貿 8cbf
40	鉾 927e	防 9632	吠 5420	頬 982c	北 5317	僕 50d5	卜 535c	墨 58a8	撲 64b2	朴 6734
50	牧 7267	睦 7766	穆 7a46	釦 91e6	勃 52c3	没 6ca1	殆 6b86	堀 5800	幌 5e4c	奔 5954
60	本 672c	翻 7ffb	凡 51e1	盆 76c6	摩 6469	磨 78e8	魔 9b54	麻 9ebb	埋 57cb	妹 59b9
70	昧 6627	枚 679a	毎 6bce	哩 54e9	槙 69d9	幕 5e55	膜 819c	枕 6795	鮪 9baa	柾 67fe
80	鱒 9c52	桝 685d	亦 4ea6	俣 4fe3	又 53c8	抹 62b9	末 672b	沫 6cab	迄 8fc4	侭 4fad
90	繭 7e6d	麿 9ebf	万 4e07	慢 6162	満 6e80					

KU 44		1	2	3	4	5	6	7	8	9
		漫 6f2b	蔓 8513	味 5473	未 672a	魅 9b45	巳 5df3	箕 7b95	岬 5cac	密 5bc6
10	蜜 871c	湊 6e4a	蓑 84d1	稔 7a14	脈 8108	妙 5999	粍 7c8d	民 6c11	眠 7720	務 52d9
20	夢 5922	無 7121	牟 725f	矛 77db	霧 9727	鵡 9d61	椋 690b	婿 5a7f	娘 5a18	冥 51a5
30	名 540d	命 547d	明 660e	盟 76df	迷 8ff7	銘 9298	鳴 9cf4	姪 59ea	牝 725d	滅 6ec5
40	免 514d	棉 68c9	綿 7dbf	緬 7dec	面 9762	麺 9eba	摸 6478	模 6a21	茂 8302	妄 5984
50	孟 5b5f	毛 6bdb	猛 731b	盲 76f2	網 7db2	耗 8017	蒙 8499	儲 5132	木 6728	黙 9ed9
60	目 76ee	杢 6762	勿 52ff	餅 9905	尤 5c24	戻 623b	籾 7c7e	貰 8cb0	問 554f	悶 60b6
70	紋 7d0b	門 9580	匁 5301	也 4e5f	冶 51b6	夜 591c	爺 723a	耶 8036	野 91ce	弥 5f25
80	矢 77e2	厄 5384	役 5f79	約 7d04	薬 85ac	訳 8a33	躍 8e8d	靖 9756	柳 67f3	薮 85ae
90	鑓 9453	愉 6109	愈 6108	油 6cb9	癒 7652					

KU 45		1	2	3	4	5	6	7	8	9
		諭 8aed	輸 8f38	唯 552f	佑 4f51	優 512a	勇 52c7	友 53cb	宥 5ba5	幽 5e7d
10	悠 60a0	憂 6182	揖 63d6	有 6709	柚 67da	湧 6e67	涌 6d8c	猶 7336	猷 7337	由 7531
20	祐 7950	裕 88d5	誘 8a98	遊 904a	邑 9091	郵 90f5	雄 96c4	融 878d	夕 5915	予 4e88
30	余 4f59	与 4e0e	誉 8a89	輿 8f3f	預 9810	傭 50ad	幼 5e7c	妖 5996	容 5bb9	庸 5eb8
40	揚 63da	揺 63fa	擁 64c1	曜 66dc	楊 694a	様 69d8	洋 6d0b	溶 6eb6	熔 7194	用 7528
50	窯 7aaf	羊 7f8a	耀 8000	葉 8449	蓉 84c9	要 8981	謡 8b21	踊 8e0a	遥 9065	陽 967d
60	養 990a	慾 617e	抑 6291	欲 6b32	沃 6c83	浴 6d74	翌 7fcc	翼 7ffc	淀 6dc0	羅 7f85
70	螺 87ba	裸 88f8	来 6765	莱 83b1	頼 983c	雷 96f7	洛 6d1b	絡 7d61	落 843d	酪 916a
80	乱 4e71	卵 5375	嵐 5d50	欄 6b04	濫 6feb	藍 85cd	蘭 862d	覧 89a7	利 5229	吏 540f
90	履 5c65	李 674e	梨 68a8	理 7406	璃 7483					

KU 46

		1	2	3	4	5	6	7	8	9
		痢 75e2	裏 88cf	裡 88e1	里 91cc	離 96e2	陸 9678	律 5f8b	率 7387	立 7acb
10	葎 844e	掠 63a0	略 7565	劉 5289	流 6d41	溜 6e9c	琉 7409	留 7559	硫 786b	粒 7c92
20	隆 9686	竜 7adc	龍 9f8d	侶 4fb6	慮 616e	旅 65c5	虜 865c	了 4e86	亮 4eae	僚 50da
30	両 4e21	凌 51cc	寮 5bee	料 6599	梁 6881	涼 6dbc	猟 731f	療 7642	瞭 77ad	稜 7a1c
40	糧 7ce7	良 826f	諒 8ad2	遼 907c	量 91cf	陵 9675	領 9818	力 529b	緑 7dd1	倫 502b
50	厘 5398	林 6797	淋 6dcb	燐 71d0	琳 7433	臨 81e8	輪 8f2a	隣 96a3	鱗 9c57	麟 9e9f
60	瑠 7460	塁 5841	涙 6d99	累 7d2f	類 985e	令 4ee4	伶 4f36	例 4f8b	冷 51b7	励 52b1
70	嶺 5dba	怜 601c	玲 73b2	礼 793c	苓 82d3	鈴 9234	隷 96b7	零 96f6	霊 970a	麗 9e97
80	齢 9f62	暦 66a6	歴 6b74	列 5217	劣 52a3	烈 70c8	裂 88c2	廉 5ec9	恋 604b	憐 6190
90	漣 6f23	煉 7149	簾 7c3e	練 7df4	聯 806f					

KU 47

		1	2	3	4	5	6	7	8	9
		蓮 84ee	連 9023	錬 932c	呂 5442	魯 9b6f	櫓 6ad3	炉 7089	賂 8cc2	路 8def
10	露 9732	労 52b4	婁 5a41	廊 5eca	弄 5f04	朗 6717	楼 697c	榔 6994	浪 6d6a	漏 6f0f
20	牢 7262	狼 72fc	篭 7bed	老 8001	聾 807e	蝋 874b	郎 90ce	六 516d	麓 9e93	禄 7984
30	肋 808b	録 9332	論 8ad6	倭 502d	和 548c	話 8a71	歪 6b6a	賄 8cc4	脇 8107	惑 60d1
40	枠 67a0	鷲 9df2	亙 4e99	亘 4e98	鰐 9c10	詫 8a6b	藁 85c1	蕨 8568	椀 6900	湾 6e7e
50	碗 7897	腕 8155								

KU 48

		1	2	3	4	5	6	7	8	9
		弌* 5f0c	丐 4e10	丕 4e15	个* 4e2a	丱 4e31	丶* 4e36	丼 4e3c	丿* 4e3f	乂 4e42
10	乖 4e56	乘 4e58	亂 4e82	亅* 4e85	豫 8c6b	亊* 4e8a	舒 8212	弍* 5f0d	于 4e8e	亞 4e9e
20	亟 4e9f	亠* 4ea0	亢 4ea2	亰* 4eb0	亳 4eb3	亶 4eb6	从* 4ece	仍 4ecd	仄 4ec4	仆 4ec6
30	仂 4ec2	仗 4ed7	仞 4ede	仭* 4eed	仟 4edf	价 4ef7	伉 4f09	佚 4f5a	估 4f30	佛 4f5b
40	佝 4f5d	佗 4f57	佇 4f47	佶 4f76	侈 4f88	侏 4f8f	侘 4f98	佻 4f7b	佩 4f69	佰 4f70
50	侑 4f91	佯 4f6f	來 4f86	侖 4f96	儘 5118	俔 4fd4	俟 4fdf	俎 4fce	俘 4fd8	俛 4fdb
60	俑 4fd1	俚 4fda	俐 4fd0	俤* 4fe4	俥* 4fe5	倚 501a	倨 5028	倔 5014	倪 502a	倥 5025
70	倅 5005	伜* 4f1c	俶 4ff6	倡 5021	倩 5029	倬 502c	俾 4ffe	俯 4fef	們 5011	倆 5006
80	偃 5043	假 5047	會 6703	偕 5055	偐* 5050	偈 5048	做 505a	偖* 5056	偬* 506c	偸* 5078
90	傀 5080	傚 509a	傅 5085	傴 50b4	傲 50b2					

KU 49		1	2	3	4	5	6	7	8	9
		僉 50c9	僊 50ca	傳 50b3	僂 50c2	僖 50d6	僞* 50de	僥 50e5	僭 50ed	僣 50e3
10	僮 50ee	價 50f9	僵 50f5	儉 5109	儁* 5101	儂 5102	儖* 5116	儕 5115	儔 5114	儚 511a
20	儡 5121	儺 513a	儷 5137	儼 513c	儻 513b	儿 513f	兀 5140	兒 5152	兌 514c	兔 5154
30	兢 5162	竸* 7af8	兩 5169	兪* 516a	兮 516e	冀 5180	冂* 5182	囘* 56d8	册* 518c	冉 5189
40	冏 518f	冑 5191	冓 5193	冕 5195	冖* 5196	冤 51a4	冦* 51a6	冢 51a2	冩* 51a9	冪 51aa
50	冫* 51ab	决* 51b3	冱 51b1	冲* 51b2	冰 51b0	况* 51b5	冽 51bd	凅 51c5	凉* 51c9	凛* 51db
60	几 51e0	處 8655	凩* 51e9	凭* 51ed	凰 51f0	凵 51f5	凾* 51fe	刄* 5204	刋* 520b	刔* 5214
70	刎 520e	刧* 5227	522a	刮 522e	刳 5233	刹* 5239	剏* 524f	剄 5244	剋 524b	剌 524c
80	剞 525e	剔 5254	剪 526a	剴 5274	剩 5269	剳* 5273	剿 527f	剽 527d	劍 528d	劔* 5294
90	劒* 5292	剱* 5271	劈 5288	劑 5291	辨 8fa8					

KU 50		1	2	3	4	5	6	7	8	9
		辦* 8fa7	劬 52ac	劭 52ad	劼 52bc	劵* 52b5	勁 52c1	勍 52cd	勗 52d7	勞 52de
10	勣 52e3	勦 52e6	飭 98ed	勠* 52e0	勳 52f3	勵 52f5	52f8	勹* 52f9	匆 5306	匈 5308
20	甸 7538	匍 530d	匐 5310	匏 530f	匕 5315	匚 531a	匣 5323	匯 532f	匱 5331	匳* 5333
30	匸* 5338	區 5340	卆* 5346	卅 5345	丗* 4e17	卉 5349	卍 534d	凖* 51d6	卞 535e	卩* 5369
40	卮 536e	夘* 5918	卻 537b	卷 5377	5382	厖 5396	厠* 53a0	厦* 53a6	厥 53a5	厮* 53ae
50	厰* 53b0	厶* 53b6	參 53c3	簒* 7c12	雙 96d9	叟 53df	曼 66fc	燮 71ee	叮 53ee	叨 53e8
60	叭 53ed	叺* 53fa	吁 5401	吽 543d	呀 5440	听 542c	吭 542d	吼 543c	吮 542e	吶 5436
70	吩 5429	吝 541d	呎 544e	咏* 548f	呵 5475	咎 548e	呟* 545f	呱 5471	呷 5477	呰 5470
80	咒 5492	呻 547b	咀 5480	呶 5476	咄 5484	咐 5490	咆 5486	哇 54c7	咢 54a2	咸 54b8
90	咥 54a5	咬 54ac	哄 54c4	哈 54c8	咨 54a8					

KU 51

		1	2	3	4	5	6	7	8	9
		咫 54ab	哂 54c2	咤* 54a4	咾 54be	咼 54bc	哘* 54d8	哥 54e5	哦 54e6	唏 550f
10	唔 5514	哽 54fd	哮 54ee	哭 54ed	哺 54fa	哢 54e2	唹 5539	啀 5540	啣 5563	啌* 554c
20	售 552e	啜 555c	啅 5545	啖 5556	啗 5557	唸 5538	唳 5533	啝* 555d	喙 5599	喀 5580
30	咯 54af	喊 558a	喟 559f	啻 557b	啾 557e	喘 5598	喞* 559e	單 55ae	啼 557c	喃 5583
40	喩* 55a9	喇 5587	喨 55a8	嗚 55da	嗅 55c5	嗟 55df	嗄 55c4	嗜 55dc	嗤 55e4	嗔 55d4
50	嘔 5614	嗷 55f7	嘖 5616	嗾 55fe	嗽 55fd	嘛 561b	嗹 55f9	噎 564e	噐* 5650	營 71df
60	嘴 5634	嘶 5636	嘲 5632	嘸 5638	噫 566b	噤 5664	嘯 562f	噬 566c	噪 566a	嚆 5686
70	嚀 5680	嚊* 568a	嚠* 56a0	嚔* 5694	嚏 568f	嚥 56a5	嚮 56ae	嚶 56b6	嚴 56b4	囂 56c2
80	嚼 56bc	囁 56c1	囃 56c3	囀 56c0	囈 56c8	囎* 56ce	囑 56d1	囓 56d3	囗 56d7	囮 56ee
90	囹 56f9	圀* 5700	囿 56ff	圄 5704	圉 5709					

KU 52		1	2	3	4	5	6	7	8	9
		圈 5708	國 570b	圍 570d	圓 5713	團 5718	圖 5716	嗇 55c7	圜 571c	圦* 5726
10	圷* 5737	圸* 5738	坎 574e	圻 573b	址 5740	坏 574f	坩 5769	埀* 57c0	垈* 5788	坡 5761
20	坿* 577f	垉* 5789	垓 5793	垠 57a0	垳* 57b3	垤 57a4	垪* 57aa	垰* 57b0	埃 57c3	埆 57c6
30	埔 57d4	埒 57d2	埓* 57d3	堊 580a	埖* 57d6	埣 57e3	堋 580b	堙 5819	堝 581d	塲* 5872
40	堡 5821	塢 5862	塋 584b	塰* 5870	毀 6bc0	塒 5852	堽 583d	塹 5879	墅 5885	墹* 58b9
50	墟 589f	墫 58ab	墺 58ba	壞 58de	墻* 58bb	墸* 58b8	墮 58ae	壅 58c5	壓 58d3	壑 58d1
60	壗* 58d7	壙 58d9	壘 58d8	壥* 58e5	壜* 58dc	壤 58e4	壟 58df	壯 58ef	壺 58fa	壹 58f9
70	壻* 58fb	壼 58fc	壽 58fd	夂* 5902	夊* 590a	夐* 5910	夛* 591b	梦* 68a6	夥 5925	夬 592c
80	夭 592d	夲* 5932	夸 5938	夾 593e	竒* 7ad2	奕 5955	奐 5950	奎 594e	奚 595a	奘 5958
90	奢 5962	奠 5960	奧 5967	奬* 596c	奩 5969					

KU 53

	0	1	2	3	4	5	6	7	8	9
		奸 5978	妁 5981	妝 599d	佞 4f5e	侫* 4fab	妣 59a3	妲 59b2	姆 59c6	姨 59e8
10	姜 59dc	妍 598d	姙* 59d9	姚 59da	娥 5a25	娟 5a1f	娑 5a11	娜 5a1c	娉 5a09	娚* 5a1a
20	婀 5a40	婬 5a6c	婉 5a49	娵 5a35	娶 5a36	婢 5a62	婪 5a6a	媚 5a9a	媼 5abc	媾 5abe
30	嫋 5acb	嫂 5ac2	媽 5abd	嫣 5ae3	嫗 5ad7	嫦 5ae6	嫩 5ae9	嫖 5ad6	嫺* 5afa	嫻 5afb
40	嬌 5b0c	嬋 5b0b	嬖 5b16	嬲 5b32	嫐* 5ad0	嬪 5b2a	嬶* 5b36	嬾 5b3e	孃 5b43	孅 5b45
50	孀 5b40	孑 5b51	孕 5b55	孚 5b5a	孛 5b5b	孥 5b65	孩 5b69	孰 5b70	孳 5b73	孵 5b75
60	學 5b78	斈* 6588	孺 5b7a	宀* 5b80	它 5b83	宦 5ba6	宸 5bb8	寃* 5bc3	寇 5bc7	寉* 5bc9
70	寔 5bd4	寐 5bd0	寤 5be4	實 5be6	寢 5be2	寞 5bde	寥 5be5	寫 5beb	寰 5bf0	寶 5bf6
80	寳* 5bf3	尅* 5c05	將 5c07	專 5c08	對 5c0d	尓* 5c13	尠* 5c20	尢 5c22	尨 5c28	尸 5c38
90	尹 5c39	屁 5c41	屆 5c46	屎 5c4e	屓* 5c53					

KU 54		1	2	3	4	5	6	7	8	9
		屐 5c50	屏 5c4f	孱 5b71	屬 5c6c	屮 5c6e	乢* 4e62	屶* 5c76	屹 5c79	岌 5c8c
10	岑 5c91	岔 5c94	妛* 599b	岫 5cab	岻* 5cbb	岶 5cb6	岼* 5cbc	岷 5cb7	峅* 5cc5	岾* 5cbe
20	峇 5cc7	峙 5cd9	峩* 5ce9	峽 5cfd	峺* 5cfa	峭 5ced	嶌* 5d8c	峪 5cea	崋 5d0b	崕* 5d15
30	崗 5d17	嵜* 5d5c	崟 5d1f	崛 5d1b	崑 5d11	崔 5d14	崢 5d22	崚 5d1a	崙 5d19	崘* 5d18
40	嵌 5d4c	嵒 5d52	嵎 5d4e	嵋 5d4b	嵬 5d6c	嵳* 5d73	嵶* 5d76	嶇 5d87	嶄 5d84	嶂 5d82
50	嶢 5da2	嶝 5d9d	嶬 5dac	嶮 5dae	嶽 5dbd	嶐* 5d90	嶷 5db7	嶼 5dbc	巉 5dc9	巍 5dcd
60	巓* 5dd3	巒 5dd2	巖 5dd6	巛* 5ddb	巫 5deb	已 5df2	巵* 5df5	帋* 5e0b	帚 5e1a	帙 5e19
70	帑 5e11	帛 5e1b	帶 5e36	帷 5e37	幄 5e44	幃 5e43	幀 5e40	幎 5e4e	幗 5e57	幔 5e54
80	幟 5e5f	幢 5e62	幤* 5e64	幇* 5e47	幵 5e75	并 5e76	幺* 5e7a	麼 9ebc	广* 5e7f	庠 5ea0
90	廁 5ec1	廂 5ec2	廈 5ec8	廐* 5ed0	廏* 5ecf					

KU 55

		1	2	3	4	5	6	7	8	9
		廖 5ed6	廣 5ee3	廝 5edd	廚 5eda	廛 5edb	廢 5ee2	廡 5ee1	廨 5ee8	廩 5ee9
10	廬 5eec	廱 5ef1	廳 5ef3	廰* 5ef0	廴* 5ef4	廸* 5ef8	廾 5efe	弃* 5f03	弉* 5f09	彝 5f5d
20	彜* 5f5c	弋 5f0b	弑* 5f11	弖* 5f16	弩 5f29	弭 5f2d	弸 5f38	彁* 5f41	彈 5f48	彌 5f4c
30	彎 5f4e	弯* 5f2f	彑* 5f51	彖 5f56	彗 5f57	彙 5f59	彡* 5f61	彭 5f6d	彳 5f73	彷 5f77
40	徃* 5f83	徂 5f82	彿 5f7f	徊 5f8a	很 5f88	徑 5f91	徇 5f87	從 5f9e	徙 5f99	徘 5f98
50	徠 5fa0	徨 5fa8	徭 5fad	徼 5fbc	忖 5fd6	忻 5ffb	忤 5fe4	忸 5ff8	忱 5ff1	忝 5fdd
60	悳* 60b3	忿 5fff	怡 6021	恠* 6060	怙 6019	怐 6010	怩 6029	怎 600e	怱* 6031	怛 601b
70	怕 6015	怫 602b	怦 6026	怏 600f	怺* 603a	恚 605a	恁 6041	恪 606a	恷* 6077	恟 605f
80	恊* 604a	恆 6046	恍 604d	恣 6063	恃 6043	恤 6064	恂 6042	恬 606c	恫 606b	恙 6059
90	悁 6081	悍 608d	惧* 60c7	悃 6083	悚 609a					

KU 56

		1	2	3	4	5	6	7	8	9
		悄 6084	悛 609b	悖 6096	悗 6097	悒 6092	悧* 60a7	悋* 608b	惡 60e1	悸 60b8
10	惠 60e0	惓 60d3	悴 60b4	忰* 5ff0	悽 60bd	惆 60c6	悵 60b5	惘 60d8	慍 614d	愕 6115
20	愆 6106	惶 60f6	惷 60f7	愀 6100	惴 60f4	惺 60fa	愃 6103	愡* 6121	惻 60fb	惱 60f1
30	愍 610d	愎 610e	慇 6147	愾 613e	愨 6128	愧 6127	慊 614a	愿 613f	愼* 613c	愬 612c
40	愴 6134	愽* 613d	慂* 6142	慄 6144	慳 6173	慷 6177	慘 6158	慙* 6159	慚 615a	慫 616b
50	慴 6174	慯* 616f	慥 6165	慱 6171	慟 615f	慝 615d	慓 6153	慵 6175	憙* 6199	憖 6196
60	憇* 6187	憬 61ac	憔 6194	憚 619a	憊 618a	憑 6191	憫 61ab	憮 61ae	懌 61cc	懊 61ca
70	應 61c9	懷 61f7	懈 61c8	懃 61c3	懆 61c6	憺 61ba	懋 61cb	罹 7f79	懍 61cd	懦 61e6
80	懣 61e3	懶 61f6	懺 61fa	懴* 61f4	懿 61ff	懽 61fd	懼 61fc	懾 61fe	戀 6200	戈 6208
90	戉 6209	戍 620d	戌 620c	戔 6214	戛 621b					

KU 57

		1	2	3	4	5	6	7	8	9
		戞* 621e	戡 6221	截 622a	戮 622e	戰 6230	戲 6232	戳 6233	扁 6241	扎 624e
10	扞 625e	扣 6263	扛 625b	扠 6260	扨* 6268	扼 627c	抂* 6282	抉 6289	找 627e	抒 6292
20	抓 6293	抖 6296	拔 62d4	抃 6283	抔 6294	拗 62d7	拑 62d1	抻 62bb	拏 62cf	拿 62ff
30	拆 62c6	擔 64d4	拈 62c8	拜 62dc	拌 62cc	拊 62ca	拂 62c2	拇 62c7	抛* 629b	拉 62c9
40	挌 630c	拮 62ee	拱 62f1	挧* 6327	挂 6302	挈 6308	拯 62ef	拵 62f5	捐 6350	挾 633e
50	捍 634d	搜 641c	捏 634f	掖 6396	掎 638e	掀 6380	掫 63ab	捶 6376	掣 63a3	掏 638f
60	掉 6389	掟 639f	掵* 63b5	捫 636b	捩 6369	掾 63be	揩 63e9	揀 63c0	揆 63c6	揣 63e3
70	揉 63c9	插 63d2	揶 63f6	揄 63c4	搖 6416	搴 6434	搆 6406	搓 6413	搦 6426	搶 6436
80	攝 651d	搗 6417	搨 6428	搏 640f	摧 6467	摯 646f	摶 6476	摎 644e	攪 652a	撕 6495
90	撓 6493	撥 64a5	撩 64a9	撈 6488	撼 64bc					

KU 58

		1	2	3	4	5	6	7	8	9
		據 64da	擒 64d2	擅 64c5	擇 64c7	撻 64bb	擘 64d8	擂 64c2	擱 64f1	擧* 64e7
10	舉 8209	擠 64e0	擡* 64e1	抬 62ac	擣 64e3	擯 64ef	攬 652c	擶* 64f6	擴 64f4	擲 64f2
20	擺 64fa	攀 6500	擽 64fd	攘 6518	攜 651c	攅* 6505	攤 6524	攣 6523	攫 652b	攴* 6534
30	攵* 6535	攷 6537	收 6536	攸 6538	畋 754b	效 6548	敖 6556	敕 6555	敍* 654d	敘 6558
40	敞 655e	敝 655d	敲 6572	數 6578	斂 6582	斃 6583	變 8b8a	斛 659b	斟 659f	斫 65ab
50	斷 65b7	旃 65c3	旆 65c6	旁 65c1	旄 65c4	旌 65cc	旒 65d2	旛 65db	旙* 65d9	无* 65e0
60	旡 65e1	旱 65f1	杲 6772	昊 660a	昃 6603	旻 65fb	杳 6773	昵 6635	昶 6636	昴 6634
70	昜 661c	晏 664f	晄* 6644	晉 6649	晁 6641	晞 665e	晝 665d	晤 6664	晧* 6667	晨 6668
80	晟 665f	晢 6662	晰 6670	暃* 6683	暈 6688	暎* 668e	暉 6689	暄 6684	暘 6698	暝 669d
90	曁* 66c1	暹 66b9	曉 66c9	暾 66be	暼* 66bc					

KU 59

	0	1	2	3	4	5	6	7	8	9
		曄 66c4	暸 66b8	曖 66d6	曚 66da	曠 66e0	昿* 663f	曦 66e6	曩 66e9	曰 66f0
10	曵* 66f5	曷 66f7	朏 670f	朖* 6716	朞* 671e	朦 6726	朧 6727	霸 9738	朮 672e	朿 673f
20	朶* 6736	杁* 6741	朸 6738	朷* 6737	杆 6746	杞 675e	杠 6760	杙 6759	杣* 6763	杤* 6764
30	枉 6789	杰 6770	枩* 67a9	杼 677c	杪 676a	枌 678c	枋 678b	枦* 67a6	枡* 67a1	枅 6785
40	枷 67b7	柯 67ef	枴 67b4	柬 67ec	枳 67b3	柩 67e9	枸 67b8	柤 67e4	柞 67de	柝 67dd
50	柢 67e2	柮 67ee	枹 67b9	柎 67ce	柆 67c6	柧 67e7	檜 6a9c	栞* 681e	框 6846	栩 6829
60	桀 6840	桍 684d	栲 6832	桎 684e	梳 68b3	栫 682b	桙* 6859	档* 6863	桷 6877	桿 687f
70	梟 689f	梏 688f	梭 68ad	梔 6894	條 689d	梛 689b	梃 6883	檮 6aae	梹* 68b9	桴 6874
80	梵 68b5	梠 68a0	梺* 68ba	椏 690f	梍* 688d	桾 687e	椁* 6901	棊* 68ca	椈 6908	棘 68d8
90	椢* 6922	椦* 6926	棡 68e1	椌 690c	棍 68cd					

KU 60		1	2	3	4	5	6	7	8	9
		棔 68d4	棧 68e7	棕 68d5	椶* 6936	椒 6912	椄 6904	棗 68d7	棣 68e3	椥 6925
10	棹 68f9	棠 68e0	棯 68ef	椨* 6928	椪 692a	椚* 691a	椣* 6923	椡* 6921	棆 68c6	楹 6979
20	楷 6977	楜 695c	楸 6978	楫 696b	楔 6954	楾* 697e	楮 696e	椹 6939	楴 6974	椽 693d
30	楙 6959	椰 6930	楡* 6961	楞 695e	楝 695d	榁* 6981	楪 696a	榲* 69b2	榮 69ae	槐 69d0
40	榿 69bf	槁 69c1	槓 69d3	榾 69be	槎 69ce	寨 5be8	槊 69ca	槝* 69dd	榻 69bb	槃 69c3
50	榧 69a7	樮* 6a2e	榑 6991	榠 69a0	榜 699c	榕 6995	榴 69b4	槞* 69de	槨 69e8	樂 6a02
60	樛 6a1b	槿 69ff	權 6b0a	槹* 69f9	槲 69f2	槧 69e7	樅 6a05	榱 69b1	樞 6a1e	槭 69ed
70	樔 6a14	槫 69eb	樊 6a0a	樒* 6a12	櫁* 6ac1	樣 6a23	樓 6a13	橄 6a44	樌* 6a0c	橲* 6a72
80	樶* 6a36	橸* 6a78	橇 6a47	橢 6a62	橙 6a59	橦 6a66	橈 6a48	樸 6a38	樢* 6a22	檐 6a90
90	檍 6a8d	檠 6aa0	檄 6a84	檢 6aa2	檣 6aa3					

KU 61

		1	2	3	4	5	6	7	8	9
		檗 6a97	蘗 8617	檻 6abb	櫃 6ac3	櫂 6ac2	檸 6ab8	檳 6ab3	檬 6aac	櫞 6ade
10	櫑 6ad1	櫟 6adf	檪* 6aaa	櫚 6ada	櫪 6aea	櫻 6afb	欅* 6b05	蘖* 8616	櫺 6afa	欒 6b12
20	欖 6b16	鬱 9b31	欟* 6b1f	欸 6b38	欷 6b37	盜 76dc	欹 6b39	飮* 98ee	歇 6b47	歃 6b43
30	歉 6b49	歐 6b50	歙 6b59	歔 6b54	歛 6b5b	歟 6b5f	歡 6b61	歸 6b78	歹 6b79	歿 6b7f
40	殀 6b80	殄 6b84	殃 6b83	殍 6b8d	殘 6b98	殕 6b95	殞 6b9e	殤 6ba4	殪 6baa	殫 6bab
50	殯 6baf	殲 6bb2	殱* 6bb1	殳 6bb3	殷 6bb7	殼 6bbc	毆 6bc6	毋 6bcb	毓 6bd3	毟* 6bdf
60	毬 6bec	毫 6beb	毳 6bf3	毯 6bef	麾 9ebe	氈 6c08	氓 6c13	气 6c14	氛 6c1b	氤 6c24
70	氣 6c23	汞 6c5e	汕 6c55	汢* 6c62	汪 6c6a	沂 6c82	沍 6c8d	沚 6c9a	沁 6c81	沛 6c9b
80	汾 6c7e	汨 6c68	汳 6c73	沒 6c92	沐 6c90	泄 6cc4	泱 6cf1	泓 6cd3	沽 6cbd	泗 6cd7
90	泅 6cc5	泝 6cdd	沮 6cae	沱 6cb1	沾 6cbe					

KU 62		1	2	3	4	5	6	7	8	9
		沺 6cba	泛 6cdb	泯 6cef	泙 6cd9	泪* 6cea	洟 6d1f	衍 884d	洶 6d36	洫 6d2b
10	洽 6d3d	洸 6d38	洙 6d19	洵 6d35	洳 6d33	洒 6d12	洌 6d0c	浣 6d63	涓 6d93	浤 6d64
20	浚 6d5a	浹 6d79	浙 6d59	涎 6d8e	涕 6d95	濤 6fe4	涅 6d85	淹 6df9	渕* 6e15	渊* 6e0a
30	涵 6db5	淇 6dc7	淦 6de6	涸 6db8	淆 6dc6	淬 6dec	淞 6dde	淌 6dcc	淨 6de8	淒 6dd2
40	淅 6dc5	淺 6dfa	淙 6dd9	淤 6de4	淕 6dd5	淪 6dea	淮 6dee	渭 6e2d	湮 6e6e	渮 6e2e
50	渙 6e19	湲 6e72	湟 6e5f	渾 6e3e	渣 6e23	湫 6e6b	渫 6e2b	湶* 6e76	湍 6e4d	渟 6e1f
60	湃 6e43	渺 6e3a	湎 6e4e	渤 6e24	滿 6eff	渝 6e1d	游 6e38	溂* 6e82	溪 6eaa	溘 6e98
70	滉 6ec9	溷 6eb7	滓 6ed3	溽 6ebd	溯 6eaf	滄 6ec4	溲 6eb2	滔 6ed4	滕 6ed5	溏 6e8f
80	溥 6ea5	滂 6ec2	溟 6e9f	潁 6f41	漑* 6f11	灌 704c	滬 6eec	滸 6ef8	滾 6efe	漿 6f3f
90	滲 6ef2	漱 6f31	滯 6eef	漲 6f32	滌 6ecc					

KU 63

		1	2	3	4	5	6	7	8	9
		漾 6f3e	漓 6f13	滷 6ef7	澆 6f86	潺 6f7a	潸 6f78	澁* 6f81	澀 6f80	潯 6f6f
10	潛 6f5b	濳* 6ff3	潭 6f6d	澂 6f82	潼 6f7c	潘 6f58	澎 6f8e	澑* 6f91	濂 6fc2	潦 6f66
20	澳 6fb3	澣 6fa3	澡 6fa1	澤 6fa4	澹 6fb9	濆 6fc6	澪 6faa	濟 6fdf	濕 6fd5	濬 6fec
30	濔 6fd4	濘 6fd8	濱 6ff1	濮 6fee	濛 6fdb	瀉 7009	瀋 700b	濺 6ffa	瀑 7011	瀁 7001
40	瀏 700f	濾 6ffe	瀛 701b	瀚 701a	潴* 6f74	瀝 701d	瀘 7018	瀟 701f	瀰 7030	瀾 703e
50	瀲 7032	灑 7051	灣 7063	炙 7099	炒 7092	炯 70af	烱* 70f1	炬 70ac	炸 70b8	炳 70b3
60	炮 70ae	烟* 70df	烋 70cb	烝 70dd	烙 70d9	焉 7109	烽 70fd	焜 711c	焙 7119	煥 7165
70	煕* 7155	熈* 7188	煦 7166	煢 7162	煌 714c	煖 7156	煬 716c	熏 718f	燻 71fb	熄 7184
80	熕* 7195	熨 71a8	熬 71ac	燗* 71d7	熹 71b9	熾 71be	燒 71d2	燉 71c9	燔 71d4	燎 71ce
90	燠 71e0	燬 71ec	燧 71e7	燵* 71f5	燼 71fc					

KU 64		1	2	3	4	5	6	7	8	9
		燹 71f9	燿 71ff	爍 720d	爐 7210	爛 721b	爨 7228	爭 722d	爬 722c	爰 7230
10	爲* 7232	爻 723b	爼* 723c	爿 723f	牀* 7240	牆 7246	牋 724b	牘 7258	牴 7274	牾 727e
20	犂* 7282	犁 7281	犇* 7287	犒 7292	犖 7296	犢 72a2	犧 72a7	犹* 72b9	犲* 72b2	狃 72c3
30	狆 72c6	狄 72c4	狎 72ce	狒 72d2	狢* 72e2	狠 72e0	狡 72e1	狹 72f9	狷 72f7	倏 500f
40	猗 7317	猊 730a	猜 731c	猖 7316	猝 731d	猴 7334	猯* 732f	猩 7329	猥 7325	猾 733e
50	獎 734e	獏* 734f	默 9ed8	獗 7357	獪 736a	獨 7368	獰 7370	獸 7378	獵 7375	獻 737b
60	獺 737a	珈 73c8	玳 73b3	珎* 73ce	玻 73bb	珀 73c0	珥 73e5	珮 73ee	珞 73de	璢* 74a2
70	琅 7405	瑯 746f	琥 7425	珸 73f8	琲 7432	琺 743a	瑕 7455	琿 743f	瑟 745f	瑙 7459
80	瑁 7441	瑜 745c	瑩 7469	瑰 7470	瑣 7463	瑪 746a	瑶* 7476	瑾 747e	璋 748b	璞 749e
90	璧 74a7	瓊 74ca	瓏 74cf	瓔 74d4	珱* 73f1					

KU 65		1	2	3	4	5	6	7	8	9
		瓠 74e0	瓣 74e3	瓧* 74e7	瓩 74e9	瓮 74ee	瓲* 74f2	瓰* 74f0	瓱* 74f1	瓸* 74f8
10	瓷 74f7	甄 7504	甃 7503	甅* 7505	甌 750c	甎* 750e	甍 750d	甕 7515	甓 7513	甞* 751e
20	甦 7526	甬 752c	甼* 753c	畄* 7544	畍* 754d	畊* 754a	畉* 7549	畛 755b	畆* 7546	畚 755a
30	畩* 7569	畤 7564	畧* 7567	畫 756b	畭* 756d	畸 7578	當 7576	疆 7586	疇 7587	畴* 7574
40	疊 758a	疉* 7589	疂* 7582	疔 7594	疚 759a	疝 759d	疥 75a5	疣 75a3	痂 75c2	疳 75b3
50	痃* 75c3	疵 75b5	疽 75bd	疸 75b8	疼 75bc	疱* 75b1	痍 75cd	痊 75ca	痒 75d2	痙 75d9
60	痣 75e3	痞 75de	痾 75fe	痿 75ff	痼 75fc	瘁 7601	痰 75f0	痺 75fa	痲 75f2	痳 75f3
70	瘋 760b	瘍 760d	瘉 7609	瘟 761f	瘧 7627	瘠 7620	瘡 7621	瘢 7622	瘤 7624	瘴 7634
80	瘰 7630	瘻* 763b	癇 7647	癈 7648	癆 7646	癜 765c	癘 7658	癡 7661	癢 7662	癨* 7668
90	癩 7669	癪 766a	癧* 7667	癬 766c	癰 7670					

KU 66		1	2	3	4	5	6	7	8	9
		癩 7672	癶* 7676	癸 7678	發 767c	皀* 7680	皃* 7683	皈 7688	皋 768b	皎 768e
10	皖 7696	皓 7693	皙 7699	皚 769a	皰 76b0	皴 76b4	皸 76b8	皹* 76b9	皺 76ba	盂 76c2
20	盍 76cd	盖* 76d6	盒 76d2	盞 76de	盡 76e1	盥 76e5	盧 76e7	盪 76ea	蘯* 862f	盻 76fb
30	眈 7708	眇 7707	眄 7704	眩 7729	眤* 7724	眞* 771e	眥 7725	眦* 7726	眛 771b	眷 7737
40	眸 7738	睇 7747	睚 775a	睨 7768	睫 776b	睛 775b	睥 7765	睿 777f	睾 777e	睹 7779
50	瞎 778e	瞋 778b	瞑 7791	瞠 77a0	瞞 779e	瞰 77b0	瞶 77b6	瞹* 77b9	瞿 77bf	瞼 77bc
60	瞽 77bd	瞻 77bb	矇 77c7	矍 77cd	矗 77d7	矚 77da	矜 77dc	矣 77e3	矮 77ee	矼 77fc
70	砌 780c	砒 7812	礦 7926	砠 7820	礪 792a	硅 7845	碎 788e	硴* 7874	碆 7886	硼 787c
80	碚 789a	碌 788c	碣 78a3	碵* 78b5	碪 78aa	碯* 78af	磑 78d1	磆* 78c6	磋 78cb	磔 78d4
90	碾 78be	碼 78bc	磅 78c5	磊 78ca	磬 78ec					

KU 67

		1	2	3	4	5	6	7	8	9
		磧 78e7	磚 78da	磽 78fd	磴 78f4	礇* 7907	礒 7912	礑 7911	礙 7919	礬 792c
10	礫 792b	祀 7940	祠 7960	祗 7957	祟 795f	祚 795a	祕 7955	祓 7953	祺 797a	祿 797f
20	禊 798a	禝* 799d	禧 79a7	齋 9f4b	禪 79aa	禮 79ae	禳 79b3	禹 79b9	禺 79ba	秉 79c9
30	秕 79d5	秧 79e7	秬 79ec	秡* 79e1	秣 79e3	稈 7a08	稍 7a0d	稘 7a18	稙 7a19	稠 7a20
40	稟 7a1f	禀* 7980	稱 7a31	稻 7a3b	稾* 7a3e	稷 7a37	穃* 7a43	穗 7a57	穉* 7a49	穡 7a61
50	穢 7a62	穩 7a69	龝* 9f9d	穰 7a70	穹 7a79	穽* 7a7d	窈 7a88	窗 7a97	窕 7a95	窘 7a98
60	窖 7a96	窩 7aa9	竈* 7ac8	窰* 7ab0	窶 7ab6	竅 7ac5	竄 7ac4	窿 7abf	邃 9083	竇 7ac7
70	竊 7aca	竍* 7acd	竏* 7acf	竕* 7ad5	竓* 7ad3	站 7ad9	竚* 7ada	竝* 7add	竡* 7ae1	竢* 7ae2
80	竦 7ae6	竭 7aed	竰* 7af0	笂* 7b02	笏 7b0f	笊 7b0a	笆 7b06	笳 7b33	笘 7b18	笙 7b19
90	笞 7b1e	笵 7b35	笨 7b28	笶* 7b36	筐 7b50					

KU 68

	0	1	2	3	4	5	6	7	8	9
		筺* 7b7a	笄 7b04	筍 7b4d	笋* 7b0b	筌 7b4c	筅 7b45	筵 7b75	筥 7b65	筴 7b74
10	筧 7b67	筰 7b70	筱 7b71	筬* 7b6c	筮 7b6e	箝 7b9d	箘 7b98	箟* 7b9f	箍 7b8d	箜 7b9c
20	箚* 7b9a	箋 7b8b	箒* 7b92	箏 7b8f	筝* 7b5d	箙 7b99	篋 7bcb	篁 7bc1	篌 7bcc	篏* 7bcf
30	箴 7bb4	篆 7bc6	篝 7bdd	篩 7be9	簑 7c11	簔* 7c14	篦 7be6	篥 7be5	籠 7c60	簀 7c00
40	簇 7c07	簓* 7c13	篳 7bf3	篷 7bf7	簗* 7c17	簍 7c0d	篶* 7bf6	簣 7c23	簧 7c27	簪 7c2a
50	簟 7c1f	簷 7c37	簫 7c2b	簽 7c3d	籌 7c4c	籃 7c43	籔 7c54	籏* 7c4f	籀 7c40	籐 7c50
60	籘* 7c58	籟 7c5f	籤 7c64	籖* 7c56	籥 7c65	籬 7c6c	籵 7c75	粃* 7c83	粐* 7c90	粤* 7ca4
70	粭* 7cad	粢 7ca2	粫* 7cab	粡 7ca1	粨 7ca8	粳 7cb3	粲 7cb2	粱 7cb1	粮* 7cae	粹 7cb9
80	粽 7cbd	糀* 7cc0	糅 7cc5	糂* 7cc2	糘* 7cd8	糒 7cd2	糜 7cdc	糢 7ce2	鬻 9b3b	糯 7cef
90	糲 7cf2	糴 7cf4	糶 7cf6	糺* 7cfa	紆 7d06					

KU 69

		1	2	3	4	5	6	7	8	9
		紂 7d02	紜 7d1c	紕 7d15	紊 7d0a	絅 7d45	絋* 7d4b	紮 7d2e	紲 7d32	紿 7d3f
10	紵 7d35	絆 7d46	絳 7d73	絖 7d56	絎 7d4e	絲 7d72	絨 7d68	絮 7d6e	絏 7d4f	絣 7d63
20	經 7d93	綉* 7d89	絛 7d5b	綏 7d8f	絽 7d7d	綛* 7d9b	綺 7dba	綮 7dae	綣 7da3	綵 7db5
30	緇 7dc7	綽 7dbd	綫* 7dab	總 7e3d	綢 7da2	綯 7daf	緜* 7ddc	綸 7db8	綟 7d9f	綰 7db0
40	緘 7dd8	緝 7ddd	緤* 7de4	緞 7dde	緻 7dfb	緲 7df2	緡 7de1	縅* 7e05	縊 7e0a	縣 7e23
50	縡 7e21	縒 7e12	縱 7e31	縟 7e1f	縉 7e09	縋 7e0b	縢 7e22	繆 7e46	繦* 7e66	縻 7e3b
60	縵 7e35	縹 7e39	繃 7e43	縷 7e37	縲 7e32	縺 7e3a	繧* 7e67	繝* 7e5d	繖 7e56	繞 7e5e
70	繙 7e59	繚 7e5a	繹 7e79	繪 7e6a	繩 7e69	繼 7e7c	繻 7e7b	纃* 7e83	緕* 7dd5	繽 7e7d
80	辮 8fae	繿* 7e7f	纈 7e88	纉* 7e89	續 7e8c	纒* 7e92	纐* 7e90	纓 7e93	纔 7e94	纖 7e96
90	纎* 7e8e	纛 7e9b	纜 7e9c	缸 7f38	缺 7f3a					

3 / JAPANESE LOCALIZATION

KU 70		1	2	3	4	5	6	7	8	9
		罅 7f45	罌 7f4c	罍 7f4d	罎* 7f4e	罐 7f50	网 7f51	罕 7f55	罔 7f54	罘 7f58
10	罟 7f5f	罠 7f60	罨 7f68	罩 7f69	罧 7f67	罸* 7f78	羂* 7f82	羆 7f86	羃 7f83	羈 7f88
20	羇 7f87	羌 7f8c	羔 7f94	羞 7f9e	羝 7f9d	羚 7f9a	羣* 7fa3	羯 7faf	羲 7fb2	羹 7fb9
30	羮* 7fae	羶 7fb6	羸 7fb8	譱* 8b71	翅 7fc5	翆* 7fc6	翊 7fca	翕 7fd5	翔 7fd4	翡 7fe1
40	翦 7fe6	翩 7fe9	翳 7ff3	翹 7ff9	飜* 98dc	耆 8006	耄 8004	耋 800b	耒 8012	耘 8018
50	耙 8019	耜 801c	耡 8021	耨 8028	耿 803f	耻* 803b	聊 804a	聆 8046	聒 8052	聘 8058
60	聚 805a	聟* 805f	聢* 8062	聨* 8068	聳 8073	聲 8072	聰 8070	聶 8076	聹 8079	聽 807d
70	聿 807f	肄 8084	肆 8086	肅 8085	肛 809b	肓 8093	肚 809a	肭 80ad	冐* 5190	肬* 80ac
80	胛 80db	胥 80e5	胙 80d9	胝 80dd	胄 80c4	胚 80da	胖 80d6	脉* 8109	胯 80ef	胱 80f1
90	脛 811b	脩 8129	脣 8123	脯 812f	腋 814b					

KU 71

		1	2	3	4	5	6	7	8	9
		隋 968b	腆 8146	脾 813e	腓 8153	腑 8151	胼 80fc	腱 8171	腮 816e	腥 8165
10	腦 8166	腴 8174	膃 8183	膈 8188	膊 818a	膀 8180	膂 8182	膠 81a0	膕 8195	膤* 81a4
20	膣 81a3	腟* 815f	膓* 8193	膩 81a9	膰 81b0	膵 81b5	膾 81be	膸* 81b8	膽 81bd	臀 81c0
30	臂 81c2	膺 81ba	臉 81c9	臍 81cd	臑 81d1	臙 81d9	臘 81d8	臈* 81c8	臚 81da	臟 81df
40	臠 81e0	臧 81e7	臺 81fa	臻 81fb	臾 81fe	舁 8201	舂 8202	舅 8205	與 8207	舊 820a
50	舍 820d	舐 8210	舖 8216	舩* 8229	舫 822b	舸 8238	舳 8233	艀 8240	艙 8259	艘 8258
60	艝* 825d	艚 825a	艟 825f	艤 8264	艢* 8262	艨 8268	艪* 826a	艫 826b	舮* 822e	艱 8271
70	艷 8277	艸 8278	艾 827e	芍 828d	芒 8292	芫 82ab	芟 829f	芻 82bb	芬 82ac	苡 82e1
80	苣 82e3	苟 82df	苒 82d2	苴 82f4	苳 82f3	苺 82fa	莓 8393	范 8303	苻 82fb	苹 82f9
90	苞 82de	茆 8306	苜 82dc	茉 8309	苙 82d9					

KU 72		1	2	3	4	5	6	7	8	9
		茵 8335	茴 8334	茖 8316	茲 8332	茱 8331	荀 8340	茹 8339	荐 8350	荅 8345
10	茯 832f	茫 832b	茗 8317	茘* 8318	莅* 8385	莚 839a	莪 83aa	莟* 839f	莢 83a2	莖 8396
20	茣* 8323	莎 838e	莇 8387	莊 838a	荼 837c	莵* 83b5	荳 8373	荵 8375	莠 83a0	莉 8389
30	莨 83a8	菴 83f4	萓 8413	菫 83cb	菎 83ce	菽 83fd	萃 8403	菘 83d8	萋 840b	菁 83c1
40	菷* 83f7	萇 8407	菠 83e0	菲 83f2	萍 840d	萢* 8422	萠* 8420	莽 83bd	萸 8438	蔆 8506
50	菻 83fb	葭 846d	萪* 842a	萼 843c	蕚* 855a	蒄* 8484	葷 8477	葫 846b	蒭* 84ad	葮 846e
60	蒂 8482	葩 8469	葆 8446	萬 842c	葯 846f	葹 8479	萵 8435	蓊 84ca	葢* 8462	蒹 84b9
70	蒿 84bf	蒟 849f	蓙* 84d9	蓍 84cd	蒻 84bb	蓚* 84da	蓐 84d0	蓁 84c1	蓆 84c6	蓖 84d6
80	蒡 84a1	蔡 8521	蓿 84ff	蓴 84f4	蔗 8517	蔘 8518	蔬 852c	蔟 851f	蔕 8515	蔔 8514
90	蓼 84fc	蕀 8540	蕣 8563	蕘 8558	蕈 8548					

KU 73

		1	2	3	4	5	6	7	8	9
		蕁 8541	蘂* 8602	蕋* 854b	蕕 8555	薀 8580	薤 85a4	薈 8588	薑 8591	薊 858a
10	薨 85a8	蕭 856d	薔 8594	薛 859b	藪 85ea	薇 8587	薜 859c	蕷 8577	蕾 857e	薐 8590
20	藉 85c9	薺 85ba	藏 85cf	薹 85b9	藐 85d0	藕 85d5	藝 85dd	藥 85e5	藜 85dc	藹 85f9
30	蘊 860a	蘓* 8613	蘋 860b	藾 85fe	藺 85fa	蘆 8606	蘢 8622	蘚 861a	蘰* 8630	蘿 863f
40	虍 864d	乕* 4e55	虔 8654	號 865f	虧 8667	虱 8671	蚓 8693	蚣 86a3	蚩 86a9	蚪 86aa
50	蚋 868b	蚌 868c	蚶 86b6	蚯 86af	蛄 86c4	蛆 86c6	蚰 86b0	蛉 86c9	蠣 8823	蚫* 86ab
60	蛔 86d4	蛞 86de	蛩 86e9	蛬 86ec	蛟 86df	蛛 86db	蛯* 86ef	蜒 8712	蜆 8706	蜈 8708
70	蜀 8700	蜃 8703	蛻 86fb	蜑 8711	蜉 8709	蜍 870d	蛹 86f9	蜊 870a	蜴 8734	蜿 873f
80	蜷 8737	蜻 873b	蜥 8725	蜩 8729	蜚 871a	蝠 8760	蝟 875f	蝸 8778	蝌 874c	蝎 874e
90	蝴 8774	蝗 8757	蝨 8768	蝮 876e	蝙 8759					

KU 74		1	2	3	4	5	6	7	8	9
		蝓 8753	蝣 8763	蝪 876a	蠅 8805	螢 87a2	螟 879f	螂 8782	螯 87af	蟋 87cb
10	螽 87bd	蟀 87c0	蟐* 87d0	雖 96d6	螫 87ab	蟄 87c4	螳 87b3	蟇* 87c7	蟆 87c6	螻 87bb
20	蟯 87ef	蟲 87f2	蟠 87e0	蠏* 880f	蠍 880d	蟾 87fe	蟶 87f6	蟷 87f7	蠎* 880e	蟒 87d2
30	蠑 8811	蠖 8816	蠕 8815	蠢 8822	蠡 8821	蠱 8831	蠶 8836	蠹 8839	蠧* 8827	蠻 883b
40	衄 8844	衂* 8842	衒 8852	衙 8859	衞* 885e	衢 8862	衫 886b	袁 8881	衾 887e	袞 889e
50	衵 8875	衽 887d	袵* 88b5	衲 8872	袂 8882	袗 8897	袒 8892	袮* 88ae	袙 8899	袢 88a2
60	袍 888d	袤 88a4	袰* 88b0	袿* 88bf	袱 88b1	裃* 88c3	裄* 88c4	裔 88d4	裘 88d8	裙 88d9
70	裝 88dd	裹 88f9	褂 8902	裼 88fc	裴 88f4	裨 88e8	裲 88f2	褄* 8904	褌 890c	褊 890a
80	褓 8913	襃* 8943	褞 891e	褥 8925	褪 892a	褫 892b	襁 8941	襄 8944	褻 893b	褶 8936
90	褸 8938	襌 894c	褝* 891d	襠 8960	襞 895e					

KU 75

		1	2	3	4	5	6	7	8	9
		襦 8966	襤 8964	襭 896d	襪 896a	襯 896f	襴 8974	襷* 8977	襾 897e	覃 8983
10	覈 8988	覊* 898a	覓 8993	覘 8998	覡 89a1	覩* 89a9	覦 89a6	覬 89ac	覯 89af	覲 89b2
20	覺 89ba	覽 89bd	覿 89bf	觀 89c0	觚 89da	觜 89dc	觝 89dd	觧* 89e7	觴 89f4	觸 89f8
30	訃 8a03	訖 8a16	訐 8a10	訌 8a0c	訛 8a1b	訝 8a1d	訥 8a25	訶 8a36	詁 8a41	詛 8a5b
40	詒 8a52	詆 8a46	詈 8a48	詼 8a7c	詭 8a6d	詬 8a6c	詢 8a62	誅 8a85	誂 8a82	誄 8a84
50	誨 8aa8	誡 8aa1	誑 8a91	誥 8aa5	誦 8aa6	誚 8a9a	誣 8aa3	諄 8ac4	諍 8acd	諂 8ac2
60	諚* 8ada	諫 8aeb	諳 8af3	諧 8ae7	諤 8ac4	諱 8af1	謔 8b14	諠 8ae0	諢 8ae2	諷 8af7
70	諞 8ade	諛 8adb	謌* 8b0c	謇 8b07	謚 8b1a	諡 8ae1	謖 8b16	謐 8b10	謗 8b17	謠 8b20
80	謳 8b33	鞫 97ab	謦 8b26	謫 8b2b	謾 8b3e	謨 8b28	譁 8b41	譌* 8b4c	譏 8b4f	譎 8b4e
90	證 8b49	譖 8b56	譛* 8b5b	譚 8b5a	譫 8b6b					

KU 76

		1	2	3	4	5	6	7	8	9
		譟	譬	譯	譴	譽	讀	讌	讎	讒
		8b5f	8b6c	8b6f	8b74	8b7d	8b80	8b8c	8b8e	8b92
10	讓	讖	讙	讚	谺*	豁	谿	豈	豌	豎
	8b93	8b96	8b99	8b9a	8c3a	8c41	8c3f	8c48	8c4c	8c4e
20	豐	豕	豢	豬	豸	豺	貂	貉	貅	貊
	8c50	8c55	8c62	8c6c	8c78	8c7a	8c82	8c89	8c85	8c8a
30	貍	貎*	貔	豼*	貘	戝*	貭*	貪	貽	貲
	8c8d	8c8e	8c94	8c7c	8c98	621d	8cad	8caa	8cbd	8cb2
40	貳	貮*	貶	賈	賁	賤	賣	賚	賽	賺
	8cb3	8cae	8cb6	8cc8	8cc1	8ce4	8ce3	8cda	8cfd	8cfa
50	賻	贄	贅	贊	贇	贏	贍	贐	齎	贓
	8cfb	8d04	8d05	8d0a	8d07	8d0f	8d0d	8d10	9f4e	8d13
60	賍*	贔	贖	赧	赭	赱*	赳	趁	趙	跂
	8ccd	8d14	8d16	8d67	8d6d	8d71	8d73	8d81	8d99	8dc2
70	趾	趺	跏	跚	跖	跌	跛	跋	跪	跫
	8dbe	8dba	8dcf	8dda	8dd6	8dcc	8ddb	8dcb	8dea	8deb
80	跟	跣	跼	踈*	踉	跿	踝	踞	踐	踟
	8ddf	8de3	8dfc	8e08	8e09	8dff	8e1d	8e1e	8e10	8e1f
90	蹂	踵	踰	踴	蹊					
	8e42	8e35	8e30	8e34	8e4a					

KU 77		1	2	3	4	5	6	7	8	9
		蹇 8e47	蹉 8e49	蹌 8e4c	蹐 8e50	蹈 8e48	蹙 8e59	蹤 8e64	蹠 8e60	踪* 8e2a
10	蹣 8e63	蹕 8e55	蹶 8e76	蹲 8e72	蹼 8e7c	躁 8e81	躇 8e87	躅 8e85	躄 8e84	躋 8e8b
20	躊 8e8a	躓 8e93	躑 8e91	躔 8e94	躙* 8e99	躪 8eaa	躡 8ea1	躬 8eac	躰* 8eb0	軆* 8ec6
30	躱* 8eb1	躾* 8ebe	軅* 8ec5	軈* 8ec8	軋 8ecb	軛 8edb	軣* 8ee3	軼 8efc	軻 8efb	軫 8eeb
40	軾 8efe	輊 8f0a	輅 8f05	輕 8f15	輒 8f12	輙* 8f19	輓 8f13	輜 8f1c	輟 8f1f	輛 8f1b
50	輌* 8f0c	輦 8f26	輳 8f33	輻 8f3b	輹 8f39	轅 8f45	轂 8f42	輾 8f3e	轌* 8f4c	轉 8f49
60	轆 8f46	轎 8f4e	轗 8f57	轜* 8f5c	轢 8f62	轣 8f63	轤 8f64	辜 8f9c	辟 8f9f	辣 8fa3
70	辭 8fad	辯 8faf	辷* 8fb7	迚* 8fda	迥 8fe5	迢 8fe2	迪 8fea	迯* 8fef	邇 9087	迴 8ff4
80	逅 9005	迹* 8ff9	迺 8ffa	逑 9011	逕 9015	逡 9021	逍 900d	逞 901e	逖 9016	逋 900b
90	逧* 9027	逶 9036	逵 9035	逹* 9039	迸 8ff8					

3 / JAPANESE LOCALIZATION

KU 78

	0	1	2	3	4	5	6	7	8	9
		遏 904f	遐 9050	遑 9051	遒 9052	逎* 900c	遉 9049	逾 903e	遖* 9056	遘 9058
10	遞 905e	遨 9068	遯 906f	遶 9076	隨 96a8	遲 9072	邂 9082	遽 907d	邁 9081	邀 9080
20	邊 908a	邉* 9089	邏 908f	邨* 90a8	邯 90af	邱 90b1	邵 90b5	郢 90e2	郤 90e4	扈 6248
30	郛 90db	鄂 9102	鄒 9112	鄙 9119	鄲 9132	鄰 9130	酊 914a	酖 9156	酘 9158	酣 9163
40	酥 9165	酩 9169	酳 9173	酲 9172	醋 918b	醉 9189	醂 9182	醢 91a2	醫 91ab	醯 91af
50	醪 91aa	醵 91b5	醴 91b4	醺 91ba	釀 91c0	釁 91c1	釉 91c9	釋 91cb	釐 91d0	釖* 91d6
60	釟* 91df	釡* 91e1	釛* 91db	釼* 91fc	釵 91f5	釶* 91f6	鈞 921e	釿 91ff	鈔 9214	鈬* 922c
70	鈕 9215	鈑 9211	鉞 925e	鉗 9257	鉅 9245	鉉 9249	鉤 9264	鉈 9248	銕 9295	鈿 923f
80	鉋 924b	鉐 9250	銜 929c	銖 9296	銓 9293	銛 929b	鉚 925a	鋏 92cf	銹* 92b9	銷 92b7
90	鋩 92e9	錏 930f	鋺 92fa	鍄* 9344	錮 932e					

KU 79

		1	2	3	4	5	6	7	8	9
		錙 9319	錢 9322	錚 931a	錣 9323	錺* 933a	錵 9335	錻* 933b	鍜 935c	鍠 9360
10	鍼 937c	鍮* 936e	鍖 9356	鎰 93b0	鎬 93ac	鎭* 93ad	鎔 9394	鎹* 93b9	鏖 93d6	鏗 93d7
20	鏨 93e8	鏥* 93e5	鏘 93d8	鏃 93c3	鏝 93dd	鏐 93d0	鏈 93c8	鏤 93e4	鐚* 941a	鐔 9414
30	鐓 9413	鐃 9403	鐇 9407	鐐 9410	鐶 9436	鐫 942b	鐵 9435	鐡* 9421	鐺 943a	鑁* 9441
40	鑒 9452	鑄 9444	鑛* 945b	鑠 9460	鑢 9462	鑞 945e	鑪 946a	鈩* 9229	鑰 9470	鑵 9475
50	鑷 9477	鑽 947d	鑚* 945a	鑼 947c	鑾 947e	钁 9481	鑿 947f	門 9582	閇* 9587	閊* 958a
60	閔 9594	閖* 9596	閘 9598	閙* 9599	閠* 95a0	閨 95a8	閧* 95a7	閭 95ad	閼 95bc	閻 95bb
70	閹 95b9	閾 95be	闊 95ca	濶* 6ff6	闃 95c3	闍 95cd	闌 95cc	闕 95d5	闔 95d4	闖 95d6
80	關 95dc	闡 95e1	闥 95e5	闢 95e2	阡 9621	阨 9628	阮 962e	阯 962f	陂 9642	陌 964c
90	陏 964f	陋 964b	陷 9677	陜 965c	陞 965e					

KU 80		1	2	3	4	5	6	7	8	9
		陝 965d	陟 965f	陦* 9666	陲 9672	陬 966c	隍 968d	隘 9698	隕 9695	隗 9697
10	險 96aa	隧 96a7	隱 96b1	隲* 96b2	隰 96b0	隴 96b4	隶* 96b6	隸 96b8	隹 96b9	雎 96ce
20	雋 96cb	雉 96c9	雍 96cd	襍* 894d	雜 96dc	霍 970d	雕 96d5	雹 96f9	霄 9704	霆 9706
30	霈 9708	霓 9713	霎 970e	霑 9711	霏 970f	霖 9716	霙 9719	霤 9724	霪 972a	霰 9730
40	霹 9739	霽 973d	霾 973e	靄 9744	靆 9746	靈 9748	靂 9742	靉 9749	靜 975c	靠 9760
50	靤* 9764	靦 9766	靨 9768	勒 52d2	靫* 976b	靱* 9771	靹* 9779	鞅 9785	靼 977c	鞁 9781
60	靺 977a	鞆* 9786	鞋 978b	鞏 978f	鞐* 9790	鞜 979c	鞨 97a8	鞦 97a6	鞣 97a3	鞳 97b3
70	鞴* 97b4	韃 97c3	韆 97c6	韈* 97c8	韋 97cb	韜 97dc	韭 97cd	齏 9f4f	韲* 97f2	竟 7adf
80	韶 97f6	韵* 97f5	頏 980f	頌 980c	頸 9838	頤 9824	頡 9821	頷 9837	頽* 983d	顆 9846
90	顏 984f	顋* 984b	顫 986b	顯 986f	顰 9870					

KU 81

		1	2	3	4	5	6	7	8	9
		顱 9871	顴 9874	顳 9873	颪* 98aa	颯 98af	颱 98b1	颶 98b6	飄 98c4	飃* 98c3
10	飆 98c6	飩 98e9	飫 98eb	餃 9903	餉 9909	餒 9912	餔 9914	餘 9918	餡 9921	餝* 991d
20	餞 991e	餤 9924	餠* 9920	餬 992c	餮 992e	餽 993d	餾 993e	饂* 9942	饉 9949	饅 9945
30	饐 9950	饋 994b	饑 9951	饒 9952	饌 994c	饕 9955	馗 9997	馘 9998	馥 99a5	馭 99ad
40	馮 99ae	馼* 99bc	駟 99df	駛 99db	駝 99dd	駘 99d8	駑 99d1	駭 99ed	駮 99ee	駱 99f1
50	駲* 99f2	駻 99fb	駸 99f8	騁 9a01	騏 9a0f	騅 9a05	駢 99e2	騙 9a19	騫 9a2b	騷 9a37
60	驅 9a45	驂 9a42	驀 9a40	驃 9a43	騾 9a3e	驕 9a55	驍 9a4d	驛 9a5b	驗 9a57	驟 9a5f
70	驢 9a62	驥 9a65	驤 9a64	驩 9a69	驫 9a6b	驪 9a6a	骭 9aad	骰 9ab0	骼 9abc	髀 9ac0
80	髏 9acf	髑 9ad1	髓 9ad3	體 9ad4	髞* 9ade	髟 9adf	髢* 9ae2	髣 9ae3	髦 9ae6	髯 9aef
90	髫 9aeb	髮 9aee	髴* 9af4	髱 9af1	髷 9af7					

KU 82		1	2	3	4	5	6	7	8	9
		髻 9afb	鬆 9b06	鬘 9b18	鬚 9b1a	鬟 9b1f	鬢 9b22	鬣 9b23	鬥 9b25	鬧 9b27
10	鬨 9b28	鬩 9b29	鬪* 9b2a	鬮 9b2e	鬯 9b2f	鬲 9b32	魄 9b44	魃 9b43	魏 9b4f	魍 9b4d
20	魎 9b4e	魑 9b51	魘 9b58	魴 9b74	鮓 9b93	鮃* 9b83	鮑 9b91	鮖* 9b96	鮗* 9b97	鮟* 9b9f
30	鮠 9ba0	鮨 9ba8	鮴* 9bb4	鯀 9bc0	鯊 9bca	鮹 9bb9	鯆 9bc6	鯏* 9bcf	鯑* 9bd1	鯒* 9bd2
40	鯣* 9be3	鯢 9be2	鯤 9be4	鯔 9bd4	鯡 9be1	鰺* 9c3a	鯲* 9bf2	鯱* 9bf1	鯰 9bf0	鰕* 9c15
50	鰔 9c14	鰉 9c09	鰓 9c13	鰌* 9c0c	鰆 9c06	鰈 9c08	鰒 9c12	鰊* 9c0a	鰄* 9c04	鰮* 9c2e
60	鰛* 9c1b	鰥 9c25	鰤 9c24	鰡 9c21	鰰* 9c30	鱇* 9c47	鰲 9c32	鱆 9c46	鰾 9c3e	鱚* 9c5a
70	鱠 9c60	鱧 9c67	鱶* 9c76	鱸 9c78	鳧 9ce7	鳬* 9cec	鳰* 9cf0	鴉 9d09	鴈 9d08	鳫* 9ceb
80	鴃 9d03	鴆 9d06	鴪* 9d2a	鴦 9d26	鶯 9daf	鴣 9d23	鴟 9d1f	鵄* 9d44	鴕 9d15	鴒 9d12
90	鵁 9d41	鴿 9d3f	鴾 9d3e	鵆* 9d46	鵈* 9d48					

KU 83

		1	2	3	4	5	6	7	8	9
		鵝 9d5d	鵞* 9d5e	鵤* 9d64	鵑 9d51	鵐* 9d50	鵙 9d59	鵲 9d72	鶉 9d89	鶇 9d87
10	鶫* 9dab	鵯 9d6f	鵺* 9d7a	鶚 9d9a	鶤 9da4	鶩 9da9	鶲 9db2	鷄* 9dc4	鷁 9dc1	鶻 9dbb
20	鶸 9db8	鶺 9dba	鷆* 9dc6	鷏 9dcf	鷂 9dc2	鷙 9dd9	鷓 9dd3	鷸 9df8	鷦 9de6	鷭 9ded
30	鷯 9def	鷽 9dfd	鸚 9e1a	鸛 9e1b	鸞 9e1e	鹵 9e75	鹹 9e79	鹽 9e7d	麁* 9e81	麈 9e88
40	麋 9e8b	麌 9e8c	麒 9e92	麕* 9e95	麑 9e91	麝 9e9d	麥 9ea5	麩 9ea9	麸* 9eb8	麪* 9eaa
50	麭 9ead	靡 9761	黌 9ecc	黎 9ece	黏 9ecf	黐 9ed0	黔 9ed4	黜 9edc	點 9ede	黝 9edd
60	黠 9ee0	黥 9ee5	黨 9ee8	黯 9eef	黴 9ef4	黶 9ef6	黷 9ef7	黹 9ef9	黻 9efb	黼 9efc
70	黽 9efd	鼇 9f07	鼈* 9f08	皷* 76b7	鼕 9f15	鼡* 9f21	鼬 9f2c	鼾 9f3e	齊 9f4a	齒 9f52
80	齔 9f54	齣 9f63	齟 9f5f	齠 9f60	齡 9f61	齦 9f66	齧 9f67	齬 9f6c	齪 9f6a	齷 9f77
90	齲 9f72	齶 9f76	龕 9f95	龜 9f9c	龠 9fa0					

KU 84

		1	2	3	4	5	6	7	8	9
		堯 582f	槇* 69c7	遙 9059	瑤 7464	凜 51dc	熙 7199			

Part 4 / PenPoint Development Tools Supplement

PENPOINT APPLICATION WRITING GUIDE / EXPANDED EDITION

PART 4 / DEVELOPMENT TOOLS SUPPLEMENT

Chapter 27 / Introduction

This manual contains updated information about the PenPoint™ development environment. It describes tools you use to create applications for the Japanese localization of the PenPoint operating system.

Much of the information in the *PenPoint Development Tools* manual is still accurate. Where information in the *PenPoint Development Tools* manual is outdated, this manual provides updates.This manual also describes new tools and utilities available in the PenPoint SDK 2.0 Japanese.

Consult the following sources for more information on the PenPoint development environment:

- *PenPoint Development Tools*
- *PenPoint Application Writing Guide: Expanded Edition*
- *About PenPoint 2.0 SDK*

To make information easy to find, this manual closely follows the organization of the original *PenPoint Development Tools* manual.

Organization of this supplement

This chapter, Introduction, describes the purpose and organization of this manual. Chapter 3, Running PenPoint on a PC, from the original *PenPoint Development Tools* has moved to a separate document called *Installing and Running PenPoint SDK 2.0.*

Chapter 28, Road Map, describes the general process of writing PenPoint applications. This chapter also points out which volumes of the PenPoint documentation will help you in this process.

Chapter 29, Creating Applications and Services, covers topics related to creating a PenPoint application or service.

Chapter 30, Debugging, describes PenPoint debugging techniques, including new tools for debugging PenPoint 2.0 Japanese applications.

Chapter 31, Tools and Utilities, describes various tools that you use to work with PenPoint directories, resource files, and bitmaps.

Chapter 28 / Road Map

This chapter describes the typical process of creating a PenPoint™ application. The steps involved are discussed below, and Figure 28-1 is a visual representation of part of the process. Although applications are explicitly discussed here, the process of creating a PenPoint service are nearly identical.

Creating a PenPoint application

Creating a PenPoint application or service typically follows these steps. Indicated after each step is where you can find more information on a particular process.

One time only tasks

You only need to perform the following steps once.

1 Install the PenPoint SDK 2.0 Japanese.
   - See *Installing and Running the PenPoint SDK 2.0 Japanese* for more information.

2 Learn PenPoint programming and user interface concepts. See the following for more information.
   - *PenPoint Application Writing Guide: Expanded Edition, Parts 1 and 6*
   - *PenPoint Development Tools*
   - *PenPoint User Interface Design Reference*

Preliminary design

You need to perform the following preliminary design tasks each time you create a PenPoint application or service.

1 Design your application taking advantage of PenPoint's extensive class library.
   - *PenPoint Architectural Reference*
   - *Part 2: PenPoint Internationalization Handbook*
   - *Part 3: PenPoint Japanese Localization Handbook*
   - *Part 5: PenPoint Architectural Reference Supplement*

2 Plan, design, and begin writing user documentation.
   - *Using PenPoint*
   - *New UI Features in PenPoint*

3 Plan, design, and begin producing supporting documents such as Help notebook documents, stationery, and sample documents.
   - *PenPoint Development Tools Supplement,* Chapter 29

4 If necessary, plan for translation and other localization services.

Creating an application

You take the following steps to create a PenPoint application. The steps here correspond to the steps in Figure 28-1.

1 Write C code, including source code (.C), header files (.H), resource files (.RC), a method table (METHODS.TBL), and a makefile (MAKEFILE). Name your resource file (one for each localization) to remind you of the localization the file corresponds to. For example, name your resource files JPN.RC and USA.RC for the Japanese and U.S. English localizations.
   - *PenPoint Architectural Reference*
   - *PenPoint Application Programming Interface*
   - *PenPoint Development Tools*
   - On-line header files in \2_0\PENPOINT\SDK\INC

2 Compile and link your code. This involves compiling source code, resource files, and your method table, as shown in Figure 28-1. You then link your object files with PenPoint libraries.
   - *PenPoint Application Writing Guide: Expanded Edition*
   - *PenPoint Development Tools*
   - Compiler documentation

Typically, use a makefile based on one of the sample application makefiles to coordinate steps 3 and 4.

3 Build your PenPoint application in its own directory underneath the PenPoint application directory (\2_0\PENPOINT\APP) or service directory (\2_0\PENPOINT\SERVICE). You must put in your application directory the necessary executable file (.EXE), dynamic link libraries (.DLL), compiled resource files (.RES), dependency identifier file (.DLC), and supporting documents.
   - *PenPoint Development Tools*

Different releases of PenPoint use different root directories. For example, PenPoint 1.0 uses \1_01 as its root.

4 Stamp your application directory with PenPoint information, including the application name, the file type (application, service, font, and so on), and the linker name.
   - *PenPoint Development Tools*

Preparing for distribution

As you prepare for distribution, remember to take the following steps to

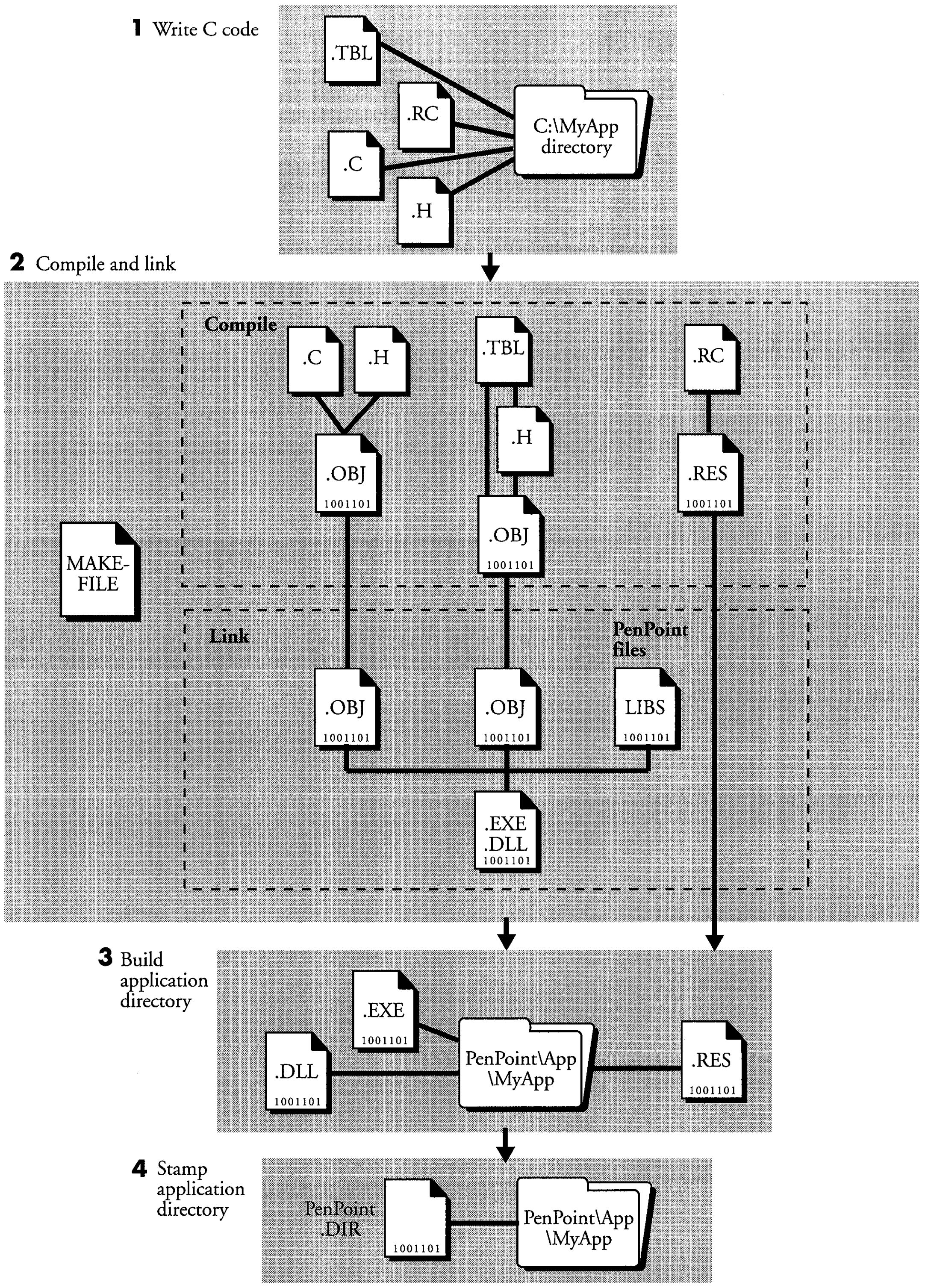
1 Contact GO Technical Services to register your classes.

2 Verify user documentation, translations, and other localization issues
   - *Part 2: PenPoint Internationalization Handbook*, Chapter 18

3 Create distribution disks.
   - *PenPoint Development Tools Supplement*, Chapter 29

Creating an application

FIGURE 28-1



Chapter 29 / Creating Applications and Services

This chapter describes the process of compiling, linking, and preparing your application or service to work under the PenPoint™ operating system.

Overview

You typically take the following steps to create a PenPoint application. Although this discussion centers on applications, the process of creating services is similar.

1. Compile your source code, resource files, and method table.
2. Link your object code with PenPoint libraries to create an executable image and dynamic link libraries.
3. Build your application by placing the executable image, dynamic link libraries, resource files, and supporting files in a directory within the PenPoint application directory. In PenPoint 2.0 Japanese, the application directory is \2_0\PENPOINT\APP.
4. Stamp your application directory and the PenPoint application directory with PenPoint file names and attributes.

Rather than performing all these steps manually, you typically create a makefile that your compiler uses to perform each of these steps. The sample applications included with the SDK all build on three standard makefiles that work with the WATCOM WMAKE utility.

- SDEFINES.MIF contains standard definitions used to compile and link PenPoint applications and services.
- SRULES.MIF contains standard rules for creating PenPoint applications.
- SVCRULES.MIF contains standard rules for creating PenPoint services.

Use these makefiles as building blocks. Namely, define variables and add rules for your own project, but do not make any changes to the sample makefiles. Probably the easiest way to write your own makefile is to modify one that comes with a sample application such as Tic-Tac-Toe.

So that you can see the steps involved in creating a typical PenPoint application, the section "Makefiles" on page 382 steps through the most important parts of SRULES.MIF.

Where to put your files

Don't use the SDK trees (for example, \2_0\PENPOINT\SDK) for your own projects.

GO reserves the right to change the organization of the SDK trees. Also, GO may provide tools for moving, copying or otherwise manipulating these trees. If you store your projects in the SDK trees, we cannot guarantee that these tools will work, nor can we guarantee that they will preserve your sources.

You might use a directory in your root directory such as \MYAPP or in a special directory containing all your applications such as \APPS\MYAPP.

Makefiles

Most of the complexity of creating a PenPoint application is contained in the file SRULES.MIF. Stepping through the key parts of this file is a good way to understand the steps involved in creating an application.

Compiling

First, the makefile compiles your source code, method table, and resource files. Compiler errors are saved in files with a .ERR extension. After the makefile finishes creating your application, list all the .ERR files in your directory to make sure there are no compiler errors.

These commands in the makefile compile your source code, method table, and resource files.

```
.c.obj:
      set WCC386=$(WCC386)
      wcc386p /Fo$*.obj $*.c > $*.err
      type $*.err
# Method table
.tbl.obj:
      set WCC=$(WCC)
      set WCC386=$(WCC386)
      $(PENPOINT_PATH)\sdk\util\clsmgr\mt $(MT_FLAGS) $< -Fo=$*.obj > $*.err
      type $*.err
# Resource compiler file
.rc.res:
      $(RC) $*.rc > $*.err
      type $*.err
```

Some variables that are common to all projects are defined in SDEFINES.MIF. For example, the variable WCC386 lists all the compiler flags required to create the appropriate object files.

Sometimes old object files get linked to your new project. This happens when the compiler encounters errors in your updated source code and aborts compilation. Type WMAKE CLEAN to delete old object files during your compile, test, debug, and recompile cycles.

Linking

After the source code is compiled into object code, it must be linked with PenPoint libraries. The makefile accomplishes this by creating a temporary WLINK file. The **%create** command creates this file, and the **%append** command adds the required lines to the file.

These lines link object files with PenPoint libraries to create executable code or dynamic link libraries.

```
$(APP_DIR) : .SYMBOLIC
      mkdir $(APP_DIR)
# The following complexity is needed to build a WLINK command file.
$(APP_DIR)\$(PROJ).exe : $(APP_DIR) $(EXE_OBJS)
      %create $(PROJ).eln
      %append $(PROJ).eln     SYSTEM PenPoint
      %append $(PROJ).eln     NAME $(APP_DIR)\$(PROJ).exe
      %append $(PROJ).eln$(LINK_DEBUG)
      for %i in ($(EXE_OBJS)) do %append $(PROJ).eln      FILE %i
      for %i in ($(EXE_LIBS)) do %append $(PROJ).eln      LIBRARY %i
      %append $(PROJ).eln     $(EXE_DATA_DIRECTIVE)
      %append $(PROJ).eln     OPTION Quiet, Map=$(PROJ).mpe, NOD, Verbose, &
      Stack=$(EXE_STACK), MODNAME='$(EXE_LNAME)'
      wlinkp @$(PROJ).eln
      del $(PROJ).eln
      copy $(PROJ).mpe $(APP_DIR)
```

Your makefile must define some of the variables used by SRULES.MIF. For example, the EXE_OBJS variable lists all of your object files that are needed to create the final executable image (.EXE). Again, the easiest way to write a makefile is to tailor the makefile of a sample application to your own project.

The makefile links the method table object code at this step in the process, but it does not link the resource object code. The resource object code stays separate from the executable code so that resource file data is cleanly separated from the rest of your application. See "Building a resource file" on page 385 for more information on how the resource file is built later in the process.

The loops in the code above adds a line to the temporary file for each of the object files and PenPoint libraries needed to create this application. The WLINKP command uses the information in the temporary file to link all the necessary object files together, and the temporary file is deleted when the linker finishes.

Finally, the makefile copies a file with an .MPE extension to your application directory. The linker creates this so-called map file to provide details about line numbers and symbol addresses created by the compiler and linker. You can view this file with a text editor to see details of how the linker created your executable file.

Stamping

The makefile uses a similar strategy for stamping the application directory. It first creates a temporary file that contains information about how to stamp your directory, passes that temporary file to the PSTAMP utility, and deletes the file when the utility is done. The following makefile rules accomplish this:

```
        %create $(PROJ).stm
        %append $(PROJ).stm $(APP_DIR)\..
        %append $(PROJ).stm /u
        %append $(PROJ).stm /n
!       ifeq RES_STAMP yes
              %append $(PROJ).stm /l $(LOCALE)
              %append $(PROJ).stm /r "$(LOCALE).rc" ""
!       else
              %append $(PROJ).stm /g "$(EXE_NAME)"
!       endif
        %append $(PROJ).stm /d $(PROJ)
        %append $(PROJ).stm /a imAttrVersion      "$(APP_VERSION)"
        %append $(PROJ).stm /a cimAttrProgramName "$(EXE_LNAME)"
        %append $(PROJ).stm /a appAttrClassName   "Application"
        %append $(PROJ).stm /a appAttrClass       10001a0
        -$(STAMP) -s $(PROJ).stm
        del $(PROJ).stm
        %create $(PROJ).stm
        %append $(PROJ).stm $(APP_DIR)
        %append $(PROJ).stm /u
        %append $(PROJ).stm /n
!       ifdef EXE_DLC
!             ifeq RES_STAMP yes
                    %append $(PROJ).stm /l $(LOCALE)
                    %append $(PROJ).stm /r "$(LOCALE).rc" ".dlc"
!             else
                    %append $(PROJ).stm /g "$(EXE_NAME).dlc"
!             endif
              %append $(PROJ).stm /D $(PROJ).dlc
!       else
!             ifeq RES_STAMP yes
                    %append $(PROJ).stm /l $(LOCALE)
                    %append $(PROJ).stm /r "$(LOCALE).rc" ".exe"
!             else
                    %append $(PROJ).stm /g "$(EXE_NAME).exe"
!             endif
              %append $(PROJ).stm /D $(PROJ).exe
!       endif
        -$(STAMP) -s $(PROJ).stm
        del $(PROJ).stm
```

The makefile first stamps a user-visible PenPoint name for your application. The standard rules use the string associated with **tagAppMgrAppFilename** as your application name. The tag and string are defined in the localized version of your resource file USA.RC or JPN.RC. If you need to stamp PenPoint names yourself, use the **-g** or **-r** option with the PSTAMP utility.

The makefile also stamps the directory with the following four attributes:

- A user-visible string describing the file type such as **Application**, **Font**, or **Service**. SRULES.MIF uses the string associated with **tagAppMgrApp-ClassName** in the appropriate version of your resource file.

- The linker name of your application. Define this in your makefile with the EXE_LNAME variable.
- The version number of your application. Define this in your makefile as the APP_VERSION.
- A special 7-digit hexadecimal number that identifies the file as an application. SRULES.MIF stamps this number for you. You don't need to set this in either your makefile or your resource file.

Table 29-1 shows more details about the information the makefile stamps. The middle column shows the tags (in lower case) or makefile variables (in upper case) you should define.

Attributes stamped by the makefile TABLE 29-1

Attribute label	*Makefile variable / Resource file tag*	*Example attribute value*
Not an attribute	**tagAppMgrDefaultDocName**	Counter Application
appAttrClassName	**tagAppMgrAppClassName**	Application
imAttrVersion	APP_VERSION	2.0
cimAttrProgramName	EXE_LNAME	GO-CNTRAPP-V2(0)
appAttrClass	None (automatically stamped)	10001A0

Use the DOS program PSTAMP to manually stamp the directory with this required information. See "PSTAMP" on page 405 for more information.

Notice that the makefile checks the value of RES_STAMP several times. You must define this variable in your own makefile. Set it to YES if the makefile should stamp your project with the application name and type in your resource file.

If you do not set RES_STAMP to YES, the makefile directs PSTAMP to use the EXE_NAME from your makefile and stamp the type as **Application**.

Building a resource file

By now, the makefile has created an executable in your project directory and stamped your directory with PenPoint attributes. The next step is to build your application resource files, as shown in the following makefile rules:

```
#
# Build the app's .res file
#
$(APP_DIR)\$(TARGET_RESFILE) : $(APP_DIR) $($(LOCALE)_RES_FILES) $(RES_FILES)
      for %i in ($($(LOCALE)_RES_FILES)) do -$(RESAPPND) %i temp.res
!ifdef RES_FILES
      for %i in ($(RES_FILES)) do -$(RESAPPND) %i temp.res
!endif
!ifdef DISTRIBUTED_DLLS
      for %i in ($(DISTRIBUTED_DLLS)) do -$(RESAPPND)
$(PENPOINT_PATH)\sdk\dll\%i\$(LOCALE).res temp.res
!endif
      copy temp.res $(APP_DIR)\$(TARGET_RESFILE)
      -del temp.res
```

The exact resource file that gets built depends on the LOCALE variable. You can specify the value of LOCALE when you call WMAKE. See "Specifying locales" on page 386 for details.

Changes from 1.0

The following things have changed in SRULES.MIF since PenPoint 1.0.

Stamping changes

In 1.0, SRULES.MIF stamped an application's name and its type ("Application") onto the executable.

In 2.0, the rules changed to allow both the name and the type to be localized. The new version of SRULES.MIF reads the application name and type from a resource file. The resource file must have a name of the form *Locale*.RC where *Locale* is JPN or USA.

All you need to do is provide **tagAppMgrAppFilename** and **tagAppMgrApp-ClassName** in the resource file for each localization. For example, if you are making a Japanese version of your application, these strings must be in JPN.RC.

Specifying locales

The WATCOM make utility (WMAKE.EXE) helps you make different localized versions of your application. When you call WMAKE, specify a LOCALE argument in the command line to make a localized version of your application. For instance, you can type:

```
wmake LOCALE=jpn
wmake LOCALE=usa
```

to create the Japanese and American versions of your application.

The LOCALE variable tells the compiler which resource files to compile and then copy to your application directory. Specifying JPN as the LOCALE directs the makefile to use the resource files specified by the variable JPN_RES_FILES. For example, with the Counter Application, the only resource file specified is JPN.RC.

You need a resource file for each of the localizations you create. If you have both American English and Japanese versions of your application, you must have files named USA.RC and JPN.RC.

In your makefile, use the three variables shown in Table 29-2 to identify which resource files to compile and copy into the application directory with the executable image.

Makefile variables TABLE 29-2

Variable	*Use*
RES_FILES	Resource files to be included with all versions of your application.
USA_RES_FILES	Resource files to be included with only the American version.
JPN_RES_FILES	Resource files to be included only with the Japanese version of your application.

For example, this fragment comes from the makefile for the NotePaper application in \2_0\PENPOINT\SDK\NPAPP:

```
#  The .res files for your project.  If you have resources, add
#  $(APP_DIR)\$(TARGET_RESFILE) to the "all" target.
      RES_FILES       = bitmap.res
      USA_RES_FILES   = usa.res
      JPN_RES_FILES   = jpn.res
# Targets
      all: $(APP_DIR)\$(PROJ).exe $(APP_DIR)\$(TARGET_RESFILE) .SYMBOLIC
```

Compiler details

GO uses the WATCOM C 9.01d/386 compiler to compile PenPoint 2.0 Japanese code. Although other compilers may work, GO does not support them.

16-bit character flag

When you compile code containing 16-bit (Unicode) or double-byte (Shift-JIS) characters, set a compiler flag so that the compiler works for 16-bit characters. For example, if you are using the WATCOM C compiler, set the compiler flag /ZK0U. All PenPoint 2.0 Japanese applications should support the 16-bit Unicode character standard. For more information, see Chapter 15 of *Part 2: PenPoint Internationalization Handbook.*

The standard makefile rules set this flag automatically in SDEFINES.MIF. The PenPoint 2.0 Japanese resource compiler RC.EXE also sets this flag because your resource files almost certainly contain Shift-JIS or Unicode strings.

DOS environment variables

You must identify the DOS PATH containing your WATCOM C compiler files. For example, you might put the following line in your AUTOEXEC.BAT:

```
path=c:\watcom\bin;
```

Use the CONTEXT batch file to set up the other DOS environment variables required to created PenPoint applications. See "CONTEXT batch file" on page 410 for more details on how to use this batch file.

Working with the method table compiler

The method table compiler, (\2_0\PENPOINT\SDK\UTIL\DOS\MT.EXE), creates a header file named METHODS.H from your source file METHODS.TBL. If METHODS.H already exists, the compiler checks if the first line of the file matches the following:

```
"// WARNING: DO NOT EDIT ..."
```

If that is the first line of the file, or if the file does not exist, the compiler creates a new METHODS.H.

If a METHODS.H file exists, but the first line is not the one above, the compiler exists with the error message:

```
mtcom ERROR: Failed while opening intermediate file (.h file exists)
```

PenPoint libraries

All the PenPoint 2.0 Japanese libraries are located in \2_0\PENPOINT\SDK\LIB. To direct the makefile to link a given PenPoint library with your code, you must include the name of the library as part of the definition of EXE_OBJS in your makefile.

The PenPoint 2.0 Japanese header files tell you which libraries you need to link. Remember to include the header file using the **#ifndef**, **#include**, and **#endif** preprocessor directives. The directives prevent files from being included more than once. For example, to use the PenPoint code that deals with memory allocation, you must have:

```
#ifndef OS_HEAP
#include <osheap.h>
#endif
```

in one of your header files. Then put the following line in your makefile:

```
EXE_OBJS = PENPOINT.LIB
```

Note that INTL.LIB and BRIDGE.LIB are new to PenPoint 2.0 Japanese.

- INTL.LIB contains all the international functions that are described in more detail in *Part 2: PenPoint Internationalization Handbook.*
- BRIDGE.LIB is virtually empty in PenPoint 2.0 Japanese. You can link this file with specially written code to allow the code to compile and run under both PenPoint 1.0 and 2.0. See the *PenPoint Bridging Handbook* for more information.

PenPoint applications

Many applications and services are stamped with their Japanese names in the 2.0 SDK. This is because no local application directory exists for each localization. Applications and services are usually found in \2_0\PENPOINT\APP and \2_0\PENPOINT\SERVICE.

To see the Shift-JIS names of these applications and services, type

```
\2_0\penpoint\sdk\util\dos\pdir -c XJIS \2_0\penpoint\app
```

This also shows you how the Japanese file names relate to DOS names.

Installing PenPoint applications

Section 3.15 of the *PenPoint Development Tools* suggests using the Installer to install your application. The Installer no longer exists. Here are some ways to install your application.

Installing automatically

To have PenPoint automatically install your application at boot time:

1 Add its PenPoint name and path to the appropriate APP.INI file.

2 If the application name contains characters you cannot type with your keyboard, just copy the application name from the application's resource file *Locale*.RC (USA.RC or JPN.RC) into APP.INI.

The appropriate file depends on whether you are installing your application for the Japanese (\2_0\PENPOINT\BOOT\JPN\APP.INI) or American English (\2_0\PENPOINT\BOOT\USA\APP.INI) localization.

Installing applications in \2_0\PENPOINT\APP

To install an application in the \2_0\PENPOINT\APP directory:

1 Open the Connections notebook.

2 Make sure you are on the Disks page.

3 Choose Applications from the View menu.

4 Tap on the Install box next to the application you want to install.

Installing applications from any connected disk

To install an application from any connected disk:

1 Open the Connections notebook.

2 Make sure you are on the Disks page.

3 Choose Layout under the Options menu.

4 Select Install in the option sheet that appears.

5 Navigate to an application on any connected disk.

6 Tap on the Install box next to the application you want to install.

Using the Settings notebook

To install an application by copying it into the Settings notebook:

1 Open the Settings notebook, and tap on the Applications button. This shows the currently installed applications.

2 To install an application, tap on the Install menu item. This brings up a screen that shows you all the applications in \2_0\PENPOINT\APP. Tap on the Install box for the application you want to install.

3 If you want to install an application not shown in this window, turn to the Disks page of the Connections notebook.

4 Select View by Directory.

5 Navigate to the application you want to install.

6 Copy the application into the Installed Applications page of the Settings notebook. Use the tap press ⋅⫰ gesture to initiate the copy, then drag the application on top of the Settings notebook.

Using the Settings notebook is easier if you also want to set preferences for software, remove software, or save any changes you have made.

Using the Connections notebook is more convenient if you also plan to set up a printer, transfer files to and from the computer, format a floppy disk, or use network resources. It also lets you install applications from any directory, not just \2_0\PENPOINT\APP.

Aside from these differences, using the Settings notebook is identical to using the Connections notebook. They are merely different user interfaces for the same process.

Copying files to the application directory

On page 51 in *PenPoint Development Tools*, the last section instructs you to create a directory called \PENPOINT\APP\EMPTYAPP, and copy EMPTYAPP.EXE into that directory.

Simply creating the directory and copying the executable file into that directory does not create a PenPoint application. You need to copy in any resource files, supporting documents, and stamp the directory with PenPoint information. See "Stamping" on page 384 for more information on how the standard makefile rules stamp PenPoint information on application directories.

Working with supporting files

In addition to the executable image and compiled resource files, your application might require supporting files such as Quick Start documents, Help notebook documents, and stationery. This section describes strategies for working with these supporting documents.

Preparing distribution disks

Each of your distribution disks for an application or service should contain certain directories. Table 29-3 shows the structure for the sample Tic-Tac-Toe application. You should put similar directories and files beginning at the root of your distribution disk.

Sample distribution disk structure TABLE 29-3

Directory and contents	*Description*
\PENPOINT\APP\TTT	Application directory
\PENPOINT\APP\TTT\PENPOINT.DIR	PenPoint directory and application information.
\PENPOINT\APP\TTT\TTT.EXE	Application executable.
\PENPOINT\APP\TTT\USA.RES	Compiled resource file for the USA localization.
\PENPOINT\APP\TTT\JPN.RES	Compiled resource file for the JPN localization.
\PENPOINT\APP\TTT\HELP	Help directory
\PENPOINT\APP\TTT\HELP\TTTHELP1	Directory containing first page (document) of help text.
\PENPOINT\APP\TTT\HELP\TTTHELP1\HELP.TXT	Actual help text, page 1
\PENPOINT\APP\TTT\HELP\TTTHELP2	Directory containing second page (document) of help text.
\PENPOINT\APP\TTT\HELP\TTTHELP2\HELP.TXT	Actual help text, page 2
\PENPOINT\APP\TTT\STATNRY	Stationery directory
\PENPOINT\APP\TTT\HELP\TTTSTAT1	Directory containing first stationery document.
\PENPOINT\APP\TTT\HELP\TTTSTAT1\TTTFTUF.TXT	Actual contents used by stationery document.
\PENPOINT\APP\TTT\HELP\TTTSTAT2	Directory containing first stationery document.
\PENPOINT\APP\TTT\HELP\TTTSTAT2\TTTFTUF.TXT	Actual contents used by stationery document.

See Chapter 11 of this manual for more information on how Tic-Tac-Toe creates and uses its supporting files. You may also want to examine the makefile for Tic-Tac-Toe to learn how to set up the \HELP and \STATNRY directories.

There is one more file in the Tic-Tac-Toe directory called TTT.MPE. This map file is created by the linker and is for debugging purposes only. You should not copy the file to your distribution disks.

Using short DOS path and file names

DOS imposes a 64-character limit on the total length of a path and file name.

The PenPoint document model creates a separate directory for each file. Furthermore, each embedded document contains a separate subdirectory beneath the directory of its parent document.

This recursive structure can create DOS file names longer than the maximum of 64 characters. For example, here is the full path to a stationery document that is part of a stationery notebook for a typical application:

```
C:\2_0\PENPOINT\APP\MY_APP\STATNRY\MY_APP_Q\NOTEBOOK\CONTENTS\USQ1SALES\DOC.RES
```

You can work around the 64-character limit by using the short, 2-character path name PenPoint creates for each directory. PenPoint can then translate the short file names into full PenPoint names using information in PENPOINT.DIR. Using this technique shortens the long path name above to the shorter path name:

```
C:\2_0\PENPOINT\APP\MY_APP\STATNRY\MQ\NB\CS\US\DOC.RES
```

To create these short DOS directory names, you must copy the document to a \PENPOINT directory in the root of any volume.

For example, follow this procedure to save a MiniNote document as a Quick Start stationery item. The sample application is called **MyApp**.

1. Before starting PenPoint, set the B flag to 800 by adding /DB800 to the **DebugSet** line in ENVIRON.INI as follows:
   ```
   DebugSet=/DD8000 /D*1 /DB800
   ```
   You can also type FS B 800 in the mini-debugger to set the flag.
2. Start PenPoint and create a MiniNote document named something like **MyApp Quick Start**. When you have finished the Quick Start document, open the Connections notebook and set the View to Directory.
3. Open the \PENPOINT directory in the root of any volume except RAM.
4. Copy the Quick Start document into \PENPOINT. Make sure you are copying into the \PENPOINT directory directly off the root of your volume. For example, the \2_0\PENPOINT directory will not work.
5. Shut down PenPoint.

6 Navigate into your application directory, and create a \STATNRY subdirectory. For example, the following creates the subdirectory in C:\2_0\PENPOINT\APP\MYAPP.

```
cd c:\2_0\penpoint\app\myapp
md statnry
cd statnry
```

7 Use the DOS utility PDIR to list the PenPoint names of the files in the \PENPOINT directory: See "PDIR" on page 407 for more information.

```
pdir c:\penpoint
```

8 Note the DOS name of the directory containing **MyApp Quick Start**. In this example, the DOS directory name is MT.

9 Copy the PenPoint document to your application's \STATNRY directory. For example, if the document is in the DOS directory MT, type the following:

```
xcopy c:\penpoint\mt mt /s/e
```

10 Append PenPoint attributes from the Quick Start document to the \STATNRY directory. See "PenPoint attribute utilities" on page 404 for details.

```
pappend penpoint.dir c:\penpoint\penpoint.dir /g "Myapp Quick Start"
```

11 Stamp the document to put a check in its menu check box. This forces the stationery document to appear in the Create menu.

```
pstamp /g "MyApp Quick Start" /A anmAttrStationeryMenu 1
```

12 Boot PenPoint and install **MyApp**. Verify that the only two stationery items for the application are **MyApp** and **MyApp Quick Start** by opening the Stationery Notebook.

GO uses this technique to save the Sample notebook and Help notebook. These files are in the \2_0\PENPOINT\BOOT\DOC directory.

Stamping stationery with different names

When you create localized versions of your application, you may want to ship different versions of your supporting documents, such as stationery and Quick Start documents. Use the procedure described above in "Using short DOS path and file names" to create the supporting documents.

However, some supporting documents you create are not language-specific. For example, the Tic-Tac-Toe sample application uses a piece of stationery to fill in a Tic-Tac-Toe board. Its stationery files are simply filled with a pattern of Xs and Os. The only difference between localized versions of the stationery is the stamped information (containing, among other things, the user-visible file name).

To create this kind of supporting document (which differs only by the stamped information), create the document in your development directory as described above.

When you build your application directory, copy the stationery files to \2_0\PENPOINT\APP\MYAPP\STATNRY and stamp the files as appropriate to the locale. For example, the following procedure stamps locally appropriate names onto stationery:

1 Add a tag and an associated string to the resource file for each localization. For example, you might use the tag **tagMyStationeryName** and add the appropriate strings to each resource file.

2 Add lines to your makefile that stamp your stationery with the string associated with the tag. Specify the name that should be stamped by prefixing the tag name with the & (ampersand) symbol. For example, the following makefile commands stamp the string associated with **tagMyStationeryName** on your stationery:

```
%create statnry.stm
%append statnry.stm -u
%append statnry.stm -n
%append statnry.stm $(PENPOINT_PATH)\app\MyApp\statnry
%append statnry.stm -l $(LOCALE)
%append statnry.stm -r $(LOCALE).rc ""
%append statnry.stm -g &tagMyStationeryName
%append statnry.stm -d $(DOSNAME)
%append statnry.stm -a anmAttrStationeryMenu 1
-$(STAMP) -s statnry.stm
del statnry.stm
```

3 Specify a locale when you call WMAKE to create your stationery. For example, the following commands create U.S. English versions of Tic-Tac-Toe stationery.

```
wmake stationery LOCALE=usa
wmake help LOCALE=usa
```

If you do not specify a locale, the makefile rules assume JPN.

Chapter 30 / Debugging

You can use a variety of tools to debug your PenPoint™ applications. This chapter provides an overview of those tools and discusses changes and improvements to the debugging tools for PenPoint 2.0 Japanese.

See Part 2 of *PenPoint Development Tools* for more information on using PenPoint's debugging tools.

Overview

There are a variety of ways to debug your application. Some common strategies are:

- Running the debug version of PenPoint 2.0 Japanese.
- Using **Debugf()** or **DPrintf()** to send text strings to the debugger stream. You can display these strings on your PenPoint screen or a second monochrome screen dedicated to displaying the debugger stream. You can also use the System Log application to write the debugger stream to a file.
- Using the PenPoint source debugger or the mini-debugger.
- Handling **msgDump**, which requests an object to format its instance data in a readable format to send it to the debugger stream.

Each of these strategies is discussed in more detail below.

Debug version of PenPoint

The PenPoint SDK 2.0 Japanese includes two versions of PenPoint, the production and debug versions. Each version contains its own set of DLLs, services, and applications.

The production version is what the end-user sees. Its files are in \...\PENPOINT\BOOT\DLL, \...\PENPOINT\BOOT\APP, and \...\PENPOINT\BOOT\SERVICE. The file PENPOINT.OS is also part of the production version. The ellipses (...) represent either 1_01 or 2_0, depending on which version of PenPoint you are developing for.

The debug version lets you see and use the following information in your debugger stream:

- Warnings from **ObjCallWarn/Ret/Jump** and **StsWarn/Ret/Jump**.
- Symbolic names for objects, classes, messages, and status values.
- Additional debugging information that may be helpful for reporting bugs to Developer Technical Services.
- Special debugging features documented in DEBUG.H.

The debug version files are in \...\PENPOINT\BOOT_DLL, \...\PENPOINT\BOOT_APP, and \...\PENPOINT\BOOT_SERVICE. The debug version of PENPOINT.OS is _PP.OS.

The drawbacks to using the debug version are:

- It requires more memory.
- PenPoint 2.0 Japanese runs more slowly.
- It is unlike the final end-user environment.
- Some debug warning messages are benign, and these may cause you undue worry about your own code.

You can modify the GO.BAT file to run either the debug or production version of PenPoint 2.0 Japanese. The default is the production version.

Sending text to the debugger stream

You can write debug information to the debugger stream using the **Debugf()** and **DPrintf()** functions. The sample code often uses this strategy to identify when a particular method handler executes. For example, here is the **CntrNewDefaults** method from the Counter Application:

```
MsgHandlerArgType(CntrNewDefaults, P_CNTR_NEW)
{
      Dbg(Debugf(U_L("Cntr:CntrNewDefaults"));)
      // Set default value in new struct.
      pArgs->cntr.initialValue = 0;
      return stsOK;
      MsgHandlerParametersNoWarning;
} /* CntrNewDefaults */
```

You can find this code in \2_0\PENPOINT\SDK\SAMPLE\CNTRAPP\CNTR.C.

Responding to **msgDump** is another way to send text to the debugger stream. Because your shipping product should not handle **msgDump**, use the #IFDEF and #ENDIF preprocessor directives to surround code that handles **msgDump** (and all other debugging code). For example, here are excerpts from the Tic-Tac-Toe method table showing how to use the directives:

```
MSG_INFO clsTttDataMethods[] = {
      msgNewDefaults,    "TttDataNewDefaults",   objCallAncestorBefore,
      . . .
#ifdef DEBUG
      msgDump,           "TttDataDump",          objCallAncestorBefore,
#endif
      . . .
};
```

Put a line in your makefile to define the name DEBUG when your application compiles. When DEBUG is defined, all the debugging code surrounded by the #IFDEF and #ENDIF preprocessor directives is compiled, so debugging information is sent to the debugger stream. Here is the line in Tic-Tac-Toe's makefile that defines the name:

```
MODE = debug
```

Look in SDEFINES.MIF to see how the MODE line influences which compiler and debugging flags are set.

See Chapter 10 in *PenPoint Development Tools* for more details on the functions that send data to the debugger stream.

Viewing the debugger stream

There are a variety of ways to view the debugger stream. Here are some of your options. See Chapter 10 in *PenPoint Development Tools* for more information.

On the PenPoint screen

To view the debugger stream on the same monitor as your PenPoint screen, you must uncomment the following line in MIL.INI:

```
    MonoDebug=off
```

On a second monitor

Make sure the line above is commented out if you want to see the debugger stream on a second, monochrome monitor. This configuration is called two- or dual-headed debugging. Be sure to set the appropriate **DebugCharSet** to indicate which character set should be used to interpret the debugger stream codes. See "Debug-CharSet" on page 398 for more details.

GO does not support viewing Shift-JIS on this second debugging monitor.

Using the System Log application

This PenPoint application saves the debugger stream to its own internal buffer. You can use the application to see the stream in a variety of ways. See "System Log" on page 398 for details.

Using a serial port

Assign a serial port to the **SerialDebugPort** variable in MIL.INI to send the debugger stream to that serial port. The port should be connected to a terminal emulator that can display the required character set. Most likely, this is another computer running a terminal emulation package. See "MIL.INI" on page 415 for details.

As a file

You can set variables in ENVIRON.INI to send the debugger stream to a file on your PC's hard drive. For example, set the following variables to save the stream to the file PENPOINT.LOG in the root directory of your PC's hard drive. The stream is flushed to the file after every 10 characters written to the stream.

```
    DebugSet=/DD8000
    DebugLog=\PENPOINT.LOG
    DebugLogFlushCount=10
```

DebugCharSet

The **DebugCharSet** variable in ENVIRON.INI controls the character set of your debugging output. Table 30-1 shows the currently permissible values.

DebugCharSet variable permissible values TABLE 30-1

Value	*Description*
ASCII	Standard 7-bit ASCII
XJIS	1990 Shift-JIS character set
437	Extended ASCII used in American PCs
850	Extended ASCII used in European PCs

If you are sending debugging information to your PenPoint monitor or a second debugging monitor, make sure it can display characters in the specified **Debug-CharSet**. If you don't, you will see unmeaningful characters.

Literal strings in **Debugf()** and **DPrintf()** appear in the specified character set. Unicode characters that do not have glyphs display as hex quads in PenPoint 2.0 Japanese. Outside of PenPoint, Unicode characters without glyphs display as /x *value*, where *value* is a 4-digit hexadecimal number. For example, the first hex quad in the margin would be displayed as /x001B.

Hex quads

```
00  F1  00
1B  F2  12
```

The default value of **DebugCharSet** depends on the value of **Locale**, another ENVIRON.INI variable. If **Locale** is JPN, the default is Shift-JIS. The default is ASCII if **Locale** is USA.

If **DebugCharSet** is set to an invalid value, the default character set is assumed.

System Log

The System Log application writes the debugger stream to a file.

See Chapter 11 of *PenPoint Development Tools* for information on using the application. Contrary to the text on page 141 of that chapter, you cannot get or set debug flags with the System Log application.

In PenPoint 2.0 Japanese, the System Log application can display Shift-JIS characters as well as ASCII text.

Note that the Device List command under View menu is not supported in the production version of PenPoint 2.0 Japanese.

Debug modes

You can run the debug version of PenPoint in either **DebugRAM** or **DebugTablet** mode. The **DebugRAM** mode is convenient for early application testing because it is much faster, while the **DebugTablet** mode is more appropriate for more refined testing because it more closely simulates a notebook computer. For example, in **DebugTablet** mode, files created in PenPoint persist between boots.

Because the **DebugTablet** mode most closely matches a real pen computer, GO encourages you to run in **DebugTablet** mode as you get closer to shipping your product.

DebugTablet

The **DebugTablet** mode, the default mode in PenPoint 2.0 Japanese, simulates a pen computer most closely. For example, files that you create will persist across boots. Also, when applications are installed, PenPoint copies executable code to a special system directory called the loader database. See Section 3.6 of *PenPoint Development Tools* for more information on **DebugTablet** mode.

If you set **SwapBoot** to 2 in ENVIRON.INI, PenPoint writes the content of your simulated notebook, complete with its documents and applications, to a swap file called \PENPOINT.SWP. The next time you boot, PenPoint reads the swap file to restore your simulated notebook. This considerably speeds up the boot process.

You specify the size of the swap file by setting **SwapFileSize** in ENVIRON.INI.

DebugRAM

In the **DebugRAM** mode, PenPoint creates its run-time file system (the Bookshelf, Notebook, and document directories) in RAM. If you use the complete Japanese font set in PenPoint 2.0 Japanese, this mode requires 12 megabytes of RAM. See *Installing and Running the PenPoint SDK 2.0* for strategies on reducing the required amount of RAM.

In **DebugRAM** mode, PenPoint does not create the loader database of executable system code, application executables, and DLLs. Instead, PenPoint pages them in from the original files. Thus, PenPoint starts from a fresh state each time you boot in **DebugRAM** mode.

You can still preserve files by copying them to your DOS disk by using the Connections notebook.

If you install an application or service from a floppy disk, and remove the disk while the application or service is running, PenPoint 2.0 Japanese may page fault. This is true only in **DebugRAM** mode.

Running PenPoint 1.0 and 2.0

If you are developing PenPoint 1.0 and 2.0 Japanese applications on the same PC, make sure only one environment uses **DebugTablet** mode. This protects you from crashes that occur if the PenPoint operating system tries to read a swap file created by a different version of PenPoint.

Because **DebugTablet** is the default mode in PenPoint 2.0 Japanese, set **DebugRAM** as your PenPoint 1.0 mode, or change the 2.0 default.

Warm booting

When you are debugging, you frequently want to replace the existing code with the most recent version. In **DebugTablet** mode, you may be able to replace the changed .EXE and .DLL files in the loader database (in \2_0\PENPOINT\SS\LR), thereby bypassing the PenPoint installer. This technique, called warm booting, saves about 100 seconds during boot.

Warm booting works only when you have:

- Cold booted with your .DLL or .EXE running.
- Made no structural changes to your project's destination directory.
- Not enabled swap booting.

The DLLs and EXEs in the loader database are the same as the files in \2_0\PENPOINT\APP or \2_0\PENPOINT\BOOT\DLL, except that:

- They are stamped with PenPoint attributes.
- Their PenPoint names are in the linker form rather than the DOS directory names. The syntax for the linker name is as follows:
  `company-project-VmajorVersion(minorVersion)`
  For example, the linker form of the dynamic link library required by the Calculator sample application is GO-CALC_ENG-V2(0).

Leave the PenPoint attributes unchanged in \2_0\PENPOINT\SS\LR\PENPOINT.DIR, and just copy in the new version of the DOS file.

The name of the files to copy are the DOS forms of your application's PenPoint name. The DOS names are usually the first eight letters of the linker name. For example, GO-CALC_ is the DOS version of the linker name GO-CALC_ENG-V2(0).

If you can guarantee that the first eight characters identify the files you want, write a makefile to perform the required copying automatically. You might write your makefile so that typing **make warmboot** from the DOS prompt takes the required steps. Your makefile might look something like this:

```
#
# Warmboot puts the newly recompiled files into the loader database when
# the user is using DebugTablet. This saves having to de-install and
# re-install the app or service to get the new version.
#
WARMEXES =
# if DLL_DIR is defined, then we are processing a service, so don't copy
# the DLL here. There is another double-colon warmboot target in the
# svcrules.mif to handle the service DLL
!    ifneq DLL_OBJS
!    ifeq DLL_DIR
WARMEXES += $(APP_DIR)\$(PROJ).dll
!    endif
!    endif
# only get the init.dll or the executable if they are built here
!    ifneq INIT_OBJS
WARMEXES += $(APP_DIR)\init.dll
!    endif
!    ifneq EXE_OBJS
WARMEXES += $(APP_DIR)\$(PROJ).exe
!    endif
warmboot :: \penpoint\ss\lr\penpoint.dir $(WARMEXES) .SYMBOLIC
!    ifneq DLL_OBJS
!    ifeq DLL_DIR
            -copy $(APP_DIR)\$(PROJ).dll \penpoint\ss\lr\$(DLL_LNAME)
!    endif
!    endif
!    ifneq INIT_OBJS
           -copy $(APP_DIR)\init.dll \penpoint\ss\lr\$(DLL_LNAME)
!    endif
!    ifneq EXE_OBJS
           -copy $(APP_DIR)\$(PROJ).exe \penpoint\ss\lr\$(EXE_LNAME)
!    endif
# must have booted before to do make warmboot, so quit here if they haven't
\penpoint\ss\lr\penpoint.dir:
           @if not exist \penpoint\ss\lr\penpoint.dir &
                @echo ******* Error! *********
           @if not exist \penpoint\ss\lr\penpoint.dir &
                @echo You must first boot PenPoint with Config=DebugTablet to
able to 'make warmboot.'
           @if not exist \penpoint\ss\lr\penpoint.dir @%quit
```

If you are working with services, you might put this rule in your standard service makefile.

```
warmboot ::  \penpoint\ss\lr\penpoint.dir $(DLL_DIR)\$(PROJ).dll .SYMBOLIC
!    ifneq DLL_DIR
               -copy $(DLL_DIR)\$(PROJ).dll \penpoint\ss\lr\$(DLL_LNAME)
!    endif
```

An 8-character name does not always uniquely identify a file. For example, if you have two applications with the PenPoint names PENCOMPANY-PROJECT1-V1(0) and PENCOMPANY-PROJECT2-V1(0), the first 8 letters does not distinguish between the two files.

In this case, use the DOS utility PDIR to determine the DOS name of your .EXE and .DLL files. Then copy the file (with the 8-character DOS name PDIR shows) into \2_0\PENPOINT\SS\LR.

Note *GDIR utility has been renamed PDIR in PenPoint 2.0 Japanese*

Using the mini-debugger

This section contains more detailed information about how to use the mini-debugger. See Chapter 12 of *PenPoint Development Tools* for more information.

Displaying Unicode

Table 12-1 on page 146 of *PenPoint Development Tools* shows a series of mini-debugger commands (**d, da, db, dd**, and **dw**) that display the contents of particular memory addresses as ASCII text.

In PenPoint 2.0 Japanese, the same commands display the memory contents as either multibyte (Shift-JIS) or Unicode characters. The interpretation is controlled by the **mdb** command, which controls various aspects of the mini-debugger.

Table 30-2 shows the currently defined controls. Simply type **mdb** followed by one of these numbers to activate a control. For example, **mdb 6** turns on Unicode interpretation of memory dumps.

Mini-debugger controls

TABLE 30-2

Control	Meaning
1	Turn off page fault protection.
2	Turn on page fault protection.
3	Turn off symbolic translation during stack traces (**st**).
4	Turn on symbolic translation during stack traces (**st**).
5	Turn off symbolic translation when displaying address information (**ai**).
6	Interpret memory contents as Unicode characters.
7	Interpret memory contents as multibyte (Shift-JIS) characters.

Disabling the mini-debugger

In the end-user version of PenPoint, the mini-debugger is disabled. You can explicitly disable the mini-debugger by setting the /DD10000 flag. When the mini-debugger is disabled, applications or services that crash are simply terminated, and operation continues.

Also, set the /DD40000 flag to disable the keys that drop you into the mini-debugger. Without this flag, CTRL-C and BREAK drop you into the mini-debugger.

A PenPoint machine set up for a user has both flags set; namely, /DD50000 is set.

Turning flag bits on and off

The mini-debugger fs command accepts + and – to enable and disable flags. The operators toggle the value of the flag specified. For example, the following lines toggle the B 800 flag.

```
fs B +800
fs B -800
```

These commands are much simpler than the source-level debugger commands, where you are responsible for the addition and subtraction of the appropriate bits.

Getting help

Type H or h to see the list of valid commands. Press the space bar to scroll through the list one line at time; press Return to scroll through an entire screen at once.

Table 12-1 in *PenPoint Development Tools* erroneously states that you can type the question mark (?) character to get help.

Chapter 31 / Tools and Utilities

This chapter describes various DOS utilities and PenPoint™ accessories that help you create PenPoint 2.0 Japanese applications and services. Many of these utilities are updates of tools found in the 1.0 SDK, so you should look in *PenPoint Development Tools* for information on how to use these tools.

The old command-line syntax is still accepted for all utilities. You only need to rename GDIR and STAMP in your 1.0 scripts, since these utilities have been renamed.

Most of the new and updated DOS utilities are in \2_0\PENPOINT\SDK\UTIL\DOS. Type -? or /? after any of these DOS utilities to see a help message.

Locales and character sets

Because you are likely to use different character sets while developing your PenPoint 2.0 Japanese application, most of the DOS utilities are sensitive to two DOS environment variables.

- CHARSET can be one of the values in Table 31-1.
- LOCALE, a combination of a country, language, and dialect, can be either USA or JPN. The locale maps to a default character set. The USA locale maps to code page 437, while JPN maps to XJIS.

Valid values for CHARSET

TABLE 31-1

Value	*Description*
ASCII	Standard 7-bit ASCII.
XJIS	Shift-JIS encoding of the 1990 JIS character set.
XJIS 1983	Shift-JIS encoding of the 1983 JIS character set.
XJIS 1978	Shift-JIS encoding of the 1978 JIS character set.
437	IBM code page 437 used in U.S. PCs.
850	IBM code page 850 used in European PCs.
LATIN1	International Standards Organization (ISO) Latin 1 character set.

Set these environment variables to reflect the character set that your keyboard uses.

You can set the variables in AUTOEXEC.BAT if you use a particular character set most of the time. Use this line, for example, to specify code page 437 as the default character set.

```
    set charset=437
```

With this variable set, the DOS utilities will interpret the characters in stamped names as ASCII characters. Other character sets and locales are supported, but the ones mentioned here are the only ones relevant to PenPoint 2.0 Japanese.

If you want to specify a character set or locale for just one time, most utilities accept a **-c** or **-l** argument. For example, you can type either of the following commands to display PenPoint directory information, with the PenPoint names interpreted as Shift-JIS characters.

```
pdir -cxjis
pdir -l jpn
```

Notice in the above example that spaces are not required between flags and their values. You can type either **-cXJIS** or **-c** XJIS to specify the character set.

You may specify explicit Unicode characters by embedding \x*hhhh* in your strings, where *hhhh* is up to 4 hexadecimal digits. For example,

```
pstamp MyDir -u -g "Q3 Sales in \x00A5" -d Q3SALES
```

stamps the directory **MyDir** with the name **Q3 Sales in ¥** because the Unicode value 0x00A5 represents the ¥ symbol.

PenPoint attribute utilities

PenPoint 2.0 Japanese provides a collection of DOS utilities to stamp directories, applications, services, and documents with PenPoint attributes. This information is stored in a file named PENPOINT.DIR in the same DOS directory. Table 31-2 shows the available utilities.

In PenPoint 2.0 Japanese, PENPOINT.DIR can contain Unicode strings, although the utilities can still read PENPOINT.DIR containing ASCII strings. That means the utilities still work with PenPoint 1.0 files. You must have the DOS4GW.EXE file in your DOS PATH to run any of these utilities.

Attribute utilities

TABLE 31-2

Name	*Purpose*
PSTAMP	Adds special PenPoint information to a DOS file or directory.
PDEL	Deletes specific directory entries from PENPOINT.DIR files.
PCOPY	Recursively copies files and directories to other PenPoint directories. Appends the appropriate entries in PENPOINT.DIR.
PDIR	Lists the PenPoint names and file systems attributes for all the files and directories in a DOS directory. Replaces GDIR from the utilities included with PenPoint 1.0.
PSYNC	Scans the current directory and removes any entries from PENPOINT.DIR for which there are no corresponding DOS files. Note that this updates the PenPoint directory information from the DOS information; it does *not* update DOS files from the information in PENPOINT.DIR.

PSTAMP

PSTAMP replaces STAMP from PenPoint 1.0. It has changed in the following ways:

- You can stamp Unicode strings by specifying the **-u** option.
- You can specify a script file of arguments using the **-s** option to help automate the stamping process. The script file must contain the same arguments you would type in at the command line. These scripts help you avoid the 128-character limit DOS imposes on commands.
- You can now delete an attribute by specifying the **-x** argument. Previously, the only way to do this was to create a new entry that did not contain the attribute you wanted to delete.
- When manipulating attributes, you can refer to entries one of three ways:
    - By PenPoint name (as it was in the STAMP from 1.0).
    - By DOS name.
    - By the string associated with **tagAppMgrAppFilename** in a resource file (JPN.RC or USA.RC).
- PSTAMP assumes you want to manipulate the PENPOINT.DIR in the current directory, so you don't need to specify a PENPOINT.DIR file each time you run PSTAMP. You can still specify PENPOINT.DIR files in other directories if you want.
- You may change the PenPoint name of an entry by specifying the **-n** flag. You must specify the new PenPoint name and the old DOS name as arguments. If the entry does not exist already, a new one is created. For example, this command changes the name of the Paint sample application from Paint Demo to My Paint Program:

  ```
  PSTAMP -N -D "PAINT.EXE" -G "My Paint Program"
  ```
- You may specify symbolic names for all system-defined attributes instead of attribute numbers. The symbolic names are the **#define** names of the attributes in the header files. All names that are defined using a function of the form **FSMake*Attr()** are valid.
- Table 31-3 shows some attributes that are commonly stamped onto items you can install in PenPoint (for example, applications and services). Remember that there are three kinds of attributes: strings, variables, and fixed (32-bit or 64-bit). See Chapter 72, Using the File System in the *Architecture Reference* for more information.

Attributes stamped on installable items

TABLE 31-3

Admin	Index	Category	Must use?	Label header file	Comment
62	0	String	Yes	**fsAttrName** FS.H	User-visible name of installable item. Must be unique within the parent directory.
260	3	String	No	**imAttrVersion** INSTLMGR.H	User-visible version string.
157	12	String	No	**appAttrClassName** APPDIR.H	User-visible installable type name (for example, Application, Service, Font, Printer).
193	2	String	Yes	**cimAttrProgramName** CODEMGR.H	Module name such as GO-ABAPP-V2(0). Must match the module name in the .LBC file used to build the module.
157	1	FIX32	Yes	**appAttrClass** APPDIR.H	Installable type. Must be set to the appropriate installation manager such as **theInstalledApps** (010001A0).

Table 31-3 shows attributes that are commonly stamped on PenPoint documents.

Attributes stamped on documents

TABLE 31-4

Admin	Index	Category	Must use?	Label header file	Comment
62	0	String	Yes	**fsAttrName** FS.H	User-visible name of document. Must be unique within the parent directory.
28	0	FIX64	Yes	**fsAttrDirIndex** FS.H	Directory index. Must be unique.
157	12	String	No	**appAttrClassName** APPDIR.H	User-visible name of document's application (for example, GOMail, MiniNote).
157	1	FIX32	Yes	**appAttrClass** APPDIR.H	Document's application class.
157	4	FIX32	Yes	**appAttrSequence** APPDIR.H	Sequence number describing the position of an embedded document.
157	3	FIX32	Yes	**appAttrNumChildren** APPDIR.H	Number of documents embedded within this document.
157	6	FIX64	Yes	**appAttrFlags** APPDIR.H	Document's file system flags (for example, moveable, **readOnly**).
157	9	String	No	**appAttrBookmark** APPDIR.H	User-visible name of a document's tab in the notebook.
157	10	String	No	**appAttrAuthor** APPDIR.H	User-visible author field.
157	10	String	No	**appAttrComments** APPDIR.H	User-visible comments field.

See "Stamping changes" on page 386 for more information about how the standard makefile rules have changed stamping behavior since 1.0.

PCOPY

PCOPY allows you to recursively copy files and directories to other PenPoint directories. If the directory you are copying contains a PENPOINT.DIR file, PCOPY updates the target PENPOINT.DIR file to include the new information.

```
PCOPY source [target-dir] [/V] [/L locale | /C charset]
```

The *source* can be a file or a directory. If *source* is a directory, PCOPY copies the contents recursively. You can also use the standard DOS wildcards * and ? to specify multiple files or directories. The *locale* and *charset* specify the locale or character set from which to translate the PenPoint name. PenPoint names, remember, are written as Unicode strings. See "Locales and character sets" on page 403 for a list of valid values.

For example, you might need to copy an application from \2_0\PENPOINT\SDK\APP, such as SSHOT, to \2_0\PENPOINT\APP. With the tools available in PenPoint 1.0, you had to use XCOPY to copy the SSHOT directory into \2_0\PENPOINT\APP, then use PAPPEND to copy SSHOT attributes from its original directory to \2_0\PENPOINT\APP.

Now, you can simply type the following to achieve the same result:

```
PCOPY \2_0\PENPOINT\SDK\APP\SSHOT \2_0\PENPOINT\APP
```

PCOPY recursively copies SSHOT to the target and updates the PENPOINT.DIR file in \2_0\PENPOINT\APP.

PCOPY cannot copy to already existing subdirectories below the target directory. For instance, you can't:

```
MD D:\2_0\PENPOINT
PCOPY C:\2_0\PENPOINT\SDK\APP D:\2_0\PENPOINT
PCOPY C:\2_0\PENPOINT\APP D:\2_0\PENPOINT
```

The second PCOPY fails because D:\2_0\PENPOINT\APP already exists.

PDIR

PDIR replaces GDIR from PenPoint 1.0. While GDIR allowed you to specify only directories, PDIR allows you to specify files as well. For example:

```
PDIR PENPOINT.BAK
```

displays PenPoint information about the file PENPOINT.BAK.

PDIR also differs from GDIR in the following ways:

- Attributes are only printed when you specify the **-a** argument rather than printing automatically.
- Unicode names that have been stamped using PSTAMP's **-u** option are interpreted as characters from the set specified in CHARSET.

If some of the characters in your Unicode strings display as spaces, there is no equivalent character in the character set you specified. Change your character set or specify the **-u** option to display unprintable Unicode characters as hex numbers.

Resource file utilities

The resource utilities from PenPoint 1.0 have been ported to 2.0 Japanese.

- RC, the resource compiler, compiles .RC files into .RES files. Applications and services actually use .RES files, not .RC files.
- RESAPPEND appends resources from one resource to another. It also compacts the target resource file by removing deleted or duplicated resources.
- RESDUMP allows you to view the contents of a resource file.

RESDEL

A new resource file utility, RESDEL, deletes specific resources from a compiled resource file. The syntax of the command:

```
RESDEL resource-file-name [ resource-ID-spec ]
```

The *resource-ID-spec* identifies a particular resource in a variety of ways. You can find a resource ID by examining the contents of a resource file using RESDUMP.

For example, if you wanted to delete a resource from the USA.RES of Tic-Tac-Toe, first examine the contents of the file by typing:

```
resdump c:\2_0\penpoint\sdk\app\ttt\usa.res > tempfile
```

Open **tempfile** with a text editor and notice that a typical resource looks like the following:

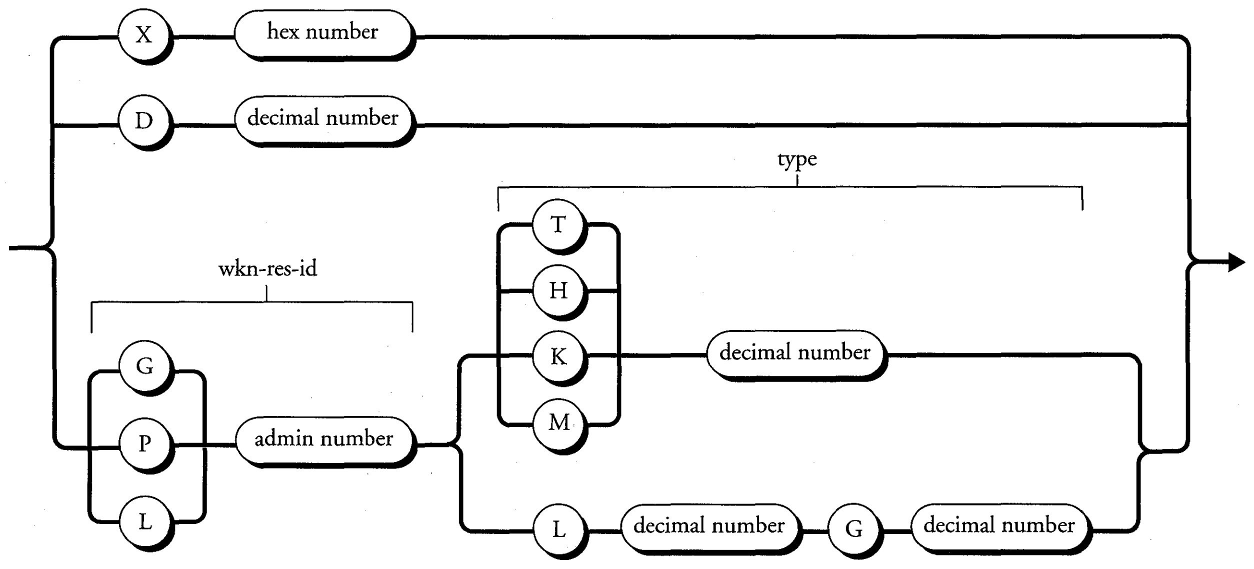
```
Resource 0 is a well-known object resource
resId = [0x0780001A WKN: Scope=Global Admin=13 Tag=15]
Objects class = [0x010002F4 WKN: Scope=Global Admin=378 Ver=1], data length=401
Min sys version = 0
```

You can specify the resource with the hexadecimal number 0x0780001A. To delete it, just type:

```
RESDEL C:\2_0\PENPOINT\SDK\APP\TTT\USA.RES 0x0780001A
```

There are more complex ways of specifying a resource, as shown in Figure 31-1.

Specifying a resource with RESDEL FIGURE 31-1



In the above example, note that the resource if a well-known object resource with global scope, an administered number of 13, and a tag number of 15. Given this information, you can delete the resource by typing:

```
RESDEL C:\2_0\PENPOINT\SDK\APP\\TTT\USA.RES G13T15
```

Type RESDEL /H for details on these alternate ways of specifying resources.

Other DOS utilities

There are a collection of DOS utilities that do not deal with PenPoint attributes or resource files.

UCONVERT

A new utility UCONVERT allows you to convert entire files from one character set to another. The syntax of the command is as follows:

```
UCONVERT [-d] [-m] source-file dest-file [source CharSet] [dest CharSet]
```

You can specify a character set as either a code page or a locale. Any character set shown in Table 30-1 in Chapter 30, Debugging, is valid. You may also specify UNI to indicate the Unicode character set. Each locale maps to a default character set:

- USA maps to code page 437 (specified as 437).
- JPN maps to Shift-JIS (specified as XJIS).

Table 31-5 shows examples of using the UCONVERT utility.

Using UCONVERT TABLE 31-5

Command	*Description*
uconvert mytext.doc mytext.unc	Puts a Unicode copy of ASCII document MYTEXT.DOC in the file MYTEXT.UNC. ASCII-to-Unicode is the default conversion.
uconvert mytext.unc mytext.jis uni xjis	Puts a Shift-JIS version of the Unicode document MYTEXT.UNC in the file MYTEXT.JIS
uconvert -d myfiles.doc myfiles.jis xjis uni	Puts a Shift-JIS version of the file MYFILES.TXT containing filenames in the file MYFILES.JIS. The -d flag is necessary when the input Shift-JIS file contains filenames.
uconvert letter.jis letter.unc jpn uni	Puts a Unicode copy of the Shift-JIS file LETTER.JIS in the file LETTER.UNC.
uconvert -m longfile.437 longfile.unc	Puts a copy of the extended ASCII file LONGFILE.437 in the Unicode file LONGFILE.UNC, converting all CR/LF combinations to the Unicode line separator character (U+2028).

RTFTRIM

You can use RTFTRIM to convert an RTF document into a form usable by the Help notebook.

RTFTRIM converts the Japanese RTF form \'*xx*\'*yy* to *uuuu* where *xx* and *yy* are the first and second bytes of a Shift-JIS character and *uuuu* is the Unicode equivalent.

RTFTRIM also introduces a new keyword \UNC that allows you to embed Unicode characters in a 7-bit RTF file. Just put \UNC*xxxx* where *xxxx* is the hex representation of the Unicode character.

This allows you to specify GO gesture glyphs in RTF help files. For example, the literal /UNCF600 represents the Unicode point for the single tap gesture. See the header file \2_0\PENPOINT\SDK\INC\GLYPH.H for a list of Unicode values for the GO gesture glyphs.

CONTEXT batch file

The CONTEXT batch file, located in \2_0\PENPOINT\SDK\UTIL\DOS, helps you set up the DOS environment variables required to run PenPoint and compile PenPoint applications and services.

Because the batch file accepts 1_01 or 2_0 as arguments, it is especially useful if you need to switch between PenPoint 1.0 and 2.0 development.

After reading the argument that represents the development environment, CONTEXT.BAT performs the following actions:

- Sets these environment variables:
    - CONTEXT
    - PENPOINT_PATH, GO_PATH, PATH
    - INCLUDE
    - LIB
- Adds \2_0\SDK\UTIL\DOS to the end of your PATH.
- Creates \PENPOINT and \PENPOINT\BOOT directories in the root of your current volume (if they don't already exist).
- Copies the ENVIRON.INI file from the appropriate \...\PENPOINT\BOOT directory into \PENPOINT\BOOT in the root.

CONTEXT.BAT assumes C: is the source drive. You can change this by reassigning the SRC_DRV variable.

If you change your path in AUTOEXEC.BAT, reboot your machine. Do not simply run AUTOEXEC.BAT to get the new path because CONTEXT.BAT sets up an environment variable that must be cleared if you change your path.

GO batch file

The GO.BAT batch file now takes two optional parameters to specify the locales to boot with:

```
go [system_locale] [user_locale]
```

When you specify a system locale, PenPoint's behavior and user interface are changed to be appropriate to the specified locale (U.S. or Japanese).

When you specify both a system and user locale, the batch file directs PenPoint 2.0 Japanese to change its behavior to match the system locale, but to change its user interface strings to match the user locale.

For example:

- To boot with Japanese behavior and strings, type the following command:
  ```
  go jpn
  ```
- To boot with Japanese behavior and English strings, type the following command:
  ```
  go jpn usa
  ```

When you type GO with no parameters, PenPoint 2.0 Japanese boots in the same state as it was last booted. If you type GO with no parameters and you are in **DebugTablet** mode, PenPoint warm boots. See "Warm booting" on page 399 for more information. Because the batch file only controls which resource files PenPoint loads, the stamped application and service names appear in the system locale language.

GO.BAT relies on LOCALE.BAT to do the locale switch. Both GO.BAT and LOCALE.BAT require utilities in the \2_0\PENPOINT\SDK\UTIL\DOS directory to switch locales.

When you specify a locale with GO.BAT (or LOCALE.BAT) the batch file recursively deletes your \PENPOINT\SS directory. This deletes any documents that you had saved in your PenPoint file system. Make sure to save the files to your hard drive if you need them.

Warning The GO.BAT and LOCALE.BAT batch files delete your PenPoint 2.0 Japanese files when you specify locales.

Currently, only two locales are supported: JPN and USA.

LOCALE batch file

GO.BAT calls LOCALE.BAT to implement the required changes. You can call LOCALE.BAT yourself if you want to change the configuration without booting PenPoint. Its syntax is similar to GO.BAT:

```
locale system_locale [user_locale]
```

LOCALE.BAT edits ENVIRON.INI and copies the appropriate MIL.RES file from the appropriate locale-specific boot directory (\2_0\PENPOINT\BOOT\JPN or \2_0\PENPOINT\BOOT\USA) to \2_0\PENPOINT\BOOT. If you want to load any of your applications or services at boot time, remember to specify them in the appropriate APP.INI or SERVICE.INI file. PenPoint 2.0 Japanese uses the initialization files in the directory corresponding to the user locale.

For example, if you are running the system with Japanese behavior and English strings, only the services in \2_0\PENPOINT\BOOT\USA\SERVICE.INI are loaded. The KKC engine is not listed in this SERVICE.INI, so you need to explicitly install this service if you want to test your application with KKC enabled. Use the Settings notebook to install the services.

Do not make the changes that LOCALE.BAT implements unless you are sure your changes are not destabilizing. The way PenPoint 2.0 Japanese handles locales will change in the future, so you should let LOCALE.BAT handle the switch.

Bitmap editor

The bitmap editor is in \2_0\PENPOINT\SDK\APP\BITMAP.

See "Installing applications from any connected disk" on page 389 for details on how to install the bitmap editor.

Here is the typical procedure for working with bitmaps:

1 Create a bitmap using the bitmap editor. See Chapter 16 of *PenPoint Development Tools* for details on how to use the bitmap editor.

2 If you are creating an icon, define an appropriate hot spot. The hot spot determines the origin of the bitmap. Make sure your hot spot is defined in a way so that the icon is clearly visible when it is drawn on-screen.

3 If you are creating a bitmap other than an application icon, tap on the Custom Resource Id choice under the Options menu. Set the following values:
   - Set the **Class** number to the administered number for your class.
   - Set the **Scope** value to match the scope of your class.
   - Set the **Id** value to the tag value you use to identify your bitmap in your header file.

4 Export the bitmap. If your bitmap is an application icon, select App or Small App from the Resource Id pop-up menu. Otherwise, select Custom. Note that bitmaps are exported as resource object (.RES) files.

5 Create a tag to identify your bitmap with the **MakeTag**() macro. Typically, this is done in a header file. Use the Id value you used in step 3.

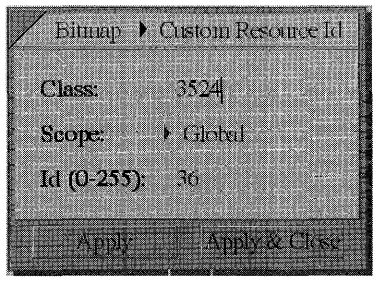
6 Create an instance of **clsIcon** or **clsIconToggle**.

7 Assign the field that identifies a bitmap to the tag you defined in step 5. For example, set the **win.tag** field of the ICON_NEW structure or the **iconToggle.offTag** of the ICON_TOGGLE_NEW structure to the tag you defined to identify your bitmap. See below for an example.

8 Set the **control.client** field of ICON_NEW to **OSThisApp**().

The following code comes from the UI Companion, a sample application in \2_0\PENPOINT\SDK\SAMPLE\UICOMP. The code shows how the frog icon on the "Lists" page of the UI Companion was created and used.

When the bitmap was created, the following values were assigned in the Custom Resource Id option sheet:


Bitmap ▸ Custom Resource Id
Class: 3524
Scope: ▸ Global
Id (0-255): 36

The values used in the option sheets come from the header file UICOMP.H:

```
#define clsUICompApp    MakeGlobalWKN (3524, 1)
...
#define tagIconFrog     MakeTag (clsUICompApp, 36)
```

The code in LISTS.C uses this tag to identify the bitmap when creating an instance of **clsIconToggle**:

```
ICON_TOGGLE_NEW         itn;
STATUS                  s;
ObjCallWarn(msgNewDefaults, clsIconToggle, &itn);
itn.iconToggle.offTag   = tagIconFrog;
itn.iconToggle.onTag    = tagIconPrince;
itn.icon.pictureSize.w  = iconSizeNormal;
itn.icon.pictureSize.h  = iconSizeNormal;
itn.control.client      = OSThisApp();
itn.border.style.edge   = bsEdgeAll;
ObjCallRet(msgNew, clsIconToggle, &itn, s);
```

See the in-line comments in UICOMP.C for more information on using bitmaps.

Font editor

The font editor for PenPoint 2.0 Japanese fonts is no longer supported. You should convert your fonts to bitmaps.

If you have fonts created with the font editor that you believe would be valuable to a large community of PenPoint programmers, contact GO to negotiate translating those fonts for use in PenPoint 2.0 Japanese.

Contact GO if you want to see the font specification. With the font specification, you can create your own fonts.

PenPoint tools

Aside from the DOS utilities, there are a number of PenPoint 2.0 Japanese applications and accessories to help you create applications and services.

MiniText

You can use MiniText as a Shift-JIS and Unicode editor. MiniText supports Japanese handwriting recognition, KKC, and RKC.

See Chapters 4 and 5 of the *Japanese Localization Handbook* for more details on how to use MiniText as a text editor.

Unicode Browser

The Unicode Browser allows you to find specific characters and put them into the input stream. It contains all the characters available in the Japanese fonts when those fonts are installed.

The first row of characters in the Unicode Browser consists of Latin characters (the letters a through z, the numbers 0 through 9, mathematical operators, accented characters, Cyrillic characters) and special fonts (gestures and other GO glyphs) installed in PenPoint 2.0 Japanese. The remaining rows of the Unicode Browser contain kanji radicals.

Tap on any character in the top level of the Browser to open a submenu. For non-kanji (the top row), the submenu contains the characters in the set represented. For kanji radicals, the submenu contains all characters that use the displayed radical as their base radical. The size of this submenu varies from radical to radical.

Tap on a character in the submenu to insert the character into the input stream at the current insertion point. Tap outside the submenu to close the submenu without any character being selected. If there is no current insertion point, the submenu closes and nothing happens.

Japanese virtual keyboard

The virtual keyboard is another PenPoint 2.0 Japanese accessory that allows users to send characters to the text stream. It offers U.S. and Japanese IBM-A01 keyboard modes. Bring up the keyboard and make the check ✓ gesture over the title bar to switch modes.

Chapter 32 / Miscellaneous

This chapter describes miscellaneous topics including:

- MIL.INI.
- ENVIRON.INI.
- Printing to a spooler.
- Long DOS file names.
- Changes to QuickHelp.
- Corrections to previous documentation.

MIL.INI

The MIL may print out some initial errors before PenPoint 2.0 Japanese boots. Because PenPoint isn't running at this time, it hasn't read ENVIRON.INI to determine whether to log to a file. To see these pre-boot errors, either use a second monochrome monitor or use MIL.INI to log to the serial port.

To see low-level output on a second monochrome monitor, set

```
LowLevelDebug=mono
```

in MIL.INI. To direct low-level output to a serial port, connect a serial port to another computer running a telecommunications package and set

```
LowLevelDebug=com1
SerialDebugPort=1
```

in MIL.INI.

Keyboards

The MIL.INI file contains a new variable that allows you to specify what kind of keyboard you are using. PenPoint 2.0 Japanese supports the following keyboard types:

- USA 101-key (IBM AT) keyboard.
- IBM A01 Japanese keyboard.
- AX Consortium keyboard.
- Toshiba 3100 desktop keyboards.
- Toshiba 3100 laptop keyboards.

See MIL.INI for valid values for the **Keyboard** variable.

The value of **Keyboard** determines how the keyboard behaves in PenPoint 2.0 Japanese. To change keyboards, you must warm or cold boot. Swap booting does not change keyboard behavior.

MonoDebug

The only reason you need to set the **MonoDebug** variable is if you have a monochrome card, but you nonetheless want PenPoint to use the VGA screen for debugging output.

The description on page 44 in *PenPoint Development Tools* may be misleading, because it suggests that you need to set the variable even if you have a VGA card.

ENVIRON.INI

This section describes changes to ENVIRON.INI since PenPoint 1.0.

Locale

You can boot PenPoint 2.0 Japanese with a system locale different than its user locale. For example, you can boot with Japanese behavior and English strings in its user interface. This configuration helps you use PenPoint and test the Japanese version of your product without having to read Japanese.

See "GO batch file" on page 410 for more details on how system and user locales differ and how to switch locales.

A new variable named **Locale** has been added to ENVIRON.INI to represent the system locale. You should not specify a value for **Locale** yourself. Instead, use the GO.BAT or LOCALE.BAT batch files to specify locales.

If you specify a user locale while running either GO.BAT or LOCALE.BAT, the batch files create a variable named **LocaleUser** in your ENVIRON.INI. Do not modify this variable. Instead, call the batch files to specify system and user locales.

PenPoint uses **LocaleUser** to determine where to search for its initialization files APP.INI, SERVICE.INI, SYSAPP.INI, and SYSCOPY.INI. When **Locale** is USA, PenPoint uses \2_0\PENPOINT\BOOT\USA to find the required initialization files. When **Locale** is JPN, PenPoint uses \2_0\PENPOINT\BOOT\JPN.

Debugging character set

The **DebugCharSet** variable in ENVIRON.INI controls the character set of your debugging output. See "DebugCharSet" on page 398 for more information.

Shutdown and standby buttons

You can put shutdown and standby buttons on the Bookshelf at boot time by assigning values to two new variables, **ShutDownButton** and **StandByButton.**

The values of the variables indicate the position of the buttons in the Bookshelf.

A value of 1 puts the button on the far left side of the Bookshelf. For example, these lines put the shutdown and standby buttons next to each other on the left side of the Bookshelf.

```
ShutDownButton=1
StandByButton=2
```

Versions and trademarks

In PenPoint 1.0, the three variables **Version, Trademark,** and **CommVersion** had multiline strings, with text items separated by the vertical bar (|) symbol.

In PenPoint 2.0 Japanese, the version and trademark information is represented with single-part strings. Everything else is stored in a PenPoint resource file.

```
Version=2.0
Copyright=1992
CommVersion=1991-1992
```

Note that the **Trademark** variable has been replaced by the **Copyright** variable.

If your code depends on the PenPoint 1.0 strings, you must change the code to reflect the new values and variable name. GO discourages you from writing code that depends on these strings.

Start application

The variable used to define the default initial application, **StartApp**, has been removed from the default ENVIRON.INI. If you want to specify your own initial application, add a line to ENVIRON.INI defining the **StartApp** variable.

Its value should be the complete path and file name of the initial application. The file name must contain only ASCII characters.

Autozoom

The **Autozoom** setting has been removed from ENVIRON.INI. It is now stored in the resource file associated with the Bookshelf application. The resource file contains the name of the document that is to be automatically zoomed.

BkshelfPath

The **BkShelfPath** variable identifies the path to the default contents of the Bookshelf. When PenPoint 2.0 Japanese boots, it copies the contents of this directory into the Bookshelf.

Uncommenting the **BkShelfPath** line in by the default ENVIRON.INI causes PenPoint to load the Help notebook and several sample documents into the main PenPoint notebook.

This description updates the description on pages 36 and 37 of the *PenPoint Development Tools*.

Debugging flags

Some common debugging flags are set in ENVIRON.INI with the **DebugSet** variable. You can set any debugging flag in **DebugSet**.

- /D*1 works only if you run the debug version of PenPoint 2.0 Japanese.
  It directs the heap manager to validate heaps after any heap allocations or deallocations. This validation degrades performance by about 15 percent and dramatically slows screen layout.
- /DD8000 sends a copy of the debugger stream to the file specified in the **DebugLog** variable in ENVIRON.INI. See "Viewing the debugger stream" on page 397 for details.

- /DB800 causes your PenPoint boot volume (the hard drive from which you booted PenPoint) or RAM (if you specified **DebugRAM** mode), to appear in the Connections notebook. Set this flag to copy files between your hard drive (or RAM) and the PenPoint file system.

Also, page 34 of *PenPoint Development Tools* says that you should modify \PENPOINT\BOOT\ENVIRON.INI to enable logging by uncommenting the line

```
#DebugSet =/D*1 /DD8000
```

You should make sure the first **DebugSet** line is commented out:

```
#DebugSet=/D*1
DebugSet=/DD8000 /DB800 /D*1
```

PenPoint ignores duplicate lines in initialization files.

BOOT.DLC

The file BOOT.DLC has moved from \2_0\PENPOINT\BOOT to \2_0\PENPOINT\BOOT\USA and \2_0\PENPOINT\BOOT\JPN.

Do not put comments in any .DLC files, including BOOT.DLC. Comments sometimes cause PenPoint to fail booting.

Where necessary, the version numbers for DLLs have been changed to 2.0. Compared to the 1.0 version, the 2.0 version of BOOT.DLC loads additional DLLs. The exact list of additional DLLs varies between locales.

If you have a .DLC file that refers to a PenPoint DLL, you must update the file to use the DLL's new version number. For example, the Notepaper App sample application uses NOTEPAPR.DLL. In 1.0, NPAPP.DLC contained these lines:

```
GO-NotePaper-V1(0) notepapr.dll
GO-NOTEPAPER_APP-V1(0) npapp.exe
```

In PenPoint 2.0 Japanese, NPAPP.DLC contains:

```
GO-NotePaper-V2(0) notepapr.dll
GO-NOTEPAPER_APP-V2(0) npapp.exe
```

Interpreting Japanese file names

Japanese names in the initialization files, such as SERVICE.INI and APP.INI, have English translations in the comments above them.

Repeated lines

Make sure you do not try to set a variable twice. PenPoint uses the first assignment if you have two lines trying to assign a value to a variable. For example, if your MIL.INI contains the lines:

```
ScreenType=Std480
ScreenType=SuperScriptII
```

PenPoint uses the **Std480** screen parameters.

Printing to a spooler

The PenPoint SDK 2.0 Japanese includes a special printer service called PRSPOOL. Use PRSPOOL to print PenPoint files to a spool file on a DOS disk. You can later copy this spool file to a printer. This procedure lets you print a PenPoint document without a tablet computer and with no printer attached to your PC. Remember that end-user versions do not support printing to a spooler.

1 Make sure you load the Out box services. Because printers create sections in the Out box, you cannot create a printer with Out box support.

2 Install PRSPOOL by uncommenting its line in SERVICE.INI.

3 Install a printer driver by uncommenting the appropriate line in your SERVICE.INI file. Alternatively, install the service by opening the Disk page of the Connections notebook. Select the Services view, and tap on an Install box to install service. HP LaserJet printers use the PCL service.

4 Turn to the Printers page of the Connections notebook. Create a new driver with the caret ∧ gesture. Choose a driver in the pop-up list.

5 Enter a name for this virtual printer.

6 Enable the printer by tapping on the Enable box.

7 Turn to the document you want to print.

8 Set any special page layout properties (headers, footers, margins, and so on) by choosing Print Setup in the document menu.

9 Print the document by tapping on the Print command. You should see the Out box icon change to full, then eventually return to empty.

10 Exit from PenPoint and go to the root of your PenPoint directories. The default root is \2_0 for PenPoint 2.0 Japanese.

11 The documents you printed will be named PRFILE, PRFILE_1, PRFILE_2, and so on in your root PenPoint directory. Print these from DOS by copying them to a port. For example, this line sends the file to the printer attached to LPT1:

```
copy PRFILE LPT1: /b
```

The /B flag tells DOS to use binary mode, which prevents DOS from interpreting print driver control characters as end of file markers.

If you print many documents, your PenPoint directory information may get out of sync. You may want to clean up your directory with a batch file like this:

```
# Delete any spooler files
del \PRFILE*.*
# Clean out redundant entries in PENPOINT.DIR
psync /B /D \ /V
```

The arguments to PSYNC direct the utility to creates a backup PENPOINT.BAK file and lets you know what files are being cleaned up.

Long DOS file names

The PenPoint document model creates a directory for each file. Each embedded document is contained in a separate subdirectory within the directory that contains the parent document. This recursive structure could create DOS file names longer than the maximum of 64 characters.

See "Using short DOS path and file names" on page 391 for strategies on creating distribution disks without violating this 64-character limit.

Do not use CHKDSK /F

Do not use the DOS utility CHKDSK with the /F flag if your PenPoint file system contains DOS path names longer than 64 characters.

The DOS utility CHKDSK skips all the files whose path names are longer than 64 characters and marks the clusters used by those as lost. Running CHKDSK with the /F flag will free those erroneously marked clusters, thereby corrupting your file system.

PenPoint returns **stsFSVolCorrupt** when it tries to read this file system. Even worse, you may lose data before seeing this warning from PenPoint if those incorrectly freed clusters are allocated and used by other files.

You can use CHKDSK without any parameters to check for path names that are too long. To navigate to those files, you need a DOS utility to shorten the path names through renaming.

For example, say you have the following path name on a disk:

```
B:\2_0\PENPOINT\APP\MY_APP\STATNRY\MY_APP_Q\NOTEBOOK\ . . . \DOC.RES
```

Running CHKDSK on B: yields the following output:

```
c:\>chkdsk b:
Errors found, F parameter not specified
Corrections will not be written to disk
1 lost allocation units found in 1 chains.
       512 bytes disk space would be freed
   1457664 bytes total disk space
      4096 bytes in 8 directories
   1453056 bytes available on disk
       512 bytes in each allocation unit
      2847 total allocation units on disk
      2838 available allocation units on disk
    655360 total bytes memory
    473856 bytes free
```

Running CHKDSK /F instead of CHKDSK would have erroneously freed the single allocation unit reported above.

Changes to QuickHelp

In PenPoint 1.0, you specified special characters like the GO gesture glyphs by changing the font to Symbol, and using the /F63 keyword.

In PenPoint 2.0 Japanese, you can specify a Unicode character representing the special character by using the \x*hhhh* where *hhhh* is a 4-digit hexadecimal Unicode value. GO has placed its gesture fonts in the Unicode corporate zone from 0xF6600 to 0xF700. The letter gestures share the same code as the corresponding letter, so the codes are scattered between 0x0041 and 0x005A. Look in the header file GLYPH.H for exact code assignments.

The keyword /F63 is no longer recognized in PenPoint 2.0 Japanese.

Working with different locales

You can boot PenPoint with different user and system locales as described in "GO batch file" on page 410.

Only you as a developer can take advantage of this locale switching behavior. GO will never ship an end-user system that supports mixed locales. Consequently, always do your final user testing with the same system and user locales.

Here are a few other things to note when you boot with a USA user locale:

- To enable kana-kanji conversion or romaji-kana conversion, you must install the KKC engine. You can install the engine turning to the Installed Software page of the Settings notebook, and tapping on the Install menu. The KKC engine icon has Japanese characters that say VACS VJE. To load the engine at boot time, add the KKC engine name to \2_0\PENPOINT\BOOT\USA\SERVICE.INI. Copy and paste the name from \2_0\PENPOINT\BOOT\JPN\SERVICE.INI
- To enable Japanese handwriting recognition, you must install the Japanese handwriting recognition engine.

Corrections to previous documentation

PenPoint Development Tools errata

TABLE 32-1

Page and section	*Old text followed by correction*
Page 25, line 3	The simulation is imperfect (no static RAM, no pen-on-screen interaction, and so on) The simulation is imperfect (for example, no pen-on-screen interaction)
Page 36 Line 6 of Table 3-5	Specifies the when to flush the debug log to a file. Specifies when to flush the debug log to a file.
Page 43, paragraph 1	Repeats next to last paragraph of previous page. Disregard it.
Page 54	The section on "Volume Selection" belongs on page 37.
Page 60, paragraph 2	Use the UniPenPort tag in MIL.INI to select a predefined protocol. Use the **UniPenType** tag in MIL.INI to select a predefined protocol.
Page 61 Line 2	You will probable have to "tune" these...for the specific characteristics of your digitizer. You will probably have to "tune" these...for the specific characteristics of your digitizer.
Page 61, Section 3.18.2.1	The digitizing resolution of the AceCat5by5 and the MM are listed as 19,500. That should be corrected to 19,685.
Page 61 Section 3.18.2	The tags are UNIPENCOMPORT...UNIPENPROTOCOL, UNIPENPROTOCOL The tags are UNIPENCOMPORT...UNIPENYPROTOCOL, UNIPENPROTOCOL.
Page 146 Section 12.2	Typing ? displays the available mini-debugger commands. Typing h displays the available mini-debugger commands.
Page 162 Section 14.1	*Part 6: File System* in the *PenPoint Architectural Reference*, explains file system... *Part 7: File System* in the *PenPoint Architectural Reference*, explains file system...

Part 5 / PenPoint Architectural Reference Supplement

Chapter 33 / Overview

This document provides material that supplements the *PenPoint Architectural Reference* manual published for version 1.0 of the PenPoint Software Developers Kit (SDK). It describes architectural concepts and API definitions and procedures that are new with both PenPoint SDK 1.0 and PenPoint SDK 2.0 Japanese. It also offers programming tips, suggests workarounds for known bugs, clarifies some concepts and procedures, and amends conceptual, procedural, and typographical errors found in the earlier manuals.

We sometimes use the names "PenPoint 2.0" and "PenPoint SDK 2.0" in this document. Because this release of PenPoint has been localized only to Japan, these terms refer to the PenPoint 2.0 Japanese operating system and the PenPoint SDK 2.0 Japanese.

About this supplement

Intended audience

This book is intended for PenPoint application developers who are familiar with the two-volume *PenPoint Architectural Reference* or who have access to it. Ideally, you should read the *Supplement* with the earlier volumes at hand. This manual makes frequent references to these manuals, particularly when it corrects or clarifies them.

If you do not have copies of the version 1.0 manuals, you can still learn much that is useful and interesting to PenPoint application developers from this document, particularly the material new to PenPoint SDK 2.0 Japanese. But to get the full value of the information presented in the following pages, you should treat the *Supplement* as a companion document to its predecessors.

Document structure

The structure of *Part 4: PenPoint Architectural Reference Supplement* is simple. Each chapter after this Overview chapter is mapped to a part of the earlier *PenPoint Architectural Reference.* Thus Chapter 28 is entitled "The Class Manager," the same title as Part 1 of the earlier document; Chapter 29 is entitled "The Application Framework," and so on. The last chapter, "International Routines and Services," covers functionality entirely new to PenPoint 2.0 so it has no counterpart in the version 1.0 manuals.

Within each chapter there are up to three major sections. The first section, "What's New," describes concepts and defines interfaces and functions that are new since the PenPoint 1.0 SDK. "Tips and Clarifications," the second section, gives some suggestions on programming and clarifies areas that might have caused confusion. The final section, "Corrections and Errata," catalogs typographical errors and amends sections in the earlier manuals that were inaccurate.

PenPoint 2.0 Japanese

Fundamental changes

The major difference between PenPoint 2.0 Japanese and earlier versions is that PenPoint 2.0 Japanese contains general modifications to support languages other than English and specific modifications to support the Japanese language. This support required three major changes:

- PenPoint expects strings to consist of 16-bit characters.
- Most text strings for display have been moved to resource files.
- Gestures are now Unicode values.

The following sections expand briefly on these changes. For a full description of these changes, please see the *Part 2: PenPoint Internationalization Handbook.*

PenPoint 2.0 Japanese supports only American English and Japanese. However, the modifications present in PenPoint 2.0 SDK Japanese provide most of the features necessary for supporting other languages in the future.

16-bit characters

Almost all strings in PenPoint 2.0 are represented by 16-bit characters, using the Unicode standard encoding. This global modification implies a number of other changes. For example, the CHAR type is 16 bits wide and all the U...() string functions of the standard C library expect 16-bit characters.

Text strings moved to resource files

Most text strings displayed in PenPoint 2.0 Japanese have been moved to resource files. There are currently two versions of each resource file in the PenPoint operating system; the file USA.RES contains American English strings; the file JPN.RES contains Japanese strings.

The resource files that PenPoint uses are determined by the setting of the **Locale** and **LocaleUser** environment variables in ENVIRON.INI. See *Part 4: PenPoint Development Tools Supplement* for more information on these variables.

While we have worked to ensure that all strings have been moved to resource files and translated to Japanese, a small number of strings might have escaped our notice. If you find one of these strings in the Japanese version, please notify GO Developer Technical Support.

Gestures are now Unicode values

Earlier versions of PenPoint encoded gestures as 32-bit IDs. In PenPoint 2.0 Japanese, gestures are encoded as 16-bit Unicode values. Unicode values further separate the character used for a gesture and its meaning.

If you use gestures, you must change the ID for each gesture to the Unicode for that gesture. The Unicode values for gestures and standard User Interface (UI) icons and symbols are in GLYPH.H.

New sample code

The PenPoint SDK 2.0 Japanese includes four new sample applications:

- Keisen Table Application (Japanese only), project name KEISEN
- Serial I/O Demo, project name SXDEMO
- Video Player, project name VIDPLAY
- UI Companion, project name UICOMP

Additionally, two samples that were previously released via CompuServe (LBDEMO and SAMPLMON) are now part of the sample code in the PenPoint SDK 2.0 Japanese.

There are also a number of changes to the 1.0 and 1.0.1 SDK samples. Most of them (with the exceptions of EMPTYAPP, HELLO, HELLOTK, BASICSVC and MILSVC) have had their text strings moved to resource files. (All of them have been ported, obviously.) All of the earlier sample code has also been updated to use the Bridging Package; the sample code runs under both the 1.0 and 2.0 Japanese SDKs.

General code and API changes

Library changes

PenPoint 2.0 Japanese contains a new library, INTL.LIB, which contains many of the functions that you use to store Unicode strings and to get information pertaining to the current locale.

Please see the documents shipped with the WATCOM compiler for changes in the PENPOINT.LIB file.

The library BRIDGE.LIB is empty in the PenPoint SDK 2.0 Japanese but is provided for makefile compatibility when using the Bridging Package. The Bridging Package allows you to compile your applications under both PenPoint 1.0 and PenPoint 2.0 Japanese. See the *PenPoint Bridging Handbook* for details.

New header files

The header files listed in Table 33-1 have been added to PenPoint SDK 2.0 Japanese. Descriptions of these files' contents occur in the appropriate chapter of this document.

New header files TABLE 33-1

File	Contents
GLYPH.H	Unicode values for PenPoint glyphs and gestures.
KKC.H	Definitions for **clsKKC**, the class that communicates with the kana-kanji conversion service.
PANOSE.H	The API for the PANOSE™ Typeface Matching System. (PANOSE is a trademark of ElseWare Corporation, Seattle, Washington.)
STDSTR.H	Tags for UI Toolkit strings.
ISR.H	Header for international routines in INTL.LIB.
ISRSTYLE.H	Definitions for styles used by routines in ISR.H (included by ISR.H).
INTL.H	Macros for building and manipulating locale values; also the U_L() macro.
ALAYOUT	Definitions for **clsAcetateLayout**, a descendent of **clsNotePaper**, used for layout.
GOLOCALE.H	Definitions for locale-related constants used by the ISR routines. Replaces LOCALE.H
CHARTYPE.H	Definitions for character types and macros for international character manipulations.
CHARTR.H	Definitions for **clsCharTranslator**, the character translator abstract class.
BRIDGE.H	Definitions that allow developers to maintain the same source code for both 1.0 and 2.0 Japanese SDKs.
KKCCT.H	Definitions for **clsKKCCharTranslator**, the character kana-kanji translator class.

Changes for resource files and tags

The following table lists the header files that have been changed to support resource files and tags.

Header files changed for resource strings TABLE 33-2

File Name	Change
APPTAG.H	All Standard Application Menus (SAMs), all standard option card titles, default document name, company, copyright for 16-bit filename and classname, used in building app dir. obsolete: **tagAppMgrDefaultDocName**, **tagAppMgrDisplayedAppName**.
APPWIN.H	Icon win Quick Help. Articles, miscellaneous strings, and errors.
BATTERY.H	Errors, warnings, and toolkit strings.
CBWIN.H	Cork board window Quick Help.
GOTO.H	Reference button Quick Help and miscellaneous strings.
HWGEST.H	Toolkit string and gesture names.
HWLETTER.H	Miscellaneous strings.
ICONWIN.H	Icon window layout option card title string.
POWERUI.H	Power button string.
PREFS.H	Toolkit and miscellaneous strings.
QHELP.H	Quick Help, toolkit and miscellaneous strings.
RCAPP.H	Root container application name and document name.
SYSTEM.H	Warnings.

Name changes of data elements

Table 33-3 presents some of the major name changes of data structures, constants, and variables since version 1.0 of PenPoint. As you can see from scanning the table, many of these name changes reflect the transition to Unicode. This list is *not* comprehensive; for instance, name changes of enumerated and defined values are not included.

Some data name changes

TABLE 33-3

Header file	*Data type*	*Old name*	*New name*
VOLGODIR.H	typedef	LV_NATIVE_NAME	LV_NATIVE_FS_NAME
XFER.H S	typedef	XFER_ASCII_METRIC	XFER_STRING_METRIC
	tag	xferASCIIMetrics	xferStringMetrics
SENDSERV.H	typedef	ADDR_BOOK_ATTR	SEND_SERV_ATTR
STDIO.H	constant	_ERR	_SFERR
TXDATA.H	typedef	BYTE_INDEX	TEXT_INDEX
XLIST.H	typedef	X2GESTURE	GWIN_GESTURE
XSHAPE.H	typedef	XS_ASCII_MATCH	XS_LATIN1_MATCH
	(see names)	U32 gestureId	CHAR16 gestureId
	typedef	XS_ASCII_MATCH	XS_TEXT_MATCH
	array	asciiMatch[xsMaxCharList]	textMatch[xsMaxCharList]
	U16	matchArraySize	matchArrayLength

Chapter 34 / Class Manager

What's new

The Class Manager contains no new API, functions, or other features.

Tips and clarifications

Using keys

Readers should read Section 2.7 of the *PenPoint Architectural Reference* with the following caveat in mind: you must use unique constants for keys on classes created by distributed DLLs. The keys must be kept private, or others may be able to subvert or delete these classes. Particularly, do not use **ObjWknKey** as the key.

Many applications use a function pointer or pointer to a method table as a key when creating a class. This will cause a problem if the class is replaced or upgraded; the new key won't match the old one, so class creation by the new DLL fails. The problem typically surfaces in distributed DLLs, because two versions of one can be running at the same time. It does not cause a problem when upgrading applications, because old and new versions of the application do not co-exist.

Don't use msgScavenged

msgScavenged is an obsolete message. It is never sent by the Class Manager, and its message number is the same as **msgFreeSubtask**.

Don't send or respond to this message. If you see **msgScavenged** in any debugging output, the actual message sent was probably **msgFreeSubtask**.

Posting msgDestroy

You can use **ObjectPost** to deliver messages at a later time, which is usually understood to be when the handling of the current message is complete. If, however, a system modal note is displayed during the execution thread of a message handler, all messages that are currently in the input queue are delivered. Thus a posted message is delivered before the current execution thread is unwound, and this case causes severe problems if the posted message is a destructive message, particularly **msgDestroy**.

PenPoint guarantees that a posted **msgDestroy** will not be delivered until the current execution thread has unwound from the current method handler. If you have code or logic that depends on a posted **msgDestroy** getting through while a system modal note is up, you will need to rethink your logic.

You should also carefully think about any other destructive messages you post which may get delivered before you unwind from the current execution thread.

Corrections and errata

ObjectSend()

Section 2.5.1 in *PenPoint Architectural Reference* contains the paragraph:

> When you send a message with **ObjectSend()**, your code's task is suspended while waiting for the message handler to return. Your task resumes operation when the message handler returns.

These statements are not quite accurate. When you send a message with **ObjectSend()**, it is true that the sending task waits for a return status. However, it can still handle incoming messages while it is waiting. After it sends a message, **ObjectSend()** responds to any one of the following events:

- The return status from the called object, after which the calling task continues to the next line.
- A task-terminated indication, upon which the task continues to the next line.
- Any incoming **ObjectSend()** messages for objects owned by the tasks, which the waiting **ObjectSend()** dispatches. Specifically, the task's flow of control jumps to the method that handles the incoming message, returns a status, and resumes waiting for one of the three events.

appVersion and minAppVersion

The fields **appVersion** in OBJ_RESTORE and **minAppVersion** in OBJ_SAVE are incorrectly documented in the file CLSMGR.H. These 16-bit fields are no longer used by PenPoint and should not be used by your code.

Change in title

Rename the title of section 4.4.5 of the *PenPoint Architectural Reference* from "Getting a Class's Class," to "Getting a Class's Ancestor."

Typographical errors

Part 1 (Class Manager)—typos TABLE 34-1

Volume, section, paragraph	*Old text on first line* *New text on second line*
I, Preface, vii	Code example: pInst->>placeHolder = -1L Code example: **pInst->placeHolder = -1L**
I, 2.4, ¶2	Code example (twice): if (s stsOK) Code example (twice): **if (s < stsOK)**
I, 4.3.1, ¶2	The message taks a pointer to an OBJ_NOTIFY_OBSERVERS structure... The message takes a pointer to an OBJ_NOTIFY_OBSERVERS structure...
I, 4.6.3, ¶2	An object must be in prepared to handle msgDestroy... An object must be prepared to handle **msgDestroy**...

Chapter 35 / Application Framework

What's new

Document recovery message

This section applies to **clsApp**.

The Application Framework includes a new **clsApp** message that enables your application to respond appropriately when its resource file is corrupted. The message is **msgAppRecover**; its **pArgs** argument points to the handle (DIR_HANDLE) of the resource file and returns STATUS.

When **msgAppRestore** fails, the application object self-sends **msgAppInit**. Respond to this message as you would when creating a document (initialize your instance data). **clsApp** then sends **msgAppRecover** to its descendants so that they can modify their instance data. It passes in the handle (DIR_HANDLE) to the resource file; if this handle is set to **objNull**, then the resource-file object was not found or was damaged.

clsApp descendants should respond to **msgAppRecover** by doing something to handle the error condition:

- They can reset their instance data to a state different than that of a just-created document.
- If they are passed a handle to the resource file, they can salvage as much data as they can. (Make sure that the handle is a valid object, and not **objNull**.)
- They can determine the cause of the error and display a message that is more informative than the standard PenPoint error message.
- They can simply return **stsOK**. By doing this, the data is lost, but the document can be re-opened without losing any embedded documents.

If descendents choose not to handle **msgAppRecover**, PenPoint displays a standard error message and does not recover the document. If this happens, the document cannot be re-opened and all embedded documents are lost.

Initialization DLL

This section applies to **clsAppMon**.

To conserve memory, you can include an initialization DLL in your application's installation procedure. An initialization DLL typically contains code that your application needs to execute only once, such as code that creates UI components.

When installation occurs, the application monitor installs the initialization DLL in the loader database, runs it once and then deinstalls the DLL code before it installs the application code. The objects created by the DLL code are saved to a resource file before the installation of the application.

To use an initialization DLL in you application's installation, complete the procedure outlined here. The examples presented are from the UI Companion sample code (UICOMP):

1 In the application project directory, create a DLL source file named INIT.C. The entry point must be **InitMain**, not **DLLMain**. The code should perform some once-only initializations, such as building the application's UI components. In the application's MAKEFILE, you can link the INIT.C file to existing source files (see item 4, below).

2 Create a file called INIT.LBC and in it list all exported functions defined in the initialization DLL. In most cases, the only exported function is **InitMain**. Each line in the file has the form:

```
++entry-point.'company ID-project-major version(minor version)'
```

Thus the sole entry in the UICOMP example of INIT.LBC is:

```
++InitMain.'GO-UICOMP-V1(0)'
```

3 Create a file that has the project name and an extension of .DLC (for example, UICOMP.DLC). This file expresses the dependencies between an application's executable file and that application's DLLs. Each line of the file pairs the name for a PenPoint executable or DLL with the DOS path to the corresponding executable or DLL file (relative to the application directory in \PENPOINT\APP). When PenPoint loads an application, it reads the .DLC file to determine which DLL files to load before it installs the application. The UICOMP.DLC file, for example, has these two lines:

```
GO-UICOMP_DLL-V1(0)     uicomp.dll
GO-UICOMP_EXE-V1(0)     uicomp.exe
```

4 Finally, include several lines in the application's MAKEFILE to specify the linker name, object files and libraries for the initialization DLL. The following example from UICOMP is typical:

```
INIT_LNAME = GO-UICOMP_DLL-V1(0)
INIT_OBJS = init.obj buttons.obj lists.obj menus.obj
INIT_LIBS = penpoint resfile
```

When you build the initialization DLL, the resulting DLL file (such as UICOMP.DLL) is placed in the same location as the application's executable. This is the default initialization DLL file. When installation begins, **msgAMLoadInitDll** is sent to the application monitor, which then looks in the application directory for the initialization DLL file. If you subclass **clsAppMonitor**, we recommend that your subclass not respond to **msgAMLoadInitDll**.

New and obsolete tags

New tags defined in APPTAG.H allow application writers to define the following text strings in a resource file:

tagAppMgrAppDefaultDocName The default document name for the application. This appears in the Create menu and the Stationery notebook. If one is not assigned by the application, the **tagAppMgrApp-Filename** string is used.

tagAppMgrAppCompany The name of the company creating the application.

tagAppMgrAppCopyright The copyright date information for the application. The Unicode code point \x00A9 can be used for the copyright symbol in the string. (This is defined in GLYPH.H, but must be literally included in the string.)

tagAppMgrAppFilename The name of the application or service. This appears in the Settings notebook and Installation time of the application. This is also the stamped name of the application when it is built. It is also the name that is entered in the APP.INI file.

tagAppMgrAppClassName The type of executable being created: Application, Service, and so on.

The resource that contains these strings is **resAppMgrAppStrings**. Two Application Framework tags are now obsolete: **tagAppMgrDefaultDocName** and **tagAppMgr-DisplayedAppName**.

Tips and clarifications

Unimplemented flag for msgPrintGetProtocol

Do not set the **paginationMethod** flag to **prPaginationScale** (PRINT_PROTOCOLS) when you send **msgPrintGetProtocol**. Pagination scale is not implemented and using it can block printing.

Note *The PenPoint Architectural Reference does not specifically mention this flag.*

Printed document and msgSave

Developers should be aware that their documents, when activated by the print wrapper, receive a **msgSave** before **msgAppOpen**. This order, of course, is the reverse for a screen document. In summary, a printed document receives **msgInit**, **msgRestore**, **msgSave** and **msgAppOpen**, in that order.

Class defaults for clsAppMonitor subclasses

When you create a subclass of **clsAppMonitor**, set the following CLASS_NEW fields after sending **msgNewDefaults** to **clsClass**:

new.cls.pMsg Assign to this field a pointer to the method table for your subclass.

new.cls.size Set this to the size of your class instance data. This data is usually defined in some structure such as MY_INST_DATA. In this case, assign **SizeOf**(MY_INST_DATA) to the size field.

new.cls.ancestor Set this to the class from which you want your class to inherit its behavior: **clsAppMon**.

new.cls.newArgsSize Set this field to the size of the structure that the **msgNewDefaults** and **msgNew** messages for class instantiation take as an argument. For some developers who have tried to subclass **clsAppMon**, this field has caused some confusion. This class has no _NEW_ONLY structure of its own and so no lamination takes place. When this situation occurs, you must provide the size of the ancestor class's **newArgs** structure. **clsAppMon**'s ancestor is **clsApp**, so you assign **Sizeof**(APP_NEW) to **newArgsSize**.

For most subclasses that you create, you may recall, you send **msgNewDefaults** and **msgNew** to **clsClass**. But when you create subclasses of **clsAppMon** and other application classes, send these messages to the application superclass **clsAppMgr**.

Page sequencing and msgAppMgrCreate

When you create a new document with **msgAppMgrCreate**, set the **sequence** field (APP_MSG_CREATE) to a sequential number, with the parent application being 0. Thus, if you want your document to be the first thing in the parent application, set sequence to 1. Page numbers are global sequence numbers and not attributes. PenPoint 2.0 Japanese keeps track of the number of children of a document, and can compute page numbers from that.

By the way, always set the **renumber** field to TRUE unless you are going to create another document with **msgAppMgrCreate** immediately afterward.

Corrections and errata

msgSave

Item 2 of the numbered list in section 8.2.5.2 of the *PenPoint Architectural Reference,* says that "**clsApp** sends **msgResWriteObject** to the resource file handle with the document's main window as the object." This is incorrect. **clsApp** sends **msgResPutObject** to the resource file handle.

Reactivating a document

Item 5 of the numbered list in section 8.2.6 of the *PenPoint Architectural Reference* (**clsApp** sending **msgResReadObject** to the resource file handle) is redundant and should be deleted.

Getting attributes for many application directories

Section 16.6 of the *PenPoint Architectural Reference* contains a few inaccuracies. First, you should not use **msgAppDirGetNextInit** in obtaining the attributes of document directories. Instead, use **msgAppDirGetNext** only.

You cannot specify a starting point in **pFirst**. PenPoint 2.0 Japanese sets and resets this member (and **pNext**) internally, regardless of what you assign to them. Instead, assign zero to the handle member and complete **attrs**, **pName**, and **fsFlags** as specified. After the last iteration through an application directory, **msgAppDirGetNext** returns **pNull** in **pNext**. Send the message once before going into the while loop because on the first iteration **pNext** is **pNull**. When it completes, **msgAppDirGet-Next** returns **stsOK** if a directory is found and **stsNoMatch** if none is found.

In addition to the foregoing errors, the section states that "you must send **msgAppDirGetNextInit** to **clsAppDir**." Instead, you send **msgAppDirGetNextInit** and **msgAppDirGetNext** to an instance of **clsAppDir**.

Terminating a document

Sections 8.2.5 and 8.2.5.1 of the *PenPoint Architectural Reference* contain a few errors in the actual order of messages that occur when a user closes a document. The Notebook does not terminate a document by sending it **msgFree**. Also, the first few sentences of 8.2.5.1 give the impression that, after an application frees its objects, it calls its ancestor (**clsApp**), which sends **msgAppSave** to save these just-freed objects.

When a user closes a document, the following exchange of messages occur up to the point at which the document (application instance) itself receives **msgFree**:

1 The Notebook sends **msgAppTerminate** to the document; the document does not handle this message, but lets it percolate up to **clsApp**.
2 If **msgAppTerminate** is sent with **pArgs** of TRUE, **clsApp** self-sends **msgFreeOK**; if the document doesn't respond to this message, **stsOK** is assumed.
3 If the document responds positively (OK to free), **clsApp** self-sends **msgAppSave**. Descendents do not usually handle **msgAppSave**.
4 **clsApp** then self-sends **msgDestroy**.
5 As a result, the document receives **msgSave** and then **msgFree**.

The descriptions in section 8.2.5.1 from the third sentence to the end of the section are correct.

Handling msgAppTerminate

Section (13.4.1.2) of the *PenPoint Architectural Reference* can be deleted. **msgApp-Terminate** is no longer sent to application monitors.

Typographical errors

Part 2 (Application Framework)—typos

TABLE 35-1

Volume, section, paragraph	*Old text on first line* *New text on second line*
I, 7.4, ¶11	Resourcetl files. Resource files.

Chapter 36 / Windows and Graphics

What's new

PANOSE typeface matching

Clients using the font API (see section 26.12 in the *PenPoint Architectural Reference*) can now access GO's implementation of the PANOSE™ Typeface Matching System. The PANOSE system defines certain values for typographical attributes in various scripts, or writing systems (GO currently supports only Latin characters and kanji). These attributes include genre (display text, decorative, symbols, and so on), weight, monospace, contrast, ratio, slant, tool type, stroke type, and so on.

Developers of text-processing applications might find the PANOSE API useful. They must specify the required PANOSE values in structure **PANOSE_MEM** in a nibble format (two values per byte). A set of macros is provided for inserting and extracting these values. Two utility functions, **PanoseToXDR** and **PanoseFromXDR** enable the conversion of PANOSE numbers to and from their XDR representations, something you must do before filing away the numbers and reading them back in to your application.(XDR stands for eXternal Data Representation.)

When you have built your PANOSE_MEM structure, insert it into the structure SYSDC_FONT_DESC. Then pass a pointer to SYSDC_FONT_DESC when sending **msgDCGetFontDesc** and **msgDCSetFontDesc**. This causes **msgDCGetFontDesc** to fetch the DC's current font state and **msgDCSetFontDesc** to set the drawing context's current font state. These messages replace **msgDCOpenFont**.

The API definitions of the PANOSE structures, functions, macros, and definitions are in PANOSE.H. The API definitions of the drawing-context messages are in SYSGRAF.H. The definition of **SYSDC_FONT_DESC** is in SYSFONT.H.

Unicode values for gestures and system UI

The file GLYPH.H contains the Unicode values for the standard gestures and for common UI icons, symbols and other graphics.

Tips and clarifications

Filing window resources

If you file a window with **msgResPutObject**, that window must not have **wsFileInline** set in its style flags.

This is likely to concern you if you allow windows and components (such as reference buttons) to be embedded in your view and you also file those objects yourself. This is because the view files the objects too, and if **wsFileInline** is set, then you'll end up with two copies when you restore them.

The solution is that whenever you deal with a embedded window being added (normally in your response to **msgWinInsertOK**), make sure that the **wsFileInline** bit is turned off and the **wsSendFile** bit is set the way you want (usually on).

Receiving msgWinVisibilityChanged

msgWinVisibilityChanged is sent only when a window's **wsVisible** flag changed. When the window is extracted, **msgWinIsVisible** will return true, but the flag will not be changed unless you explicitly change it.

The comments for the message in WIN.H are not correct, because they imply that it will be sent on insertion. **msgWinExtracted** is sent on extraction, and then you can check the **wsVisible** flag.

Windows and WKNs

Developers should not create window instances with private or process well-known UIDs. The window system maintains a PenPoint-global database of all windows, and it expects each UID to be unique.

You can use private well-known UIDs for classes, not for instances.

Corrections and errata

The current grafic

The third paragraph of section 27.5.2 the *PenPoint Architectural Reference* states in the:

> Note that **clsPicSeg** allocates the memory for the grafic-dependent data structure from the process heap, but it is up to the client to free it with **OSHeapBlockFree()**.

This statement is not complete. By system default, if **pData** is **pNull**, **clsPicSeg** allocates the memory for the data structure from the local process heap (**osProcessHeapId**); otherwise, it uses the heap that you pass in. If you want to have the heap shared between your process and another process, assign **osProcessSharedHeapId** as the default heap when you create your **clsPicSeg** object (**new.object.heap**). You must call **OSHeapBlockFree()** to deallocate both **osProcessHeapId** and **osProcessSharedHeapId.**

Repaint

Section 28.2.3 in the *PenPoint Architectural Reference* begins with two sentences that describe an obsolete message:

> A TIFF object repaints when it receives **msgPicSegRedraw**. Since a TIFF object isn't a window and isn't bound to one, you must pass in a drawing context as the message argument to **msgPicSegRedraw**.

These descriptions are no longer true. To repaint a TIFF object in its display list, a **clsPicSeg** object now must send it **msgPicSegPaintObject**, passing the TIFF object a pointer to PIC_SEG_PAINT_OBJECT. Specify the painting rectangle in logical units and assign the UID of the drawing context or **PicSeg** object to the **picSeg**. Ignore all other fields.

Using a bitmap

Replace the second paragraph of section 28.1.1 with in the *PenPoint Architectural Reference* the following text:

Having created a bitmap, you usually want to get it on the screen in some form. One way to display a bitmap is to send an instance of **clsBitmap msgBitmapCache-ImageDefaults**. This message takes a pointer to the SYSDC_IMAGE_CACHE structure used by **msgDcCacheImage. clsBitmap** fills in the structure with default values. You can send the bitmap **msgDcCacheImage**, and then send it **msgDc-CopyImage** to have the sampled image stored in the bitmap rendered in the window.

You can also display bitmaps by creating instances of **clsIcon** or one of its descendants. See *Part 4: PenPoint Development Tools Supplement* for a step-by-step procedure. The UI Companion sample application uses this procedure to create an instance of **clsIconToggle** to display some bitmaps. The source code and comments are in \2_0\PENPOINT\SDK\INC\SAMPLE\UICOMP\UICOMP.C.

Typographical errors

Part 3 (Windows and Graphics)—typos

TABLE 36-1

Volume, section, paragraph	*Old text on first line* *New text on second line*
I, 22.1, ¶3	However, every application requires that you design at least one custom sublcass of clsWin... However, every application must use a subclass of **clsWin**...
I, 23.6, ¶1	Applications somtimes require windows to have a particular size... Applications sometimes require windows to have a particular size...
I,26.12.3.1,¶2	The function SysDcFontID performs this algorithm... The function **SysDcFontID()** performs this algorithm...

Chapter 37 / UI Toolkit

What's new

UI components with built-in KKC translation

clsField and **clsIP** objects have built-in support for translation of Japanese characters. If you want to create your own UI client of the kana-kanji conversion (KKC) character translator, see "The character translator classes" on page 453. If you want to subclass **clsKCC** to make your own interface to a KKC engine, see "Kana-kanji conversion class" on page 451.

References to page numbers are to the PenPoint Architectural Reference, Volume I.

Text highlighting and "dirtying"

clsLabel now provides a new flag and message with which you can give text selections one of two highlight styles. One highlight style (strong highlighting) is the standard dark grey rectangle with inverted text. The other highlight style (weak highlighting) encloses the selection in light grey and underlines it.

The new message is **msgLabelProvideHighlight**. Because **clsLabel** self-sends this message, you must create a subclassed instance of **clsLabel**. When you change the LABEL_NEW_ONLY structure defaults for this object, set the **label.style.getHighlight** flag to TRUE. (Set only the **label.style.stringSelected** flag to TRUE if you want only the standard highlighting style.)

During repaint operations (**msgWinRepaint** or **msgBorderPaintForeground**), **clsLabel** self-sends **msgLabelProvideHighlight** if the **getHighlight** or **stringSelected** flags are set. For each area to be highlighted, your subclassed object must specify the span information in a LABEL_SPAN structure and highlight style that **clsLabel** needs to draw the highlight graphic. (**clsLabel** ignores this highlight field if the **stringSelected** flag is set to TRUE.)

Next, put these LABEL_SPAN blocks in the **spanBuf** array of a LABEL_HIGHLIGHT structure, sorted by increasing index. Set the **pSpans** field of LABEL_HIGHLIGHT to point to the start of this buffer. When you send **msgLabelProvideHighlight** to **clsLabel**, pass it a pointer this LABEL_HIGHLIGHT structure. (**clsLabel** sets the SYSDC_RGB fields of the structure.)

Another new message, **msgLabelDirtySpan**, is related to **msgLabelProvideHighlight**. Send **clsLabel** this message, passing it a pointer to LABEL_SPAN, to have it dirty the area indicated by the span. The highlight information for the span is ignored.

Standard strings

STDSTR.H contains tag definitions for the standard UI Toolkit strings that were moved to resource files (such as "OK," "Apply and Close," and "Contents").

clsKbdFrame

A new class, **clsKbdFrame**, provides generalized behavior for simulated keyboards. It supports input filtering and the queuing of character events. PenPoint's virtual keyboard and its Unicode Browser, for example, make use of **clsKbdFrame**. The immediate ancestor of **clsKbdFrame** is **clsFrame**. API definition for **clsKbdFrame** is in KBDFRAME.H.

Some of the queuing functionality of **clsKbdFrame** will be replaced in future PenPoint versions by generalized improvements to the input system.

Acetate Layout and Markup classes

Two classes have been added to PenPoint to increase the markup functionality of **clsNotePaper**: **clsAcetateLayout** and **clsMarkup**. Essentially, **clsMarkup** implements the transparent, scalable, and rotatable data-drawing layer, and **clsAcetateLayout** overlays an instance of **clsMarkup** on top of an application window. You can also use **clsAcetateLayout** to lay one window over another, thereby enhancing gesture handling.

Using clsMarkup and clsAcetateLayout together

When you want to add a markup layer to your application, you can use **clsMarkup** and **clsAcetateLayout** in a complementary way. **clsAcetateLayout** synchronizes the scrolling of a **clsMarkup** window that is layered over a client application window. It also handles document embedding so that documents are embedded only in the client window.

You can think of **clsAcetateLayout** as an intermediate layer in the window hierarchy. It mediates between the **clsMarkup** view and the application's own view. **clsAcetateLayout** treats the application window as its client (in this context, a client is a window that is subordinate to the acetate layout in the window hierarchy).

To implement this behavior, insert the instances of **clsAcetateLayout** and **clsMarkup** when your application handles **msgAppOpen**. After creating the application's scrolling window, instantiate the Acetate Layout, after setting its **client** field to the scrolling window. The Acetate Layout, in turn, has as its child the application's view (that is, the Opaque View) and the Markup View, layered so that the Opaque View lies below the Markup View.

clsAcetateLayout

When you want to lay a window atop another window, you must typically put up with a lot of drudgery to handle pass-through of gestures and to synchronize the behavior of the two windows. **clsAcetateLayout** enables an application to implement a markup overlay atop its window without having to implement gesture and event pass-through or graphical markup.

Do not confuse **clsAcetateLayout** with the PenPoint windowing system's "acetate" layer.

As a subclass of **clsCustomLayout**, **clsAcetateLayout** allows an application to place and correctly lay out a client window (usually a markup layer) atop the application window. To synchronize the two windows, it handles the messages described in Table 37-1.

***clsAcetateLayout* synchronization messages** TABLE 37-1

Message	*Takes*	*Description*
msgScrollbarVertScroll	P_SCROLLBAR_SCROLL	Client should perform vertical scroll.
msgScrollbarHorizScroll	P_SCROLLBAR_SCROLL	Client should perform horizontal scroll.
msgScrollbarProvideVertInfo	P_SCROLLBAR_PROVIDE	Client should provide the document and view information for a vertical scroll.
nsgScrollbarProvideHorizinfo	P_SCROLLBAR_PROVIDE	Client should provide the document and view information for a horizontal scroll.
msgScrollWinProvideSize	P_SCROLL_WIN_SIZE	Self-sent to determine bubble location and size.
msgScrollWinProvideDelta	P_SCROLL_WIN_DELTA	Self-sent so that descendants can normalize the scroll.

clsMarkup

Developers can use **clsAcetateLayout** to help them use **clsMarkup**. **clsMarkup** is a subclass of **clsNotePaper** optimized so that developers can have graphical markup tools for a document without requiring their applications to know markup. **clsMarkup** is a transparent instance of **clsNotePaper** with the additional ability to scale and rotate. **clsMarkup** provides useful annotation functions, but does not perform smart markup; that is, the annotations are not tied to the marked-up application data. You could use **clsMark** (or an appropriate subclass) to tie the items together.

Tips and clarifications

clsBorder tracks on pen down

clsBorder doesn't start tracking until it receives **msgPenDown**. If your application consumes **msgPenDown** as part of a press ↓ gesture (to create a move icon, for example), it must self-send a new **msgPenDown**, which tells **clsBorder** to start tracking.

Progress bars

When advancing a progress bar, you must advance it by an amount greater than zero, or your application may page fault.

XList handlers must handle msgGWinGesture

If a gesture window has timeout events enabled, and a hold timeout is initiated by the user, the gesture window converts the **inputHoldTimeout** event directly into a **msgGWinGesture**, rather than going through the normal protocol of sending out an **XList** that will get self-converted to a gesture. If your application is processing **XLists** rather than gestures, you must add a handler for **msgGWinGesture**.

Field change

As a result of better hand printing recognition, there is a change in FIELD.H related to character box memory. **fstBoxMemoryFour** is replaced with **fstBox-MemoryTwo**, which uses two characters of box memory. Existing code that uses **fstBoxMemoryFour** will still compile; however, it will only use two-character box memory.

Bug in clsLabel

After you create a label object, **clsLabel** resets the **label.style.infoType** field to **lsInfoString**. This causes problems particularly if you had earlier set **infoType** to **lsInfoStringId** so that you could read strings from a resource file; attempts to get new strings for the label object by sending **msgLabelSetStringId** do not succeed, Until this bug is fixed, work around it by setting the **label.style.infoType** field to **lsInfoStringId** before sending **msgLabelSetStringId.**

Bug in clsToggleTable

Instances of **clsToggleTable** have handlers that override certain **clsControl** messages. As **clsControl** defines it, four of these messages (**msgControlGetDirty**, **msg-ControlSetDirty**, **msgControlGetEnable** and **msgControlSetEnable**) take as P_ARGS a pointer to a 16-bit BOOLEAN value. But **clsToggleTable** defines the P_ARGS for these same messages as a pointer to a U32 data type, bits of which it reads or toggles before returning the bitmask to the caller. Page faults can occur as a result of this discrepancy, particular with the **msg...Get...** messages.

As a workaround for **msgControlGetDirty** and **msgControlGetEnable,** declare a U32 variable for the return value and pass a pointer to it. For **msgControlSetDirty** and **msgControlSetEnable**, just be aware of the discrepancy when you send these messages to **clsToggleTable.**

Corrections and errata

UI Toolkit programming details

The first paragraph of section 31.5.4 in the *PenPoint Architectural Reference* suggests that you can create your application's UI in a separate INIT.DLL and when the application's "**DLLMain** routine is called by the Installer, create the UI." The entry point for INIT.DLL should be named **InitMain**, not **DLLMain; DLLMain** gives INIT.DLL its own process.

For information on creating an initialization DLL for your application, see "Initialization DLL" on page 435 of this document.

Incorrect table reference

On the top of page 386 of *PenPoint Architectural Reference* (section 34.4.2) the definition of the **constraint** field refers to Table 34-2. It should be Table 34-3.

Providing custom backgrounds

The description of the BORDER_BACKGROUND fields in section 33.4.5 in the *PenPoint Architectural Reference* omits the **borderInk** field. The value you assign to this field specifies the color of a graphic object's bordering line. Typical values are color constants such as **bsInkGray66** and **bsInkBlack** (the default).

Typographical errors

Part 4 (UI Toolkit) — typos

TABLE 37-2

Volume, section, paragraph	*Old text on first Line* *New text on second line*
I, 46.6, ¶17	Assuming that clsMyView does not create a custom sheet, othen... Assuming that clsMyView does not create a custom sheet, ...
I, Table 40.3	Title: SCROLLWIN_STYLE Styles *Title*: SCROLL_WIN_STYLE Styles
	verticalScrollbar vertScrollbar

Chapter 38 / Input and Handwriting Recognition

What's new

Two sections in this chapter present information on two new character-translation components: the kana-kanji conversion class (**clsKKC**) and the character-translation classes, **clsCharTranslator** and **clsKKCCharTranslator**. The information in these sections pertain to developers who:

- Want their application to handle text entry, particularly direct text entry ("The character translator classes").
- Want to write their own classes to support kana-kanji conversion ("Kana-kanji conversion class" and "The character translator class").
- Want to implement their own KKC engine ("Kana-kanji conversion class").

Kana-kanji conversion class

The kana-kanji conversion class (**clsKKC**) provides default superclass behavior for kana-kanji conversion (KKC) engines. PenPoint's KKC engine, developed for PenPoint 2.0 Japanese, is based on this superclass. **clsKKC** inherits from **clsService**; KKC engines are implemented as services. Table 38-1 shows the relationship of **clsKKC** and the other classes related to translation of Japanese characters.

The API defined in KKC.H is primarily for developers who want to port exiting KKC engines to PenPoint 2.0 Japanese. In PenPoint, all KKC engines inherit from **clsKKC**. This class provides substantial default behavior for its descendents, thereby simplifying the work of porting.

Developers may also want to be direct clients of a **clsKKC** service and provide their own user interface to the conversion engine. Although this is possible, the character translator API for KKC (defined in CHARTR.H and KKCCT.H) already provides a rich, high-level, international protocol and a sophisticated user interface that is build into objects of **clsField** and **clsIP**.

The PenPoint KKC engine has a RKC (romaji-kana) component that converts romaji into hiragana characters as they are typed or written. Then, given the proper gesture or keyboard command, the engine converts the hiragana or katakana characters in the proximate bunsetsu (phrase context) into a list of kanji alternatives for each kana character. It presents these alternative characters to the user in a pop-up window.

Generally, **clsKKC** and its subclasses operate by processing data contained in an **XList**. For KKC engines, an **XList** must contain only two types of elements, **xtText** and **xtKKCSpan**. The **xtText** elements contain unconverted text and **xtKKCSpan** elements hold text that has already been converted.

A KKC span in an **XList** contains, in addition to the converted text, information such as conversion alternatives and the decomposition of text into stem and ending. While there can be multiple conversion alternatives associated within a given span, only one of them can be the display choice.

A KKC engine service can always construct a display string from the **XList**. The display string consists of the ordered concatenation of each **xtText** element and each **xtKKCSpan** display choice for that element. The engine operates on the **XList** by specifying indices into the display string.

The API definition in KKC.H provides many more details about using **clsKKC** interfaces, both for porting purposes and as a client. For information on writing services, refer to Part 13 of the *Penpoint Architectural Reference.* For information on using services, see Chapter 94 of Part 10, "Connectivity." If you are interested in porting KKC engines, you can contact GO Technical Support to obtain a copy of the "KKC Porting Kit."

clsKKC messages

clsKKC messages TABLE 38-1

Message	*Takes*	*Description*
msgKKCConvertSingle	P_KKC_CONVERT	Produces a list of conversion alternatives for a range of text. Subclass responsibility.
msgKKCConvertMultiple	P_KKC_CONVERT	Converts all unconverted text. Subclasses have the option of implementing this message.
msgKKCConvertRange	P_KKC_CONVERT	Converts all text in the specified range as a single span. Subclasses have the option of implementing this message.
msgKKCUnconvertSingle	P_KKC_CONVERT	Converts specified kanji text back to hirigana. Subclasses may optionally implement this message to provide "reverse henkan" functionality.
msgKKCAccepted	P_XLIST	Client sends when user accepts current choice. A subclass option.
msgKKCGetMetrics	P_KKC_GET_METRICS	Fetches information about the current **XList**. Useful in determining the length of the display string and the number of KKC spans and text elements.
msgKKCSetChoice	P_KKC_SET_CHOICE	Changes the current choice for the given span. Superclass responsibility.
msgKKCAlterSpan	P_KKC_ALTER_SPAN	Extends or shortens the boundaries of a converted string. Superclass responsibility.
msgKKCChangeText	P_KKC_CHANGE_TEXT	Inserts, deletes and replaces text in the display string. Superclass responsibility.
msgKKCGetChars	P_KKC_GET_CHARS	Extracts a substring from the given **XList** and puts it in a buffer. Superclass responsibility.

clsKKC messages TABLE 38-1 (continued)

Message	*Takes*	*Description*
msgKKCInsertSpan	P_KKC_INSERT_SPAN	Creates, initializes and inserts a new **xtKKCSpan.** A KKC engine can self-send this message to have the superclass manipulate the **XList** when the engine needs to supply KKC results to the client object.
msgKKCFindElement	P_KKC_FIND_ELEMENT	Finds the element that contains the character of the display string specified by an index. A KKC engine self-sends this message to convert an incoming display index into usable text. The superclass passes back the element index, the string index and the element itself.
msgKKCDumpXlist	P_XLIST	Prints the contents of the **XList** to the debugging console. Valid only in DEBUG mode.
msgKKCInitialize	P_KCC_INITIALIZE	Performs service initialization for a KKC engine. Subclasses should send this message to **clsKKC** as part of their **DLLMain()**. Clients should never send this message.
msgKKCRKC	P_KKC_RKC	Subclasses can implement this message so that they can change the default behavior of the superclass' romaji-to-kana conversion algorithm.

The character translator classes

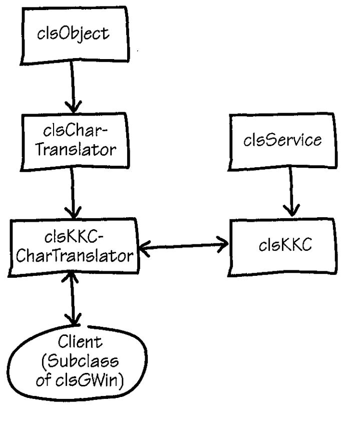
Character translators assist in the translation of characters from one set to another and in the presentation of translation alternatives to users. Working together with their client UI components, character translators create the user interface for translations of character sets. They also act as intermediaries between their UI clients and the engines that perform the character-set conversions.

PenPoint SDK 2.0 Japanese has two new classes related to character-set translation. The abstract class **clsCharTranslator** specifies the standard interfaces and implements the standard methods for the translations. It inherits from **clsObject.**

Because **clsCharTranslator** is an abstract class, only a subclass of it can realize the latent functionality for a particular character set. PenPoint SDK 2.0 Japanese provides the KKC character translator class for this purpose. **clsKKCCharTranslator** is a subclass of **clsCharTranslator**; objects of this class act as clients to kana-kanji conversion engines through interfaces defined in **clsKKC.**

The sketch to the right depicts the relationship of these classes. The API definition for **clsCharTranslator** is in CHARTR.H. The API definition for **clsKKCCharTranslator** is in KKCCT.H.

Relation of translator classes



Creating a client of KKC character translator

Because a character translator requests gesture information, its clients usually are instances of **clsGWin** or one of **clsGWin**'s subclasses (although, strictly speaking, they don't need to be). Client objects exchange messages with the KKC character translator class, **clsKKCCharTranslator** (and, by inheritance, with **clsChar-Translator**). The KKC character translator requests translation services from the KKC conversion-engine class (**clsKKC**) and receives back translation alternatives, which it forwards to the client.

This section is intended for developers who, instead of using **clsField** or **clsIP**, want to create their own **clsGWin** descendent client of the KKC character translator.

Developers who want to implement their own UI interface for kana-kanji character conversion in an application must set certain fields in their client object's _NEW_ONLY structure that derive from **clsGWin**. Then they must have their client object observe a specific protocol.

During **msgInit**, every instance of **clsGWin** (or one of its subclasses) can create a character translator that handles translation requests. You must first indicate that you want a translator created by setting the **new.gwin.style.useCharTranslator** field TRUE. Then assign to **new.gwin.charTrLocaleId** field the locale identifier (LOCALE_ID) for the translator. If you specify no locale ID (by setting **charTr-LocaleId** to zero), **clsGWin** creates a translator appropriate to the system locale. The default translator for Japan (**locJpn**) is an instance of **clsKKCCharTranslator.**

Rather than generalizing the protocol for all possible client objects of **clsGWin**, we can describe the protocol as a client object of **clsIP** actually implements it:

1 The user writes a few kana characters in an insertion pad, then requests KKC with the right up ⅃ gesture. When the pad receives the gesture, it self-sends the message **msgCharTransGesture**.

2 Rather than handling the message itself, **clsIP** allows **clsGWin** to handle the message. In turn, **clsGWin** sends the message to the character translator it created when it responded to **msgInit**. Again, for PenPoint SDK 2.0 Japanese, the default translator is an instance of **clsKKCCharTranslator**.

3 When the character translator (an instance of **clsKKCCharTranslator**) receives the gesture information, it determines whether the gesture is relevant to character translation. Since the right up ⅃ gesture explicitly requests KKC, it sends the message **msgCharTransGetClientBuffer** to the client (**clsIP**) requesting a portion of its buffer.

4 The client sends the requested characters in response to **msgCharTransGet-ClientBuffer.**

5 The translator communicates with **clsKKC**, the front-end to the actual service that provides KKC. In this case, a translation is needed, so the translator sends **msgCharTransModifyBuffer** with the translation to the client.

6 Using information sent with **msgCharTransModifyBuffer**, the insertion pad updates its internal buffer and user interface to display the translated character. Note that the translated character is highlighted. The P_ARGS sent with **msgCharTransModifyBuffer** also contains highlighting information. See *Part 6: PenPoint User Interface Design Reference Supplement* for details on how character highlighting should behave during KKC.

7 The user then requests a list of alternatives by tapping on the highlighted character. The insertion pad self-sends **msgCharTransGesture**, again allowing the message to be handled by **clsGWin**.

8 The translator receives the message from **clsGWin** and queries **clsKKC** for character alternatives. It also asks the client where the character alternatives pop-up box should be placed by sending **msgCharTransProvideListXY**.

The insertion pad calculates the coordinates of the upper-left corner of the pop-up box. The pop-up box should appear directly below the original character.

9 If the user selects an alternative from the pop-up box, the translator sends **msgCharTransModifyBuffer** to the client. The insertion pad should then update its buffer and user interface.

10 When the user taps OK to dismiss the insertion pad, the pad should self-send **msgCharTransGoQuiescent** to synchronize the character counts between the text view and the character translator. This step ensures the correct setting and clearing of the weak and strong character highlights.

The description above does not exhaust the messages involved in the character translation protocol. For example, it did not mention any of the messages that support keyboard input. These are the most important messages involved in the protocol:

The client should self-send the following four messages when appropriate. However, the client should not define a method to handle the message. Rather, the client should allow the message to be passed up to **clsGWin**.

- Self-send **msgCharTransKey** each time the user presses a key.
- Self-send **msgCharTransChar** each time the user edits an existing buffer (for example, when the user inserts or deletes a character). As the user writes new characters, you normally do not send this message until the user makes the translation gesture. However, when the user is typing, you send each character with this message.
- Self-send **msgCharTransGoQuiescent** to cancel the current translation. When the user taps outside an insertion pad, for example, **clsIP** self-sends **msgCharTransGoQuiescent**.
- Self-send **msgCharTransGesture** each time the user makes a gesture on your text.

The client should respond to the following messages sent by the character translator:

- **msgCharTransModifyBuffer**, which contains information on how to translate characters. The client should respond by updating your text buffer and user interface, including updating strong and weak highlighting. The character translator passes you a CHAR_TRANS_MODIFY structure.
- **msgCharTransGetClientBuffer**, which asks your window instance for some text from the client's buffer. Pass the requested text to the character translator as part of a CHAR_TRANS_GET_BUF structure.
- **msgCharTransProvideListXY**, which asks the client where to put the character alternative list. The client should compute the coordinates so that the pop-up box appears below the original character.

Highlighting information

When a character translator sends **msgCharTransModifyBuffer** to a client, that client should examine the **highlight** fields in CHAR_TRAN_HIGHLIGHT. It should begin weak highlighting from **weakStart** and extend it for **weakLen**. Strong highlighting should begin from **strongStart** and extend for **strongLen**. Previous highlighting information is provided in **oldWeakLen**, **oldStrongStart** and **oldStrongLen**.

Some of the length fields can hold 0, indicating that highlighting can be removed. For example, if there is no strong highlighting required, **strongLen** is 0. The client may need to clear any old highlighting that is specified by the **old...** fields. Note that if the character translator just wants to change highlighting information, it will send this message with the **delete** and **insert** lengths of CHAR_TRAN_HIGHLIGHT set to 0, but with the highlight information changed.

Class Character Translator messages

clsCharTranslator messages TABLE 38-2

Message	*Takes...*	*Comments*
msgNewDefaults	P_CHAR_TRANS_NEW	CHAR_TRANS_NEW is passed in with all arguments set to zero (or **pNull**, as appropriate).
msgNew	P_CHAR_TRANS_NEW	Creates a character translator. If **pArgs->charTrans.pBindings** is **pNull**, the default bindings (from the system preferences) are used.
msgCharTransKeyEvent	P_CHAR_TRANS_KEY_EVENT	Clients self-send this message to notify a character translator of a keyboard event. If the character translator does not use the key, it returns an error message. Otherwise, the client should not use the key because it is being used by the translator.

clsCharTranslator messages

TABLE 38-2 (continued)

Message	*Takes...*	*Comments*
msgCharTransChar	P_CHAR_TRANS_CHAR	The client self-sends this message to notify a character translator of a character about to be changed. The client then receives back **msgCharTransModifyBuffer**; it should examine the arguments passed it in P_CHAR_TRANS_MODIFY to determine exactly what the character translator wants to have changed.
msgCharTransSetMark	P_CHAR_TRANS_SET_MARK	The character translator sends this message to the client to notify that it (the translator) is beginning to collect characters at the given position.
msgCharTransGetClientBuf	P_CHAR_TRANS_GET_BUF	The translator sends this message to the client to request a copy of the characters in the client's buffer. The client should copy **length** characters from **startPosition** into **buf**. If fewer than **length** characters are available, the client must end the string that it copies with a null character.
msgCharTransModifyBuffer	P_CHAR_TRANS_MODIFY	**clsCharTranslator** (or a subclass of it) sends this message to the client to tell it how to modify its buffer. The client should delete **length** characters beginning at **first** and replace them with **bufLen** characters from **buf**. The client should also adjust its highlighting according to the values in the highlight structure (CHAR_TRAN_HIGHLIGHT). See "Highlighting information" on page 456.
msgCharTransProvideListXY	P_CHAR_TRANS_LIST_XY	Sent to the client to request the X-Y coordinates for the top left corner of the pop up menu for the alternatives list. Current UI guidelines dictate that the client should compute the coordinates so that the menu pops up below the character. The coordinates are in relation to the root window.
msgCharTransListActivate	P_CHAR_TRANS_LIST_XY	Self-sent to subclasses to activate the alternatives list. **pArgs->charPosition** has the character position in the client's buffer and **pArgs->xy** has the root window X-Y coordinates for the list.

Return of translation alternatives

The text subclasses for handwriting translation, **clsXText** and **clsXWord**, include new flags that request the translation object to return information in addition to the best-guess translation. Most other translation flags (**hwxFlags**) govern which of the various scoring rules the translation object applies when it chooses the best translation. The new translation flags specify which additional data that object is to return:

- **xltReturnAltWords**: Return the highest ranking alternative word translations.
- **xltReturnAltChars**: Return the alternative characters in each position of the best-guess translation.
- **xltReturnStrokeIds**: Return the strokes that belong with each character of the best-guess translation.

You can set one or more of these flags in the _NEW structures for both **clsXText** and **clsXWord** (**pArgs->xlate.hwxFlags**) when you create the translation object. You can also set and clear them with **msgXlateSetFlags** and **msgXlateClearFlags** any time before the translation object has received the first stroke from the scribble. To find out the current **hwxFlags** settings, send **msgXlateGetFlags** to the translation object.

To get the information on alternatives, send **msgXlateData**, to the **clsXText** or **clsXWord** object as usual; pass it via P_XLATE_DATA an identifier of the heap from which memory is to be allocated for the **Xlist** elements. The translation object returns the requested information in linked **Xlist** elements (see Table 38-1).

xltReturnAltWords The translation object returns, along with the best-guess translation for a word (as defined by **Locale**), a list of alternative word translations, ranked in order of their scores. The **Xlist** element requested by this flag is of type **xtTextAltWords**, which points to a WORD_LIST structure that contains the alternative word choices.

xltReturnAltChars The translator returns a list of all alternatives for each of the characters in the best-guess word translation. Each alternative's plausibility is determined by the translator's shape matcher. The **Xlist** element requested by this flag is of type **xtTextAltChars**, and it points to a XL_CHAR_LIST structure that holds alternative character information.

xltReturnStrokeIds The translator returns the pen strokes that underlie every best-guess and alternate character in the word translation. The **Xlist** element requested by this flag is of type **xtTextStrokeIdList**, and it points to a structure of XL_STROKE_ID_LIST.

If you set the **xltReturnStrokeIds** flag, you might also want to set **xltReturnAltChars**. You can use the stroke-count information returned via the **xtTextAltChars Xlist** element to interpret the stroke IDs returned via the **xtTextStrokeIdList Xlist** element.

The linked list of **Xlist** elements returned for each type of alternative information (word, character, and stroke ID) is often extended beyond a single translated word. The translator can link that word's sequence of **Xlist** elements with the **Xlist** elements returned for the next translated word.

All **hwxFlags** are defined in the header file for the abstract superclass, **clsXtract/ clsXlate** (XLATE.H).

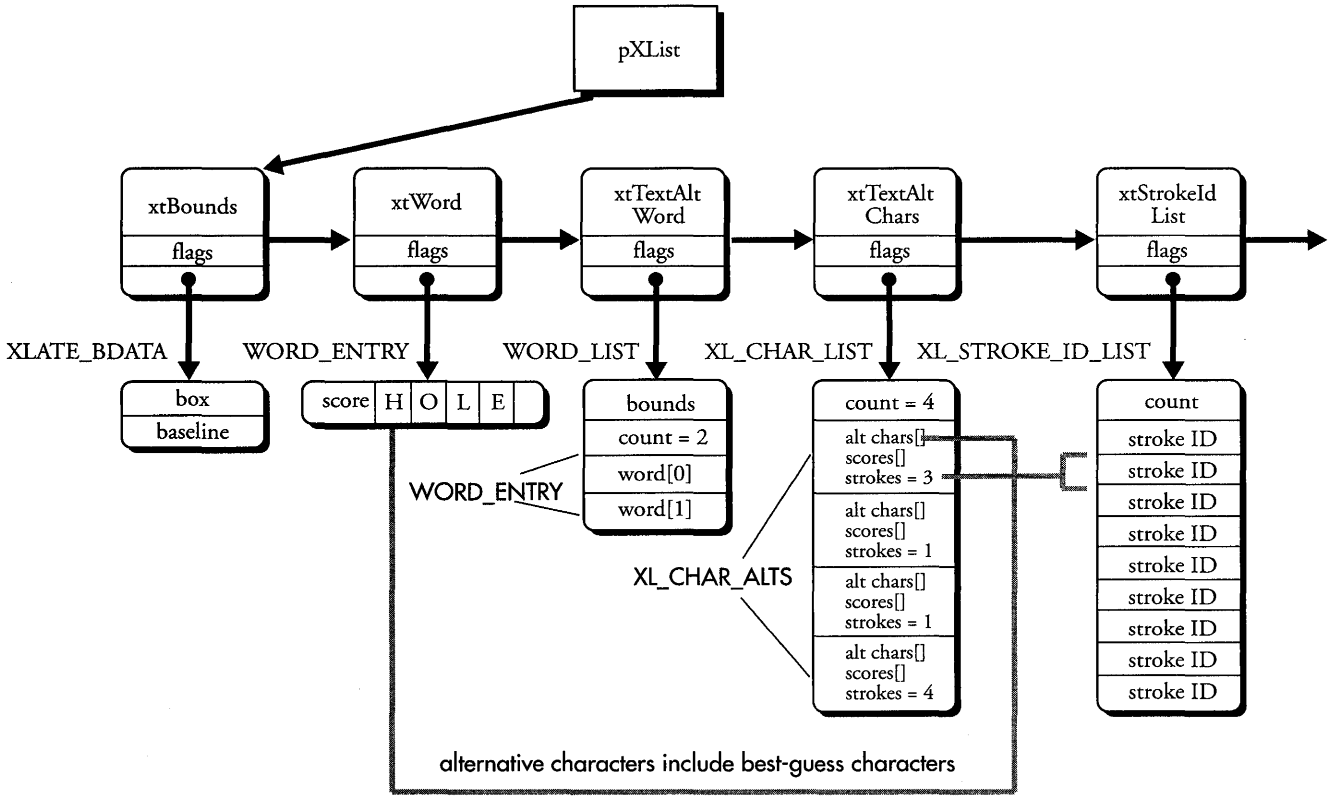
Handwriting changes

Handwriting customization has been removed as a feature from PenPoint 2.0 Japanese. Because this version of PenPoint does not use the GOWrite engine, there is no implementation of handwriting customization.

The header file HWCUSTOM.H remains, as does the hook to the Customize... button on the Settings notebook's Installed Handwriting page. ISVs who wish to use customization may write their own **clsFrame** descendants conforming to the HWCUSTOM.H header; customization will proceed as it did in PenPoint 1.0 and

Translation alternatives returned by msgXlateData

FIGURE 38-1



PenPoint 1.01. But the customization classes that GO provided in PenPoint 1.0 and PenPoint 1.01 are not part of PenPoint 2.0 Japanese.

Letter practice removed

The header file HWLETTER.H remains, as does the hook to the Practice... button on the Settings notebook's Installed Handwriting page. ISVs who wish to use letter practice may write their own **clsFrame** descendants conforming to the HWLETTER.H header; letter practice will proceed as it did in PenPoint 1.0 and 1.01. But the letter practice classes that GO provided in PenPoint 1.0 and 1.01 are not part of PenPoint 2.0.

Changed and obsolete gesture names

Changed gesture names

TABLE 38-3

Gesture...	*Is Now...*
xgsLLCorner	xgsDownRight
xgsLLCornerFlick	xgsDownRightFlick
xgsLRCorner	xgsDownLeft
xgsLRCornerFlick	xgsDownLeftFlick
xgsULCorner	xgsUpRight

Obsolete gesture names TABLE 38-4

xgsAsterisk	xgsBordersOn	xgsCircleDblTap
xgsDblArrow	xgsDblDownCaret	xgsDownTriangle
xgsFlickTapDown	xgsFlickTapLeft	xgsFlickTapRight
xgsFlickTapUp	xgsInfinity	xgsLeftCaret
xgsLineCaretLeft	xgsLineCaretRight	xgsLineDblCaret
xgsParagraph	xgsPigtailHorz	xgsPlusTap
xgsPolyline	xgsRect	xgsRightCaret
xgsRoundRect	xgsSpline	xgsUpCaretDblDot
xgsUpTriangle		

Tips and clarifications

clsAnimSPaper metrics

Instances of **clsAnimSPaper** will crash (divide by zero) if they are redrawn with the delay and interstroke metrics both set to zero. To avoid the problem, ensure that the interstroke is always non-zero whenever the delay is zero.

Transparent input

If you want transparent input (**inputTransparent** set), you must make sure that **inputLRContinue** is clear.

Corrections and errata

Adding a filter

Section 53.3.2 in the *PenPoint Architectural Reference* on **InputFilterAdd()** shows an incomplete prototype and does not describe two of the function's arguments (page 571). The actual prototype is:

```
STATUS EXPORTED InputFilterAdd(
      OBJECT            newFilter,
      INPUT_FLAGS       inputEventFlags,
      FILTER_FLAGS      filterFlags,
      U8                priority
);
```

The arguments are defined as follows:

newFilter The UID of the filter object to be placed on the filter list.

inputEventFlags By setting flag bits in this U32 field, you indicate those input events (and related messages) that you want your filter to handle. Examples of input events that you can specify are **inputTip**, **inputEnter**, and **inputTap**. See INPUT.H or Table 53-2 in the *PenPoint Architectural Reference* for a list of these flags.

filterFlags The flags set in this U32 field control event distribution to your filter. Currently developers can set only one flag, **iflSendMyWindowOnly**. By setting it you instruct the input system to withhold messages from the filter unless the event happened in the filter or in one of that filter's window children or window ancestors.

priority A value from 0 to 255 that indicates the relative priority of the filter. This value specifies the position of the filter in the list.

Typographical errors

Part 5 (Input and Handwriting Translation)—typos

TABLE 38-5

Volume, section, paragraph	*Old text on first line* *New text on second line*
I, 52.4, ¶1	...win.input.flags... ...win.flags.input...

Chapter 39 / Text

What's new

Gesture targeting

In PenPoint 2.0 Japanese, **clsTextView** targets gestures differently than in other PenPoint versions. In PenPoint 2.0 Japanese, where a gesture would expand its target to a word (bunsetsu in the Japanese version), the gesture targets the character where the hotpoint of the gesture was. As in PenPoint 1.0x or PenPoint 2.0 running with U.S. behavior, however, if the gesture is over a selection, then the selection is the target.

Type the following to run PenPoint with U.S. behavior:

go usa usa

Note, however, that GO has no plans to ship a end-user U.S. localization of PenPoint 2.0.

Table 39-1 lists the gestures that have new (non-bunsetsu) targets in PenPoint 2.0 Japanese. The Like-Type gestures select similar contiguous characters (that is, those characters that are all kana or all kanji) instead of following the standard bunsetsu selection rules.

New gesture targets TABLE 39-1

Gesture Type	*Gesture*	*Description*	*Target*
Insertion	⌟	New paragraph	between characters
	⌟̳	New line	between characters
	⩓	Embedder	between characters
	∧	Floating input pad	between characters
	∧̇	Embedded input pad	between characters
Selection[1]	[	Select to left	between characters
	]	Select to right	between characters
Like-Type	F	Find selected word	similar contiguous characters
	B	Bold	similar contiguous characters
	I	Italic	similar contiguous characters
	N	Normal	similar contiguous characters
	U	Underline	similar contiguous characters
	↑	Increase font size	similar contiguous characters
	↓	Decrease font size	similar contiguous characters

1. In PenPoint 2.0 Japanese, these selection gestures have additional behavior. Making a right bracket gesture before a selection is equivalent to a single tap (select character); making a left bracket gesture after a selection is equivalent to a double tap (select word).

In terms of API, a new atom type, **atomLikeType**, enables the selection of a span that conforms to the international style **intlDlmtWordStyleWord**. (This new atom type is in TXTDATA.H.) In addition, move and copy operations for PenPoint 2.0 Japanese now target to the character instead of to the word.

Font substitution algorithm

When users now type or write in a Latin or non-Latin font, and a certain character is missing, the PenPoint 2.0 Japanese operating system substitutes a character from the closest matching font. This substitution is based on an algorithm that uses the PANOSE™ Typeface Matching System.

PANOSE is a trademark of ElseWare Corporation, Seattle, Washington

Each font has a selection of Unicode points that identify its characters, which are mapped to glyphs. If a user-requested glyph is not available in a logical font, the font-substitution algorithm uses the PANOSE number of that font to get an ordered list of related fonts, sorted by distance from the original font. It scans the fonts in this list until it finds the glyph, and then substitutes it.

With the Heisei or Mincho fonts selected, PenPoint 2.0 Japanese converts typed or written Latin letters and numbers to the closest matching Latin font, and displays proportionally spaced (hankaku) glyphs. If you don't want this substitution, you can select the appropriate option from the Convert menu or make the right arrow gesture to convert the Latin Unicode points to the Compatibility zone equivalents that display full-width monospaced glyphs. Typing or writing kanji or kana with a Japanese font selected results in no substitution. The full-width (zenkaku) glyphs are used for display.

Initial spaces are always displayed as half-width unless the user converts them through the Convert menu.

Hankaku/zenkaku implementation

In **clsTextView**, all Latin letters including the space character (0x0020) and the Latin punctuation characters (period, question mark, and so on) default to their hankaku form. All other kana, kanji, and Japanese punctuation characters default to their zenkaku form. However, the keyboard driver for the JPN keyboard has mode switches with which you can control the kinds of characters generated. In addition, users can convert characters to all hankaku or all zenkaku via the Gestures or MiniText menus.

Unicode import type

MiniText now supports the Unicode file type (specified in FILETYPE.H). If the import file has the .UNC extension, the code points will be interpreted as Unicode. Other imported files are treated as either 7-bit RTF or Shift-JIS depending on the header. If the file starts with a valid RTF header, it will be read as 7-bit RTF, otherwise it will be treated as a Shift-JIS file. Because there is no standard for exporting 8-bit (that is, Shift-JIS) characters in RTF format, PenPoint no longer exports RTF in the 2.0 Japanese version; it does export text as Unicode and Shift-JIS.

No white space correction in Japanese version

Because Japanese doesn't delimit words with spaces, PenPoint 2.0 Japanese does not correct white space during:

- Move/copy and delete operations.
- While accepting translated text from either an embedded IP or a floating IP.

Taboo and bunsetsu rules

Lines breaks follow taboo processing rules for Japanese text. In addition to the English or European characters that cannot start or end a line, extra Japanese characters have been added that cannot start or end a line. For example, you can't end a line with an open bracket ([) and you can't start a line with a close bracket (]).

Word selection follows the Japanese rules for selecting bunsetsu.

Using msgTextModify

Because text views set the **gWin style.useCharTranslator** to TRUE, the text view character count must be synchronized with the character translator to ensure the correct setting and clearing of the weak and strong highlights. Always end a KKC session with **msgCharTransGoQuiescent** before you send a **msgTextModify** message to a text view. To end the KKC session, send a message similar to the following:

```
    ObjectCallWarn(msgCharTransGoQuiescent, self, pNull);
```

In this message, **self** is the text view object. **clsGWin** handles this message by self-sending messages to clear the weak and strong highlights before the **msgTextModify** message can change the character counts. Character offsets are thus synchronized between the character translator and the text view object.

The text view itself ends the KKC session in response to gestures that add and delete characters. See CHARTR.H for more information on the character translator.

Corrections and errata

Typographical errors

Part 6 (Text)—typos TABLE 39-2

Volume, section, paragraph	*Old text on first line* / *New text on second line*

I, Chpt 67 intro, ¶2	Code example: s = ObjectCall(msgWinInsert, new.object.id, &new.win);
	Code example: s = ObjectCall(msgWinInsert, new.object.uid, &new.win);

Chapter 40 / The File System

What's new

Stamped file system attributes

The following file system attributes are stamped on installable items (applications, services, fonts, and so on) and documents.

Stamped attributes—PenPoint 2.0 installable items TABLE 40-1

Admin/ index	Type	Label	Header file	Comment
62/0	USTR	fsAttrName	FS.H	The visible name of the installable. It must be unique within the parent directory. It has a minor programmatic use to handle collisions with items that are already installed. It is mandatory.
260/3	USTR	imAttrVersion	INSTLMGR.H	The visible version string. It is not used in source code and is optional.
157/12	USTR	appAttrClassName	APPDIR.H	The visible installable type name (such as Application, Font, Printer). It is not used in source code and is optional.
193/2	USTR	cimAttrProgramName	CODEMGR.H	The module name, for example, GO-ABAPP-V2(0). This name must match the module name in the .LBC file used to build the module. It is used in source code.
157/1	FIX	appAttrClass	APP.H	The installable type. This attribute must be set to the installation manager that controls this type of installable: **theInstalledApps**, which is 010001A0. It is mandatory.

Stamped attributes—PenPoint 2.0 documents TABLE 40-2

Admin/ index	Type	Label	Header file	Comment
62/0	USTR	fsAttrName	FS.H	The visible name of the document. It must be unique within the parent directory and is mandatory.
28/0	FIX64	fsAttrDirIndex	FS.H	The directory index. It must be unique and is mandatory.
157/12	USTR	appAttrClassName	APPDIR.H	The visible name of the document's application, for example, MiniText or GOMail. It is not used programmatically and is optional.
157/1	FIX	appAttrClass	APPDIR.H	The document's application class. It is mandatory.
157/4	FIX	appAttrSequence	APPDIR.H	The sequence number, which reflects the position of the document within its embeddor. It is mandatory.
157/3	FIX	appAttrNumChildren	APPDIR.H	The number of documents that are embedded within this document. It is mandatory.
157/6	FIX64	appAttrFlags	APPDIR.H	The document's flags, such as **moveable** and **readOnly**. It is mandatory.

Stamped attributes—PenPoint 2.0 documents TABLE 40-2 (continued)

Admin/ index	Type	Label	Header file	Comment
157/9	USTR	**appAttrBookmark**	APPDIR.H	The visible name of the document's tab in the Notebook. It is optional.
157/10	USTR	**appAttrAuthor**	APPDIR.H	The visible author field. It is optional.
157/10	USTR	**appAttrComments**	APPDIR.H	The visible comments field. It is optional.

Tips and clarifications

Open handles on files

When your application has finished with a file or directory, it must free the handle on the node, especially if the node is on a floppy or other removable media. If you don't do this and a user ejects the disk, they will continually get a number of prompts for the disk, which cannot be cancelled.

msgFSSetSize does not reposition file pointer

If you use **msgFSSetSize** to truncate a file, the file position will not be changed during the call. To write at the end of the file, you must seek to the end of file.

StdioStreamUnbind

If you use **StdioStreamBind**() and a read, a subsequent **StdioStreamUnbind**() will change the file pointer. Thus, if you plan to rebind and pick up where you left off, you must get the current file location with an **ftell**() before unbinding so you can reset the file pointer after rebinding.

Memory mapped file problem

It's not a good idea to implement a memory-mapped file in a document directory. Why? Let's say someone launches a document from an extended bookshelf on a floppy disk. You start writing to the memory map, and then the user ejects the floppy disk. This causes the memory manager to choke with a page fault.

Because all application directories live on **theSelectedVolume**, memory-mapped files in the global application directory are still okay. This seems to be the standard implementation for memory-mapped files anyway, such as a shared PIM database.

Corrections and errata

Locators

Section 70.6 in the *PenPoint Architectural Reference* describes implicit and explicit locators, but does not mention flat locators. Flat locators hold an entire locator string in a linear (flat) structure. In PenPoint 2.0 Japanese, flat locators are defined by structure FS_FLAT_LOCATOR and are used by browser objects (**clsBrowser**). See Chapter 80 for descriptions of how instances of **clsBrowser** use flat locators.

lseek() and msgFSSeek

The WATCOM C Library Reference for PenPoint states that when calling **lseek()**, the requested file position may be beyond the end of the file.

This is not true in PenPoint 2.0 Japanese. When the requested position is beyond the end of file and the file pointer is currently positioned at the end of file, both **lseek()** and **msgFSSeek** return errors. When the requested position is beyond the end of file and the file pointer is not positioned at the end, they move the file pointer to the end and do not return an error status.

The workaround for this problem is to use **chsize()** or **msgFSSetSize** to extend the file, then use **lseek()** or **msgFSSeek**.

Typographical errors

Part 7 (File System)—typos

TABLE 40-3

Volume, section, paragraph	*Old text on first Line) New text on second line*	
II, 69.2.2, ¶5	Code example: OF_GET *(Remove)*	get;
II, 72.1.1, Table 72-2	fsDenyWriters fsDenyWriters	Deny access to readers Deny access to writers.

Chapter 41 / System Services

What's new

String composition functions

PenPoint SDK 2.0 Japanese has added six new routines to its Compose Text package of functions that allow you to compose formatted text strings. The new functions allow the construction of counted strings (as opposed to null-terminated ones) and enable you to compose strings using resource files. These functions are:

This section replaces section 75.3.3 in the *PenPoint Architectural Reference.*

- SComposeTextN
- VSComposeTextN
- SComposeTextL
- VSComposeTextL
- SComposeTextNL
- VSComposeTextNL

"Function definitions" on page 473 describes these functions in detail.

These Compose Text functions are similar to standard C **stdio** functions such as **printf**() and **sprintf**(). They use positional format codes to copy a format argument into an output string, after performing the required substitutions for the format codes. Use these functions rather than **sprintf**() to create strings in your user interface.

The Compose Text functions feature format codes other than those for the usual data-type conversions. One format code and convention enables conditional insertion of singular or plural word forms, such as "is" or "are." Other format codes make it possible to specify text strings and string lists stored as resources.

Although they accomplish the same thing, the Compose Text functions come in several varieties. Some allow you to include the strings for composition as arguments, others require pointers to those strings, and other functions get the strings from resource files. In addition, some functions terminate the composed string with a null character and others do not. The API definitions for these functions are in CMPSTEXT.H. The functions themselves are in SYSUTIL.LIB.

The Compose Text functions are ideally suited for internationalization, particularly because they use resource files.

Format codes

The format string used in string composition contains one or more format codes. Format strings can also contain literal text, though they need not. A format code starts with a caret character (^), has one or more digits in the middle, and concludes with a single letter.

The string arguments (as literal text, pointers or resource file identifiers) follow the format string. The digits of the format code specify which argument to insert in that position and the letter indicates the type of the argument. For instance, format code **^2s** directs a function to insert the second argument as a string.

The following example fills **buffer** with the string "a B b A c":

```
SComposeText(&buffer, &size, heap, U_L"a ^2s b ^1s c", "A", "B");
```

Compose Text format code types

TABLE 41-1

Type Code	*Description*
s	String. The argument or arguments are pointers to text strings.
r	Resource ID of a string resource.
l	Group number and indexed list resource ID for string list. The group number and the list resource ID must be two separate arguments (in that order).
d	U32 argument printed as a decimal number.
x	U32 argument printed as a hexadecimal number.
^	Literal ^ character (^^) for putting ^ in a string. There is no number.
{...|...}	This delimiter format type permits you to conditionally insert singular or plural word forms into text strings based on the value of an argument. Insert the singular and plural forms of a word, in that order and separated by a | character, between the braces. When you use this format type, the Compose Text function examines the specified argument. If its value is 1, the function inserts the first string; otherwise, it inserts the second string.

As an example of the {...|...} format type, the following function call generates "There is 1 apple" if **numApples** is equal to 1 and "There are 5 apples" if **numApples** is equal to 5:

```
{
SComposeText(&buffer, &size, heap,
      U_L("There ^1{is|are} ^1d ^1{apple|apples}"), numApples);
 }
```

Function arguments and memory management

The first three arguments for all Compose Text functions are identical:

- A handle (type PP_CHAR) to the buffer that will contain the composed string.
- A pointer (type P_U32) to the size of the buffer (In) or to the length of the text string in the buffer (Out).
- An identifier of the heap used to allocate memory for the buffer (OS_HEAP_ID).

All Compose Text functions return the length of the generated string in the **length** argument. For those functions that compose null-terminated strings, the null is not counted in the length.

The Compose Text functions give you two ways to supply the buffer memory:

- You can supply a buffer handle and buffer length and set the heap ID to null. If this technique is used, and the buffer is too small to hold the results, an error status is returned.
- You can specify a valid heap ID to have the function allocate memory for the buffer from the specified heap. You must free the memory when finished with **OSHeapBlockFree()**. If you pass **null** for the buffer length when specifying a heap ID, you do not get the actual length of the string back.

Function definitions

SComposeText Composes a null-terminated text string from a format and arguments.

VSComposeText Composes a null-terminated text string from a format and a pointer to an argument list.

SComposeTextL Composes a null-terminated text string from a resource-file format and arguments. Unlike **SComposeText**, this function fetches the format string from a string array in a resource file.

VSComposeTextL Composes a null-terminated text string from a resourcified format and a pointer to an argument list. This function differs from **VSComposeText** in that it fetches the format string from a string array in a resource file.

SComposeTextN Composes a counted string from a format and arguments. Unlike **SComposeText**, the generated string is not terminated with a null character.

VSComposeTextN Composes a counted string from a format and a pointer to an argument list. Unlike **VSComposeText**, the generated string is not terminated with a null character.

SComposeTextNL Composes a counted string from a resource-file format and arguments. Unlike **SComposeTextL**, the generated string is not terminated with a null character.

VSComposeTextNL Composes a counted string from a resource-file format and a pointer to an argument list. Unlike **VSComposeTextL**, the generated string is not terminated with a null character.

Getting the current locale

To determine the locale the PenPoint 2.0 Japanese operating system is currently running in, send **msgSysGetLocale** to the system. The locale ID and a string describing the locale are returned. **msgSysGetLocale** is defined in SYSTEM.H.

Multibyte/Unicode conversion routines

The international conversion functions **IntlMBToUnicode()** and **IntlUnicode-ToMB()** replace the PenPoint 2.0 Alpha functions **Ustrcpy8to16()** and **Ustrcpy16to8()**, respectively. Note that this a replacement of functionality, not a renaming of functions. The interfaces for these new, replacement functions are in ISR.H. Chapter 47, International Services and Routines, describes these and similar international conversion functions.

Corrections and errata

Ugetc and Uungetc bugs

Ugetc() and **Uungetc()** are WATCOM's Unicode implementations of **getc()** and **ungetc()**. **Ugetc()** is supposed to return the next Unicode character from the input stream. **Ungetc()** is supposed to push that character back onto the stream. There is currently a bug in these routines. **Ugetc()** loses the upper half of the Unicode character after **bufsiz** (512) bytes have been read. **Uungetc()** faults when it tries to push this character back just after a new buffer is filled.

To work around this bug, include the following code immediately after the **#include** for STDIO.H:

```
#undef Ugetc
#define Ugetc(_fp) FixedUgetc((_fp))
static int FixedUgetc(FILE *file)
{
int tempChar, tempChar2;
tempChar = getc(file);
if (tempChar == EOF) return EOF;
tempChar &= 0x00FF;
tempChar2 = getc(file);
if (tempChar2 == EOF) return EOF;
return (tempChar2 << 8 | tempChar); }
#undef Uungetc
#define Uungetc(_ch, _fp) \
(CHAR) (ungetc((_ch) >> 8, (_fp)), ungetc((_ch)&0xFF, (_fp)))
```

Renaming of 16-bit utility functions

The following functions have been renamed:

Renamed counted string functions TABLE 41-2

Version 1.0	*Version 2.0*
Umemccpy	Uchrccpy
Umemchr	Uchrchr
Umemcmp	Ustrncmp
Umemcpy	Ustrncpy
Umemicmp	Ustrnicmp
Umemset	Uchrset

Note that **Ustrncmp**, **Ustrncpy** and **Ustrnicmp**, unlike their predecessors, do not copy null as any other character. They treat nulls as early termination, copying the null and then stopping.

These other 16-bit WATCOM functions have also been renamed:

Renamed WATCOM functions

TABLE 41-3

Version 1.0	*Version 2.0*	*Version 1.0*	*Version 2.0*
asctime16	_uasctime	assert16	_uassert
atof16	_uatof	atol16	_uatol
chdir16	_uchdir	creat16	_ucreat
ctime16	_uctime	fdopen16	_ufdopen
fgetc16	_ufgetc	fgetchar16	_ufgetchar
fgets16	_ufgets	fopen16	_ufopen
fprintf16	_ufprintf	fputc16	_ufputc
fputchar16	_ufputchar	fputs16	_ufputs
freopen16	_ufreopen	fscanf16	_ufscanf
getc16	_ugetc	getchar16	_ugetchar
getcwd16	_ugetcwd	getenv16	_ugetenv
gets16	_ugets	isalnum16	_uisalnum
isalpha16	_uisalpha	isascii16	_uisascii
iscntrl16	_uiscntrl	isdigit16	_uisdigit
isgraph16	_uisgraph	islower16	_uislower
isprint16	_uisprint	ispunct16	_uispunct
isspace16	_uisspace	isupper16	_uisupper
isxdigit16	_uisxdigit	itoa16	_uitoa
ltoa16	_ultoa	memccpy16	_uchrccpy
memchr16	_uchrchr	memcmp16	_ustrncmp
memcpy16	_ustrncpy	memicmp16	_ustrnicmp
memset16	_uchrset	open16	_uopen
printf16	_uprintf	putc16	_uputc
putchar16	_uputchar	puts16	_uputs
remove16	_uremove	rename16	_urename
rmdir16	_urmdir	scanf16	_uscanf
setenv16	_usetenv	sopen16	_usopen
sprintf16	_usprintf	sscanf16	_usscanf
strcat16	_ustrcat	strchr16	_ustrchr
strcmp16	_ustrcmp	strcmpi16	_ustrcmpi
strcpy16	_ustrcpy	strcspn16	_ustrcspn
strdup16	_ustrdup	strerror16	_ustrerror
stricmp16	_ustricmp	strlen16	_ustrlen
strlwr16	_ustrlwr	strncat16	_ustrncat
strncmp16	_ustrncmp	strncpy16	_ustrncpy
strnicmp16	_ustrnicmp	strnset16	_ustrnset
strpbrk16	_ustrpbrk	strrchr16	_ustrrchr

Renamed WATCOM functions TABLE 41-3 (continued)

Version 1.0	Version 2.0	Version 1.0	Version 2.0
strrev16	_ustrrev	strset16	_ustrset
strspn16	_ustrspn	strstr16	_ustrstr
strtod16	_ustrtod	strtok16	_ustrtok
strtol16	_ustrtol	strtoul16	_ustrtoul
strupr16	_ustrupr	swab16	_uswab
tmpnam16	_utmpnam	tolower16	_utolower
toupper16	_utoupper	ubprintf16	_uubprintf
ultoa16	_uultoa	ungetc16	_uungetc
utoa16	_uutoa	vfprintf16	_uvfprintf
vfscanf16	_uvfscanf	vprintf16	_uvprintf
vscanf16	_uvscanf	vsprintf16	_uvsprintf
vsscanf16	_uvsscanf	_Ubprintf	U_bprintf
_ubprintf	_u_bprintf	_Ufullpath	U_fullpath
_Umakepath	U_makepath	_umemccpy	_uchrccpy
_umemchr	_uchrchr	_umemcmp	_ustrncmp
_umemcpy	_ustrncpy	_umemicmp	_ustrnicmp
_umemset	_uchrset	_Usplitpath	U_splitpath
_Usplitpath2	U_splitpath2	_Uvbprintf	U_vbprintf
_uvbprintf	_u_vbprintf		

ecvt and fcvt

These functions are no longer available in PenPoint 2.0 Japanese.

HASH.H

The value in **hashTableMaxFillPct** has changed from 80 to 98. Now it also includes **HashFunctionString8**() and **HashCompareString8**() for 8-bit strings.

SYSTEM.H

Both **sysSysServiceFile** (SYSSERV.INI) and **sysResFile** (PENPOINT.RES) have been removed. These two files are combined in *locale*.RES (for example, JPN.RES). SYSTEM.H also includes **sysLocaleIndependentResFile** (ALL.RES), **msgSysGetLocale**, a SYS_LOCALE structure, an 8-bit **sysGoldenMaster_8** and a resource ID for warnings.

OSMemInfo, OSMemUseInfo, OSMemAvailable

OSMemInfo() is obsolete and has been replaced by **OSMemUseInfo**().

OSMemAvailable() returns the amount of swappable memory that can be allocated before the caution zone is reached. This is the point at which the system begins putting up notes warning that memory is getting low.

Typographical errors

Part 8 (System Services)—typos

TABLE 41-4

Volume, section, paragraph	*Old text on first line* *New text on second line*
II, 74.6.5, ¶4	A timer request can continue to count down after a PenPoint computer is powered on. A timer request can continue to count down after a PenPoint computer is powered off.

Chapter 42 / Utility Classes

What's New

Matching hiragana or katakana text

In search and replace operations on Japanese text, you can request **theSearchManager** to match text based on the type—hiragana or katakana—of the specified find string. If the types are different, the target text is passed over, even if the senses are identical.

To effect this search refinement, set the **matchHiraKata** flag TRUE. This flag is in the SR_FLAGS structure, which is itself part of the SR_METRICS structure. Then call **msgSRInvokeSearch** (SR_METRICS is part of the argument structure SR_INVOKE_SEARCH). **msgSRRememberMetrics** also uses a pointer to SR_METRICS as an argument.

Note the **matchHiraKata** flag is automatically set TRUE when the user selects the Same Hira/Kata Sense option under Match.

Adding gestures to Quick Help strings

RTF is no longer needed to embed gesture glyphs in Quick Help text strings in your resource files. Because the representation of strings is now Unicode-based, all you must do is type the Unicode code point for a gesture glyph (in hexadecimal) where you want the gesture to appear. (These glyphs are defined in GLYPH.H.) You no longer need to specify **\\f63** to enter a gesture font and **\\f0** to return from it.

You can still use RTF formatting commands in Quick Help text (such as **\\line**), and you can still use the **\qh** macro to set up an RTF header (although it no longer does any font mapping). However, you must remove all occurrences of {**\\f63** *c*} (where *c* is the gesture symbol) and **\\f0** from your Quick Help strings and replace the \\63 sequence with the correct Unicode values.

The following code fragment shows a typical use of a Unicode-specified gesture in Quick Help strings:

```
editMenuTag,
U-L("Edit Command||")
U-L("{\\qh Tap Edit to display an edit pad ")
U-L("with the selected text.\\line ")
U-L("\\line ")
U-L("You can also put the selected text in an edit ")
U-L("pad by drawing a Circle \xF621 gesture ")
U-L("on the selection.}",)
```

Note that you must type || to separate the title and the body of text, and that you must preface each string with the **L** (or **U_L**) macro to provide 16-bit compatibility.

This information on gestures in Quick Help strings replaces sections 84.3.3.1 and 84.3.3.2 in the *PenPoint Architectural Reference.*

clsNotePaper changes

API changes

clsNPData includes two new messages, **msgGetScribbleClass** and **msgGetTextClass**. When you subclass **clsNPTextItem** or **clsNPScribbleItem**, you should also subclass **clsNPData** and override the handlers for these two messages to have them return the appropriate class. These message handlers are required because **clsNotePaper** instantiates text and scribble objects on its own, and needs to be told when to use a subclass. For this reason, you should not subclass **clsNPItem** directly; you should only subclass **clsNPTextItem** and **clsNPScribbleItem.**

If you want to add different types of graphical objects, you can treat **clsScribbleItem** in an abstract manner. However, if your implementation is not complete, make sure that it satisfies all requirements of the **NPScribbleItem** API.

File format changes

In addition to API changes, **clsNotePaper** includes changes in file formats, particularly for file import and export. It now imports Unicode text files. As with MiniText, it requires a file suffix of .UNC for Unicode files. For Shift-JIS it expects files to have suffixes of .SJS. As before, it expects ASCII files to have .TXT suffixes.

Import of .TXT files depends on locale. In the Japan locale, **clsNotePaper** imports the file using the Shift-JIS interpretation of the ASCII character set. On the import of files, **clsNotePaper** also supports the same word or bunsetsu handling (depending on locale) that MiniText performs. It does not perform any taboo processing.

Export can be to Unicode or multibyte text files (Shift-JIS/ASCII). The multibyte format that is used depends on the locale setting of the pen computer (Shift-JIS in Japan, ASCII elsewhere).

Tips and clarifications

Cannot intercept export messages

Applications that need to modify their document's files on export cannot detect an export operation. If your application modifies the contents of its documents on export, it must provide its own menu button to perform special export operations.

msgImportQuery can arrive twice

Under certain race conditions, an application can get **msgImportQuery** twice. Message handlers should not assume that they will receive **msgImportQuery** only once.

New stream disconnected status

stsStreamDisconnected has been added to STREAM.H to report disconnected conditions in **clsMILAsyncSIO**.

Typically, SIO clients should not attempt stream calls unless they are connected. However, if the connected state of SIO changes to disconnected while in the middle of an SIO stream call, the stream call will return **stsStreamDisconnected** instead of **stsFailed**.

The difference is important. **stsFailed** return means that a client should or could retry. However a **stsStreamDisconnected** return means that a client must not retry. For one, there is no reason to retry since the call will continue to return immediately with the same **stsStreamDisconnected** status, possibly for ever, even if the cable is reconnected. (The connection functionality in PenPoint 2.0 Japanese involves messages so it requires that the message input queue be available.)

SIO clients will need to wait for the connected state to change before attempting any stream call to SIO that can be done in a couple of different ways (they may be observers of **theSerialDevices** service manager or may poll the service manager for connected state information; see SERVICE.H or SERVMGR.H).

clsTable bug

There is a known bug in **clsTable** that causes the called table object to return **stsOK** when it should return an error status. If you set the **tblRowPos** field of the **pArgs** structures to **Nil**(TBL_ROW_POS) when sending **msgTBLColGetData** or **msgTBLRowGetData**, **stsOK** is returned. **Nil**(TBL_ROW_POS) is undefined for these messages, and should cause the called object to return an error.

Clients must test the **tblRowPos** field value in their TBL_COL_GET_SET_DATA and TBL_GET_SET_ROW structures to ensure that it is not **Nil** before sending the messages.

msgTBLColGetData and **msgTBLRowGetData** are described in the *PenPoint Architectural Reference* in Table 90-1 (section 90.5, "Using Table Messages") and section 90.13, "Getting Data." **clsTable** interfaces are defined in TS.H.

Known bugs in the NotePaper component

Developers should be aware of the following bugs in MiniNote/NotePaper:

- Setting the paper width to 99999 confuses the horizontal scroll bar.
- Setting the line width to zero confuses MiniNote so that further width changes do not take effect.
- Copying a selection that contains an embedded document from MiniText into MiniNote results in the embedded document being replaced by a check mark.
- The circle tap ⊙ gesture is not targeted, while the circle ○ gesture is.

Corrections and errata

Getting the current selection

Modify the fourth sentence of section 80.2.2 (in the "Using clsBrowser" chapter in the *PenPoint Architectural Reference*) so that it reads: "These two messages take a pointer to a FS_FLAT_LOCATOR structure in which the called object returns the path or name of the current selection." The messages referred to are **msgBrowserSelection** and **msgBrowserSelectionDir.**

Classes that respond to search messages

Section 86.3 in the *PenPoint Architectural Reference* states that "**clsText** is the only class that responds to the search and replace messages." You should replace **clsText** with **clsTextView** in this sentence.

Reading and writing streams

Sections 79.3 and 79.4 in the *PenPoint Architectural Reference* are accurate in their descriptions of how to use **msgStreamRead** and **msgStreamReadTimeOut** to read communications streams. But some implications raised should be clarified. For a serial communications stream, there is no reliable notion of "end of stream"; conceptually, the byte stream is continuous, with no beginning or end. To read such a stream, you should send **msgStreamReadTimeOut** only. Moreover, if you want to make a nonblocking read of a serial stream, set the **timeOut** field (STREAM_READ_WRITE_TIMEOUT) to zero before sending **msgStreamReadTimeOut.**

Using the PenPoint gesture font

Table 84-2 on pages 188-190 of the *PenPoint Architectural Reference* lists the tags, symbols, and ASCII values for the PenPoint gesture font. It is no longer accurate. Gesture tags are now associated with glyph tags that represent Unicode values. GLYPH.H contains the current list of glyphs and their associated Unicode values; XGESTURE.H contains the current list of gesture tags and their associated glyphs.

Chapter 43 / Connectivity

What's new

clsModem has been re-implemented as a service. As a result, much of the material covered in Chapter 97 of the *PenPoint Architectural Reference*, "Data Modem Interface," is no longer valid. This section describes the major conceptual changes and summarizes the new procedure for using a modem service. Except where noted, it replaces Chapter 97. Refer to MODEM.H for complete API definitions.

Finding, binding to, and opening a modem

Since a modem is a service instance, you locate it, bind to it, and open it as you would any other service. In this case, you must send a series of **clsServiceMgr** messages to the predefined service manager for modems, **theModems**. (A modem is automatically associated with a serial port, so you no longer need to bind to and open a serial port explicitly, as Chapter 97 describes.)

The following list summarizes the **clsServiceMgr** messages you must send initially. These messages are described in greater detail in Chapter 94 in the *PenPoint Architectural Reference.*

1 Find the modem service by sending **msgIMFind**. You pass **theModems** service manager the name of the service; in the **pArgs** structure (IM_FIND), assign a pointer to the service name to the **pName** field. If the service is found, you get back a handle to that service. If it is not found, the return status is **sts-NoMatch.**

  If your application lets users choose a modem, send **msgIMGetList** to **theModems** to get a list of UIDs for modem services. Then send **msgIMGetName** to get the name of each service in the list and display these names in a list. When a user selects one, assign the name to the **pName** field of IM_FIND and send **msgIMFind.**

2 Bind to the service instance so that the service manager can add your application to the observer list for the service. Your application should bind to the modem service so that, when the status of the modem changes, **theModems** will notify your application and all other observers (see Table 43-6, "Client and observer notification messages," on page 491). You bind to the modem service instance by sending **msgSMBind** to **theModems.** In the message argument structure (SM_BIND), set the **handle** field to the value returned by **msgIMFind** and set the **caller** field to **self.**

3 Open the service instance by sending **msgSMOpen** to the modem service manager (**theModems**). This message takes a pointer to an SM_OPEN_CLOSE structure that contains the handle returned by **msgIMFind**, the **caller** (**self**) and a pointer to an argument structure (**pArgs**) containing data specific to the modem service.

4 If **msgSMOpen** returns **stsOK**, it also returns the UID of the opened modem service in the service field of SM_OPEN_CLOSE. Assign this UID to a variable of type OBJECT and specify this object in subsequent messages to the modem until you close the modem.

The following code fragment demonstrates the locating, binding and opening procedure:

```
IM_FIND          imf;
SM_BIND          smb;
SM_OPEN          smo;
OBJECT           myModem;
STATUS           s;
imf.pName = U_L("Hayes2400");
ObjCallRet(msgIMFind, theModems, &imf, s);  // find/get the modem handle
smb.handle = imf.handle;
smb.caller = self;
ObjCallRet(msgSMBind, theModems, &smb, s); // bind to modem service
smo.handle = imf.handle;
smo.caller = self;
ObjCallRet(msgSMOpen, theModems, &smo, s); // open modem
myModem = smo.service;
```

Instead of steps 1 to 3 above, you can send **msgSMAccess** to **theModems** and get back the UID of the modem service in the **service** field of a SM_ACCESS structure. When your are finished with the modem service, send **msgSMRelease** to unbind and close it. See SERVMGR.H for more information about these messages.

Initialization

The object that opens a modem service (for example, your application) becomes its client. Before it begins sending and receiving data through the modem, the client should initialize the modem firmware and the serial I/O port.

Applying the default settings

You reset the modem firmware and the I/O port state to the default settings by sending the modem-service instance **msgModemReset**. This message takes no arguments. After sending this message, you can change the reset defaults selectively. A typical usage, following our previous example, would be:

```
ObjCallRet(msgModemReset, myModem, Nil(P_ARGS), s);
```

The default modem firmware settings are:

- Auto-answer disabled.
- Busy tone detection enabled, or as current modem option card settings.
- Command termination = carriage return (ASCII 13).
- Dialing mode from dialing environment.

- Dial tone detection enabled, or as current modem option card settings.
- Enable carrier upon connect.
- Escape code = ASCII 43.
- Local character echo disabled.
- Send command result codes (words).
- Send verbal result codes.
- Speaker control on until carrier detected, or as current modem option card settings.
- Speaker volume medium, or as current modem option card settings.

Default I/O port state settings

TABLE 43-1

Setting	*SIO_METRICS Field*	*Default value*
baud rate	baud	Highest supported data mode baud rate or, if not available, 2400
data bits	line.dataBits	8 bits (**sioEightBits**)
stop bits	line.stopBits	1 bit (**sioOneStopBits**)
parity	line.parity	no parity (**sioNoParity**)
RTS	controlOut.rts	true
DTR	controlOut.dtr	true
XON char	flowChar.xonChar	0x11
XOFF char	flowChar.xoffChar	0x13
flow control	flowType.flowControl	off (**sioNoFlowControl**)

Setting I/O port state options

You can change the default I/O port state settings by sending **msgSioSetMetrics** to a modem service. To discover what these settings are, prior to altering them, send **msgSioGetMetrics** to the modem-service object. Other **clsMILAsyncSIODevice** messages that **clsModem** handles are **msSioInit**, **msgSioBreakSend**, **msgSioControlInStatus**, **msgSioInputBufferStatus**, and **msgSioInputBufferFlush**. Refer to SIO.H for descriptions of these messages.

The following code fragment demonstrates a typical use of **msgSioSetMetrics**:

```
SIO_METRICS smetrics;
/* Initialize serial port to preferences */
ObjCallWarn(msgSioGetMetrics, myModem, &smetrics);
smetrics.baud = (U32)9600;
smetrics.line.dataBits = sioSevenBits;
smetrics.line.stopBits = sioTwoStopBits;
smetrics.flowType.flowControl = sioNoFlowControl;
ObjCallWarn(msgSioSetMetrics, myModem, &smetrics);
```

Initializing the modem

After the serial I/O port has been initialized, you can send **clsModem** messages to set the desired features, control flags, and attributes of the modem. You can either make these settings as a group by sending **msgSvcSetMetrics**. Or you can initialize the modem by sending discrete messages (listed below).

If you elect to use **msgSvcSetMetrics**, you might first want to send **msgSvcGetMetrics** to the modem-service object to obtain the current settings. Both messages take a pointer to the argument structure SVC_GET_SET_METRICS, whose **pMetrics** field points to a buffer containing MODEM_METRICS. This metrics structure consists of a collection of enumerated types that list mutually exclusive settings for various modem features. These enumerated data types are also used as argument structures in the messages that set individual options in the modem service.

Table 43-2 describes the enumeration fields that make up MODEM_METRICS and that are used by the messages that set discrete options. Table Table 43-3 lists these discrete initialization messages; note that most of these messages take as **pArgs** the enumerated value itself (which is 32 bits) and not a pointer to that value.

MODEM_METRICS fields TABLE 43-2

Type	*Field*	*Possible Settings (default emphasized)*
MODEM_DIAL_MODE	**mdmDialMode**	Dialing mode: pulse, touch-tone, client supplies mode embedded in dial string, *use current dialing environment mode or current modem firmware dialing mode.*
MODEM_DUPLEX_MODE	**mdmDuplexMode**	Half duplex, *full duplex.*
MODEM_SPEAKER_CONTROL	**mdmSpeakerControl**	Modem speaker: off, on, *off until carrier detection.*
MODEM_SPEAKER_VOLUME	**mdmSpeakerVolume**	Speaker volume: whisper, low, *medium,* high.
MODEM_TONE_DETECTION	**mdmToneDetection**	Busy tone and dial tone: detect neither, *detect both,* detect busy tone only, detect dial tone only.
MODEM_ANSWER_MODE	**mdmAnswerMode**	Type of calls to answer and report connection about: *data mode,* fax mode, voice mode.
MODEM_AUTO_ANSWER	**mdmAutoAnswer**	Enable/*disable* auto-answer.
U32	**mdmAutoAnswerRings**	Number of rings before modem answers
MODEM_MNP_MODE	**mdmMNPMode**	Set MNP mode: *disable,* both modems must support MNP levels 1-4, attempt to establish MNP connection, LAPM connection.
MODEM_MNP_COMPRESSION	**mdmMNPCompression**	Enable/*disable* MNP (class 5) compression.
MODEM_MNP_BREAK_TYPE	**mdmMNPBreakType**	How to handle breaks: don't send break to remote, empty data buffers before sending break, *send break when it's received,* send break relative to data to be sent.
MODEM_MNP_FLOW_CONTROL	**mdmMNPFlowControl**	Flow control for MNP mode: *none,* XON/XOFF, RTS/CTR.

Discrete modem initialization messages TABLE 43-3

Message	*Takes*	*Description*
msgModemSetAnswerMode	MODEM_ANSWER_MODE	Filters the type of incoming call to answer and to report connection on. (Some modems do not have this capability.)
msgModemSetAutoAnswer	P_MODEM_SET_AUTO_ANSWER	Disables or enables auto-answer mode. The argument passed in is a pointer to a structure containing MODEM_AUTO_ANSWER and an S32 field for the number of rings before answering.
msgModemSetDialType	MODEM_DIAL_MODE	Sets the mode for dialing
msgModemSetDuplex	MODEM_DUPLEX_MODE	Sets the duplex mode (half or full) for inter-modem communication.
msgModemSetMNPBreakType	MODEM_MNP_BREAK_TYPE	Specifies how the modem handles a break character when in MNP mode.
msgModemSetMNPCompression	MODEM_MNP_COMPRESSION	Sets MNP class 5 compression off and on.
msgModemSetMNPFlowControl	MODEM_MNP_FLOW_CONTROL	Sets the type of flow control to use when in MNP mode.
msgModemSetMNPMode	MODEM_MNP_MODE	Sets the MNP mode of operation.
msgModemSetSpeakerControl	MODEM_SPEAKER_CONTROL	Controls the behavior of the modem speaker.
msgModemSetSpeakerVolume	MODEM_SPEAKER_VOLUME	Sets the volume of the modem speaker.

Response mode

In your code's modem-initialization section or at any time while a modem service is open, may also want to set the response mode of the modem service. The response mode, set with **msgModemSetResponseBehavior**, affects how the modem-service object responds to its client:

Respond via status In this mode, the client sends a message to the modem service, which then sends a command to the modem and then blocks, waiting for the response from the modem. If a timeout period (specified in the **pArgs** structure) elapses, **stsTimeout** is returned. Respond via status is the default response mode; the default timeout periods are 2.5 seconds for commands and 30 seconds for the connection.

Respond via message notification In this mode, the modem service acts as it does in Respond via status mode. But in addition, it sends **msgModemResponse** to the client. This mode is useful when you want to return to handle other work (such as handling certain abort commands) without waiting for a return. The client can post a request (via **Object-PostAsync**) to the modem, thereby freeing up the execution thread so it can process input events. See Table 43-6, "Client and observer notification messages," on page 491 for more on **msgModemResponse**.

Transparent This mode essentially disables the modem service's response processing. Responses to modem commands remain unaltered in the input data stream. It is the responsibility of the client to read and interpret these responses. They must ensure that the expected sequences of commands are sent (via **msgModemSendCommand** or through the discrete command messages).

The advantage of transparent mode is that there is less overhead in processing characters received from a remote modem. Aside from better performance, your application is less likely to have character overruns at high baud rates. If decide to operate in transparent mode, you might want to switch to it after initializing the modem and establishing a connection in one of the other modes.

A related message, **msgModemGetResponseBehavior**, passes back to the client the current modem response mode and timeout values. Refer to MODEM.H for details on **msgModemSetResponseBehavior** and **msgModemSetResponseBehavior**.

Establishing a connection (outbound)

If you are initiating an exchange of data, you use **clsModem** messages to dial and connect with the remote modem.

To dial another modem, send **msgModemDial** to the modem service. This message takes a pointer to a MODEM_DIAL structure that contains the field **dialString**, which is of type DIALENV_DIAL_STRING. Assign to this field the phone number (in the form of a text string) of the remote modem.

The phone number usually contains the number to dial. It can also contain a number of dial string modifiers defined by the AT command set (although this is not required). These dial string modifiers are described in the section, "Dial string modifiers", in Chapter 97 of the *PenPoint Architectural Reference.*

Although this phone-number string would normally be something that a user enters or selects from an address book, the following code fragment shows how dialing would occur with a hard-coded string.

```
MODEM_DIAL  dial;
STATUS      s;
      .
      .
dial.dialString = U_L"(415-345-7400");
ObjPostAsync(msgModemDial, myModem, &dial, s);
```

If **msgModemDial** returns **stsOK**, a connection is established with the remote modem. You can begin sending and receiving data. You can also send **msgGetConnectionInfo** to find out the details of the connection. When you are finished, send the modem service **msgModemHangUp** (it takes no arguments). This message terminates the connection and hangs up the phone.

Waiting for a connection (inbound)

When a remote modem attempts to make a connection with your local modem, you can instruct the modem-service object to automatically answer the phone or you can answer the phone yourself with **clsModem** messages.

To instruct the modem to answer the phone automatically, set the MODEM_AUTO_ANSWER enumerated type to **mdmAutoAnswerEnabled**, specify the desired number of rings to wait, and send **msgSvcSetMetrics** or **msgModem-SetAutoAnswer** (see "Initializing the modem," on page 486). When another modem dials your modem's number and the phone rings, the modem service takes the phone off-hook and sends the client the notification message **msgModemConnected**. The client can then send **msgModemGetConnectionInfo** to get more information about the connection.

To answer the phone yourself, set MODEM_AUTO_ANSWER to **mdmAutoAnswerDisabled** and send **msgSvcSetMetrics** or **msgModemSet-AutoAnswer**. When the phone rings, the modem service notifies the client via **msgModemRingDetected**. The client answers the phone by sending **msgModem-Answer** to the modem service (no arguments required). Once connection is established, the client can send **msgModemGetConnectionInfo** to get more information about the connection.

Transmitting and receiving data

Once you have established a connection with a remote modem, you can begin reading or writing data. You effect these functions through **clsStream** messages. Send **msgStreamWriteTimeout** to transmit data to the modem service (and ultimately to the remote modem); send **msgStreamReadTimeout** to read the stream of data coming into the modem service. See Chapter 79 in the *PenPoint Architectural Reference* or STREAM.H for more information on these messages.

If the connection between modems is lost, the client receives **msgModem-Disconnected.**

Terminating the modem service

To end a modem service, first make sure there is no connection established. Then send **msgSMClose** to **theModems** to close the service. This message takes a pointer to a SM_BIND structure, which contains:

handle Set to the service handle (obtained early in the procedure via **msgIMFind**).

caller Set to the UID of the modem-service client (usually **self**).

When the message completes successfully, it returns **stsOK**. Finally, remove your client from the service's observer list by sending **smgSMUnbind** to the service manager for modems (**theModems**). If you had opened and bound a modem service through sending **msgSMAccess**, close and unbind that service by sending **msgSMRelease.**

clsModem messages

See "Initializing the modem," on page 486 for a description of **msgSvcSetMetrics**. Refer to Table 43-3 for a list of discrete modem initialization messages.

Modem service creation and initialization messages TABLE 43-4

Message	*Takes*	*Description*
msgNewDefaults	P_MODEM_NEW	Initializes the MODEM_NEW structure to default values.
msgNew	P_MODEM_NEW	Creates a new instance of a modem service.

Modem service request messages TABLE 43-5

Message	*Takes*	*Description*
msgModemAnswer	nothing	Immediately answers a telephone call.
msgModemDial	P_MODEM_DIAL	Dials a remote modem and attempts to establish a connection.
msgModemGetConnectionInfo	P_MODEM_CONNECTION_INFO	Passes back information about the current connection. This information consists of baud rate, connection type (standard, LAPM and MNP) and MNP class (if applicable).
msgModemGetResponseBehavior	P_MODEM_REPONSE_BEHAVIOR	Passes back the current modem response mode and the current command-to-response timeout values.
msgModemHangUp	nothing	Hangs up and disconnects to terminate a connection.
msgModemOffHook	nothing	Picks up the phone line.
msgModemOnline	nothing	Forces the modem online into data mode.
msgModemReset	nothing	Resets the modem firmware, I/O port state and service state to default values.
msgModemSendCommand	P_MODEM_SEND_COMMAND	Sends a command to the modem. The command strings are from the AT command set (see section 97.4 in the *PenPoint Architectural Reference*). In the argument structure you can also send a timeout value that supersedes any timeout specified via **msgModemSetResponse-Behavior**. The response to the command is returned via the argument structure. Clients should use this message only to obtain modem behavior unavailable through other messages in the **clsModem** API. They are responsible for ensuring that commands altering modem registers do not adversely affect **clsModem**.
msgModemSetCommandState	nothing	Sets the modem into command mode.

Modem service request messages

TABLE 43-5 (continued)

Message	*Takes*	*Description*
msgModemSetResponseBehavior	P_MODEM_RESPONSE_BEHAVIOR	Set the modem's response mode and the command-to-response timeout values. (See "Response mode," on page 487 for a description of available response modes.)
msgModemSetSignallingModes	P_MODEM_SIGNALLING_MODES	Restricts the operation of the modem within specified voiceband and wideband signalling modes or standards.

The modem service sends two kinds of notification messages, one to its client and one to its observers. There are several client notification messages, but only one observer notification message, **msgModemActivity**.

In order for the client to receive client notification messages, the response mode must be set to Respond via status (**msgModemSetResponseBehavior**). Observer notification messages are sent to all objects on the modem service's observer list; they can be objects other than the client of the modem service.

Client and observer notification messages

TABLE 43-6

Message	*Takes*	*Comments*
msgModemActivity	MODEM_ACTIVITY	Observer notification. Informs observers of a change in modem activity. Passes a pointer to **MODEM_ACTIVITY**, which enumerates possible modem states.
msgModemResponse	P_MODEM_RESPONSE_INFO	Client notification. Provides the response to a previous command or request sent to the modem object. The response behavior must be set to **mdmResponseViaMessage** via **msgModemSetResponseBehavior**. A pointer is passed in to **MODEM_ REPONSE_INFO**, which enumerates possible responses.
msgModemConnected	nothing	Notifies client that the modem has connected with a remote modem.
msgModemDisconnected	nothing	Notifies client that the current connection has been terminated.
msgModemRingDetected	nothing	Notifies client that a ring indication has been received from the modem.
msgModemTransmissionError	nothing	Notifies client that an error was detected during the transmission (sending or receiving) of data. The modem service typically sends this message as a result of a data-framing error or some other error generated by low-level modem link protocol.
msgModemErrorDetected	nothing	Notifies client that an unexpected error indication was received from the modem.

Corrections and errata

Reading and writing with the serial port

Section 95.2.4 of the *PenPoint Architectural Reference* erroneously calls the pointer-to-buffer field of STREAM_READ_WRITE **pReadBuffer**. It should be called **pBuf**, and its definition should read:

> A pointer to a buffer that receives data read from the stream or that contains data to be written to the stream. On **msgStreamRead**, the buffer must be big enough to hold at least **numBytes** of data.

Note that this correction applies also to the definition of the STREAM_READ_WRITE fields in section 79.3 in the *PenPoint Architectural Reference.*

Predefined service managers

GO defines a number of service managers in UID.H. Table 43-7 defines the service managers only listed in section 94.2.1 in the *PenPoint Architectural Reference.*

Predefined service managers TABLE 43-7

Service Manager	*Function*
theMILDevices	Maintains and manages the list of current MIL services (device drivers).
theParallelDevices	Maintains and manages the list of current parallel port devices.
theAppleTalkDevices	Maintains and manages the list of current AppleTalk port devices.
theSerialDevices	Maintains and manages the list of current serial port devices.
thePrinterDevices	Maintains and manages the list of all devices that support printers.
thePrinters	Maintains and manages the list of all current printers.
theSendableServices	Maintains and manages the list of all services that have interfaces with the Send Manager and whose names appear in the Send menu (for example, fax and E-mail).
theTransportHandlers	Maintains and manages the list of current transport-level network protocol handlers.
theLinkHandlers	Maintains and manages the list of current data-communication services for physical network devices (such as LocalTalk).
theHWXEngines	Maintains and manages the current list of installable handwriting-translation engines.
theModems	Maintains and manages the list of instances that handle communication over a type of modem.
theHighSpeedPacketHandlers	Maintains and manages the services that perform high-speed packet transfer over parallel and serial ports.
theDatabases	Maintains and manages services that implement PIA databases.

Typographical errors

Part 10 (Connectivity)—typos

TABLE 43-8

Volume, section, paragraph	*Old text on first Line* *New text on second line*
II, 94.1,¶ 3	The services architecture can be though of as being.. The services architecture can be thought of as being...
II, 99.3.4, ¶1	You must add any servicespecific behaviors... You must add any service-specific behaviors...

Chapter 44 / Resources

What's new

Resource file utility routines

New functions defined in RESUTIL.H help your application read strings in from its resource files (**theProcessResList**). Table 44-1 defines all the resource file utility functions and marks those that are new.

Resource file utility routines TABLE 44-1

Function	New?	Comments
ResUtilLoadObject		Loads an object from **theProcessResList**.
ResUtilLoadString		Loads a string from **theProcessResList**. You can allocate memory by specifying a buffer and a length *or* by specifying a heap to allocate from.
ResUtilGetString	Yes	Gets a string item from **theProcessResList**.
ResUtilAllocString	Yes	Reads a string item from **theProcessResList** and puts it in allocated memory.
ResUtilLoadListString		Loads a string from a string array in the application resource list (**theProcessResList**). It uses the group and indexed resource ID to construct the resource ID of a string list and the index into it. You can allocate memory by specifying a buffer and a length *or* by specifying a heap to allocate from.
ResUtilGetListString	Yes	Gets an item from a string list in the application's resource list. It uses the group and indexed resource ID to construct the resource ID of a string list and the index into it. You can allocate memory by specifying a buffer and a length.
ResUtilAllocListString	Yes	Gets an item from a string list in the application's resource list. It uses the group and indexed resource ID to construct the resource ID of a string list and the index into it. You can allocate memory by specifying a heap.

All of these functions are shortcuts to using **msgResReadData**. They are implemented in RESFILE.LIB.

New system preferences

The system preference file (PREF.H) contains several new preferences and some new functions for accessing and manipulating preferences. These items include:

- New preferences for fully enclosed (Japanese-style) character box height and width.
- New preference for import/export data exchange format (1983 JIS vs. 1978 JIS).

- The function **PrefsIntlDateToString()**, which returns a string containing the Unicode representation of the formatted date based on the current user preference. Use this function instead of **PrefsDateToString()**.
- The function **PrefsIntlTimeToString()**, which returns a string containing the Unicode representation of the formatted time based on the current user preference. Use this function instead of **PrefsTimeToString()**.
- New tag **tagBSAppAutoZoomDocument**, which identifies the document to be automatically zoomed when PenPoint boots. This replaces the **AutoZoom** string in ENVIRON.INI.

New resource group

A new resource group in PenPoint SDK 2.0, called **resGrpMisc**, allows developers to read in strings that fall into a miscellaneous category (that is, resources that are not Toolkit or Quick Help strings).

New and renamed string resource agents

In PenPoint SDK 2.0, the names **resStringResAgent** and **resStringArrayResAgent** now refer to 16-bit string resources. The two resource agents for strings that in PenPoint 1.0 had these names have been renamed: **resStringResAgent** is now **resString8ResAgent**, while **resStringArrayResAgent** is now **resString8ArrayResAgent**.

Tips and clarifications

msgResWriteData does not copy pData

You send **msgResWriteData** to file some data in an object file, passing a pointer to the data to be written. If this is inside a **msgSave** handler, **clsResFile** has not been written when **msgResWriteData** returns; **clsResFile** has just queued it for writing in the future.

If (P_RES_WRITE_DATA)**pArgs->pData** points to data on the stack, it will probably be corrupt when **clsResFile** unwinds and writes the data. If **pData** points to allocated memory, there's no easy way to know when it's safe to free it (you could post **self** a message to free it).

clsResFile's queued-write behavior is not a complicated multi-threading subtask. Object filing starts when someone tells **clsResFile** to write an object; **clsResFile** tells that object to save. If one object sends a message to write another object or data while it's saving, **clsResFile** queues the write. When **msgSave** returns, control is returned to **clsResFile**, which then processes queued writes. It all takes place by **ObjectCall** within one task. Just remember that **clsResFile** is in the driver's seat.

Saving bitmap editor resources

If while using the bitmap editor you save a bitmap to a resource file, make sure the file has a name different from any existing resource files. If you save a bitmap to an existing resource file, the bitmap will be appended to the existing file.

Corrections and errata

Typographical errors

Part 11 (Resources)—typos

TABLE 44-2

Volume, section, paragraph	*Old text on first line* *New text on second line*
II,102.2, ¶6	Code example (twice): ObjectCall(...,clsFileHandle,...) *Code example (twice)*: ObjectCall(...,clsResFile,...

Chapter 45 / Installation API

What's new

KKC engine installation

Several new API enhancements make possible the installation and deinstallation of kana-kanji conversion (KKC) engines. In the user interface, these changes show up in the KKC engines page of the Installed Software section of the Settings notebook. This page lists the installed KKC engines and lets users perform the normal operations for setting the current engine, deleting an engine, and so on. When users tap the Install menu item, or when they select the KKC engines view within the Connections notebook, PenPoint 2.0 Japanese creates a disk viewer that shows all installable KKC engines on the selected volume.

Since KKC Engines are PenPoint services, the Disk Viewer shows all installable services on the volume, including KKC Engines. If you install an ordinary service from the KKC page (or vice versa), it finds its way to the proper Installed Software page.

Install Manager class

PenPoint SDK 2.0 has a new class of installation managers, **clsKKCInstallMgr**, that handles the installation and deinstallation of KKC Engines. This class is a subclass of **clsServiceInstallMgr**. The new public header file KKCIMGR.H documents the API for implementing **clsKKCInstallMgr**.

At boot time, the INSTALL.DLL creates **clsKKCInstallMgr** and a single well-known instance of it, called **theInstalledKKCEngines**. The **clsKKCInstallMgr** makes the first KKC Engine installed the current engine.

Before installation, KKC engines are in the same directory as services (\2_0\PENPOINT\SERVICE). Once they are installed, however, KKC engines live in their own subdirectory, \PENPOINT\KKC. In order for a service to be recognized as a KKC engine, its directory must be stamped differently: **appAttrClass** must have the value **theInstalledKKCEngines** (01000416) instead of **theInstalledServices** (01000240).

KKCCT class

The KKC Character Translator class participates as a client in the protocol defined by **clsKKCInstallMgr**. This protocol enables clients to open and close engines and provides dynamic notification of engine changes. User interface elements dynamically track the users' preferences for the current engine anywhere that KKC happens in the system. See KKCIMGR.H for definitions of the protocol.

KKC class

clsKKC provides certain default behavior for deinstallation. Specifically, it responds to **msgSvcTerminate**, **msgSvcTerminateOK**, and **msgSvcClassGetMetrics**.

Installation routing via appAttrClass

To facilitate the boot-time installation of services and applications, the processing of .INI files has changed. Instead of the INI filename determining the destination installation manager, each installable in an INI file is sent to the installation manager designated by the **appAttrClass** attribute on the installable.

If the attribute is not a valid installation class, the item is sent to the installation manager designated by the name of the INI file being processed. Although this change implies that APP.INI and SERVICE.INI could be collapsed into a single file, they remain separate.

Tips and clarifications

Other installation information

There are messages in SYSTEM.H to locate the active area, and messages in AUXNBMGR.H to locate particular auxiliary notebooks on the Bookshelf.

PenPoint 2.0 by default doesn't consider **theSelectedVolume** something that it displays in a browser, hence the user can't see **theSelectedVolume** in Connections or any other list of attached volumes. This is why in **DebugTablet** mode, you can't install software from the hard drive. Setting the B800 debugging flag overrides this.

Corrections and errata

Installation clarifications for production PenPoint

Chapter 110, "Organization of Distribution Volumes," in the *PenPoint Architectural Reference* doesn't mention the \PENPOINT\SYS\LOADER database of code. This is a key concept on a pen computer. There's no \PENPOINT\APP; instead application code is copied to \PENPOINT\SYS\LOADER and renamed after the EXE and DLL lname strings. On a cold boot, PenPoint uses attributes stamped on files in the loader database to determine the order in which to load these .DLL and .EXE files—there's no APP.INI on a pen computer.

The hierarchy descriptions in Chapter 110 are for the PenPoint configuration for the SDK, not the pre-set configuration on pen computer installation disks or the actual configuration of a pen computer running PenPoint.

On a running PenPoint machine, there typically is nothing *but* \PENPOINT\SYS and \PENPOINT\BOOT\ENVIRON.INI. There's no other .INI files in the BOOT directory and there is no \PENPOINT\APP.

Erroneous Directory

Pages 383 and 388 of the *PenPoint Architectural Reference,* have diagrams showing a \PENPOINT\SYS\DOC directory. The directory is \PENPOINT\SYS\Bookshelf.

Dynamic Link Libraries

Chapter 111 of the *PenPoint Architectural Reference* on Dynamic Link Libraries (DLLs) requires several corrections and clarifications.

Minor version numbers

Section 111.3 in the *PenPoint Architectural Reference* claims that the minor version number (the one in parentheses in the dll-id string) is optional and "is ignored by the operating system when it determines whether a DLL is already loaded in the PenPoint computer." Section 111.5 in the *PenPoint Architectural Reference* says that "the application monitor does not compare minor version numbers." Both statements are wrong. The application monitor does take the minor version number into account if the rest of both **dll-ids** are identical. If a DLL is being installed and a DLL with the same company name, module name and major version number is already installed, the application monitor installs the new DLL only if it has the higher minor version number.

Deinstallation

Section 111.5 in the *PenPoint Architectural Reference* states that "when an application is deinstalled, the application monitor again opens the corresponding .DLC file and compares its **dll-ids** against the currently loaded **dll-ids**." Not true. Once an application is installed on a running PenPoint system, there might not be any "corresponding .DLC file" to open (for example, the file is on an installation disk). So, when it deinstalls an application, the application monitor gets the application's **dll-ids** from the attributes stamped in the \PENPOINT\SYS\LOADER database for that application.

What happens next is as described in 111.5. The application monitor matches the to-be-deinstalled **dll-id** against the application's **dll-ids** and, if there is a match, decrements the **dll-id** reference counter. If the reference counter becomes zero, the application monitor deinstalls the DLL.

Naming conventions

Section 111.4in the *PenPoint Architectural Reference* is mistaken about the naming conventions for an application directory, an application's .EXE file and its .DLC file. It is not true that "the .DLC file must have the same name as the application directory and the executable file." A simple application might have only an executable file (that is, no DLLs); in this case, the name of the executable file should be the same as that for the application directory. Otherwise, the .DLC file should have the same name as the application directory; the name of the executable file can be any valid DOS filename, as long as it's referenced in the .DLC file.

In light of these changes, the examples cited in section 111.4 need to be updated. An application with the PenPoint name Graph it Right is stored in the directory \PENPOINT\APP\Graph It Right. If the application has no DLL files, the executable file is named Graph It Right.EXE. If the application has DLLs, there is a Graph It Right.DLC file in the application directory; this file lists the **dll-ids** and DOS pathnames of the application's DLL files and executable file in order of dependency.

Let's say (as in the section 111.4 example) that the executable file in the Graph It Right directory is named GRAPHER.EXE. The application's one DLL file is named FORMS.DLL. Therefore, the application's .DLC file is name Graph It Right.DLC (not grapher, as in the example) and it contains two lines:

```
GO-forms_dll-V1(2)     FORMS.DLL
GO-Grapher_exe-V1      GRAPHER.EXE
```

Minimum operating system version

In its last sentence, section 111.8 in the *PenPoint Architectural Reference* says that "an application can specify the minimum operating system version it will run under at installation time." Although an application can no longer specify the minimum operating system version, it can still accomplish the same end.

PenPoint SDK 1.0 has in the **pArgs** structure for **msgSave** (OBJ_SAVE) the fields **minAppVer** and **minSysVer**. The **minAppVer**, once used to specify the minimum version of PenPoint for an application, is now obsolete. (**minSysVer** is used for a different purpose: it helps to prevent the restoration of a system-synchronized object from a resource file to an older version of PenPoint that may not understand the filed format.)

An application, at initialization time, can send the message **msgGetSysVersion** to obtain the current version number of PenPoint. If that version is incompatible with the application, the application can then do something in response, such as displaying a warning message and exiting.

Chapter 46 / Writing PenPoint Services

What's new

PenPoint 2.0 Japanese contains no new features, API, or functions for inclusion in this chapter.

Tips and clarifications

MIL services and other services

Some readers of the *PenPoint Architectural Reference* thought that section 93.4.2 did not clearly explain the difference between MIL services and other services. One one level, the difference is in function: MIL services generally implement objects that act as device drivers for a type of device. Other services operate at a more abstract level: they are removed from the hardware but have interfaces with MIL services.

But there is a more essential difference. The principle difference between MIL services and other services lies in the use of protected memory. A MIL service comprises two DLLs. One runs as a protected task in Ring 0 (protected) memory. Written in procedural code, this DLL controls and responds to the device itself. Another DLL in Ring 3 memory provides the API for applications and other services. This DLL, written in object-oriented code, mediates between these client objects (applications and regular services) and the Ring 0 code.

Non-MIL services can implement many things. The following is a partial list:

- Connectivity services (for printing, faxing, E-mail, and so on).
- Network protocol stacks (using chain of targeted services).
- Installable file systems.
- Database engines.
- Handwriting engines.

Writing a MIL service is not a trivial matter. If you intend to write a MIL service, refer to the HDK documentation for information on how to proceed.

theServiceManagers

There is a problem with the well-known list **theServiceManagers** in PenPoint. Its entries are not the UIDs of the currently existing service managers. Rather, the entries are pointers into stack frames that have disappeared. When a service manager is created, it does add an entry to **theServiceManagers**, but what it adds is a bogus pointer into the stack rather than its own UID.

This situation leads to a couple of effects:

- If your service A targets service B and tries to bind to it before service B's service manager exists, the delayed bind does *not* automatically happen (as documented and expected) when B's service manager gets created. That's because A is waiting for B's service manager UID to get added to **theServiceManager**, and that never happens. If B's service manager does exist, there's no problem.
- If (as documented in SERVMGR.H) you observe **theServiceManagers** you will get notified (via **msgListNotifyAddition**) when a service manager is created. However, there's no way of knowing which service manager it is.

If you're waiting for a particular service manager to be created, first observe **theServiceManagers**. When you receive **msgListNotifyAddition**, check **pArgs->list** to make sure **theServiceManagers** is sending the notification. Then send **msgObjectValid** to **clsObject**, passing in the UID of the service manager that you're waiting for; the response from **clsObject** tells you whether this service manager now exists.

Responding to msgTrackProvideMetrics

When a service has the **autoMsgPass** style bit set to TRUE, it forwards all messages that are not **clsObject**, **clsService**, or **clsOption** messages to its target service. One such message can be **msgTrackProvideMetrics**. For example, if the service is the client of a **clsFrame** window, it will receive **msgTrackProvideMetrics** when the window is dragged around on the screen.

Most services do not need to handle **msgTrackProvideMetrics**. However, if the service's target is not opened, a **stsSvcTargetNotOpen** will be returned when **clsService** attempts to forward the message to the target. This could create a problem if the caller (of **msgTrackProvideMetrics**) is expecting either **stsOK** or **stsMessageIgnored** before proceeding further. In our **clsFrame** example, **msgTrackProvideMetrics** is sent to the frame's client (the service) after **clsFrame** already provides an adequate track metrics. The idea is to let the frame's client have a crack at poking the track metrics. If the service returns **stsSvcTargetNotOpen** instead of **stsMessageIgnored**, when the user drags the **clsFrame** window, a black box might appear briefly, but no tracker is created, and thus the window remains where it was.

To fix this problem, simply return **stsMessageIgnored** in response to **msgTrackProvideMetrics**. In your method table, you can say:

```
{msgTrackProvideMetrics, "StsMessageIgnoredMsgHandler",
},
```

See CLSMGR.H for examples of other default message handlers returning **stsOK**, **stsFailed**, and so on.

Deinstalling dependent services and applications

PenPoint 2.0 Japanese allows you to bundle applications and services together. For example, you might ship a database user interface application along with the database service. The service, in this case, is referred to as the *dependent* service. This association is made by putting the database service and the SERVICE.INI file in the application's directory. Deinstallation of the application and service happens when the application is deinstalled.

If you chose not to bundle your application and service together, you need to be very careful about the programmatic dependencies between the two items. You must be especially careful to test deinstallation, because it is possible to write code in such a way that once the application is deinstalled, the service would fail to deinstall, thereby leaving the service permanently installed.

Corrections and errata

The Service Class and class instances

You should use this function-interface approach *only* if you are having performance problems with your service.

The last paragraph of section 116.4.2 in the *PenPoint Architectural Reference* says that "the service can also tell its openers the entry points to specific procedural interfaces (if any)." But it neither describes how this is done nor refers to another section containing the general procedure. If you want your service to provide its clients with a function interface, thereby saving the overhead of object calls, it should respond to **msgSvcGetFunctions.** (This message is merely mentioned in Table 117-1 of the *PenPoint Architectural Reference.*) To do this, it must be a subclassed instance of **clsService** (which, by default, returns a null pointer.) Your service should pass back to the opener a pointer to a table of function entry points. The format of this pointer block is up to the service to define. See SERVMISC.H for more information about **msgSvcGetFunctions.**

Handling msgSvcOpenDefaultsRequested

In section 117.7.2.4 of the *PenPoint Architectural Reference*, delete the last sentence of the third paragraph. (This sentence is in parentheses and begins "The METHOD.TBL in the TESTSVC directory does not specify an ancestor call for **msgSvcOpenDefaultsRequested...**") The method table for the TESTSVC sample service *does* specify **objCallAncestorBefore** for **msgSvcOpenDefaultsRequested.**

In box and Out box changes

There is an In box/Out box bug that can affect an existing client of the In box/Out box service. The fix for this bug changes how a client enables or disables an Out box service.

The change adds an explicit message for a client to programmatically enable an Out box service. In 1.0 (and 1.0a), enabling an Out box service means that the service becomes the owner of the target. For example:

```
ObjCallRet(msgSvcGetTarget, anOutboxService, &getTarget, s);
setOwner.handle = getTarget.targetHandle;
setOwner.owner = anOutboxService;
ObjCallRet(msgSMSetOwner, getTarget.target.manager, &setOwner, s);
```

In PenPoint 1.01 and PenPoint 2.0 Japanese, the above code becomes:

```
ObjCallRet(msgIOBXSvcEnableService, anOutboxService, (P_ARGS)TRUE, s);
```

And the default behavior of **msgIOBXSvcEnableService** is to do exactly what is done in PenPoint 1.01 and PenPoint 2.0, namely to make **anOutboxService** the owner of its target.

Chapter 47 / International Services and Routines

Falling under the umbrella designation of International Services and Routines are a collection of functions, macros, structures and defined values that help developers internationalize application code. With the international routines, you can write one code base for your application that works naturally in several countries.

Note that, despite the title of "International Services and Routines," there are no "international" services in version 2.0 of Penpoint. In future versions, routines will call services to implement internationalization more thoroughly.

Internationalization in PenPoint makes use of both the international routines and resource files. Resource files can hold the Unicode text strings—in multiple languages—that an application requires for its UI and other purposes.

Your application will probably need to use the international routines if it must take into account anything that is culturally dependent, such as time and date formats, hyphenation rules, units of measure, and so on. (Of course, your application must also use Unicode for its strings.)

Part 2: PenPoint Internationalization Handbook, describes all aspects of writing your code for an international market.

International and related header files

Header files

TABLE 47-1

Header File	*Description*
ISR.H	Prototypes the functions and defines structures for the PenPoint international routines. Some routines perform text delimiting, hyphenation, data-type conversion and the conversion and parsing of dates and times. Other routines compare, sort and compress strings, and perform conversions involving dialects, character sets, units of measure and other linguistic elements. The routines themselves are in INTL.LIB. See "International routines" on page 508 for descriptions of these routines.
ISRSTYLE.H	Contains definitions of style values used as parameters in the ISR.H routines. Most international routines allow the client to specify a style for the operation. Styles modify the locale and must be appropriate to the routine. For example, some styles enable the client to specify dates in one of the four (or more) formats possible in each western language.
GOLOCALE.H	Defines the locale values used as parameters in the ISR.H routines. These values are of type LOCALE_ID, which is itself composed of three values: language, dialect, and country. (See description of INTL.H, below.) Values for weights and measures, currencies, time zones, eras and other international units are also defined. In addition to a list of current locale values, GOLOCALE.H specifies the required resource IDs for the lists containing the text strings, contains some macros that make tags for locale values, and defines some common locales. Note that this header file was named LOCALE.H in earlier versions of PenPoint.

Header files

TABLE 47-1 (continued)

Header File	*Description*
CHARTYPE.H	This file defines some classification tables for Unicode characters and provides a set of macros for testing and manipulating characters as specified in those tables. These macros are similar to those defined in CTYPE.H for the conversion and testing of ASCII characters. See "Character conversion and testing macros" on page 512.
INTL.H	This file defines the LOCALE_ID used as a parameter in the international routines. It also contains the macros that you can use to create, modify and access locale values. LOCALE_ID is an unsigned 32-bit type with three 8-bit fields for the language, dialect and country values defined in GOLOCALE.H. Currently, the other bits are reserved for future use.

The macros in INTL.H are:

```
intlLIDMakeLocaleId(l,d,c) // l=language, d=dialect, c=country
intlLIDGetLanguage(locale) // (values defined in GOLOCALE)
intlLIDGetDialect(locale)
intlLIDGetCountry(locale)
intlLIDSetLanguage(locale,v) // v = GOLOCALE value
intlLIDSetDialect(locale,v)
intlLIDSetCountry(locale,v)
```

International routines

Most functions require you to supply a locale and a style as arguments. The locale is a 32-bit type (LOCALE_ID) containing three 8-bit values; one identifies the country whose conventions the routine should observe and the remaining two identify the language and dialect that the routine must process. (Currently, no dialect values are defined, but they could easily be added in the future.)

In most cases the international routines require a style value as a parameter. A style value modifies the way a routine processes a locale, because there can be variations of linguistic forms within countries and languages. Styles are 32-bit types containing 16-bit segments; the low-order segment identifies a base style and the high-order segment contains flags that modify the base style.

The locale values are defined in GOLOCALE.H. You can compose LOCALE_ID structures by using the **intlLIDMakeLocaleId**() and related macros in INTL.H. You use bitwise operators and **intlStyleMask** and **intlStyleFlagsMask** to set the base value and one or more of the flags defined in ISRSTYLE.H in an unsigned 32-bit integer.

Those routines that take a locale as argument provide a macro that substitutes **intlDefaultLocale** for the locale. The default locale requests the current system locale. The macro has the same name as the function, but the prefix is **Loc** instead of **Intl**. The following example shows a typical macro definition:

```
#define LocDelimitWord(tx,s,st) \
            IntlDelimitWord(tx,s,intlDefaultLocale,st)
```

Most of the international functions come in nearly identical pairs. The **Intl...** versions work on null-terminated strings and the **IntlN...** versions work on counted strings. The **IntlN...** functions take an extra argument that specifies the legnth of the passed string. When you read the definitions of international routines in the following sections, remember that there is a counted-string counterpart to the null-terminating function listed (unless specifically noted otherwise). In other words, the definition for **IntlDelimitWord** applies equally to **IntlNDelimitWord**.

Some of the functions that handle null-terminated strings return the required length of an output buffer. Remember that these counts do not include the null character, so if you are using these functions to allocate memory for these buffers, make sure to add one to the count.

Delimiting and hyphenation routines

Delimiting routines

TABLE 47-2

Function	*Description*
IntlDelimitWord	Finds a "word" in a string. The style argument controls the internal definition of a word.
IntlDelimitSentence	Finds a "sentence" in a string. The style argument determines what is considered a sentence.

Hyphenation routines

TABLE 47-3

Function	*Description*
IntlBreakLine	Given a string of a certain length, this routine calculates a line break that is valid for the locale and returns the hyphenation information in a structure of type INTL_BREAK_LINE. This information includes the position at which to make the line break, the number of characters to delete from each side of the break, and the characters to insert on each side of the break point.

Time conversion routines

These routines pass in or receive back time information in an argument of INTL_TIME. This structure is a superset of the standard **tm** structure. In addition to the standard **tm** fields, it includes the time zone as a posix string (**pTz**) and an **era** field that can put the year in a context other than *anno Domini* (AD).

Time conversion functions

TABLE 47-4

Function	*Description*
IntlSecToTimeStruct	Converts the time, in seconds since 0:00 January 1, 1970 UTC (GMT), into an international time structure (INTL_TIME). Use the **time()** function to get the current time in seconds. For the current release of PenPoint, set the **pTimeZone** field (the target time zone) to **pNull**.
IntlTimeStructToSec	Converts an international time structure (INTL_TIME) to the time in seconds since 0:00 January 1, 1970 UTC (GMT). Currently, this function works only on times in the current time zone.
IntlOSDateTimeToIntlTime	Converts the time in the PenPoint system format into the international time structure (INTL_TIME). Since the system time is always a modern Gregorian date, the era in INTL_TIME is always set to **itcEraAD**.
IntlIntlTimeToOSDateTime	Converts the time specified in the international time structure (INTL_TIME) into the PenPoint system format. Since the system time is always a modern Gregorian date, you must set the era in INTL_TIME to **itcEraAD**.

Note that these last two functions, **IntlOSDateTimeToIntlTime** and **IntlIntl-TimeToOSDateTime**, should be used only by code that is involved in setting the system clock. Other code should use the **time()** function and then do conversions to and from international time with **IntlSecToTimeStruct** and **IntlTime-StructToSec**.

Formatting routines

These functions take an input value and convert it into a string. They return the length of the generated string or, if **pString** is **pNull**, they do no formatting but do return the size required for the output buffer.

Calls to **IntlFormatS32** and **IntlFormatNumber** include two extra arguments, one to indicate the minimum number of integer digits and the other to specify the maximum number of fractional digits. For S32 values, the fraction displayed is always zero, unless the **intlFmtNumScale** flag is set. This allows scaling, particularly for displaying currency.

Calls to **IntlFormatDate** and **IntlFormatTime** include as an argument a resource tag for a format string. If this tag is not NIL, the routine fetches the format from **theProcessResList** with a resource group of **resGrpTK**. You can use the format with Compose Text functions to generate the output string.

Formatting functions

TABLE 47-5

Function	*Description*
IntlFormatS32	Converts a signed integer to a string.
IntlFormatNumber	Converts a floating point number to a string.
IntlFormatDate	Converts a time structure to a date string.
IntlFormatTime	Converts a time structure to a time string.

Parsing routines

These routines convert an input string into the value of the requested type. They return the length of the parsed string. You can treat the string as a single item or as a set of tokens of known type (see ISR.H for further details). Note that the date and time parsing routines only set values for the date elements that they find in the string; they do not set default date values. For example, "September" results only in **IntlParseDate** setting the **mon** field and nothing else. You should therefore initialize the INTL_TIME structure with **intlTimeStructInit** before making a call, so that you can find out which fields were filled.

Parsing functions

TABLE 47-6

Function	*Description*
IntlParseS32	Converts a string to a signed integer.
IntlParseNumber	Converts a string to a floating point number.
IntlParseDate	Converts a string to a date as contained in an international time (INTL_TIME) structure.
IntlParseTime	Converts a string to a time as contained in an international time (INTL_TIME) structure.

Collation routines

Sort and compare functions

TABLE 47-7

Function	Description
IntlCompare	Compares two strings in a linguistically correct method according to the locale. You can use it in searching or in sorting a list (although the **IntlSort** routine does that already).
IntlSort	Sorts an array of strings in a linguistically correct way, according to the locale.

String conversion routines

The **IntlStrConvert** routine converts Unicode strings from one stylistic or linguistic format to another, such as between upper and lower case, katakana and hiragana and composed characters and floating diacritics. Unlike the character-conversion macros in CHARTYPE.H, these routines handle conversions that affect the lengths of strings, or that depend on locale, on context, or on a dictionary for some characters.

You can use the Unicode string conversion routines in three ways:

- In a single call.
- Writing from a single input buffer.
- Using extended input.

See ISR.H for further details on these methods.

Character set conversion routines

Both functions return the number of target characters that were produced (not counting the null) unless there is an error. If **IntlMBToUnicode**, for instance, finds an unknown character, it converts it to 0xFFFD unless you specify a flag to override this conversion. (The actual character displayed depends on the character set.) The style parameter specifies the character set to convert from.

The difference in the behavior between the null-terminating and counted-string versions of these functions is significant in this area. If, for example, there is no null character in the string given to **IntlMBToUnicode**, the output buffer fills up and the string is truncated. **IntNMBToUnicode** updates the source length to be the number of characters processed and returns normally. To verify that the string was processed, you can then compare the number of characters passed in with the number returned.

By the way, do not confuse these routines with the **IntlStrConvert** function, which only does conversions *within* Unicode.

Character set conversion functions

TABLE 47-8

Function	Description
IntlMBToUnicode	Converts a multibyte string to a Unicode string.
IntlUnicodeToMB	Converts a Unicode string to a multibyte string.

String compression routines

String compression functions TABLE 47-9

Function	*Description*
IntlCompressUnicode	Converts a Unicode string into a compressed form stored in a counted array of bytes. If successful, it returns the number of bytes of compressed data. The output buffer must be big enough to hold the compressed data, so you can call either routine first with a null pointer for the destination, and the required buffer size is return. (Remember to add 1 for the null character when allocating memory for this buffer.)
IntlUncompressUnicode	Uncompresses a compressed Unicode string. It returns the number of characters produced, unless there is an error. The output buffer must be big enough to hold the uncompressed data, so you can call either routine first with a null pointer for the destination, and the required buffer size is return. (Remember to add 1 for the null character when allocating memory for this buffer.)

Units conversion routine

The **IntlConvertUnits** routine converts an input value specified in one unit of measure to a value in another unit of measure. (These unit-of-measure definitions are in GOLOCALE.H.) It does not require locale or style as arguments.

Character conversion and testing macros

These macros are defined in CHARTYPE.H:

CHARTYPE macros TABLE 47-10

Macro	*Description*
IntlGetCharType	Returns the 16-bit flags that apply to a character from the set defined in the section "Character flags" on page 514. These flags are intended to be "international," rather than specific to any particular language.
IntlCharToUpper	If a character is in lower case, it returns the upper-case equivalent; if the character is not in lower case, it returns the character unchanged. One-to-one single character conversions do not work for all characters.
IntlCharToLower	If a character is in upper case, it returns the lower-case equivalent; if the character is not in upper case, it returns the character unchanged. One-to-one single character conversions do not work for all characters.
IntlCharIsUpper	Returns TRUE if the character passed is an uppercase character.
IntlCharIsLower	Returns TRUE if the character passed is a lowercase character.
IntlCharToFullWidth	If a character is half-width, it returns the full-width equivalent; if the character is not half-width, it returns the character unchanged. This macro maps only the romaji and katakana characters, not the hangul ones.
IntlCharToHalfWidth	If a character is full-width, it returns the half-width equivalent; if the character is not full-width, it returns the character unchanged. This macro maps only the romaji and katakana characters, not the hangul ones.
IntlCharIsFullWidth	Returns TRUE if the character passed is a full-width character.
IntlCharIsHalfWidth	Returns TRUE if the character passed is a half-width character.
IntlCharIsKatakana	Returns TRUE if the character passed is a katakana character. Includes both hankaku and zenkaku katakana.
IntlCharIsHiragana	Returns TRUE if the character passed is a hiragana character.

CHARTYPE macros

TABLE 47-10 (continued)

Macro	*Description*
IntlCharIsHan	Returns TRUE if the character passed is a kanji, hanja or hanzi character. XJIS gaiji characters are mapped into the private use area: 0xf300 to 0xf5fc.
IntlCharIsCompatibilityZone	Returns TRUE if the character passed is in the Compatibility zone.
IntlCharIsGOCorporate	Returns TRUE if the character passed is in the GO Corporate zone.
IntlCharIsSpace	returns TRUE if the character passed is a space character of any kind. This does not include cursor movement control characters such as tab and linefeed.
IntlCharIsAlphabetic	Returns TRUE if the character passed is a character from an alphabetic script. These include Latin, Greek, Cyrillic, and Latin characters from the Compatibility zone and excludes digits, punctuation, spaces, and so on.
IntlCharIsAlphanumeric	Returns TRUE if the character passed is a character from an alphabetic script or an international digit (for example, [0-9]). This includes Latin, Greek, Cyrillic, and Latin characters from the Compatibility zone and excludes punctuation, spaces, and so on.
IntlCharIsFloating	Returns TRUE if the character passed is a floating diacritic. These include bound graphemes such as circumflex, accent acute, and daku ten.
IntlCharIsComposed	Returns TRUE if the character passed is a composed character. Composed characters can reasonably be represented by a base character and a floating diacritic.
IntlCharIsPunctuation	Returns TRUE if the character passed is a punctuation character.
IntlCharIsGraphic	Returns TRUE if the character passed corresponds to a glyph. The character cannot be a control, spacing, or undefined character.
IntlCharIsPrinting	Returns TRUE if the character passed corresponds to a glyph or space. The character cannot be a control or undefined character.
IntlCharIsControl	Returns TRUE if the character passed is a control character.
IntlCharIsDecimalDigit	Returns TRUE if the character passed is an international digit ([0-9]).
IntlCharIsHexadecimalDigit	Returns TRUE if the character passed is a hex digit ([0-9a-fA-F]).

Character flags

The following list defines the flags that can be associated with characters. You can obtain the flags for a character through the **IntlCharGetFlags** macro.

```
intlCharTypeSentEndflag0// Can end a sentence
intlCharTypeLineBrkflag1// Breaks a line
intlCharTypeSpaceflag2// White Space (not tab, line Brk)
intlCharTypeNumberflag3// valid char in a number
intlCharTypeWordflag4// valid char in a word
intlCharTypeCantStartLineflag5// Can't start a line
intlCharTypeCantEndLineflag6// Can't end a line
intlCharTypeAlphabeticflag7// Latin, Greek, or Cyrillic
intlCharTypeFloatingflag8// Floating Diacritic
intlCharTypeComposedflag9// Composed Character
intlCharTypePunctuationflag10// Punctuation Character
intlCharTypeGraphicflag11// Any printing, non-space char
intlCharTypeDecimalDigitflag12// Decimal digit
```

External tables

The character conversion and testing macros operate on a set of Unicode tables for each alphabet. These tables are defined in CHARTYPE.H.

Part 6 / PenPoint User Design Reference Supplement

Chapter 48 / Introduction

This document is intended for software developers who are developing products for PenPoint 2.0 Japanese. It provides a description of the user interface from the end-user's point of view. However, this document is not intended as a user guide to PenPoint.

The structure of this document is designed to be both useful for finding specific information and for reading large sections at a time. A considerable amount of cross-referencing takes place, allowing information to appear in only one place wherever possible.

Information has been presented in a way that focuses on the "component" level, often to the exclusion of the "big-picture" level that is provided by end-user documentation. Many of these components are used in multiple places throughout PenPoint, and this provides the rationale behind the structure.

The major change between version 1.0 and 2.0 of the PenPoint operating system is support of the Japanese language. Specifically, PenPoint 2.0 Japanese is the Japanese version of PenPoint, as well as the foundation for a future international versions of the operating system. Changes to functional components from PenPoint 1.0 can be found in the sections labelled "Change Notes."

This document relies heavily on the existing set of PenPoint 1.0 documentation, especially the *PenPoint User Interface Design Reference.*

Chapter 49 / The Notebook

The Notebook application is the principal user interface to the PenPoint system provided by GO Corporation, the NUI (Notebook User Interface). It serves as the organizing metaphor for the user as well as the underlying component for the Stationery notebook and other system-level applications.

Table of Contents

The Table of Contents (TOC) of the PenPoint 2.0 Japanese notebook is used as the central location for management of documents. The TOC is an application itself that runs inside the Notebook application (**clsSectApp**).

Operational model

The TOC is the central application in the PenPoint Notebook, appearing on the first page of the Notebook and controlling creation and navigation of the Notebook contents. The user can not delete the TOC page from the Notebook.

The user can only select one item (document or section) at a time in the TOC. The user can re-order pages by moving or copying document icons. To move the document, the user presses the document title and drags it a new location. Similarly, to copy the document, the user tap presses the document title and drags the marquee to a new location.

When the user creates a new document, the icon appears in one of three places, depending on how the user creates the document:

- At the gesture point if the user uses the caret ∧ gesture.
- Below the current selection if the user taps on the menu item Create.
- After the last page if there is nothing selected when the user taps on Create.

The user turns to documents in the Notebook with the tap gesture. A double tap gesture floats (if float is enabled) the document over the TOC. The user may tap on the page number or the icon or button (depending on the view) to the left of the document name. A tap on the document name selects the text. A double tap on a section name expands the section in the TOC. The circle ○ gesture is used to edit document names.

Standard elements

The TOC contains a list of the documents and sections currently in the Notebook, along with the pages on which they are found. Section pages contain another TOC for that section. The Layout menu controls the view of the TOC.

Table of Contents gestures

Gestures used in the Table of Contents TABLE 49-1

Gesture	*Name*	*Keyboard*	*Action*
ỵ	Tap	Enter	Turns to page (on page # or icon). Selects name of page (on page).
ỵ̣	Double tap	Ctrl + Enter	Floats document (if float enabled). Open/closes section.
|	Flick up or *down*	Page Up Page Down	Scrolls up or down.
||	Double flick up or *down*	Ctrl + Home Ctrl + End	*Up* scrolls to end of TOC. *Down* scrolls to beginning of TOC.
X	Cross out	Delete	Deletes target page.
∧	Caret		Opens Create menu at target point.

Menus

There are five menus in the TOC: Document, Edit, Options, View, and Create.

Document menu

Send Document Submenu of available services. Selected documents are sent via selected service.

About Contents Information sheets on Table of Contents.

Edit menu

Move Places selection into Move mode.

Copy Places selection into Copy mode.

Delete Deletes selection. Confirm note is opened.

Rename Opens Edit Pad with selection name available for editing.

Options menu

Document Only available when a document is selected. Opens Document option sheet.

Section Only available when a section is selected. Opens Section option sheet.

Layout Opens Layout option sheet.

Controls Opens Controls option sheet.

View menu

Expand Opens all sections one level if no selection. Opens selected section one level.

Collapse Collapses all sections if no selection. Collapses selected section.

Turn To Turns to selected document or section.

Bring To Floats selected document or section.

Create menu

Document List A list of all documents in Stationery notebook that have Menu checked. When the user selects one, PenPoint creates a new document of that type after currently selected document or at end of TOC if no selection.

Section Creates a new section after current selection or at end of TOC if no selection.

Option sheets

There are 4 option sheets available from the Table of Contents: Document, Section, Layout, and Controls. The Document/Section sheets are only visible when a Document/Section is selected. The main TOC page (as opposed to the Section TOC pages) is the only page in the Notebook that does not have the standard Access and Comments option sheets. The Access Speed for the main TOC is always set to Accelerated.

Document

The Document option sheet provides access to the fixed and variable attributes of a selected document. These include:

Title The title of the document; not editable here.

Type The type of document (the application it represents); not editable.

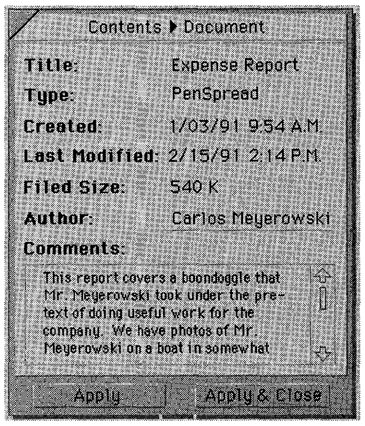
Created The creation date and time; not editable.

Last Modified The date and time of the last access (last checkpoint or page-turn away from document); not editable.

Filed Size The size of the file on disk in kilobytes; not editable.

Author The author; editable.

Comments Comments; editable.



Title:	Expense Report
Type:	PenSpread
Created:	1/03/91 9:54 A.M.
Last Modified:	2/15/91 2:14 P.M.
Filed Size:	540 K
Author:	Carlos Meyerowski

Comments:

This report covers a boondoggle that Mr. Meyerowski took under the pre-text of doing useful work for the company. We have photos of Mr. Meyerowski on a boat in somewhat

Section

The Section option sheet is the same as the Document option sheet described in the preceding section, only the title changes from Document to Section. The option sheet changes its content as the user changes the selection.

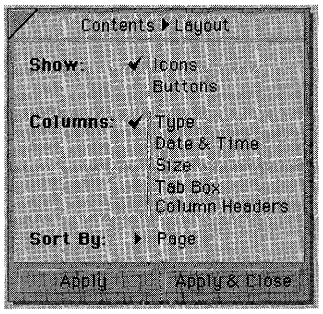
Layout

The Layout option sheet provides controls for the visual display of the TOC. The Show choice allows the user to select either icons or square buttons or neither as the indicator to the left of the document name. The default is Icons.

The Columns checklist is a multiple choice list that sets up the columns displayed in the TOC as well as the Column Headers. The Document name and Page number always appear, and the user cannot remove them. The default is Column Headers.

The Sort pop-up selects the sort criteria for the TOC. There are five choices: Page, Name, Type, Date, Size. The default is Page. Sort direction is ascending for Page, Name, and Type; descending for Date and Size. Sorting only affects the display in the TOC; it does not re-order the pages of the Notebook.

In PenPoint 2.0 Japanese the sort order is phonetic, with kanji being indexed by the first phonetic of their Chinese reading according to the JIS standard. Non-JIS characters follow in Unicode order.



Contents ▸ Layout

Show: [✓ Icons] Buttons

Columns: [✓ Type] Date & Time, Size, Tab Box, Column Headers

Sort By: ▸ Page

Controls

The Controls option sheet is the standard PenPoint Controls sheet with the only choices being: Menu Line, Scroll Margins, and Cork Margin. The default state is Menu Line on, Scroll Margins on, and Cork Margin off.

Tabs

Change notes

- Tabs do not contain vertical text.
- Contains zenkaku by default.

Chapter 50 / The Bookshelf

Help

Change notes

There are no functional changes to the Help system from 1.0 to 2.0.

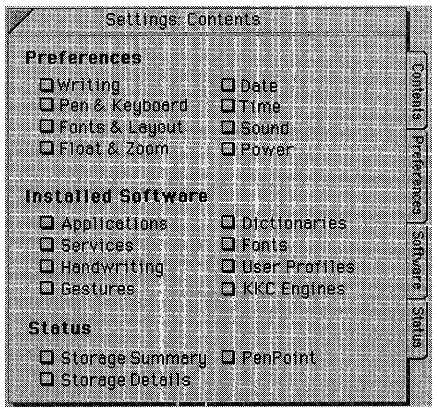
Settings

Operational model

The Settings notebook gives the user a single place to go to view and modify settings both for the PenPoint 2.0 Japanese operating system as a whole and for whatever software is currently installed in the system.

While it employs the notebook metaphor, the Settings notebook differs from the normal PenPoint data notebook in two ways:

- It is optimized for quick navigation through a small number of pages with only one level of sections. Instead of local contents pages for each section, each page has a pop-up menu in its title line allowing the user to turn to any other page in that section.
- It is not editable in any way by the user. Neither the pages nor the tabs can be deleted, re-ordered, or renamed, and the table of contents has no menu line and no display options.
- It doesn't have page numbers.



Settings: Contents

Preferences
- ☐ Writing
- ☐ Pen & Keyboard
- ☐ Fonts & Layout
- ☐ Float & Zoom
- ☐ Date
- ☐ Time
- ☐ Sound
- ☐ Power

Installed Software
- ☐ Applications
- ☐ Services
- ☐ Handwriting
- ☐ Gestures
- ☐ Dictionaries
- ☐ Fonts
- ☐ User Profiles
- ☐ KKC Engines

Status
- ☐ Storage Summary
- ☐ Storage Details
- ☐ PenPoint

Change notes

- Pen section changed to Pen & Keyboard.
- Writing changed to Writing Style and moved to fifth position from the first position.
- Keyboard choice added to Pen & Keyboard with pop-up: American, Japanese A01.
- Import/Export choice added to Fonts & Layout with pop-up: 1990 JIS, 1978 JIS.
- Kana-Kanji Conversion section added to Software section.
- Practice... button (and facility) removed from Handwriting sheet.
- Date formats changed.
- Time formats changed.

Preferences

The user uses the Preferences section of the Settings notebook for viewing and setting system-wide user preferences. It contains 8 sections.

Pen & Keyboard

There are 5 preferences for pen and keyboard input:

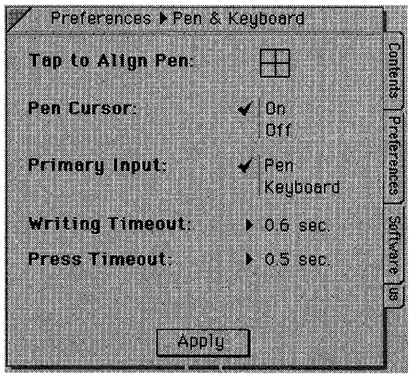
Tap to Align Pen User taps in center of square to align pen.

Pen Cursor User can turn pen cursor on or off (default = off).

Primary Input Determines primary input device, pen or keyboard (default = pen).

Writing Timeout Interval system pauses after user lifts pen from screen before translating input (default = 0.6 seconds, range = 0.2 to 1.0 in 0.1 increments, 1.2, 1.5, 2.0 seconds).

Press Timeout Interval user must touch the pen to the screen before the press gesture is recognized (default = 0.5 seconds, range = 0.2 to 1.0 seconds in 0.1 increments).



Tap to Align Pen:

Pen Cursor: [✓ On] Off

Primary Input: [✓ Pen] Keyboard

Writing Timeout: ▸ 0.6 sec

Press Timeout: ▸ 0.5 sec

Fonts & Layout

PenPoint Font Font used by system and applications for text (not in edit fields) (default = Mincho, choices = Gothic, Roman, Courier, Sans Serif, Mincho).

Field Font Font used by system for translated text (default = Mincho, choices = Gothic, Roman, Courier, Sans Serif, Mincho).

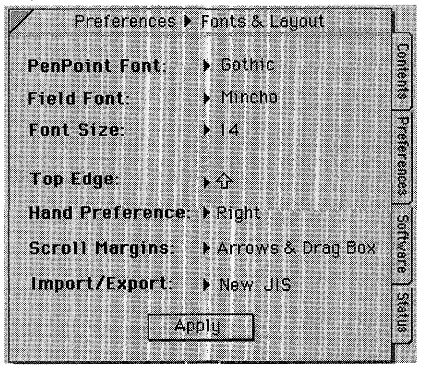
Font Size Font size (in points) used as default for both system text and field text (default = 12 pt., choices = 10, 12, 14, 16, 18, 20, 24 pt.).

Top Edge Determines orientation of screen. Menu has four arrows, one pointing to each edge of computer. The user chooses one arrow to re-orient the display so that the edge is at the top. After the user taps the Apply button, the arrow again points to the top edge.

Hand Preference Hand Preferences for screen layout. Effects placement of scrollbars (default = Right, choices = Left, Right).

Scroll Margins Allows user to choose either a traditional scroll margin with arrows and a drag box or a simple margin for flicking (default = Arrows & Drag Box, choices = Tap & Flick Area, Arrows & Drag Box).

Import/Export Determines the data exchange format to use (default = New JIS, choices = New JIS, JIS).

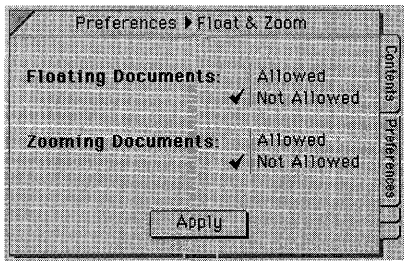

Preferences ▸ Fonts & Layout

PenPoint Font: ▸ Gothic
Field Font: ▸ Mincho
Font Size: ▸ 14
Top Edge: ▸ ⇧
Hand Preference: ▸ Right
Scroll Margins: ▸ Arrows & Drag Box
Import/Export: ▸ New JIS

Float & Zoom

Floating Documents Determines if the user can float documents or not (default = Not Allowed, choices = Allowed, Not Allowed).

Zooming Documents Determines if the user can zoom documents or not (default = Not Allowed, choices = Allowed, Not Allowed).


Preferences ▸ Float & Zoom

Floating Documents: Allowed [✓ Not Allowed]
Zooming Documents: Allowed [✓ Not Allowed]

Writing

Writing Style Determines the type of character input allowed (default = Mixed Case, choices = Upper Case Only, Mixed Case [applies to Roman text only]).

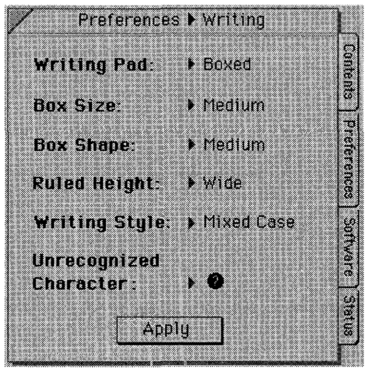
Writing Pad Determines type of box used in edit pads (default = Boxed, choices = Ruled/Boxed, Ruled, Boxed).

Box Size Determines size of box in edit pads with boxes (default = Medium, choices = Very Small, Small, Medium, Large, Very Large).

Box Shape Determines the proportional shape of the box in edit pads (default = Medium, choices = Short, Medium, Tall).

Ruled Height Determines the height of space above ruled line in edit pads (default = Medium, choices = Very Small, Small, Medium, Large, Very Large).

Unrecognized Character Determines character used to indicate unrecognized character, default = ●, choices = ●, _).



Writing Pad: ▸ Boxed
Box Size: ▸ Medium
Box Shape: ▸ Medium
Ruled Height: ▸ Wide
Writing Style: ▸ Mixed Case
Unrecognized Character: ▸ ●

Time

Current Time Current time as read from the system clock.

Time Zone Local time zone, with hours difference from GMT (default = +9 Tokyo).

Format Determines the time format that the user would like PenPoint and applications to display.

Style Determines the style of the time display that the user would like PenPoint and applications to display.

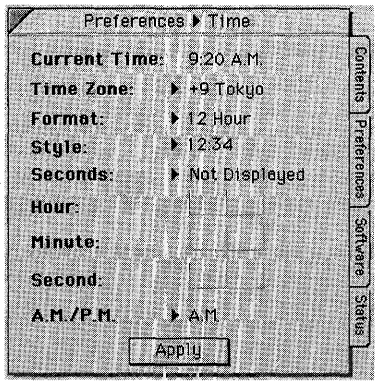
Seconds Determines if the user wants seconds displayed as part of time (default = Not Displayed, choices = Displayed, Not Displayed).

Hour Write-in field to allow user to set the hour (range = 1 to 12 for 12-hour format, 0 to 23 for 24-hour format).

Minute Write-in field to allow user to set the minutes (range = 0 to 59).

Second Write-in field to allow user to set the seconds (range = 0 to 59).

A.M./P.M. Sets A.M./P.M. for time (available only when format = 12 hours).



Current Time: 9:20 A.M.
Time Zone: ▸ +9 Tokyo
Format: ▸ 12 Hour
Style: ▸ 12:34
Seconds: ▸ Not Displayed
Hour:
Minute:
Second:
A.M./P.M. ▸ A.M.

Date

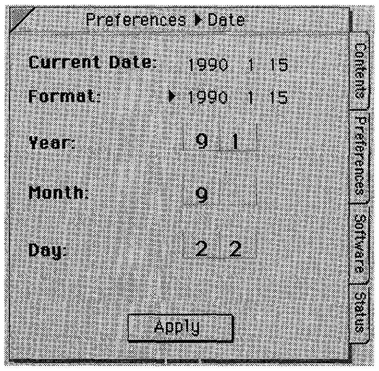
Current Date Current date as set in the system clock, displayed in format set below.

Format Determines default format for display of date.

Month Determines month of year (choices = January to December).

Day Write-in field for day of month (range = 1 to number of days in selected month).

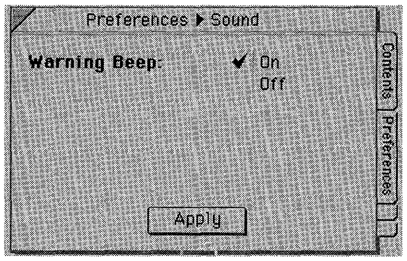
Year Write-in field for last two digits of year, assumes 19 preceding these digits (range = 70 to 99).



Current Date: 1990 1 15
Format: ▸ 1990 1 15
Year: 9 1
Month: 9
Day: 2 2

Sound

Warning Beep Determines if warning beep sound is made on error conditions (default = On, choices = On/Off).


Preferences ▶ Sound
Warning Beep: [✓ On] Off

Power

If a machine does not have suspend capability, then Manual Standby and Auto Standby choices are not visible.

Manual Standby Button that puts the processor into suspend mode.

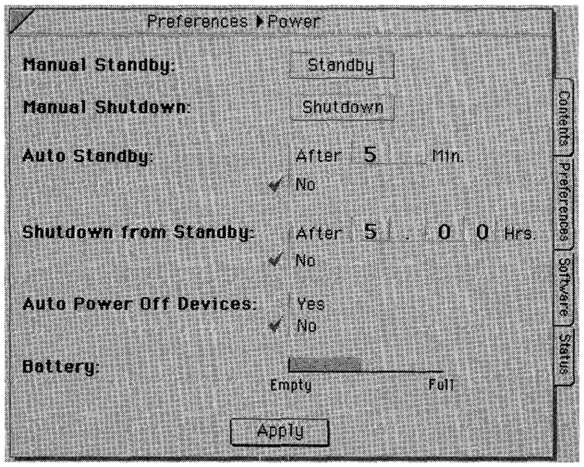
Manual Shutdown Button that shuts the system down, unless Auto Standby is on, then processor goes into suspend mode.

Auto Standby Determines if PenPoint automatically suspends the processor whenever there is no input from the user for the specified period (default = No, range = 1 to 99 minutes).

Shutdown from Standby Determines if PenPoint automatically shuts the processor down whenever the system remains suspended (in Standby) for the specified period (default = No, range = 0.10 and 9.0 hours).

Auto Power-Off Devices Determines if the computer's main circuitry and peripherals automatically power down when not in use. The manufacturer specifies the period of inactivity after which each device powers down (default = No, choices = Yes, No).

Battery Gauge of how much power remains in the battery.

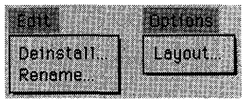

Preferences ▶ Power
Manual Standby: Standby
Manual Shutdown: Shutdown
Auto Standby: After 5 Min. [✓ No]
Shutdown from Standby: After 5 . 0 0 Hrs. [✓ No]
Auto Power Off Devices: Yes [✓ No]
Battery: Empty Full

Installed software

The second section of the Settings notebook is for installed software. The Installed Software section has a page for each category of installable software. From these pages the user can:

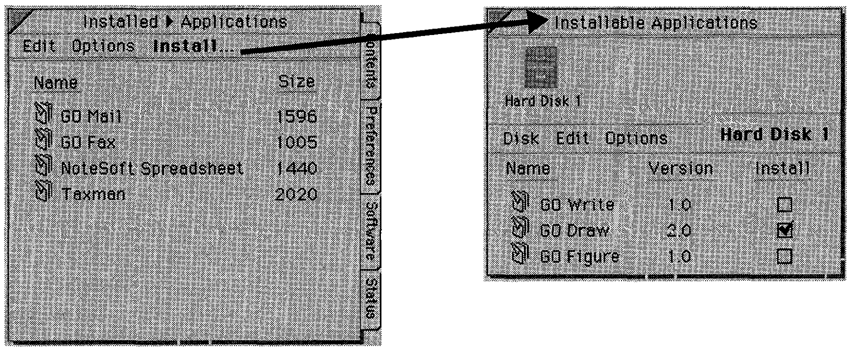
- See what's currently installed on the machine.
- Perform housekeeping functions such as deinstalling, saving to disk, restoring from disk.
- Set any options that the installed software provides. For example, an application may provide options that apply to all instances of the application.

There are seven pages (categories) to the Installed Software section of the Settings notebook. All seven have the same two menus: Edit and Options.



Applications

The Applications page shows the applications currently installed in the system. The Install... button on the right of the menu line (common to all the pages of the Installed Software section) displays a sheet showing the installable software of the appropriate category. The user taps on the checkbox to install (or deinstall) any of the installable items.



Name	Size
GO Mail	1596
GO Fax	1005
NoteSoft Spreadsheet	1440
Taxman	2020

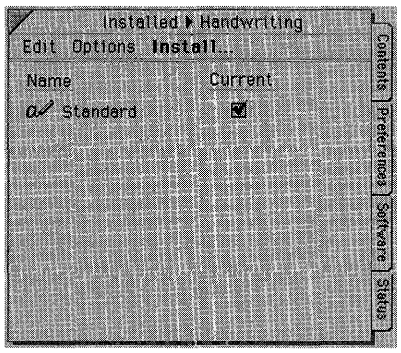
Name	Version	Install
GO Write	1.0	☐
GO Draw	2.0	☑
GO Figure	1.0	☐

Services

The Services page shows the installed services. Services include printer drivers, network connections, and other system-level programs.

Handwriting

The Installed Handwriting page shows the installed handwriting profiles. Each profile is associated with a handwriting engine and indicates the Current profile.

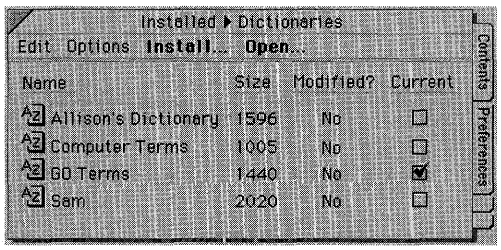

Installed ▸ Handwriting

Edit Options **Install...**

Name	Current
Standard	☑

Dictionaries

The Dictionaries page shows the installed dictionaries. Tapping the Open... button opens the selected dictionary (or current dictionary, if none is selected) into its own pop-up sheet.

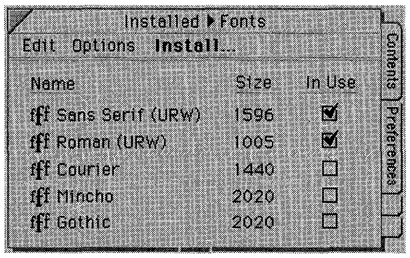

Installed ▸ Dictionaries

Edit Options **Install...** **Open...**

Name	Size	Modified?	Current
Allison's Dictionary	1596	No	☐
Computer Terms	1005	No	☐
GO Terms	1440	No	☑
Sam	2020	No	☐

Fonts

The Fonts page shows the installed fonts for the system.

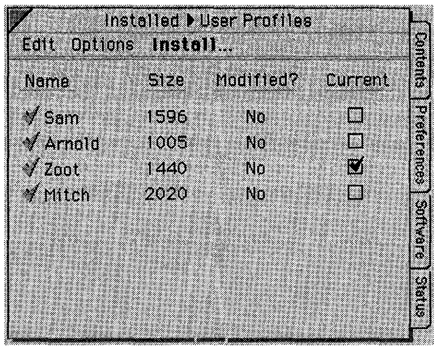

Installed ▸ Fonts

Edit Options **Install...**

Name	Size	In Use
Sans Serif (URW)	1596	☑
Roman (URW)	1005	☑
Courier	1440	☐
Mincho	2020	☐
Gothic	2020	☐

User Profiles

The User Profiles page shows all the profiles established for a machine. A user profile consists of all the preference settings for that user. Tapping the checkbox to make a profile current is equivalent to applying all the user's preferences in a single step.



Name	Size	Modified?	Current
✓ Sam	1596	No	☐
✓ Arnold	1005	No	☐
✓ Zoot	1440	No	☑
✓ Mitch	2020	No	☐

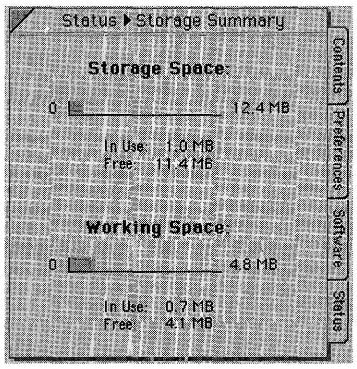
Status

Storage Summary

The Storage Summary page shows the user how much space is currently being used in the machine for storage, and how much remains to create and work with documents. This page is dynamic and updates to reflect changes in usage when it is open.

Storage Space The amount of space available for documents, accessories, and installed software. This is the space available to create new documents, copy existing documents, import files from disk, or install new software.

Working Space The amount of space available for active documents. Active documents include open documents and accessories, and also any documents for which the user has set Access Speed to Accelerated on the Access sheet.



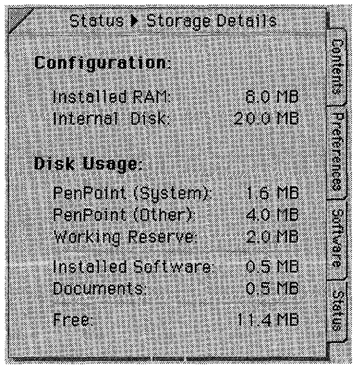
```
Storage Space:
0                12.4 MB
In Use:  1.0 MB
Free:   11.4 MB

Working Space:
0                 4.8 MB
In Use:  0.7 MB
Free:    4.1 MB
```

Storage Details

The Storage Details page presents a finer-grained view of the storage space usage than the Storage Summary page.

This page lists:

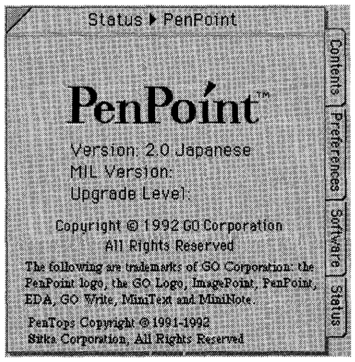
- Amount of installed RAM (MB).
- Size of internal disk (MB).
- Amount of space on internal disk used by PenPoint operating system.
- Amount of space on internal disk used by other PenPoint files.
- Amount of space on internal disk reserved as working space.
- Amount of space on internal disk used by the installed software (exclusive of PenPoint).
- Amount of space on internal disk used by documents.
- Amount of free space available on internal disk.



```
Status ▸ Storage Details

Configuration:
  Installed RAM:          8.0 MB
  Internal Disk:         20.0 MB

Disk Usage:
  PenPoint (System):      1.6 MB
  PenPoint (Other):       4.0 MB
  Working Reserve:        2.0 MB
  Installed Software:     0.5 MB
  Documents:              0.5 MB

  Free:                  11.4 MB
```

PenPoint

This page indicates the version of PenPoint currently running, along with the Copyright notice for the system.



Status ▸ PenPoint

PenPoint™

Version: 2.0 Japanese
MIL Version:
Upgrade Level:

Copyright © 1992 GO Corporation
All Rights Reserved

The following are trademarks of GO Corporation: the PenPoint logo, the GO Logo, ImagePoint, PenPoint, EDA, GO Write, MiniText and MiniNote.

PenTops Copyright © 1991-1992
Sitka Corporation, All Rights Reserved

Accessories

Unicode Browser

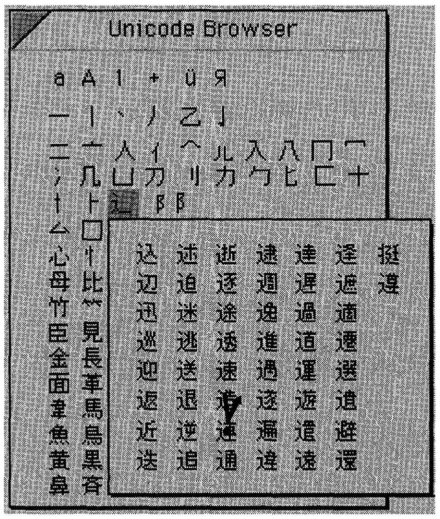
The Unicode Browser is an accessory to help the user find specific characters and put them into the input stream. It contains all the characters available in the Japanese fonts when those fonts are installed. (The figure below is not an exact representation.)

The first row of characters in the Unicode Browser are hiragana, katakana, roman alphabet, punctuation, accented upper case roman alphabet, accented lower case roman alphabet, gesture font, GO UI glyph font. The remaining rows of the browser contain the kanji radicals.

Tapping on any character in the top level of the Browser opens a submenu. For non-kanji (the top row), the submenu contains the characters in the set represented. For kanji radicals, the submenu contains all characters that use the top-level radical as their base radical. The size of this submenu varies from radical to radical.

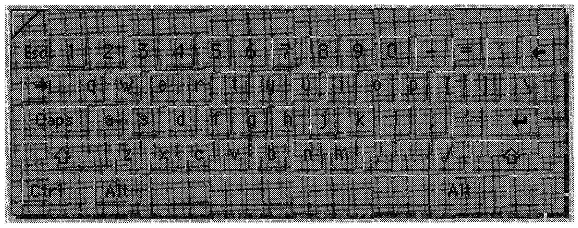
Tapping on a character in the submenu inserts that character into the input stream at the current insertion point. After the user makes a selection, the submenu closes, and the Browser remains open. Tapping outside the submenu closes the submenu without any character being selected. If there is no current insertion point, the submenu closes and nothing happens.

The Unicode Browser only accepts the tap ✓ gesture as a valid gesture on the top level and the second-level submenu. The close corner closes the Browser.



Keyboard

The PenPoint™ operating system provides a virtual keyboard. The virtual keyboard enters text directly into the input stream.


Esc 1 2 3 4 5 6 7 8 9 0 - = ` ←
→ q w e r t y u i o p [ ] \
Caps a s d f g h j k l ; ' ↵
⇧ z x c v b n m , . / ⇧
Ctrl Alt Alt

Chapter 51 / Overall System Changes

This chapter describes changes that affect aspects of the PenPoint user interface throughout the system.

Option sheets

The option sheet model has not changed between PenPoint 1.0 and PenPoint 2.0. You can find additional information about option sheets in this document in:

- "MiniText," below.
- "MiniNote" on page 539.

MiniText

MiniText is a simple text processor provided with PenPoint 2.0 Japanese. It provides multifont capabilities and simple formatting tools. MiniText is implemented as a "wrapper" around the text component of PenPoint. Because it is based on the core text component, it has changed to support Japanese in PenPoint 2.0.

Like most word-processor applications, MiniText uses a blank page into which the user can begin entering text. Input is accomplished in either in-line edit pads, through pop-up edit pads, or with the keyboard. The user selects text and then operates on it for most commands.

MiniText provides text editor and formatting features in a simple interface. The user can format characters and paragraphs, set tabs, as well as embed signatures for letters and other documents.

Change notes

These items have changed in MiniText:

- Units are changed from inches to centimeters.
- Case menu is removed.
- Convert menu is added.
- Default font becomes 12 pt. Mincho.
- Convert text gestures (H, Z) added.
- Greater keyboard support provided.
- Proof & Spell functionality only works on English words.
- Text wraps according to Japanese Taboo processing protocols.
- Fully justified text according to Japanese methods.
- Hankaku and zenkaku spaces added to Insert menu.

Option sheets

MiniText provides four option sheets. The user customizes the text and the display of the text with these sheets. All the functionality in these option sheets is part of the PenPoint text component and is available to any application that uses the text component.

Character option sheet

The Character option sheet contains three controls: Font, Size, and Style. The user controls the attributes of the characters in a document with this sheet.

Font A pop-up list of the fonts currently installed in PenPoint 2.0 Japanese. The default is Mincho.

Size A pop-up list of the font sizes available for the selected font. The default size is 12 pt.

Style A multiple checklist of the styles that can be applied to a font. The default is that no additional styles are added to text.

Paragraph option sheet

The Paragraph option sheet contains eight controls. The user sets the attributes of paragraphs in the document with this sheet.

Alignment The user aligns paragraphs left, right, center, or justified.

Line Height Height in centimeters.

Between Lines Space between lines, in centimeters.

1st Line Offset The user sets the amount of space that the first line is offset.

Left margin In centimeters.

Right Margin In centimeters.

Space Before Space left before a paragraph, in centimeters.

Space After Space left after a paragraph, in centimeters.

Tab Stops option sheet

The Tab Stops option sheet allows the user to create the tabs for a paragraph. There can be a maximum of 31 tabs. The user adds new tabs with the caret $\wedge$ gesture on this option sheet and deletes existing tabs with the cross out χ gesture. The user specifies tab stops in centimeters.

Display option sheet

The Display option sheet contains two controls that affect the display of the Mini-Text document.

Magnify Text on Screen Magnification of document display, in text points added to the current point size.

Show Special Characters Toggles display of tab, line break, and paragraph break characters.

Gestures

Non-core gestures used in MiniText

TABLE 51-1

Gesture	Name	Keyboard	Action
˙ʏ	Double tap		Selects a word. (Selects a bunsetu in Japanese.)
⁚ʏ	Triple tap		Selects a sentence.
⁘ʏ	Quadruple tap		Selects a paragraph.
‖	Double flick (four directions)	Ctrl + Home Ctrl + End	Up scrolls to end of text. *Down* scrolls to beginning of text. *Left* scrolls to right edge. *Right* scrolls to left edge.
≈	Scratch out	Delete	Deletes any character touched by gesture.
⊖	Circle line		Brings up empty editing pad to replace word or selection.
♀	Circle flick down		Searches towards the end of the text for the next occurrence of the word (or selection) under the gesture.
♁	Circle flick up		Searches towards the beginning of the text for the next occurrence of the word (or selection) under the gesture.
˙^	Caret tap		Creates an embedded insertion pad.
↑ ↓	Arrow up or *down*		Increases or *decreases* the point size for the word or selection by the increments in the Character option sheet.
┌	Up right	Space bar	Inserts a single character.
┘	Down left	Enter	Inserts a paragraph break.
╛	Down left flick		Inserts a line break.
╘	Down right flick	Tab	Inserts a tab.
┘‖	Right up flick		To upper case (romaji only).
┘	Right up		To initial caps (romaji only).
┐	Right down		To lower case (romaji only).
←	Arrow left		On selection, converts text to hankaku (Japanese only).[1]
→	Arrow right		On selection, converts text to zenkaku (Japanese only).[1]
B			Makes the word or selection bold.[1]
F			Brings up the Find sheet, set to search from the point of the gesture.
I			Italicizes the word or section.[1]
P			Proofs the word (English words only).
N			Makes the word or selection "normal"—that is, turns off bold, italic, and underlined[1] (does not turn off hankaku/zenkaku attribute).
S			Begins spell-checking words from the point of the gesture (English words only).
U			Underlines the word or selection.[1]
H			On selection, converts text to hankaku (Japanese only).[1]
Z			On selection, converts text to zenkaku (Japanese only).[1]

1. The selection works only on characters of the same type, for example all hanakaku. When there are mixed characters, only characters of the selected type are acted upon.

Menus

The menu structure of MiniText provides the dual command path for the gestures and option sheets.

Edit menu

Most of the Edit menu commands provide the standard functionality described in the *PenPoint User Interface Design Reference.* Those commands specific to MiniText are:

Undo Disabled when keyboard KKC is active.

Edit Opens an edit pad with the current selection in it.

Proof Opens a proof pad with current word and suggested alternates; only works on English words.

Spell Only works on English words.

Options menu

Character Opens Character option sheet ("Character option sheet" on page 536).

Paragraph Opens Paragraph option sheet ("Paragraph option sheet" on page 536).

Tab Stops Opens Tab Stops option sheet ("Tab Stops option sheet" on page 536).

Display Opens Display option sheet ("Display option sheet" on page 536).

Controls Opens Controls option sheet.

Access Opens Access option sheet.

Comments Opens Comments option sheet.

View menu

The View menu contains two checklists. Selections immediately update the view of the document.

Screen Format/Printer Format Toggles the view of the document between Screen Format, text wraps to the right edge of the current viewing region and Printer Format, text wraps to the right of the currently selected paper size.

Magnification Checklist of magnification factors. This is the same as the magnification found in the Display option sheet (see page 536). Magnification only applies when the Screen Format view has been selected, although control is always available.

Insert menu

Tab Inserts a tab at the insertion point.

Hankaku Space Inserts a hankaku space at the insertion point.

Zenkaku Space Inserts a zenkaku space at the insertion point.

Line Break Inserts a line break after the insertion point.

Paragraph Break Inserts a new paragraph at the insertion point.

Page Break Inserts a new page after the insertion point.

Pop-up Pad Opens a pop-up edit pad; contents go to insertion point.

Embedded Pad Opens an embedded edit pad at insertion point.

Signature Pad Places a signature (ink only) pad at insertion point.

Convert menu

The Convert menu provides functionality for converting text between hankaku and zenkaku.

To Hankaku Converts selected text to hankaku.

To Zenkaku Converts selected text to zenkaku.

To Uppercase Converts selected text to all uppercase characters (romaji only).

To Lowercase Converts selected text to all lowercase characters (romaji only).

Initial Caps Converts selected text to an initial capital letter at the start of each word (romaji only).

MiniNote

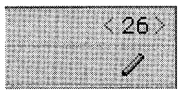
MiniNote is an ink processor implemented around PenPoint's ink component (**clsNotePaper**).

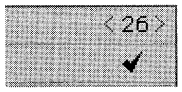
MiniNote has two input modes: ink mode and gesture mode. The default mode is ink mode. In ink mode, pen input is accepted as is and stored as ink, with the following exceptions:

- The double tap gesture over an ink object selects the object.
- The scratch out gesture deletes the object.
- The user can make gestures over selected objects.
- The tap press gesture over white space begins an area select.
- The flick left right gesture toggles between the two modes.
- The caret tap gesture for Date/Time menu.

Gesture mode allows the core gestures anywhere on the screen.

The screen is presented as a blank sheet of paper (lined in some predefined stationery) on which the user can begin writing. The menu line contains a mode toggle on the right side. The pen icon indicates ink mode. The check icon indicates gesture mode. The icons are a simple toggle switch that responds to a tap gesture.





Gesture margin

MiniNote has a gesture margin that facilitates line-oriented operations.

The margin also provides additional mode feedback: it is grey in ink mode and white in gesture mode.

Most of the gestures that are accepted in the body of the document can be made in the gesture margin, where they are interpreted as applying to the entire line. For example, double tap .ʸ selects the line, circle ○ edits the line, down left ┘ tidies the line, and tap press selects the line.

Other useful margin gestures include right down ─┐ and right up ─┘, to open and close white space by the amount of the vertical leg of the gesture. Left down and left up └─ are also accepted, and are more easily made if the gesture margin is on the right.

Option sheets

MiniNote has two unique Option sheets: Paper and Pen.

Paper option sheet

The Paper option sheet allows the user to specify the size and appearance of the paper upon which he or she is writing.

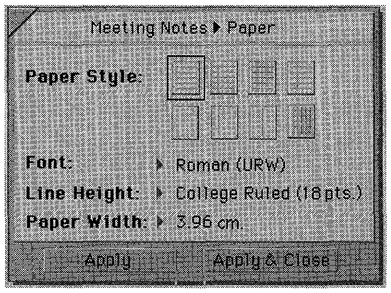
Paper Style Allows the user to specify the ruling of the paper from among the 8 choices available. This is a boxed choice list.

Font Allows the user to select the font for all the translated text in a Mini-Note document. The choices are the installed fonts in the system.

Line Height Contains two fixed line heights: College Ruled (18 pts.), and Standard (24 pts.), as well as a text field for entering an arbitrary height in points.

Paper Width Has an overwrite field containing the current paper width and two commands that affect the contents of the field. The overwrite field is scaled in centimeters.

- **Same as Document** The current width of the viewing region into the overwrite field so that a horizontal scroll margin is not needed.
- **Same as Print Settings** The current paper width (taken from the Paper Size control on the Print sheet) into the overwrite field.

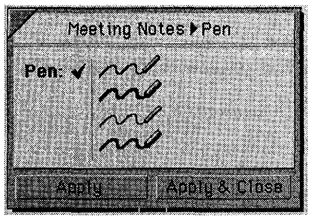

Meeting Notes ▸ Paper

Paper Style:

Font: ▸ Roman (URW)
Line Height: ▸ College Ruled (18pts.)
Paper Width: ▸ 3.96 cm.

Pen option sheet

The Pen option sheet controls the pen width and color. The choices presented in this sheet are the same as those for the Pen menu.



MiniNote gestures

Gestures that work in MiniNote

TABLE 51-2

Gestures	Name	Action
These gestures work in ink and gesture mode		
⁚	Double tap	Selects a single object.
+	Plus	Toggles selected state.
≈	Scratch out	Deletes any ink or text objects touched by the gesture.
·!	Tap press	Initiates an area select.
⇌	Flick left right	Toggles between ink and gesture modes.
⋀̇	Caret tap	Pops up insertion menu with current date and time.
ϑ	Pigtail	Deletes ink or text at the pen-down point.
✗	Undo	Reverses the effect of the most recent operation.
These gestures work in gesture mode or over the selection in ink mode		
·	Tap	Selects a single object.
⁚	Double tap	Selects a word or drawing.
⁝	Triple tap	Selects line.
+	Plus	Toggles selected state.
≈	Scratch out	Deletes any ink or text objects touched by the gesture.
!	Press	Initiates a move.
·!	Tap press	Initiates a copy.
⇌	Flick left right	Toggles between ink and gesture modes.
⋀̇	Caret tap	Pops up insertion menu with current date and time.
ϑ	Pigtail	Deletes ink or text at the pen-down point.
[ ]	Brackets	Adjust an existing selection.
X	Cross out	Deletes objects or selection.
∧	Caret	Pops up an insertion menu with current date and time.
○	Circle	Edits text, translates, and edits ink. All the selected ink is translated and appears in a pop-up edit pad.
⊙	Circle tap	Translates ink to text without displaying an edit pad.
✓	Check	Displays the option sheet for the selection or object.
┐┌	Right down Left down	Opens white space as determined by the length of the vertical leg.
┘└	Right up Left up	Closes space as determined by the length of the vertical leg.

Gestures that work in MiniNote TABLE 51-2 (continued)

Gestures	Name	Action
|	Flick (four directions)	Scrolls to edge.
||	Double flick (four directions)	Scrolls to beginning/end.
⊖	Circle line	Brings up an empty edit pad for the word.
⌟	Down left	Tidies selected lines (evens out the space between all objects).
⌞	Down right	Ungroups the selected scribbles, inserts space.
⌜	Up right	Right aligns the selected object.
⌝	Up left	Left aligns the selected object.
B		Makes the word or selection bold.
F		Brings up the Find sheet, set to start from the point of the gesture.
P		Proofs a word (English words only).
S		Begins spell-checking from point of gesture (English words only).
N		Makes the word or selection "normal"—turns off bold attribute.
U		Groups two adjacent scribbles. Start the gesture on one scribble and finish on the other.

Edit menu

Delete Deletes selected objects, closes gap.

Clear Deletes selected objects, leaves gap.

Insert Line Inserts a blank line above the selection.

Translate Translates selection.

Translate & Edit Translates selection and opens edit pad with translation (Reads Edit for already translated selection).

Options menu

Paper Opens Paper option sheet ("Paper option sheet" on page 540).

Pen Opens Pen option sheet ("Pen option sheet" on page 541).

Arrange menu

Tidy Moves selected objects to left to even gaps between objects.

Center Centers the selected objects.

Align Left Multiple line only. Shifts lines left to begin in same column as left-most line.

Align Right Multiple line only. Shifts lines right so that they all end in same column as right-most line.

Group Joins selected objects of like type (scribbles with scribbles, text with text) into a single object.

Ungroup Breaks a grouped object into constituent scribbles.

Pen menu

See "Pen option sheet" on page 541 for illustrations.

Edit pads

The **edit pad** (Input Pad) is one of the most universally used controls in PenPoint. It provides the optimized interface for precise and modal handwriting entry. Because of the intimate connection between edit pads and the language being input, edit pads have changed significantly to accommodate Japanese in PenPoint 2.0. This section describes both English language edit pads and Japanese language edit pads. In PenPoint 2.0 Japanese, the language of the text determines the input pad provided by the system. Edit pads are created with **clsIP**.

Operational model

The circle ○ gesture opens an edit pad. A circle opens an input pad for the object that was the target of the gesture. Edit pads are system modal. PenPoint selects the object being edited so that the user can see what he or she is operating on in context. A caret ∧ gesture provides a blank edit pad (often referred to as a writing pad).

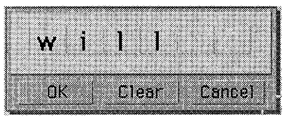
Change notes

Japanese edit pads introduced.

English edit pads

The English edit pads are the standard tool for entering and editing English text. In PenPoint 2.0 Japanese, they are used whenever romaji text is edited.

Standard elements



Edit pads come in three styles: boxed, ruled, and ruled/boxed. The boxed style is segmented into character spaces, and the ruled style is not. The ruled/boxed style is ruled when blank, and becomes boxed for all editing. Edit pads have three buttons: OK, Clear, and Cancel.

Input pads in version 2.0 are always in overwrite mode. Additionally, context checking is on during the first translation from ink to text.

The buttons for boxed pads work as follows:

OK Translates untranslated strokes or closes and accepts completely the translated text. If the pad is empty when the OK button is pressed, the pad is closed and nothing is entered into the text stream. If there are all untranslated strokes in the pad, then OK translates them. Subsequent strokes are translated automatically.

Clear Clears all text/ink from the writing area.

Cancel Closes pad without accepting any changes made.

In ruled pads, the OK button both translates the ink and puts the text into the text stream.

In ruled/boxed pads, the OK button presents a boxed pad for editing of the translated text.

Edit pad gestures

You can use these gestures in input pads:

Gestures used in edit pads

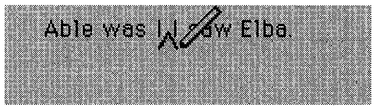
TABLE 51-3

Gesture	Name	Keyboard	Action
ꝗ	Pigtail	Delete or Backspace	Deletes single character.
⩧	Scratch out		Deletes every character it touches.
└	Down right	Space	Inserts spaces equal to the length of the right stroke.

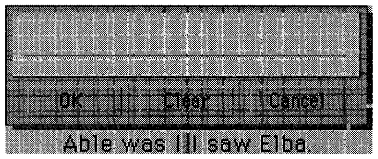
This example shows the typical steps you would go through to enter and translate text in an edit pad.

Translating text in edit pads

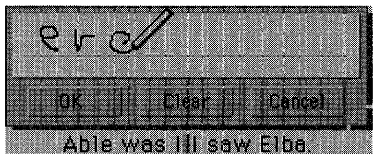
EXAMPLE 51-1



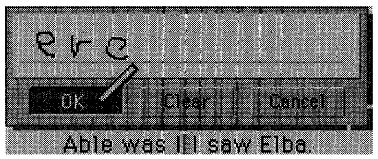
User makes caret gesture, with tip of caret targeted where new text is to be inserted.



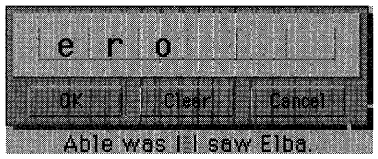
Blank ruled pad pops up.



User writes in the pad.



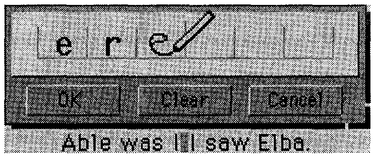
The user taps on OK to translate the text.



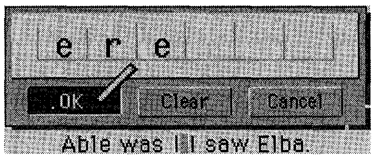
The translated text is presented in overwrite boxes for easy correction.

Translating text in edit pads

EXAMPLE 51-1 (continued)



User corrects characters as needed by overwriting.



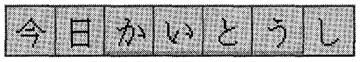
When all corrections have been made, user taps OK to insert contents of pad into text stream.

Japanese edit pads

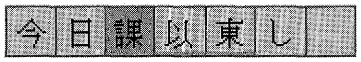
We designed the Japanese edit pads to make correction of near-miss translations easy and make kana-kanji conversion through both pen and keyboard easier and more accessible.

Standard elements

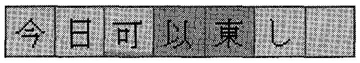
The Japanese input pad supports kana-kanji conversion (KKC) in a seamless and easy-to-use fashion. A new type of highlight, **weak highlight**, has been introduced to help support KKC. Weak highlight occurs after character entry by keyboard:

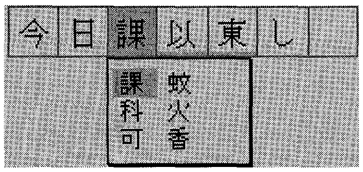


When a phrase (press spacebar once) has been selected, it gets a **strong highlight**:

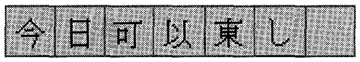


The user can press the spacebar again to get a **choice list** of other phrases to substitute. To select one, use the arrow keys to get to desired choice and use enter key to select:

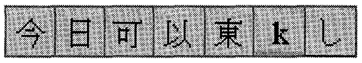




When the entire phrase has been explicitly accepted through a return, the text changes to the no highlight state, ready to accept further keyboard input:



If the user enters a new character from the keyboard, a **weak selection** point appears in the text box:

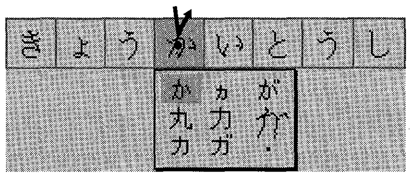


All forms of text entry devices used with the keyboard (boxed fields, write-in fields, and in-line text views) use these same states—weak highlight, strong highlight, weak selection.

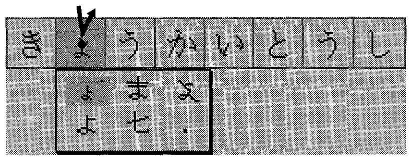
The Japanese input pad is larger than the English input pad to better accommodate the characters and the choice list.

Character alternative list

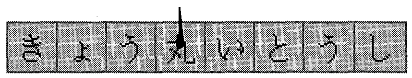
The character alternative list allows the user to select alternatives to the displayed character. This is important because of the occurrence of character "look-a-likes" and homophones among the character set.

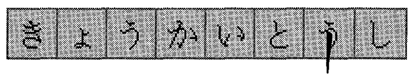


The list contains the known look-a-likes first, in stable order, and always contains a period and katakana middle dot as the last two entries. The choices are laid in an approximately square matrix with the current choice highlighted when the list is opened. The actual character that was written is not necessarily always the first choice in the list. The common endings are removed from the KKC alternatives. The user can access the choices on the list without opening the list through the use of gestures. The last two glyphs in the list are period and katakana middle dot, so that these characters can be accessed in overwrite mode through the list.



The user can use the arrows key to navigate through this list. The up/down arrow keys will move through the list, wrapping from column to column. The left/right arrow keys will wrap from row to row.





Unicode to character conversion

If the user knows the Unicode equivalent for a character, they may enter the 4-digit code in an input pad and convert it to the character with the right up ┘ gesture. If the code is also a valid Shift-JIS or ku-ten code, the converted characters will appear as alternatives in the choice list.

Japanese edit pad gestures

Gestures used in Japanese edit pads

TABLE 51-4

Gesture	Name	Keyboard	Action
Ṿ	Tap	Space (after conversion has taken place)	Open list of alternatives.
|	Flick	Up, down, left, right	Next choice. Previous choice (substitute in place).
┘	Right up	Space	Converts characters (also converts Unicode to character).
┘|	Right up flick		Alternate KKC.
┐	Right down		Reverse convert ("SaiHenkan").
←	Left arrow	Alt + left arrow	Shorten phrase.
→	Right arrow	Alt + right arrow	Extend phrase.

Menus

Change notes

Highlighting of menu items when selected item is black in 2.0 instead of grey as in 1.0.

Part 7 / Sample Code

This chapter lists the source code of the sample applications referred to in the preceding chapters of this book, and describes some of the other applications included with the PenPoint™ SDK. If you have installed the SDK, you'll find the sample code in subdirectories of PENPOINT\SDK\SAMPLE.

The following sample applications are described and their code is listed in this chapter.

The following sample applications are described in this part, but their code is not listed. The sample code is part of the SDK and is in 12-0\PENPOINT\SDK\SAMPLE.

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, Empty Application 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 Empty Application, you can create, open, float, zoom, close, rename, embed, and destroy Empty Application documents.

Objectives

Empty Application is used in the *PenPoint Application Writing Guide* to show how to compile, install, and run applications. This sample application also shows how to:

- Use **Debugf()** and **#ifdef** DEBUG and **#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

Files used

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

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

EMPTYAPP.C The application class's code and initialization.

METHODS.TBL

```
/****************************************************************************
 File: methods.tbl
 (C) Copyright 1992 by 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  $
  $Author:   aloomis  $
    $Date:   27 Jul 1992 10:59:38   $
 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
 (C) Copyright 1992 by 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  $
   $Author:   aloomis  $
     $Date:   16 Sep 1992 12:42:20  $
 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
#ifndef INTL_INCLUDED
#include <intl.h>               // for international routines
#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(U_L("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.
    //
    ObjCallWarn (msgNewDefaults, clsAppMgr, &new);
    new.object.uid              = clsEmptyApp;
    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;
    Ustrcpy(new.appMgr.company, U_L("GO Corporation"));
    Ustrcpy(new.appMgr.defaultDocName, U_L("Empty App Document"));
    ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);
    //
    // Turn on message tracing if flag is set.
    //
    if (DbgFlagGet('F', 0x1L)) {
        Debugf(U_L("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 (
    S32         argc,
    CHAR *      argv[],
    U32         processCount)
{
    Dbg(Debugf(U_L("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 */
```

Hello World (toolkit)

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:

1. Create a window and draw text in it using ImagePoint calls.
2. 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 approach for programs that need to do a lot of their own drawing, such as free-form 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**, the UI Toolkit label class, to display the words "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, that shows how to create a layout object (a window with knowledge of how to lay out 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 HELLOTK.C before compiling the application (see the README.TXT file in PENPOINT\SDK\SAMPLE\HELLOTK 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

Files used

The code for Hello World (toolkit) is in PENPOINT\SDK\SAMPLE\HELLOTK. The files are:

METHODS.TBL	the method table for **clsHelloWorld.**
HELLOTK.C	source code (actually a copy of either HELLOTK1.C or HELLOTK2.C) which the makefile compiles.
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

```
/****************************************************************************
 File: methods.tbl
 (C) Copyright 1992 by 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  $
  $Author:   aloomis  $
    $Date:   16 Sep 1992 12:51:12  $
 Methods.tbl contains the method table for clsHelloWorld (toolkit version).
****************************************************************************/
//
//  Include files
//
```

```
#ifndef APP_INCLUDED
#include <app.h>
#endif
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
 (C) Copyright 1992 by 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:   1.12  $
   $Author:   kcatlin  $
     $Date:   12 Aug 1992 17:49:16  $
 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.
 DEBUG FLAGS:
 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 LABEL_INCLUDED
#include <label.h>              // for label.
#endif
#ifndef FRAME_INCLUDED
#include <frame.h>              // for frame metrics.
#endif
#ifndef INTL_INCLUDED
#include <intl.h>               // for international routines.
#endif
#ifndef _STRING_H_INCLUDED
#include <string.h>             // for strcpy().
#endif
#include <methods.h>            // method function prototypes generated by MT
#define clsHelloWorld   wknGDTb // 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(U_L("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            = U_L("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(U_L("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(U_L("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.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;
    Ustrcpy(new.appMgr.company, U_L("GO Corporation"));
    ObjCallRet(msgNew, clsAppMgr, &new, s);
    if (DbgFlagGet('F', 0x20L)) {
        Dbg(Debugf(U_L("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 (
    S32         argc,
    CHAR *      argv[],
    U32         processCount)
{
    Dbg(Debugf(U_L("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
 (C) Copyright 1992 by 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:   1.13  $
   $Author:   kcatlin  $
     $Date:   12 Aug 1992 17:49:26  $
 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.  (The other version of hellotk.c does not use custom layout or
 create more than one toolkit window.)
 It does not have any state it needs to save.
 It does not have any instance data.
 DEBUG FLAGS:
 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 LABEL_INCLUDED
#include <label.h>              // for label.
#endif
#ifndef FRAME_INCLUDED
#include <frame.h>              // for frame metrics (and clayout.h)
#endif
#ifndef INTL_INCLUDED
#include <intl.h>               // for international routines.
#endif
#ifndef _STRING_H_INCLUDED
#include <string.h>             // for strcpy().
#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(U_L("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(U_L("HelloTK: Create the client Win"));)
    ObjCallWarn(msgNewDefaults, clsCustomLayout, &cn);
    // 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 = U_L("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(U_L("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.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;
    Ustrcpy(new.appMgr.company, U_L("GO Corporation"));
    ObjCallRet(msgNew, clsAppMgr, &new, s);
    if (DbgFlagGet('F', 0x20L)) {
        Dbg(Debugf(U_L("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 (
    S32         argc,
    CHAR *      argv[],
    U32         processCount)
{
    Dbg(Debugf(U_L("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 */
```

Hello World (custom window)

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:

1. Create a window and draw text in it using ImagePoint calls.
2. 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 free-form 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 (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 re-usable 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

Files used

The code for Hello World (custom window) is in PENPOINT\SDK\SAMPLE\HELLO. The files are:

HELTBL.TBL	the method table for the application class.
HELWTBL.TBL	the method table for the window class.
DLLINIT.C	the routine to initialize the DLL.
HELLO.C	the source code for the application.
HELLOWIN.C	the source code for the window class.
HELLOWIN.H	the header file for the window class.

HELTBL.TBL

```
/****************************************************************************
 File: heltbl.tbl
 (C) Copyright 1992 by 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:   1.8  $
  $Author:   aloomis  $
    $Date:   16 Sep 1992 12:44:50  $
 heltbl.tbl contains the method table for clsHelloWorld.
****************************************************************************/
//
//  Include files
```

```
//
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif
#ifndef APP_INCLUDED
#include <app.h>
#endif
MSG_INFO clsHelloWorldMethods [] = {
    msgAppOpen,                 "HelloOpen",       objCallAncestorAfter,
    msgAppClose,                "HelloClose",      objCallAncestorBefore,
    0
};
CLASS_INFO classInfo[] = {
    "clsHelloWorldTable",       clsHelloWorldMethods,   0,
    0
};
```

HELWTBL.TBL

```
/****************************************************************************
 File: helwtbl.tbl
 (C) Copyright 1992 by 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:   1.8  $
  $Author:   aloomis  $
    $Date:   16 Sep 1992 12:45:02  $
 helwtbl.tbl contains the method table for clsHelloWin.
****************************************************************************/
//
//  Include files
//
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif
#ifndef WIN_INCLUDED
#include <win.h>
#endif
MSG_INFO clsHelloWinMethods [] = {
    msgInit,                    "HelloWinInit",     objCallAncestorBefore,
    msgFree,                    "HelloWinFree",     objCallAncestorAfter,
    msgWinRepaint,              "HelloWinRepaint",  0,
    0
};
CLASS_INFO classInfo[] = {
    "clsHelloWinTable",         clsHelloWinMethods,     0,
    0
};
```

DLLINIT.C

```
/****************************************************************************
 File: dllinit.c
 (C) Copyright 1992 by 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:   1.9  $
   $Author:   kcatlin  $
     $Date:   12 Aug 1992 17:09:56  $
 This file contains the initialization routine for the Hello World dll.
****************************************************************************/
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif
#ifndef INTL_INCLUDED
#include <intl.h>
#endif
#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif
// The creation routines for each class in this dll.
STATUS ClsHelloWinInit  (void);
/****************************************************************************
    DLLMain

    Initialize DLL
****************************************************************************/
STATUS EXPORTED DLLMain (void)
{
    STATUS  s;
    Dbg(Debugf(U_L("Beginning hello.dll initialization."));)
    StsRet(ClsHelloWinInit(), s);
    Dbg(Debugf(U_L("Completed hello.dll initialization"));)
    return stsOK;
} /* DLLMain */
```

HELLO.C

```
/****************************************************************************
 File: hello.c
 (C) Copyright 1992 by 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:   1.11  $
   $Author:   kcatlin  $
     $Date:   12 Aug 1992 17:10:06  $
 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.
 DEBUG FLAGS:
 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
#ifndef INTL_INCLUDED
#include <intl.h>               // for frame metrics.
#endif
#ifndef HELLOWIN_INCLUDED
#include <hellowin.h>           // clsHelloWin's UID & msgNew args.
#endif
#ifndef _STRING_H_INCLUDED
#include <string.h>             // for strcpy().
#endif
#include <heltbl.h>             // method definitions
/****************************************************************************
 *                      Global variables and Defines                        *
 ****************************************************************************/
#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(U_L("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.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;
    Ustrcpy(new.appMgr.company, U_L("GO Corporation"));
    ObjCallRet(msgNew, clsAppMgr, &new, s);
    if (DbgFlagGet('F', 0x10L)) {
        Dbg(Debugf(U_L("Turning on message tracing for clsHelloWorld"));)
        (void)ObjCallWarn(msgTrace, clsHelloWorld, (P_ARGS) true);
    }
    return stsOK;
} /* ClsHelloInit */
/****************************************************************************
    main

    Main application entry point.
****************************************************************************/
void CDECL main (
    S32         argc,
    CHAR *      argv[],
    U32         processCount)
{
    Dbg(Debugf(U_L("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.C

```
/****************************************************************************
 File: hellowin.c
 (C) Copyright 1992 by 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:   1.14  $
   $Author:   aloomis  $
     $Date:   16 Sep 1992 12:45:12  $
 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.
****************************************************************************/
#ifndef DEBUG_INCLUDED
```

```
#include <debug.h>
#endif
#ifndef WIN_INCLUDED
#include <win.h>
#endif
#ifndef SYSGRAF_INCLUDED
#include <sysgraf.h>
#endif
#ifndef SYSFONT_INCLUDED
#include <sysfont.h>
#endif
#ifndef INTL_INCLUDED
#include <intl.h>
#endif
#include <helwtbl.h>            // method definitions
#ifndef HELLOWIN_INCLUDED
#include <hellowin.h>           // clsHelloWin's UID and msgNew args.
#endif
#ifndef OSHEAP_INCLUDED
#include <osheap.h>
#endif
#ifndef GOMATH_INCLUDED
#include <gomath.h>             // for scale calc. in fixed point.
#endif
#ifndef _STRING_H_INCLUDED
#include <string.h>             // for memset().
#endif
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                                    *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/****************************************************************************
    MESSAGE HANDLER: 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(data));
    // Create a dc.
    ObjCallRet(msgNewWithDefaults, clsSysDrwCtx, &dn, s);
    data.dc = dn.object.uid;
    // Rounded lines, thickness of zero.
    ObjectCall(msgDcSetLineThickness, data.dc, (P_ARGS)0);
    if (DbgFlagGet('F', 0x40L)) {
        Dbg(Debugf(U_L("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;
    // suppress compiler warnings about unused parameters
    MsgHandlerParametersNoWarning;
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(tx));
    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            = U_L("World");
    tx.lenText      = Ustrlen(tx.pText);
    ObjectCall(msgDcMeasureText, pData->dc, &tx);
    textWidth = tx.cp.x;
    tx.cp.x         = 0;
    tx.cp.y         = 0;
    tx.pText            = U_L("Hello");
    tx.lenText      = Ustrlen(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(U_L("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(U_L("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      = Ustrlen(tx.pText);
    ObjectCall(msgDcDrawText, pData->dc, &tx);
    // Display "World".
    tx.cp.x         = worldAdjust;
    tx.cp.y         = 0;
    tx.pText            = U_L("World");
    tx.lenText      = Ustrlen(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.ascenderPos / 2)) / 2;
    dotRect.origin.y = lineThickness / 2;
    dotRect.size.w = dotRect.size.h = fm.ascenderPos / 2;
    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(U_L("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)) {
        Dbg(Debugf(U_L("Turning on message tracing for clsHelloWin"));)
        (void)ObjCallWarn(msgTrace, clsHelloWin, (P_ARGS) true);
    }
    return stsOK;
} /* ClsHelloWinInit */
```

HELLOWIN.H

```
/****************************************************************************
 File: hellowin.h
 (C) Copyright 1992 by 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:   1.9  $
   $Author:   kcatlin  $
     $Date:   12 Aug 1992 17:10:30  $
 This file contains the API definition for clsHelloWin.
 clsHelloWin inherits from clsWin.
 It has no messages or msgNew arguments.
****************************************************************************/
#ifndef HELLOWIN_INCLUDED
#define HELLOWIN_INCLUDED
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif
#ifndef WIN_INCLUDED
#include <win.h>
#endif
/****************************************************************************
 *                      Global variables and Defines                        *
 ****************************************************************************/
#define clsHelloWin             MakeWKN(2165,1,wknGlobal)
/****************************************************************************
 *                      Common #defines and typedefs                        *
 ****************************************************************************/
#define helloWinNewFields   \
    winNewFields

typedef struct {
    helloWinNewFields
} HELLO_WIN_NEW, *P_HELLO_WIN_NEW;
#endif
```

Counter Application

Counter Application displays a number 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 the basis for many of the early labs in the PenPoint Programming Workshop. This sample application also shows how to:

- Save and restore application state.
- Memory-map state data.
- Separate text strings from your code and put them in separate resource files so that they can be translated into other languages without requiring recompilation.
- Define tags for strings used as resources.
- Retrieve strings from resource files using **ResUtilGetListString()**.
- Compose "international" strings using **SComposeText()**.
- Use strings contained in a resource file for menu buttons.

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

Files used

The code for Counter Application is in PENPOINT\SDK\SAMPLE\CNTRAPP. The files are:

METHODS.TBL	method tables for Counter Application.
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**.
JPN.RC	strings for the Japanese version (not listed here for typographical reasons).
USA.RC	strings for the USA version.

METHODS.TBL

```
/****************************************************************************
 File: methods.tbl
 (C) Copyright 1992 by 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:   1.7  $
   $Author:   aloomis  $
     $Date:   27 Jul 1992 10:49:36  $
 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.C

```
/****************************************************************************
 File: cntr.c
 (C) Copyright 1992 by 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:   1.9.1.0  $
   $Author:   aloomis  $
     $Date:   13 Nov 1992 12:00:40  $
 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 INTL_INCLUDED
#include <intl.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)
{
    Dbg(Debugf(U_L("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;
    Dbg(Debugf(U_L("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;

    Dbg(Debugf(U_L("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;

    Dbg(Debugf(U_L("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)
{
    Dbg(Debugf(U_L("Cntr:CntrFree"));)

    return stsOK;
    MsgHandlerParametersNoWarning;
} /* CntrFree */

/****************************************************************************
    CntrGetValue

    Respond to msgCntrGetValue.
****************************************************************************/
MsgHandlerWithTypes(CntrGetValue, P_CNTR_INFO, P_CNTR_INST)
{

    Dbg(Debugf(U_L("Cntr:CntrGetValue"));)

    pArgs->value = pData->currentValue;

    return stsOK;
    MsgHandlerParametersNoWarning;
} /* CntrGetValue */

/****************************************************************************
    CntrIncr

    Respond to msgCntrIncr.
****************************************************************************/
MsgHandlerWithTypes (CntrIncr, P_ARGS, P_CNTR_INST)
{
    CNTR_INST inst;

    Dbg(Debugf(U_L("Cntr:CntrIncr"));)

    inst = *pData;

    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 */
```

CNTR.H

```
/****************************************************************************
 File: cntr.h
 (C) Copyright 1992 by 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:   1.7  $
   $Author:   aloomis  $
     $Date:   27 Jul 1992 10:48:50  $
 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
```

CNTRAPP.C

```
/****************************************************************************
 File: cntrapp.c
 (C) Copyright 1992 by 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:   1.14.1.0  $
   $Author:   aloomis  $
     $Date:   13 Nov 1992 12:01:16  $
 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 CMPSTEXT_INCLUDED
#include <cmpstext.h>
#endif
#ifndef RESUTIL_INCLUDED
#include <resutil.h>
#endif
#ifndef CNTR_INCLUDED
#include <cntr.h>
#endif
#ifndef CNTRAPP_INCLUDED
#include <cntrapp.h>
#endif
#ifndef INTL_INCLUDED
#include <intl.h>
#endif
#ifndef BRIDGE_INCLUDED
#include <bridge.h>
#endif
#include <methods.h>
#include <string.h>
#include <stdio.h>
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                        Defines, Types, Globals, Etc                      *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
//
// You have to set a maximum size to map for a memory-mapped file, so pick
// something reasonable that's larger than the actual file size.
//
#define cntrAppMemoryMapSize    512
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;
/*
 * Here we use tags that are associated with strings in a resource file
 * for the name of our menu and the menu items.
 *
 * When using tags in a TKTable, the fifth field must be an id that gives
 * the type of the tag.  If there is an item already in the fifth field,
 * you can 'or' the two field items, and the system will know which one
 * to use.
 */
static const TK_TABLE_ENTRY CntrAppMenuBar[] = {
    {tagCntrMenu, 0, 0, 0, tkMenuPullDown | tkLabelStringId, clsMenuButton},
        {tagCntrDec, msgCntrAppChangeFormat, dec, 0, tkLabelStringId},
        {tagCntrOct, msgCntrAppChangeFormat, oct, 0, tkLabelStringId},
        {tagCntrHex, msgCntrAppChangeFormat, hex, 0, tkLabelStringId},
        {pNull},
    {pNull}
};

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                           Local Functions
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/****************************************************************************
 BuildString
 Local function to build a label string
****************************************************************************/
STATUS LOCAL BuildString(
    P_CHAR        p,
    P_CNTRAPP_INST pData)
{
    CNTR_INFO ci;
    STATUS    s;
    U32       size;
    CHAR      buffer[MINSTRLEN];
//  CHAR      resStr[MAXSTRLEN];
    ObjCallRet(msgCntrGetValue, pData->counter, &ci ,s);
    /*
     * Construct representation-dependent string for value of counter.
     */
    switch (*(pData->pFormat)) {
        case dec:
            Usprintf(buffer, U_L("%d"), ci.value);
            break;
        case oct:
            Usprintf(buffer, U_L("%o"), ci.value);
            break;
        case hex:
            Usprintf(buffer, U_L("%x"), ci.value);
            break;
        default:
            size = sizeof(p) / sizeof(CHAR);
            ResUtilGetListString(p, size, resGrpMisc, tagCntrUnknown);
            return stsOK;
            break;
    }
    /*
     * Retrieve format string from resource file, and construct display
     * string from format string and counter value.
     */
    size = MAXSTRLEN;
    SComposeTextL (&p, &size, pNull, resGrpMisc, tagCntrMessage, buffer);
    return stsOK;
} /* BuildString */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Message Handlers                                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/****************************************************************************
    CntrAppInit
    Respond to msgInit.
****************************************************************************/
MsgHandler(CntrAppInit)
{
    CNTRAPP_INST inst;
    Dbg(Debugf(U_L("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;
    Dbg(Debugf(U_L("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;
    Dbg(Debugf(U_L("CntrApp:CntrAppRestore"));)
    //
    // Get handle for format file, and save the handle.
    // The default for fsn.fs.locator.uid is theWorkingDir, which
    // is the document's directory.
    //
    ObjCallWarn(msgNewDefaults, clsFileHandle, &fsn);
    fsn.fs.locator.pPath = U_L("formatfile");
    ObjCallRet(msgNew, clsFileHandle, &fsn, s);
    inst.fileHandle = fsn.object.uid;
    //
    // Map the file to memory
    //
    ObjCallRet(msgFSMemoryMapSetSize, fsn.object.uid, \
                (P_ARGS)(SIZEOF)cntrAppMemoryMapSize, s);
    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;
    Dbg(Debugf(U_L("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.
****************************************************************************/
MsgHandlerWithTypes (CntrAppAppInit, P_ARGS, P_CNTRAPP_INST)
{
    CNTR_NEW                cn;
    FS_NEW                  fsn;
    STREAM_READ_WRITE       fsWrite;
    CNTRAPP_DISPLAY_FORMAT  format;
    CNTRAPP_INST           inst;
    STATUS                  s;
    Dbg(Debugf(U_L("CntrApp:CntrAppAppInit"));)
    inst = *pData;
    //
    // Create the counter object.
    //
    ObjCallWarn(msgNewDefaults, clsCntr, &cn);
    cn.cntr.initialValue = 42;
    ObjCallRet(msgNew, clsCntr, &cn, s);
    inst.counter = cn.object.uid;
    //
    // Create a file, fill it with a default value
    // The default for fsn.fs.locator.uid is theWorkingDir, which
    // is the document's directory.
    //
    ObjCallWarn(msgNewDefaults, clsFileHandle, &fsn);
    fsn.fs.locator.pPath = U_L("formatfile");
    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(msgFSMemoryMapSetSize, fsn.object.uid, \
                (P_ARGS)(SIZEOF)cntrAppMemoryMapSize, s);
    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[MAXSTRLEN];
    Dbg(Debugf(U_L("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.
    ObjCallWarn (msgNewDefaults, clsLabel, &ln);
    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;
    Dbg(Debugf(U_L("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[MAXSTRLEN];
    Dbg(Debugf(U_L("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);
#ifdef PP1_0
    strcpy(new.appMgr.defaultDocName, "Counter Application");
    strcpy(new.appMgr.company, "GO Corporation");
    new.appMgr.copyright = "1992 GO Corporation, All Rights Reserved";
#endif // PP1_0
    ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);
    return stsOK;
Error:
    return s;
} /* ClsCntrAppInit */
/****************************************************************************
    main
    Main application entry point.
****************************************************************************/
void CDECL
main(
    S32         argc,
    CHAR *      argv[],
    U32         processCount)
{
    if (processCount == 0) {
        StsWarn(ClsCntrAppInit());
        AppMonitorMain(clsCntrApp, objNull);
    } else {
        StsWarn(ClsCntrInit());
        AppMain();
    }
    Unused(argc); Unused(argv); // Suppress compiler's "unused parameter"
warnings
} /* main */
```

CNTRAPP.H

```
/****************************************************************************
 File: cntrapp.h

 (C) Copyright 1992 by 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:   1.7  $
   $Author:   aloomis  $
     $Date:   27 Jul 1992 10:49:02  $

 This file contains definitions for clsCntrApp.
****************************************************************************/
#ifndef CNTRAPP_INCLUDED
#define CNTRAPP_INCLUDED

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif
/****************************************************************************
 *                      Global Variables and Defines                        *
 ****************************************************************************/
// Define a well known UID for the app
#define clsCntrApp MakeWKN(555, 1, wknGlobal)

/****************************************************************************
 *                       Common Defines and Typedefs                        *
 ****************************************************************************/
/*
 * The RES_IDs for the resource lists used with the TAGs.
 */
#define resCntrTK           MakeListResId (clsCntrApp, resGrpTK, 0)
#define resCntrMisc         MakeListResId (clsCntrApp, resGrpMisc, 0)
/*
 * TAGs used to identify toolkit strings.
 */
#define tagCntrMenu         MakeTag (clsCntrApp, 0)
#define tagCntrDec          MakeTag (clsCntrApp, 1)
#define tagCntrOct          MakeTag (clsCntrApp, 2)
#define tagCntrHex          MakeTag (clsCntrApp, 3)
/*
 * TAGs used to identify miscellaneous CNTRAPP strings.
 */
#define tagCntrMessage      MakeTag (clsCntrApp, 4)
#define tagCntrUnknown      MakeTag (clsCntrApp, 5)
#define MAXSTRLEN           30
#define MINSTRLEN           5
/****************************************************************************
 *                        Messages for clsCntrApp                           *
 ****************************************************************************/
#define msgCntrAppChangeFormat MakeMsg(clsCntrApp,1)

#endif // CNTRAPP_INCLUDED
```

USA.RC

```
/****************************************************************************
 File: usa.rc
 (C) Copyright 1992 by 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:   1.7  $
 $Author:   kcatlin  $
 $Date:   17 Aug 1992 11:04:14  $
 usa.rc is the English language resource file for the Counter Application.
 In addition to the standard application resources, the Counter App also
 makes use of toolkit resources to store the string values for the
 Representation menu and the menu items, and miscellaneous resources to
 store the messages that are printed out in the drawing area.  The
 Counter App does not use Quick Help strings or Standard Message strings.
****************************************************************************/
#ifndef RESCMPLR_INCLUDED
#include <rescmplr.h>
#endif
#ifndef APPTAG_INCLUDED
#include <apptag.h>         // Resource ID & TAGs for app framework
#endif
```

```
#ifndef INTL_INCLUDED
#include <intl.h>
#endif
#ifndef BRIDGE_INCLUDED
#include <bridge.h>
#endif
#ifndef CNTRAPP_INCLUDED
#include "cntrapp.h"        // Resource IDs & TAGs for this project.
#endif
/****************************************************************************
                A P P   F r a m e w o r k   S t r i n g s
****************************************************************************/
static RC_TAGGED_STRING     appStrings[] = {
    // Default document name
    tagAppMgrAppDefaultDocName,     U_L("Counter Application"),
    // The company that produced the program.
    tagAppMgrAppCompany,            U_L("GO Corporation"),
    // The copyright string.
    tagAppMgrAppCopyright,
    U_L("\x00A9 Copyright 1992 by GO Corporation, All Rights Reserved."),
    // User-visible filename.  32 chars or less.
    tagAppMgrAppFilename,
    U_L("Counter Application"),
    // User-visible file type.  32 chars or less.
    tagAppMgrAppClassName,
    U_L("Application"),
    Nil(TAG)            // end of list marker
};
static RC_INPUT     app = {
    resAppMgrAppStrings,    // standard resource ID for APP strings
    appStrings,             // pointer to string array
    0,                      // data length; ignored for string arrays
    resTaggedStringArrayResAgent // How to interpret the data pointer
};
/****************************************************************************
                       T o o l k i t   S t r i n g s
****************************************************************************/
/*
 * Strings used by toolkit elements in CNTRAPP.  In this case, there are
 * only the Representation menu and its menu items.
 */
static RC_TAGGED_STRING     tkStrings[] = {
    // Representation menu
    tagCntrMenu,        U_L("Representation"),
    // Decimal menu item
    tagCntrDec,         U_L("Dec"),
    // Octal menu item
    tagCntrOct,         U_L("Oct"),
    // Hexagonal menu item
    tagCntrHex,         U_L("Hex"),
    Nil(TAG)
};
static RC_INPUT     tk = {
    resCntrTK,
    tkStrings,
    0,
    resTaggedStringArrayResAgent
};
/****************************************************************************
                      Q u i c k   H e l p   S t r i n g s
                                 (not used)
****************************************************************************/
/****************************************************************************
                 M i s c e l l a n e o u s   S t r i n g s
****************************************************************************/
static RC_TAGGED_STRING     miscStrings[] = {
    //
    // Message used to display counter value.  The '^1s' arguement allows
    // the code to fill in the appropriate value based on the user's menu
    // choice.
    //
    tagCntrMessage,     U_L("The counter value is: ^1s"),
    //
    // Message indicating an unknown representation type.
    //
    tagCntrUnknown,     U_L("Representation type unknown."),
    Nil(TAG)
};
static RC_INPUT     misc = {
    resCntrMisc,
    miscStrings,
    0,
    resTaggedStringArrayResAgent
};
/****************************************************************************
              S t a n d a r d   M e s s a g e   S t r i n g s
                                 (not used)
****************************************************************************/
/****************************************************************************
                     L i s t   o f   R e s o u r c e s
****************************************************************************/
P_RC_INPUT      resInput [] = {
    &app,               // the Application Framework strings
    &tk,                // the TK strings for CNTRAPP
    &misc,              // the Misc strings for CNTRAPP
    pNull               // End of list.
};
```

Tic-Tac-Toe

Tic-Tac-Toe displays a tic-tac-toe 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 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, move/copy, 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 (move, copy, delete, and undo, for example).
- Add application-specific menus.
- Add application-specific 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 application-specific document icons.
- Provide customized undo strings.
- Use **ClsSymbolsInit()**.
- Specify an application version number.

Class overview

Tic-Tac-Toe 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

Files used

The code for Tic-Tac-Toe is in PENPOINT\SDK\SAMPLE\TTT. The files are:

METHODS.TBL	the method tables for all of the Tic-Tac-Toe classes.
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	debugging-related message handlers.
TTTMBAR.C	menu bar-related message handlers.
TTTPRIV.H	private include file for Tic-Tac-Toe.

TTTUTIL.C utility functions.

TTTVIEW.C **clsTttView**'s code and initialization.

TTTVIEW.H header file for the view class.

TTTVOPT.C **clsTttView**'s option card-related message handlers.

TTTVXFER.C **clsTttView**'s data transfer-related message handlers.

S_TTT.C symbol name definitions and call to **ClsSymbolsInit**() (this file is generated automatically).

JPN.RC strings for the Japanese version (not listed here for typographical reasons).

USA.RC strings for the USA version.

FILLED.TXT stationery file (filled with Xs and Os).

RULES.TXT help file (containing the rules for the game).

STRAT.TXT help file (containing a strategy for playing the game).

XSONLY.TXT stationery file (partially filled, with Xs only).

METHODS.TBL

```
/****************************************************************************
 File: methods.tbl
 (C) Copyright 1992 by 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:   1.7  $
 $Author:   kcatlin  $
 $Date:   13 Jul 1992 10:31:50  $
 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,
    msgSelSelect,               "TttViewSelSelect",     objCallAncestorAfter,
    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,
    msgControlProvideEnable,    "TttAppProvideEnable",  0,
    0
};
```

```
CLASS_INFO classInfo[] = {
    "clsTttViewTable",  clsTttViewMethods,  0,
    "clsTttDataTable",  clsTttDataMethods,  0,
    "clsTttAppTable",   clsTttAppMethods,   0,
    0
};
```

TTTAPP.C

```
/****************************************************************************
 File: tttapp.c
 (C) Copyright 1992 by 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:   1.13  $
 $Author:   aloomis  $
 $Date:   16 Sep 1992 16:45:20  $
 This file contains the implementation of the application class.
****************************************************************************/

#ifndef APP_INCLUDED
#include <app.h>
#endif
#ifndef APPTAG_INCLUDED
#include <apptag.h>
#endif
#ifndef RESFILE_INCLUDED
#include <resfile.h>
#endif
#ifndef FRAME_INCLUDED
#include <frame.h>
#endif
#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif
#ifndef TTTVIEW_INCLUDED
#include <tttview.h>
#endif
#ifndef TTTAPP_INCLUDED
#include <tttapp.h>
#endif
#ifndef TTTDATA_INCLUDED
```

```
#include <tttdata.h>
#endif
#ifndef TTTPRIV_INCLUDED
#include <tttpriv.h>
#endif
#ifndef APPMGR_INCLUDED
#include <appmgr.h>
#endif
#ifndef OSHEAP_INCLUDED
#include <osheap.h>
#endif
#ifndef FS_INCLUDED
#include <fs.h>
#endif
#ifndef OPTION_INCLUDED
#include <option.h>
#endif
#ifndef INTL_INCLUDED
#include <intl.h>
#endif

#include <string.h>
#include <methods.h>
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Defines, Types, Globals, Etc                     *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
typedef struct TTT_APP_INST {
    U32 placeHolder;
} TTT_APP_INST,
  * P_TTT_APP_INST,
  * * PP_TTT_APP_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 TTT_APP_INST
TTT_APP_FILED_0, * P_TTT_APP_FILED_0;
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Utility Routines                                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/****************************************************************************
    TttAppFiledData0FromInstData
    Computes filed data from instance data.
****************************************************************************/
STATIC void PASCAL
TttAppFiledData0FromInstData(
    P_TTT_APP_INST      pInst,
    P_TTT_APP_FILED_0   pFiled)
{
    *pFiled = *pInst;
} /* TttAppFiledData0FromInstData */
/****************************************************************************
    TttAppInstDataFromFiledData0
    Computes instance data from filed data.
****************************************************************************/
STATIC void PASCAL
TttAppInstDataFromFiledData0(
    P_TTT_APP_FILED_0   pFiled,
    P_TTT_APP_INST      pInst)
{
    *pInst = *pFiled;
} /* TttAppInstDataFromFiledData0 */
/****************************************************************************
    TttAppCheckStationery
    The stationery file is deleted if and only if (1) no errors occur
    during this process and (2) the file is successfully read as a
    stationery file.
****************************************************************************/
#define DbgTttAppCheckStationery(x) \
    TttDbgHelper(U_L("TttAppCheckStationery"),tttAppDbgSet,0x1,x)
#define STATIONERY_FILE_NAME   U_L("tttstuff.txt")
STATIC STATUS PASCAL
TttAppCheckStationery(
    OBJECT          dataObject)
{
    FS_NEW          fNew;
    TTT_DATA_READ   dataRead;
    BOOLEAN         fileHandleCreated;
    BOOLEAN         deleteTheFile;
    STATUS          s;
    DbgTttAppCheckStationery((U_L("")))
    //
    // Initialize for error recovery and freeing resources.  Set return
    // values to something reasonable.
    //
    fNew.object.uid = objNull;
    fileHandleCreated = false;
    deleteTheFile = false;
    //
    // Look for the magic file.  If the file doesn't exist, we're done.
    //
    ObjCallJmp(msgNewDefaults, clsFileHandle, &fNew, s, Error);
    fNew.fs.mode = fsReadOnly;
    fNew.fs.exist = fsNoExistGenError | fsExistOpen;
    fNew.fs.locator.uid = theWorkingDir;
    fNew.fs.locator.pPath = STATIONERY_FILE_NAME;
    s = ObjectCall(msgNew, clsFileHandle, &fNew);
    if (s == stsFSNodeNotFound) {
        DbgTttAppCheckStationery((U_L("file not found; s=0x%lx"),s))
        goto NormalExit;
    } else if (s >= stsOK) {
        fileHandleCreated = true;
        DbgTttAppCheckStationery((U_L("file is found")))
    } else {
        DbgTttAppCheckStationery((U_L("Funny status when looking for file")))
        goto Error;
    }
    //
    // Ask the data object to read the file.  If the file is
    // successfully read as stationery, set up to delete the file.
    // If the file is not successfully read as stationery, simply continue.
    //
    dataRead.fileHandle = fNew.object.uid;
    s = ObjectCall(msgTttDataRead, dataObject, &dataRead);
    if ((s >= stsOK) AND (dataRead.successful)) {
        deleteTheFile = true;
    }
NormalExit:
    //
    // Be sure to close the file handle first;  otherwise the file
    // delete will fail.
    //
    if (fileHandleCreated) {
        ObjCallWarn(msgDestroy, fNew.object.uid, pNull);
    }
    //
    // Perhaps delete the file.  Have to make sure that the file
    // is not read-only before deleting it.  (Alternatively, I could use
    // msgForceDelete, but that's risky.)
    //
    if (deleteTheFile) {
        FS_GET_SET_ATTR     set;
        FS_ATTR_LABEL       label = fsAttrFlags;
        FS_NODE_FLAGS_ATTR  attrs;
        //
        // Turn off readOnly.  Don't bother error checking;  even
        // if something goes wrong, we'll go ahead and try to delete
        // the file, since it might not be readOnly anyhow.
        //
        attrs.mask = fsNodeReadOnly;
        attrs.flags = 0;
        set.pPath = STATIONERY_FILE_NAME;
        set.numAttrs = 1;
        set.pAttrLabels = &label;
        set.pAttrValues = &attrs;
        set.pAttrSizes = pNull;
        ObjCallWarn(msgFSSetAttr, theWorkingDir, &set);
        //
        // Delete the file.  Don't error check.  It would be unfortunate
        // if the file gets left lying around, but there's nothing we can
        // do about it anyhow.  And even if the file is left around,
        // this routine is only called once in an application's lifetime
        // and so there's no risk that we'll use the stationery file
        // instead of the proper file.
        //
        ObjCallWarn(msgFSDelete, theWorkingDir, STATIONERY_FILE_NAME);
        DbgTttAppCheckStationery((U_L("stationery file deleted")))
    }
    DbgTttAppCheckStationery((U_L("returns stsOK")))
    return stsOK;
Error:
    if (fNew.object.uid) {
        ObjCallWarn(msgDestroy, fNew.object.uid, pNull);
    }
    DbgTttAppCheckStationery((U_L("Error; returns 0x%lx"),s))
    return s;
} /* TttAppCheckStationery */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                            Message Handlers                              *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/****************************************************************************
    TttAppInit

    Initialize instance data of new object.

    Note: clsmgr has already initialized instance data to zeros.
****************************************************************************/
#define DbgTttAppInit(x) \
    TttDbgHelper(U_L("TttAppInit"),tttAppDbgSet,0x2,x)
MsgHandler(TttAppInit)
{
    P_TTT_APP_INST  pInst;
    STATUS          s;
    DbgTttAppInit((U_L("")))
    //
    // Initialize for error recovery.
    //
    pInst = pNull;
    //
    // Allocate, initialize, and record instance data.
    //
    StsJmp(OSHeapBlockAlloc(osProcessHeapId, SizeOf(*pInst), &pInst), \
            s, Error);
    pInst->placeHolder = -1L;
    ObjectWrite(self, ctx, &pInst);
    DbgTttAppInit((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;

Error:
    if (pInst) {
        OSHeapBlockFree(pInst);
    }
    DbgTttAppInit((U_L("Error; returns 0x%lx"),s))
    return s;
} /* TttAppInit */
/****************************************************************************
    TttAppFree

    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 DbgTttAppFree(x) \
    TttDbgHelper(U_L("TttAppFree"),tttAppDbgSet,0x4,x)
MsgHandlerWithTypes(TttAppFree, P_ARGS, PP_TTT_APP_INST)
{
    DbgTttAppFree((U_L("")))
    OSHeapBlockFree(*pData);
    DbgTttAppFree((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* TttAppFree */
/****************************************************************************
    TttAppSave

    Save self to a file.
****************************************************************************/
#define DbgTttAppSave(x) \
    TttDbgHelper(U_L("TttAppSave"),tttAppDbgSet,0x8,x)
MsgHandlerWithTypes(TttAppSave, P_OBJ_SAVE, PP_TTT_APP_INST)
{
    TTT_APP_FILED_0 filed;
    STATUS          s;
    DbgTttAppSave((U_L("")))
    StsJmp(TttUtilWriteVersion(pArgs->file, CURRENT_VERSION), s, Error);
    TttAppFiledData0FromInstData(*pData, &filed);
    StsJmp(TttUtilWrite(pArgs->file, SizeOf(filed), &filed), s, Error);
    DbgTttAppSave((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttAppSave((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttAppSave */
/****************************************************************************
    TttAppRestore

    Restore self from a file.
    Note: the app object has already received msgInit -- the App Framework
    has to create it to send it messages before sending it msgRestore.
    Thus the app object has already allocated its instance data in msgInit.
    This is unlike other objects which are typically recreated at msgRestore.
****************************************************************************/
#define DbgTttAppRestore(x) \
    TttDbgHelper(U_L("TttAppRestore"),tttAppDbgSet,0x10,x)
MsgHandlerWithTypes(TttAppRestore, P_OBJ_RESTORE, PP_TTT_APP_INST)
{
    TTT_APP_FILED_0 filed;
    STATUS              s;
    TTT_VERSION         version;
    DbgTttAppRestore((U_L("")))
    //
    // Read version, then read filed data.  (Currently there's only
    // only one legitimate file format, so no checking of the version
    // need be done.)
    //
    // Then convert filed data.
    //

    StsRet(TttUtilReadVersion(pArgs->file, MIN_VERSION, MAX_VERSION, \
            &version), s);
    StsRet(TttUtilRead(pArgs->file, SizeOf(filed), &filed), s);
    TttAppInstDataFromFiledData0(&filed, *pData);
    DbgTttAppRestore((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* TttAppRestore */
/****************************************************************************
    TttAppDump

    Respond to msgDump.
****************************************************************************/
#ifdef DEBUG

MsgHandlerWithTypes(TttAppDump, P_ARGS, PP_TTT_APP_INST)
```

```
{
    Debugf(U_L("TttAppDump: placeHolder=%ld"), (U32)((*pData)->placeHolder));
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* TttAppDump */
#endif // DEBUG
/****************************************************************************
    TttAppAppInit

    Respond to msgAppInit.  Perform one-time app life-cyle initializations.
****************************************************************************/
#define DbgTttAppAppInit(x) \
    TttDbgHelper(U_L("TttAppAppInit"),tttAppDbgSet,0x20,x)
MsgHandlerWithTypes(TttAppAppInit, P_ARGS, PP_TTT_APP_INST)
{
    APP_METRICS     am;
    TTT_VIEW_NEW    tttViewNew;
    BOOLEAN         responsibleForView;
    BOOLEAN         responsibleForScrollWin;
    OBJECT          dataObject;
    OBJECT          scrollWin;
    STATUS          s;
    DbgTttAppAppInit((U_L("")))
    //
    // Initialize for error recovery.
    //
    tttViewNew.object.uid = objNull;
    scrollWin = objNull;
    responsibleForView = false;
    responsibleForScrollWin = false;
    //
    // Create and initialize view.  This creates and initializes
    // data object as well.
    //
    ObjCallJmp(msgNewDefaults, clsTttView, &tttViewNew, s, Error);
    ObjCallJmp(msgNew, clsTttView, &tttViewNew, s, Error);
    responsibleForView = true;
    //
    // Check for stationery.
    //
    ObjCallJmp(msgViewGetDataObject, tttViewNew.object.uid, \
            &dataObject, s, Error);
    StsJmp(TttAppCheckStationery(dataObject), s, Error);
    //
    // Create and initialize scrollWin.
    //
    StsJmp(TttUtilCreateScrollWin(tttViewNew.object.uid, &scrollWin), \
            s, Error);
    responsibleForScrollWin = true;
    responsibleForView = false;
    //
    // Make the scrollWin be the frame's client win.
    //
    ObjCallJmp(msgAppGetMetrics, self, &am, s, Error);
    ObjCallJmp(msgFrameSetClientWin, am.mainWin, (P_ARGS)scrollWin, s, Error);
    responsibleForScrollWin = false;
    DbgTttAppAppInit((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    if (responsibleForView AND tttViewNew.object.uid) {
        ObjCallWarn(msgDestroy, tttViewNew.object.uid, pNull);
    }
    if (responsibleForScrollWin AND scrollWin) {
        ObjCallWarn(msgDestroy, scrollWin, pNull);
    }
    DbgTttAppAppInit((U_L("Error; returns 0x%lx"),s))
    return s;
} /* TttAppAppInit */
/****************************************************************************
    TttAppOpen

    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.
****************************************************************************/
#define DbgTttAppOpen(x) \
    TttDbgHelper(U_L("TttAppOpen"),tttAppDbgSet,0x40,x)
//
// Really a P_TK_TABLE_ENTRY
//
extern P_UNKNOWN tttMenuBar;
MsgHandlerWithTypes(TttAppOpen, P_ARGS, PP_TTT_APP_INST)
{
    APP_METRICS     am;
    OBJECT          menu;
    BOOLEAN         menuAdded;
    STATUS          s;
    DbgTttAppOpen((U_L("")))
    //
    // Initialize for error recovery.
    //
    menu = objNull;
    menuAdded = false;
    //
    // Get app and frame metrics.
    //
    ObjCallJmp(msgAppGetMetrics, self, &am, s, Error);
    //
    // Create and add menu bar.
    //
    StsJmp(TttUtilCreateMenu(am.mainWin, self, tttMenuBar, &menu), s, Error);
    DbgTttAppOpen((U_L("menu=0x%lx"),menu));
    ObjCallJmp(msgAppCreateMenuBar, self, &menu, s, Error);
    StsJmp(TttUtilAdjustMenu(menu), s, Error);
    ObjCallJmp(msgFrameSetMenuBar, am.mainWin, (P_ARGS)menu, s, Error);
    menuAdded = true;
    DbgTttAppOpen((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    if (menuAdded) {
        ObjCallWarn(msgFrameDestroyMenuBar, am.mainWin, pNull);
    }  else if (menu) {
        ObjCallWarn(msgDestroy, menu, pNull);
    }
    DbgTttAppOpen((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttAppOpen */

/****************************************************************************
    TttAppClose

    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.
****************************************************************************/
#define DbgTttAppClose(x) \
    TttDbgHelper(U_L("TttAppClose"),tttAppDbgSet,0x80,x)
MsgHandlerWithTypes(TttAppClose, P_ARGS, PP_TTT_APP_INST)
{
    APP_METRICS     am;
    STATUS          s;
    DbgTttAppClose((U_L("")))
    //
    // Get the frame.  Extract the menu bar from the frame.  Then
    // free the menu bar.
    //
    ObjCallJmp(msgAppGetMetrics, self, &am, s, Error);
    ObjCallJmp(msgFrameDestroyMenuBar, am.mainWin, pNull, s, Error);
    DbgTttAppClose((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttAppClose((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttAppClose */

/****************************************************************************
    TttAppSelectAll
****************************************************************************/
#define DbgTttAppSelectAll(x) \
    TttDbgHelper(U_L("TttAppSelectAll"),tttAppDbgSet,0x200,x)
MsgHandlerWithTypes(TttAppSelectAll, P_ARGS, PP_TTT_APP_INST)
{
    OBJECT  view;
    STATUS  s;
    DbgTttAppSelectAll((U_L("")))
    StsJmp(TttUtilGetComponents(self, tttGetView, pNull, &view, pNull),
            s, Error);
    ObjCallJmp(msgTttViewTakeSel, view, pNull, s, Error);
    DbgTttAppSelectAll((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttAppSelectAll((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttAppSelectAll */

/****************************************************************************
    TttAppProvideEnable

    Respond to msgControlProvideEnable.
****************************************************************************/
MsgHandlerWithTypes(TttAppProvideEnable, P_CONTROL_PROVIDE_ENABLE,
PP_TTT_APP_INST)
{
    switch (pArgs->tag) {
        case (tagAppMenuSelectAll):
            pArgs->enable = true;
            break;
        default:
            return ObjectCallAncestorCtx(ctx);
    }
    return stsOK;
    MsgHandlerParametersNoWarning;
}
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                           Installation                                  *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/****************************************************************************
    ClsTttAppInit

    Install the application.
****************************************************************************/
STATUS PASCAL
ClsTttAppInit (void)
{
    APP_MGR_NEW new;
    STATUS      s;
    ObjCallJmp(msgNewDefaults, clsAppMgr, &new, s, Error);
    new.object.uid                = clsTttApp;
    new.cls.pMsg                  = clsTttAppTable;
    new.cls.ancestor              = clsApp;
    new.cls.size                  = SizeOf(P_TTT_APP_INST);
    new.cls.newArgsSize           = SizeOf(APP_NEW);
    new.appMgr.flags.stationery   = true;
    new.appMgr.flags.accessory    = false;
#ifdef PP1_0            // manually copy in "About" information
    Ustrcpy(new.appMgr.defaultDocName, U_L("Tic-Tac-Toe"));
    Ustrcpy(new.appMgr.company, U_L("GO Corporation"));
    // 00A9 is the "circle-c" copyright symbol
    new.appMgr.copyright =
            U_L("\x00A9 1991-1992 GO Corporation, All Rights Reserved.");
#endif // PP1_0
    ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);
    return stsOK;
Error:
    return s;
} /* ClsTttAppInit  */
/****************************************************************************
    main

    Main application entry point.
****************************************************************************/
STATUS EXPORTED TttSymbolsInit(void);
void CDECL
main(
    S32         argc,
    CHAR *      argv[],
    U32         processCount)
{
    if (processCount == 0) {
        TttSymbolsInit();
        // Initialize global classes.
        StsWarn(ClsTttAppInit());
        AppMonitorMain(clsTttApp, objNull);
    }   else {
        // Initialize private classes
        StsWarn(ClsTttViewInit());
        StsWarn(ClsTttDataInit());
        AppMain();
    }
    // Suppress compiler's "unused parameter" warnings
    Unused(argc); Unused(argv);
} /* main */
```

TTTAPP.H

```
/****************************************************************************
 File: tttapp.h
 (C) Copyright 1992 by 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  $
 $Author:   kcatlin  $
 $Date:   13 Jul 1992 10:33:02  $
 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.C

```
/****************************************************************************
 File: tttdata.c
 (C) Copyright 1992 by 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  $
 $Author:   aloomis  $
 $Date:   16 Sep 1992 16:45:34  $
 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
#ifndef INTL_INCLUDED
#include <intl.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(U_L("TttDataNotifyObservers"),tttDataDbgSet,0x1,x)
STATIC STATUS PASCAL
TttDataNotifyObservers(
    OBJECT                  self,
    P_ARGS                  pArgs)
{
    OBJ_NOTIFY_OBSERVERS    nobs;
    STATUS                  s;
    DbgTttDataNotifyObservers((U_L("")))
    nobs.msg = msgTttDataChanged;
    nobs.pArgs = pArgs;
    nobs.lenSend = SizeOf(TTT_DATA_CHANGED);
    ObjCallJmp(msgNotifyObservers, self, &nobs, s, Error);
    DbgTttDataNotifyObservers((U_L("return stsOK")))
    return stsOK;
Error:
    DbgTttDataNotifyObservers((U_L("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(U_L("TttDataRecordStateForUndo"),tttDataDbgSet,0x2,x)
STATIC STATUS PASCAL
TttDataRecordStateForUndo(
    OBJECT              self,
    TAG                 undoTag,
    PP_TTT_DATA_INST    pData)
{
    UNDO_ITEM           item;
    STATUS              s;
    DbgTttDataRecordStateForUndo((U_L("")))
    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((U_L("return stsOK")))
    return stsOK;
Error:
    DbgTttDataRecordStateForUndo((U_L("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(U_L("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((U_L("")))
    //
    // Perhaps record undo information
    //
    if (recordUndo) {
        StsJmp(TttDataRecordStateForUndo(self, pArgs->undoTag, pData),
                s, Error);
    }
    //
    // Change data.
    //
    (*pData)->metrics = *pArgs;
    DbgTttDataPrivSetMetrics((U_L("returns stsOK")))
    return stsOK;
Error:
    DbgTttDataPrivSetMetrics((U_L("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(U_L("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((U_L("")))
    //
    // Perhaps record undo information
    //
    if (recordUndo) {
        StsJmp(TttDataRecordStateForUndo(self, (TAG)pNull, pData), s, Error);
    }
    //
    // Change data.
    //
    (*pData)->metrics.squares[pArgs->row][pArgs->col] = pArgs->value;

    DbgTttDataPrivSetSquare((U_L("returns stsOK")))
    return stsOK;
Error:
    DbgTttDataPrivSetSquare((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttDataPrivSetSquare */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Message Handlers                                *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/****************************************************************************
    TttDataNewDefaults

    Respond to msgNewDefaults.
****************************************************************************/
#define DbgTttDataNewDefaults(x) \
    TttDbgHelper(U_L("TttDataNewDefaults"),tttDataDbgSet,0x10,x)

MsgHandlerWithTypes(TttDataNewDefaults, P_TTT_DATA_NEW, PP_TTT_DATA_INST)
{
    U16 row;
    U16 col;

    DbgTttDataNewDefaults((U_L("")))
    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((U_L("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(U_L("TttDataInit"),tttDataDbgSet,0x20,x)

MsgHandlerWithTypes(TttDataInit, P_TTT_DATA_NEW, PP_TTT_DATA_INST)
{
    P_TTT_DATA_INST pInst;
    STATUS          s;

    DbgTttDataInit((U_L("")))
    //
    // 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((U_L("return stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    if (pInst) {
        OSHeapBlockFree(pInst);
    }
    DbgTttDataInit((U_L("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(U_L("TttDataFree"),tttDataDbgSet,0x40,x)
MsgHandlerWithTypes(TttDataFree, P_ARGS, PP_TTT_DATA_INST)
{
    STATUS  s;
    DbgTttDataFree((U_L("")))
    StsJmp(OSHeapBlockFree(*pData), s, Error);
    DbgTttDataFree((U_L("return stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttDataFree((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttDataFree */
/****************************************************************************
    TttDataSave

    Save self to a file.
****************************************************************************/
#define DbgTttDataSave(x) \
    TttDbgHelper(U_L("TttDataSave"),tttDataDbgSet,0x80,x)
MsgHandlerWithTypes(TttDataSave, P_OBJ_SAVE, PP_TTT_DATA_INST)
{
    TTT_DATA_FILED_0    filed;
    STATUS              s;

    DbgTttDataSave((U_L("")))
    StsJmp(TttUtilWriteVersion(pArgs->file, CURRENT_VERSION), s, Error);
    TttDataFiledData0FromInstData(*pData, &filed);
    StsJmp(TttUtilWrite(pArgs->file, SizeOf(filed), &filed), s, Error);
    DbgTttDataSave((U_L("return stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttDataSave((U_L("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(U_L("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((U_L("")))
    //
    // 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((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    if (pInst) {
        OSHeapBlockFree(pInst);
    }
    DbgTttDataRestore((U_L("Error; returns 0x%lx"),s))
    return s;
} /* TttDataRestore */
/****************************************************************************
    TttDataDump

    Respond to msgDump.
****************************************************************************/
#ifdef DEBUG
MsgHandlerWithTypes(TttDataDump, P_ARGS, PP_TTT_DATA_INST)
```

```
{
    Debugf(U_L("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(U_L("TttDataGetMetrics"),tttDataDbgSet,0x200,x)
MsgHandlerWithTypes(TttDataGetMetrics, P_TTT_DATA_METRICS, PP_TTT_DATA_INST)
{
    DbgTttDataGetMetrics((U_L("")))
    *pArgs = (*pData)->metrics;
    DbgTttDataGetMetrics((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* TttDataGetMetrics */
/****************************************************************************
    TttDataSetMetrics
****************************************************************************/
#define DbgTttDataSetMetrics(x) \
    TttDbgHelper(U_L("TttDataSetMetrics"),tttDataDbgSet,0x400,x)
MsgHandlerWithTypes(TttDataSetMetrics, P_TTT_DATA_METRICS, PP_TTT_DATA_INST)
{
    BOOLEAN transactionOpen;
    STATUS  s;
    DbgTttDataSetMetrics((U_L("")))
    // 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((U_L("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((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttDataSetMetrics */
/****************************************************************************
    TttDataSetSquare
    Handles both msgTttDataSetMetrics and msgTttDataSetSquare
****************************************************************************/
#define DbgTttDataSetSquare(x) \
    TttDbgHelper(U_L("TttDataSetSquare"),tttDataDbgSet,0x800,x)
MsgHandlerWithTypes(TttDataSetSquare, P_TTT_DATA_SET_SQUARE, PP_TTT_DATA_INST)
{
    TTT_DATA_CHANGED    changed;
    BOOLEAN             transactionOpen;
    STATUS              s;
    DbgTttDataSetSquare((U_L("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((U_L("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((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttDataSetSquare */

/****************************************************************************
    TttDataRead

    Handles msgTttDataRead, which is sent to read in a stationery file.
    Note that stationery files have 8-bit characters.
****************************************************************************/
#define DbgTttDataRead(x) \
    TttDbgHelper(U_L("TttDataRead"),tttDataDbgSet,0x1000,x)
#define N_CHARS 9
MsgHandlerWithTypes(TttDataRead, P_TTT_DATA_READ, PP_TTT_DATA_INST)
{
    STREAM_READ_WRITE   read;
    CHAR8               buf[N_CHARS+1];
    U16                 row;
    U16                 col;
    STATUS              s;
    DbgTttDataRead((U_L("")))
    //
    // Read in the 9 chars that must be present.  If there are fewer
    // than 9 chars, treat the attempt to read as a failure.
    //
    read.numBytes = sizeof(CHAR8) * N_CHARS;
    read.pBuf = buf;
    ObjCallJmp(msgStreamRead, pArgs->fileHandle, &read, s, Error);
    buf[N_CHARS] = U_L('\0');
    DbgTttDataRead((U_L("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++) {
            CHAR8       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((U_L("return stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttDataRead((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttDataRead */

/****************************************************************************
    TttDataUndoItem
****************************************************************************/
#define DbgTttDataUndoItem(x) \
    TttDbgHelper(U_L("TttDataUndoItem"),tttDataDbgSet,0x2000,x)
MsgHandlerWithTypes(TttDataUndoItem, P_UNDO_ITEM, PP_TTT_DATA_INST)
{
    STATUS  s;
    DbgTttDataUndoItem((U_L("")))

    if (pArgs->subclass != clsTttData) {
        DbgTttDataUndoItem((U_L("not clsTttData; give to ancestor")))
        return ObjectCallAncestorCtx(ctx);
    }
    StsJmp(TttDataPrivSetMetrics(self, pArgs->pData, pData, false),
            s, Error);
    StsJmp(TttDataNotifyObservers(self, pNull), s, Error);
    DbgTttDataUndoItem((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttDataUndoItem((U_L("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.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 */
```

TTTDATA.H

```
/****************************************************************************
 File: tttdata.h

 (C) Copyright 1992 by 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  $
 $Author:   kcatlin  $
 $Date:   13 Jul 1992 10:33:16  $
 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                                                          *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#define numberOfRows        5
#define numberOfColumns     5
//
// Tags used by for undo strings
//
#define     tagTttDataUndoDelete        MakeTag(clsTttData, 0)
#define     tagTttDataUndoMoveCopy      MakeTag(clsTttData, 1)
//
// The RES_ID for the resource list used with the TAGs.
//
#define resTttDataTK        MakeListResId(clsTttData, resGrpTK, 0)
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                            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)U_L(' '))
#define tttX     ((TTT_SQUARE_VALUE)U_L('X'))
#define tttO     ((TTT_SQUARE_VALUE)U_L('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
```

TTTDBG.C

```
/****************************************************************************
 File: tttdbg.c
 (C) Copyright 1992 by 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  $
 $Author:   kcatlin  $
 $Date:   28 Jul 1992 11:20:40  $
 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 INTL_INCLUDED
#include <intl.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(U_L("Setting tracing of app %s"),    turnTraceOn ? U_L("On") :
U_L("Off"));
        target = self;
    } else if (targetArg == 1) {
        Debugf(U_L("Setting tracing of view %s"), turnTraceOn ? U_L("On") :
U_L("Off"));
        target = view;
    } else {
       Debugf(U_L("Setting tracing of dataObject %s"), turnTraceOn ? U_L("On")
: U_L("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
```

TTTMBAR.C

```
/****************************************************************************
 File: tttmbar.c
 (C) Copyright 1992 by 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  $
 $Author:   kcatlin  $
 $Date:   28 Jul 1992 11:19:22  $
 This file contains some of the implementation of TttApp's menu bar.
****************************************************************************/
#ifndef TKTABLE_INCLUDED
#include <tktable.h>
#endif
#ifndef INTL_INCLUDED
#include <intl.h>
#endif
#ifndef TTTPRIV_INCLUDED
#include <tttpriv.h>
#endif
#ifndef TTTAPP_INCLUDED
#include <tttapp.h>
#endif
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Defines, Types, Globals, Etc                  *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#ifdef DEBUG
static const TK_TABLE_ENTRY dumpTreeMenu[] = {
  {U_L("From Frame"),   msgTttAppDumpWindowTree, 0},
  {U_L("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 const TK_TABLE_ENTRY traceMenu[] = {
  {U_L("Trace App On"),     msgTttAppChangeTracing, MakeU32(0,1)},
  {U_L("Trace App Off"),       msgTttAppChangeTracing, MakeU32(0,0)},
  {U_L("Trace View On"),       msgTttAppChangeTracing, MakeU32(1,1)},
  {U_L("Trace View Off"),   msgTttAppChangeTracing, MakeU32(1,0)},
  {U_L("Trace Data On"),       msgTttAppChangeTracing, MakeU32(2,1)},
  {U_L("Trace Data Off"),   msgTttAppChangeTracing, MakeU32(2,0)},
  {pNull}
};
static const TK_TABLE_ENTRY debugMenu[] = {
  {U_L("Dump View"),           msgTttAppDumpView,          0},
  {U_L("Dump Data"),           msgTttAppDumpDataObject,    0},
  {U_L("Dump App"),        msgDump,                   0},
  {U_L("Dump Window Tree"), (U32)dumpTreeMenu,         0, 0, tkMenuPullRight},
  {U_L("Trace"),               (U32)traceMenu,   0, 0, tkMenuPullRight |
tkBorderEdgeTop},
  {U_L("Force Repaint"),       msgTttAppForceRepaint,      0, 0,
tkBorderEdgeTop},
  {pNull}
};
#endif // DEBUG
static const TK_TABLE_ENTRY    tttRealMenuBar[] = {
#ifdef DEBUG
  {U_L("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;
```

TTTPRIV.H

```
/****************************************************************************
 File: tttpriv.h
 (C) Copyright 1992 by 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  $
 $Author:   ehoogerb  $
 $Date:   22 Oct 1992 16:46:54  $
 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(U_L("%s: "),str); Debugf
x;})
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Common Defines                                  *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Common Typedefs                                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
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

void PASCAL
TttUtilInitTextOutput(
    P_SYSDC_TEXT_OUTPUT p,
    U16                     align,
    P_CHAR                      buf);

P_CHAR 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.
//
// 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.  Thus, clsTttApp
// is a global well-known class; since neither clsTttData or clsTttView
// are referenced outside of this application, they are private classes.
//
// Of course, for any application (including those constructed from this
// template) that uses a global UID, you must contact GO Customer Services
// to receive a UID that is unique across all PenPoint applications.  When
// setting up a class as a private well-known, you can use any number you
// like as the UID, as long as the number is unique to the class for your
// application.
//
// Global WKN UIDs also allocated to tic-tac-toe, but not yet used:
//
//                          2223, 2224, 2225
//
#define clsTttApp           MakeGlobalWKN (2222,1)
#define clsTttData          MakePrivateWKN (1,1)
#define clsTttView          MakePrivateWKN (2,1)

//
// Debug flag sets
```

```
//
#define tttAppDbgSet          0xC0
#define tttDataDbgSet       0xC1
#define tttUtilDbgSet       0xC2
#define tttViewDbgSet       0xC3
#define tttViewOptsDbgSet   0xC4
#define tttViewXferDbgSet   0xC5
// tags for stationery and help file names
#define tagTttStationery1              MakeTag(clsTttApp, 0)
#define tagTttStationery2              MakeTag(clsTttApp, 1)
#define tagStrategyHelp                MakeTag(clsTttApp, 2)
#define tagRulesHelp                   MakeTag(clsTttApp, 3)
#endif  // TTTPRIV_INCLUDED
```

TTTUTIL.C

```
/****************************************************************************
 File: tttutil.c
 (C) Copyright 1992 by 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:   1.9  $
 $Author:   aloomis  $
 $Date:   16 Sep 1992 16:46:04   $
 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
#ifndef INTL_INCLUDED
#include <intl.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 const P_CHAR valueStrings[] = {
    U_L("_"), U_L("X"), U_L("O"), U_L("Unknown")};
P_CHAR 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(U_L("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((U_L("*pCount=%ld new=%ld=0x%lx"), (U32)(*pCount), \
            (U32)new, (U32)new))
    for (i=0; i<*pCount; i++) {
        if (pValues[i] == new) {
            DbgTttUtilInsertUnique((U_L("found it at %ld; returning"),(U32)i))
            return;
        }
    }
    pValues[*pCount] = new;
    *pCount = *pCount + 1;
    DbgTttUtilInsertUnique((U_L("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(U_L("TttUtilAdjustMenu"),tttUtilDbgSet,0x2,x)
//
// Tags which correspond to the menu items that this application
// will not implement.
//
static const TAG disableTags[] = {
    tagAppMenuSearch,
    tagAppMenuSpell};
#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((U_L("")))
    memset (parentMenus, 0, SizeOf(parentMenus));
    parentMenuCount = 0;
    for (i=0; i<N_DISABLE_TAGS; i++) {
        DbgTttUtilAdjustMenu((U_L("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((U_L("destroyed it; parent=0x%lx"),wm.parent))
        } else {
            DbgTttUtilAdjustMenu((U_L("didn't find tag!")))
        }
    }
    //
    // Adjust the breaks and re-layout each affected menu
    //
    for (i=0; i<parentMenuCount; i++) {
        DbgTttUtilAdjustMenu((U_L("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((U_L("returns stsOK")))
    return stsOK;
Error:
    DbgTttUtilAdjustMenu((U_L("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(U_L("TttUtilReadVersion"),tttUtilDbgSet,0x4,x)
STATUS PASCAL
TttUtilReadVersion(
    OBJECT              file,
    TTT_VERSION         minVersion,
    TTT_VERSION         maxVersion,
    P_TTT_VERSION       pVersion)
{
    STATUS              s;
    DbgTttUtilReadVersion((U_L("min=%ld max=%ld"),(U32)minVersion,
(U32)maxVersion))
    StsJmp(TttUtilRead(file, SizeOf(*pVersion), pVersion), s, Error);
    if ((*pVersion < minVersion) OR (*pVersion > maxVersion)) {
        DbgTttUtilReadVersion((U_L("version mismatch;
v=%ld"),(U32)(*pVersion)))
        s = stsIncompatibleVersion;
        goto Error;
    }
    DbgTttUtilReadVersion((U_L("version=%ld; return stsOK"),(U32)(*pVersion)))
    return stsOK;
Error:
    DbgTttUtilReadVersion((U_L("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 */

/****************************************************************************
    TttUtilInitTextOutput
****************************************************************************/
void PASCAL
TttUtilInitTextOutput(
    P_SYSDC_TEXT_OUTPUT p,
    U16                 align,
    P_CHAR              buf)
{
    memset(p, 0, SizeOf(*p));
    p->spaceChar = U_L(' ');
    p->stop = maxS32;
    p->alignChr = align;
    p->pText = buf;
    if (buf) {
        p->lenText = Ustrlen(buf);
    }
} /* TttUtilInitTextOutput */
```

TTTVIEW.C

```
/****************************************************************************
 File: tttview.c
 (C) Copyright 1992 by 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:   1.17  $
 $Author:   aloomis  $
 $Date:   23 Oct 1992 15:44:48  $
 This file contains the implementation of clsTttView.
****************************************************************************/
#ifndef APPTAG_INCLUDED
#include <apptag.h>
#endif
#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif
#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
#ifndef INTL_INCLUDED
#include <intl.h>
#endif
#ifndef BRIDGE_INCLUDED
#include <bridge.h>
#endif
#include <string.h>
#include <methods.h>

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          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(U_L("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((U_L("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((U_L("return stsOK")))
    return stsOK;
Error:
    if (dcNew.object.uid) {
        ObjCallWarn(msgDestroy, dcNew.object.uid, pNull);
    }
    DbgTttViewCreateDC((U_L("Error; returns 0x%lx"),s))
    return s;
} /* TttViewCreateDC */
/****************************************************************************
    TttViewGestureSetSquare

    Handles all gestures that set the value
    of a single square.
****************************************************************************/
#define DbgTttViewGestureSetSquare(x) \
    TttDbgHelper(U_L("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((U_L("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((U_L("return stsOK")))
    return stsOK;
Error:
    DbgTttViewGestureSetSquare((U_L("Error; returns 0x%lx"),s))
    return s;
} /* TttViewGestureSetSquare */
/****************************************************************************
    TttViewInitAndRestoreCommon

    Has code common to Init and Restore
****************************************************************************/
#define DbgTttViewInitAndRestoreCommon(x) \
    TttDbgHelper(U_L("TttViewInitAndRestoreCommon"),tttViewDbgSet,0x4,x)
STATIC STATUS PASCAL
TttViewInitAndRestoreCommon(
    VIEW            self,
    P_TTT_VIEW_INST pInst)
{
    STATUS          s;
    DbgTttViewInitAndRestoreCommon((U_L("self=0x%lx"),self))
    //
    // Initialize for Error Recovery
    //
    pInst->dc = objNull;
    //
    // Recreate the dc
    //
    StsJmp(TttViewCreateDC(self, &(pInst->dc)), s, Error);
    DbgTttViewInitAndRestoreCommon((U_L("return stsOK")))
    return stsOK;
Error:
    DbgTttViewInitAndRestoreCommon((U_L("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(U_L("TttViewSingleKey"),tttViewDbgSet,0x8,x)
STATIC STATUS PASCAL
TttViewSingleKey(
    OBJECT              self,
    CHAR                    keyCode)
{
    TTT_DATA_SET_SQUARE set;
    OBJECT              dataObject;
    BOOLEAN             inputIgnored;
    STATUS              s;
    DbgTttViewSingleKey((U_L("")))
    ObjCallJmp(msgViewGetDataObject, self, &dataObject, s, Error);
    inputIgnored = TRUE;
    if ((keyCode == U_L('x')) OR (keyCode == U_L('X'))) {
        inputIgnored = FALSE;
        set.value = tttX;
    } else if ((keyCode == U_L('o')) OR (keyCode == U_L('O'))) {
        inputIgnored = FALSE;
        set.value = tttO;
    } else if (keyCode == U_L(' ')) {
        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((U_L("return 0x%lx"),(U32)s))
    return s;
Error:
    DbgTttViewSingleKey((U_L("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(U_L("TttViewKeyInput"),tttViewDbgSet,0x10,x)
STATIC STATUS PASCAL
TttViewKeyInput(
    OBJECT          self,
    P_INPUT_EVENT   pArgs)
{
    STATUS          s;
    DbgTttViewKeyInput((U_L("self=0x%lx"),self))
    ASSERT((ClsNum(pArgs->devCode) == ClsNum(clsKey)), \
            U_L("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((U_L("return 0x%lx"),s))
    return s;

} /* TttViewKeyInput */
/****************************************************************************
    TttViewComputeSizes
****************************************************************************/
#define DbgTttViewComputeSizes(x) \
    TttDbgHelper(U_L("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((U_L("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((U_L("nBW=%ld nBH=%ld"),p->normalBoxWidth, p-
>normalBoxHeight))
    DbgTttViewComputeSizes((U_L("vert [%ld %ld %ld %ld]  [%ld %ld %ld %ld]"),
            p->vertLines[0], p->vertLines[1]));
    DbgTttViewComputeSizes((U_L("horiz [%ld %ld %ld %ld]  [%ld %ld %ld %ld]"),
            p->horizLines[0], p->horizLines[1]));
    DbgTttViewComputeSizes((
            U_L("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((
            U_L("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((
            U_L("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(U_L("TttViewNewDefaults"),tttViewDbgSet,0x80,x)
MsgHandlerWithTypes(TttViewNewDefaults, P_TTT_VIEW_NEW, PP_TTT_VIEW_INST)
{
    DbgTttViewNewDefaults((U_L("self=0x%lx"),self))
    pArgs->win.flags.input |= inputHoldTimeout;
    pArgs->gWin.helpId = tagTttView;
    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((U_L("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(U_L("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((U_L("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((U_L("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((U_L("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(U_L("TttViewFree"),tttViewDbgSet,0x200,x)
MsgHandlerWithTypes(TttViewFree, P_ARGS, PP_TTT_VIEW_INST)
{
    OBJECT  dataObject;
    DbgTttViewFree((U_L("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((U_L("return stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* TttViewFree */

/****************************************************************************
    TttViewSave

    Save self to a file.
****************************************************************************/
#define DbgTttViewSave(x) \
    TttDbgHelper(U_L("TttViewSave"),tttViewDbgSet,0x400,x)
MsgHandlerWithTypes(TttViewSave, P_OBJ_SAVE, PP_TTT_VIEW_INST)
{
    TTT_VIEW_FILED_0    filed;
    STATUS              s;
    DbgTttViewSave((U_L("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((U_L("return stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewSave((U_L("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(U_L("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((U_L("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((U_L("return stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    if (pInst) {
        OSHeapBlockFree(pInst);
    }
    DbgTttViewRestore((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttViewRestore */

/****************************************************************************
    TttViewDump

    Respond to msgDump.
****************************************************************************/
#ifdef DEBUG
MsgHandlerWithTypes(TttViewDump, P_ARGS, PP_TTT_VIEW_INST)
{
    Debugf(U_L("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(U_L("TttViewDataChanged"),tttViewDbgSet,0x1000,x)
MsgHandlerWithTypes(TttViewDataChanged, P_TTT_DATA_CHANGED, PP_TTT_VIEW_INST)
{
    STATUS s;
    DbgTttViewDataChanged((U_L("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((U_L("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((U_L("return stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewDataChanged((U_L("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(U_L("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;
    STATUS              s;
    BOOLEAN             drawIt;
    DbgTttViewRepaint((U_L("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.
//
s = ObjectCall(msgSelIsSelected, self, pNull);
if (s == stsOK) {
    DbgTttViewRepaint((U_L("self is selected")))
    ObjectCall(msgDcSetBackgroundRGB, (*pData)->dc, \
            (P_ARGS)sysDcRGBGray33);
} else {
    DbgTttViewRepaint((U_L("self is not selected")))
    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((U_L("vertical i=%ld; overlap"),(U32)i))
        ObjectCall(msgDcDrawRectangle, (*pData)->dc, \
                &(sizes.vertLines[i]));
    } else {
        DbgTttViewRepaint((U_L("vertical i=%ld; no overlap"),(U32)i))
    }
}
//
// Paint the horizontal lines
//
for (i=0; i<2; i++) {
    if (Rect32sIntersect(&dirtyRect, &(sizes.horizLines[i]))) {
        DbgTttViewRepaint((U_L("horizontal i=%ld; overlap"),(U32)i))
        ObjectCall(msgDcDrawRectangle, (*pData)->dc, \
                &(sizes.horizLines[i]));
    } else {
        DbgTttViewRepaint((U_L("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 = U_L("X");
tx.lenText = Ustrlen(tx.pText);
ObjectCall(msgDcMeasureText, (*pData)->dc, (P_ARGS)&tx);
sizeX = tx.cp;
DbgTttViewRepaint((U_L("measure X=[%ld %ld]"),sizeX.x, sizeX.y))
TttUtilInitTextOutput(&tx, sysDcAlignChrTop, pNull);
tx.pText = U_L("O");
tx.lenText = Ustrlen(tx.pText);
ObjectCall(msgDcMeasureText, (*pData)->dc, (P_ARGS)&tx);
sizeO = tx.cp;
DbgTttViewRepaint((U_L("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((U_L("row=%ld col=%ld; overlap"), \
                    (U32)row, (U32)col));
            if (dm.squares[row][col] == tttX) {
                drawIt = TRUE;
                tx.pText = U_L("X");
                tx.lenText = Ustrlen(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 = U_L("O");
                tx.lenText = Ustrlen(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((U_L("blank cell")))
                drawIt = FALSE;
            }
            if (drawIt) {
                ObjCallJmp(msgDcDrawText, (*pData)->dc, &tx, s, Error);
            }
        } else {
            DbgTttViewRepaint((U_L("row=%ld col=%ld; no overlap"), \
                    (U32)row, (U32)col));
        }
    }
  }
  //
  // Balance the msgWinBeginRepaint
  //
  ObjCallWarn(msgWinEndRepaint, (*pData)->dc, pNull);
  DbgTttViewRepaint((U_L("return stsOK")))
  return stsOK;
  MsgHandlerParametersNoWarning;
Error:
  if (endRepaintNeeded) {
      ObjCallWarn(msgWinEndRepaint, (*pData)->dc, pNull);
  }
  DbgTttViewRepaint((U_L("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(U_L("TttViewGetDesiredSize"),tttViewDbgSet,0x2000,x)
MsgHandlerArgType(TttViewGetDesiredSize, P_WIN_METRICS)
{
  pArgs->bounds.size.w    = desiredWidth;
  pArgs->bounds.size.h    = desiredHeight;
  DbgTttViewGetDesiredSize((U_L("return stsOK")))
  return stsOK;
  MsgHandlerParametersNoWarning;
} /* TttViewGetDesiredSize */
```

```
/****************************************************************************
  TttViewGesture

  Let ancestor handle unrecognized gestures.
****************************************************************************/
#define DbgTttViewGesture(x) \
  TttDbgHelper(U_L("TttViewGesture"),tttViewDbgSet,0x8000,x)
MsgHandlerWithTypes(TttViewGesture, P_GWIN_GESTURE, PP_TTT_VIEW_INST)
{
  STATUS      s;
//  OBJECT      owner;
#ifdef DEBUG
  {
      P_CLS_SYMBUF    mb;
      #ifdef PP1_0
      DbgTttViewGesture((U_L("self=0x%lx msg=0x%lx %s"), self, pArgs->msg,
              ClsMsgToString(pArgs->msg,mb)))
      #else
      DbgTttViewGesture((U_L("self=0x%lx msg=0x%lx %s"), self, pArgs-
>gesture,
              ClsMsgToString(pArgs->gesture,mb)))
      #endif
  }
#endif // DEBUG
#ifdef PP1_0
  switch(pArgs->msg) {
#else
  switch(pArgs->gesture) {
#endif

      case xgs1Tap:
          ObjCallJmp(msgTttViewToggleSel, self, pNull, s, Error);
          break;
      case xgsCross:
          StsJmp(TttViewGestureSetSquare(self, pArgs, tttX), s, Error);
          break;
      case xgsCircle:
          StsJmp(TttViewGestureSetSquare(self, pArgs, tttO), s, Error);
          break;
      case xgsPigtailVert:
          StsJmp(TttViewGestureSetSquare(self, pArgs, tttBlank), \
                  s, Error);
          break;
      case xgsCheck:
      case xgsUGesture:
          // Make sure there is a selection.
          s = ObjectCall(msgSelIsSelected, self, pNull);
          if (s == stsNoMatch) {
              ObjCallJmp(msgTttViewTakeSel, self, pNull, s, Error);
              ObjCallJmp(msgWinUpdate, self, pNull, s, Error);
```

```
            }
            // Then call the ancestor.
            ObjCallAncestorCtxJmp(ctx, s, Error);
            break;
        default:
            DbgTttViewGesture((U_L("Letting ancestor handle gesture")))
            return ObjCallAncestorCtxWarn(ctx);
    }
    DbgTttViewGesture((U_L("return stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewGesture((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttViewGesture */
/****************************************************************************
    TttViewSelYield

     msgSelYield from selection manager.
****************************************************************************/
#define DbgTttViewSelYield(x) \
    TttDbgHelper(U_L("TttViewSelYield"),tttViewDbgSet,0x10000,x)
MsgHandlerWithTypes(TttViewSelYield, P_ARGS, PP_TTT_VIEW_INST)
{
    STATUS      s;
    DbgTttViewSelYield((U_L("self=0x%lx"),self))
    StsJmp(TttViewNeedRepaint(self), s, Error);
    DbgTttViewSelYield((U_L("return stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewSelYield((U_L("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(U_L("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((U_L("")))
    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((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewSelDelete((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttViewSelDelete */
/****************************************************************************
    TttViewGetMetrics
****************************************************************************/
#define DbgTttViewGetMetrics(x) \
    TttDbgHelper(U_L("TttViewGetMetrics"),tttViewDbgSet,0x100000,x)
MsgHandlerWithTypes(TttViewGetMetrics, P_TTT_VIEW_METRICS, PP_TTT_VIEW_INST)
{
    DbgTttViewGetMetrics((U_L("self=0x%lx"),self))
    *pArgs = (*pData)->metrics;
    DbgTttViewGetMetrics((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* TttViewGetMetrics */
/****************************************************************************
    TttViewSetMetrics
****************************************************************************/
#define DbgTttViewSetMetrics(x) \
    TttDbgHelper(U_L("TttViewSetMetrics"),tttViewDbgSet,0x200000,x)
MsgHandlerWithTypes(TttViewSetMetrics, P_TTT_VIEW_METRICS, PP_TTT_VIEW_INST)
{
    DbgTttViewSetMetrics((U_L("self=0x%lx"),self))
    (*pData)->metrics = *pArgs;
    TttViewNeedRepaint(self);
    DbgTttViewSetMetrics((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
} /* TttViewSetMetrics */
/****************************************************************************
    TttViewToggleSel
     msgTttViewToggleSel
```

```
****************************************************************************/
#define DbgTttViewToggleSel(x) \
    TttDbgHelper(U_L("TttViewToggleSel"),tttViewDbgSet,0x400000,x)
MsgHandlerWithTypes(TttViewToggleSel, P_ARGS, PP_TTT_VIEW_INST)
{
    STATUS      s;
    DbgTttViewToggleSel((U_L("self=0x%lx"),self))
    s = ObjectCall(msgSelIsSelected, self, pNull);
    if (s == stsOK) {
        DbgTttViewToggleSel((U_L("View is selected; deselect it")))
        ObjCallJmp(msgSelSetOwner, theSelectionManager, pNull, s, Error);
        if (self == InputGetTarget()) {
            StsJmp(InputSetTarget(objNull, inputAllRealEventsFlags), s, Error);
        }
    } else {
        DbgTttViewToggleSel((U_L("View is not selected; select it")))
        ObjCallJmp(msgSelSelect, self, pNull, s, Error);
    }
    StsJmp(TttViewNeedRepaint(self), s, Error);
    DbgTttViewToggleSel((U_L("return stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewToggleSel((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttViewToggleSel */
/****************************************************************************
    TttViewTakeSel
****************************************************************************/
#define DbgTttViewTakeSel(x) \
    TttDbgHelper(U_L("TttViewTakeSel"),tttViewDbgSet,0x800000,x)
MsgHandlerWithTypes(TttViewTakeSel, P_ARGS, PP_TTT_VIEW_INST)
{
    STATUS  s;
    DbgTttViewTakeSel((U_L("self=0x%lx"),self))
    s = ObjectCall(msgSelIsSelected, self, pNull);
    if (s == stsNoMatch) {
        DbgTttViewTakeSel((U_L("self is not selected; taking")))
        ObjCallJmp(msgSelSelect, self, pNull, s, Error);
        StsJmp(TttViewNeedRepaint(self), s, Error);
    } else {
        DbgTttViewTakeSel((U_L("self is already selected; doing nothing")))
    }
    DbgTttViewTakeSel((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewTakeSel((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttViewTakeSel */
/****************************************************************************
    TttViewInputEvent
     msgInputEvent.
****************************************************************************/
#define DbgTttViewInputEvent(x) \
    TttDbgHelper(U_L("TttViewInputEvent"),tttViewDbgSet,0x1000000,x)
MsgHandlerWithTypes(TttViewInputEvent, P_INPUT_EVENT, PP_TTT_VIEW_INST)
{
    STATUS      s;
    switch (ClsNum(pArgs->devCode)) {
        case ClsNum(clsKey):
            s = TttViewKeyInput(self, pArgs);
            break;
        default:
            s = ObjectCallAncestorCtx(ctx);
            break;
    }
    return s;
    MsgHandlerParametersNoWarning;
} /* TttViewInputEvent */
/****************************************************************************
    TttViewSelSelect
     msgSelSelect.
****************************************************************************/
MsgHandler(TttViewSelSelect)
{
    STATUS  s;
    //
    // If the view is not selected, force it to repaint.
    // (This code is needed for the move/copy protocol; otherwise,
    // tapping or press-tapping on a ttt board does not highlight
    // the board's selection properly).
    //

    s = ObjectCall(msgSelIsSelected, self, pNull);

    if (s == stsNoMatch) {
        StsWarn(TttViewNeedRepaint(self));
    }
    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.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 */
```

TTTVIEW.H

```
/****************************************************************************
 File: tttview.h
 (C) Copyright 1992 by 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:   1.7  $
 $Author:   kcatlin  $
 $Date:   13 Jul 1992 10:31:36  $
 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 and its option sheet.
//
#define tagTttView                MakeTag(clsTttView, 0)
#define tagTttViewCard            MakeTag(clsTttView, 1)
#define tagCardTitle                  MakeTag(clsTttView, 2)
#define tagCardLineThickness      MakeTag(clsTttView, 3)
//
// The RES_IDs for the resource lists used with the TAGs.
//
#define resTttViewQHelp     MakeListResId(clsTttView, resGrpQhelp, 0)
#define resTttViewTK            MakeListResId(clsTttView, resGrpTK, 0)

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                         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
```

TTTVOPT.C

```
/****************************************************************************
 File: tttvopt.c
 (C) Copyright 1992 by 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  $
 $Author:   aloomis  $
 $Date:   20 Aug 1992 20:33:10  $
 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
#ifndef INTL_INCLUDED
#include <intl.h>
#endif
#ifndef BRIDGE_INCLUDED
#include <bridge.h>
#endif
#ifndef RESUTIL_INCLUDED
#include <resutil.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.
// Note that the two controls (a label and an integer field) share the
// same quick help id.
//
static const TK_TABLE_ENTRY cardEntries[] = {
    {tagCardLineThickness, 0, 0, 0, tkLabelStringId, 0, tagCardLineThickness},
    {U_L("1"), 1, 1, tagCardLineThickness, 0, clsIntegerField,
        tagCardLineThickness},
    {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(U_L("TttViewOptionAddCards"),tttViewOptsDbgSet,0x4,x)
MsgHandlerWithTypes(TttViewOptionAddCards, P_OPTION_TAG, PP_TTT_VIEW_INST)
{
    OPTION_CARD         card;
    STATUS              s;
    DbgTttViewOptionAddCards((U_L("")))
    //
    // Create the card.
    //
    card.tag = tagTttViewCard;
    card.win = objNull;
    card.pName = ResUtilAllocListString(osProcessHeapId, resGrpTK,
                        tagCardTitle);
    card.client = self;
    ObjCallJmp(msgOptionAddLastCard, pArgs->option, &card, s, Error);
    DbgTttViewOptionAddCards((U_L("return stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewOptionAddCards((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttViewOptionAddCards */
/****************************************************************************
    TttViewOptionProvideCard

    Handles msgOptionProvideCardWin.
****************************************************************************/
#define DbgTttViewOptionProvideCard(x) \
    TttDbgHelper(U_L("TttViewOptionProvideCard"),tttViewOptsDbgSet,0x8,x)
```

```
MsgHandlerWithTypes(TttViewOptionProvideCard, P_OPTION_CARD, P_UNKNOWN)
{
    STATUS              s;
    OPTION_TABLE_NEW    otn;
    DbgTttViewOptionProvideCard((U_L("")))
    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((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttViewOptionProvideCard */
/****************************************************************************
    TttViewOptionRefreshCard

    Handles msgOptionRefreshCard
****************************************************************************/
#define DbgTttViewOptionRefreshCard(x) \
    TttDbgHelper(U_L("TttViewOptionRefreshCard"),tttViewOptsDbgSet,0x10,x)
MsgHandlerWithTypes(TttViewOptionRefreshCard, P_OPTION_CARD, PP_TTT_VIEW_INST)
{
    OBJECT              view;
    TTT_VIEW_METRICS    vm;
    OBJECT              control;
    STATUS              s;
    DbgTttViewOptionRefreshCard((U_L("")))
    //
    // See note [1] at the beginning of this file.
    //
    if (pArgs->tag != tagTttViewCard) {
        DbgTttViewOptionRefreshCard((U_L("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((U_L("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((U_L("return stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewOptionRefreshCard((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttViewOptionRefreshCard */
/****************************************************************************
    TttViewOptionApplyCard

    Handles msgOptionApplyCard

    Note:  Perhaps this should be an undoable operation.
****************************************************************************/
#define DbgTttViewOptionApplyCard(x) \
    TttDbgHelper(U_L("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((U_L("")))
    //
    // See note [1] at the beginning of this file.
    //
    if (pArgs->tag != tagTttViewCard) {
        DbgTttViewOptionRefreshCard((U_L("unrecognized card; call ancestor")))
        return ObjCallAncestorCtxWarn(ctx);
    }
    //
    // Collect info needed to apply card.
    //
    StsJmp(TttUtilGetComponents(OSThisApp(), tttGetView, objNull, \
            &view, objNull), s, Error);
    DbgTttViewOptionApplyCard((U_L("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((U_L("\"Line Thickness\" is dirty;
value=%ld"),value))
        ObjCallJmp(msgTttViewGetMetrics, view, &vm, s, Error);
        vm.lineThickness = value;
        ObjCallJmp(msgTttViewSetMetrics, view, &vm, s, Error);
    } else {
        DbgTttViewOptionApplyCard((U_L("\"Line Thickness\" is not dirty")))
    }

    DbgTttViewOptionApplyCard((U_L("return stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewOptionApplyCard((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttViewOptionApplyCard */
/****************************************************************************
    TttViewOptionApplicableCard
    Handles msgOptionApplicableCard
****************************************************************************/
#define DbgTttViewOptionApplicableCard(x) \
    TttDbgHelper(U_L("TttViewOptionApplicableCard"),tttViewOptsDbgSet,0x80,x)
MsgHandlerWithTypes(TttViewOptionApplicableCard, P_OPTION_CARD, \
        PP_TTT_VIEW_INST)
{
    OBJECT      owner;
    STATUS      s;
    DbgTttViewOptionApplicableCard((U_L("")))
    //
    // 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((U_L("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((U_L("owner is self; return stsOK")))
        return stsOK;
    } else {
        DbgTttViewOptionApplicableCard((U_L("owner is not self; return
stsFailed")))
        return stsFailed;
    }
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewOptionApplicableCard((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttViewOptionApplicableCard */
```

TTTVXFER.C

```
/****************************************************************************
 File: tttvxfer.c
 (C) Copyright 1992 by 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:   1.8  $
 $Author:   kcatlin  $
 $Date:   28 Jul 1992 11:20:14  $
 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
#ifndef INTL_INCLUDED
#include <intl.h>
#endif
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                           Defines, Types, Globals, Etc                     *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                           Utility Routines                                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                           Message Handlers                                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/****************************************************************************
    TttViewSelBeginMoveAndCopy
    Handles both msgSelBeginMove and msgSelBeginCopy
****************************************************************************/
#define DbgTttViewSelBeginMoveAndCopy(x) \
    TttDbgHelper(U_L("TttViewSelBeginMoveAndCopy"),tttViewXferDbgSet,0x1,x)
MsgHandlerWithTypes(TttViewSelBeginMoveAndCopy, P_XY32, PP_TTT_VIEW_INST)
{
    EMBEDDED_WIN_BEGIN_MOVE_COPY    bmc;
    STATUS                          s;
    DbgTttViewSelBeginMoveAndCopy((U_L("self=0x%lx"),self))
    //
    // If we don't handle this message, the default behavior is to
    // draw a marquee around the entire selection. For ttt, the marquee
    // would stretch around the entire board, which is too large to be
    // be easily dragged into another document. So, we handle this message,
    // and set the bounds of the move/copy area to an empty rectangle.
    // msgEmbeddedWinBeginMove/Copy will know to display a move/copy icon
    // instead of drawing the marquee.
    //
    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((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewSelBeginMoveAndCopy((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttViewSelBeginMoveAndCopy */

/****************************************************************************
    TttViewXferGet
****************************************************************************/
#define DbgTttViewXferGet(x) \
    TttDbgHelper(U_L("TttViewXferGet"),tttViewXferDbgSet,0x2,x)
MsgHandlerWithTypes(TttViewXferGet, P_XFER_FIXED_BUF, PP_TTT_VIEW_INST)
{
    STATUS  s;
    DbgTttViewXferGet((U_L("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] = U_L('\0');
        s = stsOK;
    } else {
        s = ObjectCallAncestorCtx(ctx);
    }
    DbgTttViewXferGet((U_L("returns 0x%lx"),s))
    return s;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewXferGet((U_L("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(U_L("TttViewXferList"),tttViewXferDbgSet,0x4,x)
MsgHandlerWithTypes(TttViewXferList, OBJECT, PP_TTT_VIEW_INST)
{
    STATUS  s;
    DbgTttViewXferList((U_L("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((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    DbgTttViewXferList((U_L("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(U_L("TttViewSelMoveAndSelCopy"),tttViewXferDbgSet,0x8,x)
MsgHandlerWithTypes(TttViewSelMoveAndSelCopy, P_XY32, PP_TTT_VIEW_INST)
{
    TAG             transferType;
    OBJECT          owner;
    XFER_LIST_NEW   listNew;
    STATUS          s;
    DbgTttViewSelMoveAndSelCopy((U_L("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((U_L("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.
    //
    if (owner == self) {
        DbgTttViewSelMoveAndSelCopy((U_L("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((U_L("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((U_L("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((U_L("returns stsOK")))
    return stsOK;
    MsgHandlerParametersNoWarning;
Error:
    if (listNew.object.uid) {
        ObjCallWarn(msgDestroy, listNew.object.uid, pNull);
    }
    DbgTttViewSelMoveAndSelCopy((U_L("Error; return 0x%lx"),s))
    return s;
} /* TttViewSelMoveAndSelCopy */
```

USA.RC

```
/****************************************************************************
 File: usa.rc
 (C) Copyright 1992 by GO Corporation, All Rights Reserved.
 $Revision:   1.8  $
 $Author:   ehoogerb  $
 $Date:   22 Oct 1992 16:47:08  $
 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 APPTAG_INCLUDED
#include <apptag.h>         // Resource ID & TAGs for app framework
#endif
#ifndef INTL_INCLUDED
#include <intl.h>
#endif
#ifndef BRIDGE_INCLUDED
#include <bridge.h>
#endif
#ifndef TTTPRIV_INCLUDED
#include "tttpriv.h"
#endif
#ifndef TTTVIEW_INCLUDED
#include "tttview.h"
#endif
#ifndef TTTDATA_INCLUDED
#include "tttdata.h"
#endif
/****************************************************************************
                A P P   F r a m e w o r k   S t r i n g s
****************************************************************************/
// Define the strings for use with the AppFramework resource for TTT.
static RC_TAGGED_STRING    appStrings[] = {
    // The company that produced the program.
    tagAppMgrAppCompany,            U_L("GO Corporation"),
    // The copyright string.
    tagAppMgrAppCopyright,
    U_L("\x00A9 Copyright 1992 by GO Corporation, All Rights Reserved."),
    // User-visible filename.  32 chars or less.
    tagAppMgrAppFilename,
    U_L("Tic-Tac-Toe"),
    // User-visible file type.  32 chars or less.
    tagAppMgrAppClassName,
    U_L("Application"),
    Nil(TAG)            // end of list marker
};
static RC_INPUT    app = {
    resAppMgrAppStrings,
    appStrings,
    0,
    resTaggedStringArrayResAgent
};
/****************************************************************************
                       T o o l k i t   S t r i n g s
****************************************************************************/
// Define the strings for use with the Toolkit resource for clsTttView
static RC_TAGGED_STRING    tttViewTKStrings[] = {
    // The title of the TTT option card
    tagCardTitle,       U_L("TTT Card"),
    // The label on the TTT option card for the line thickness control
    tagCardLineThickness,   U_L("Line Thickness:"),
    Nil(TAG)
```

```
};
static RC_INPUT     tttViewTK = {
    resTttViewTK,
    tttViewTKStrings,
    0,
    resTaggedStringArrayResAgent
};
// Define the strings for use with the Toolkit resource for clsTttData
static RC_TAGGED_STRING     tttDataTKStrings[] = {
    // Undo menu string to undo a delete
    tagTttDataUndoDelete,   U_L("Undo Delete"),
    // Undo menu string to undo a move or copy
    tagTttDataUndoMoveCopy, U_L("Undo Move/Copy"),
    Nil(TAG)
};
static RC_INPUT     tttDataTK = {
    resTttDataTK,
    tttDataTKStrings,
    0,
    resTaggedStringArrayResAgent
};
/****************************************************************************
                     Q u i c k   H e l p   S t r i n g s
****************************************************************************/
// Define the quick help resource for TTT.
static RC_TAGGED_STRING     tttViewQHelpStrings[] = {
    // Quick help for TTT's option card to change the line thickness.
    tagTttViewCard,
    U_L("TTT Card||")
    U_L("Use this option card to change the thickness of the lines ")
    U_L("on the Tic-Tac-Toe board."),
    // Quick help for the line thickness control in TTT's option card.
    tagCardLineThickness,
    U_L("Line Thickness||")
    U_L("Change the line thickness by writing in a number from 1-9."),
    // Quick Help for the TTT window.
    tagTttView,
    U_L("Tic-Tac-Toe||")
    U_L("The Tic-Tac-Toe window lets you to make X's and O's in a Tic-Tac-Toe ")
    U_L("grid. You can write X's and O's and make move, copy ")
    U_L("and pigtail delete gestures.\n\n")
    U_L("It does not recognize a completed game, either tied or won.\n\n")
    U_L("To clear the game and start again, tap Select All in the Edit menu, ")
    U_L("then tap Delete."),
    Nil(TAG)
};
static RC_INPUT     tttViewQHelp = {
    resTttViewQHelp,
    tttViewQHelpStrings,
    0,
    resTaggedStringArrayResAgent
};
/****************************************************************************
                     Strings for pstamp to use
****************************************************************************/
// Define the strings for use with the AppFramework resource for TTT.
static RC_TAGGED_STRING     stampStrings[] = {
    // User-visible file name of Stationery already filled in. 32 chars or less
    tagTttStationery1,
    U_L("Tic-Tac-Toe (filled)"),
    // User-visible file name of Stationery with X's. 32 chars or less
    tagTttStationery2,
    U_L("Tic-Tac-Toe (X's)"),
    // User-visible file name of the strategy help file. 32 chars or less
    tagStrategyHelp,
    U_L("Tic-Tac-Toe Strategy"),
    // User-visible file name of the rules help file. 32 chars or less
    tagRulesHelp,
    U_L("Tic-Tac-Toe Rules"),
    Nil(TAG)            // end of list marker
};
static RC_INPUT     pstamp = {
    MakeListResId(clsTttApp, resGrpMisc, 0),
    stampStrings,
    0,
    resTaggedStringArrayResAgent
};
/****************************************************************************
                     L i s t   o f   R e s o u r c e s
****************************************************************************/
// List all of the resources so that RC can find them.
P_RC_INPUT      resInput [] = {
    &app,               // the Application Framework strings
    &tttViewQHelp,      // the Quick Help for clsTttView
    &tttViewTK,         // the Toolkit Strings for clsTttView
    &tttDataTK,         // the Toolkit Strings for clsTttData
    &pstamp,            // strings for pstamp to use
    pNull               // End of list.
};
```

FILLED.TXT

```
xoxoxoxox stationery for tttapp
```

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.
```

XSONLY.TXT

```
x x x x x stationery for tttapp
```

Template Application

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

PenPoint applications rely on **clsApp** to create and display their main window, save state, terminate the application instance, and so on. There are seventy-some messages that **clsApp** responds to. You will never have to worry about handling most of these, however, every application developer needs to create a descendant of **clsApp** and have the descendant handle several important messages. Template Application illustrates the messages that most applications will need to be concerned with. You will find a description of these messages and how to handle them in the block comments for each of the methods. In addition, these messages are fully documented in the PenPoint Architectural Reference, and the APP.H header file.

The PenPoint Application Framework defines a number of flags that you can use to explore the classes that it contains and the messages that they respond to. These flags are documented in the header comment for TEMPLTAP.C.

Objectives

Template Application serves as a shell of an application and can be used as the starting point for a real application.

This sample application also shows how to:

- File instance data.
- Create the standard menu bar and add application-specific menus.
- Create an icon window as a client window.
- Associate a resource file with your application.
- Use resource files to define the standard application resources.
- Define tags and lists for the strings used as resources.

Class overview

Template Application defines two classes: **clsTemplateApp** and **clsFoo**. It makes use of the following classes:

clsApp
clsAppMgr
clsClass
clsIconWin
clsMenu
clsObject

Files used

The code for Template Application is in PENPOINT\SDK\SAMPLE\TEMPLTAP. The files are:

METHODS.TBL	the list of messages that the classes respond to, and the associated message handlers to call.
FOO.C	the source code for **clsFoo.**
FOO.H	the header file for **clsFoo.**
TEMPLTAP.C	the source code for the application class.
TEMPLTAP.H	the header file for the application class
JPN.RC	strings for the Japanese version (not listed here for typographical reasons).
USA.RC	strings for the USA version.

METHODS.TBL

```
/****************************************************************************
 File: methods.tbl
 (C) Copyright 1992 by 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  $
   $Author:   aloomis  $
```

```
    $Date:   16 Sep 1992 16:06:08  $
 This file contains the method table definitions for templtap.exe.
****************************************************************************/
#ifndef APP_INCLUDED
#include <app.h>
#endif
#ifndef TEMPLTAP_INCLUDED
#include <templtap.h>
#endif
#ifndef FOO_INCLUDED
#include <foo.h>
#endif
MSG_INFO clsTemplateAppMethods[] = {
#ifdef DEBUG
{msgDump,               "TemplateAppDump",          objCallAncestorBefore   },
#endif
{msgInit,               "TemplateAppInit",          objCallAncestorBefore   },
{msgFree,               "TemplateAppFree",          objCallAncestorAfter    },
{msgSave,               "TemplateAppSave",          objCallAncestorBefore   },
{msgRestore,            "TemplateAppRestore",       objCallAncestorBefore   },
{msgAppInit,            "TemplateAppAppInit",       objCallAncestorBefore   },
{msgAppOpen,            "TemplateAppOpen",          objCallAncestorAfter    },
{msgAppClose,           "TemplateAppClose",         objCallAncestorBefore   },
{msgAppCreateClientWin, "TemplateAppCreateClientWin",                       },
{msgAppCreateMenuBar,   "TemplateAppCreateMenuBar",                         },
{msgAppRevert,          "TemplateAppRevert",        objCallAncestorBefore   },
{msgAppSelectAll,       "TemplateAppSelectAll",                             },
{msgTemplateAppGetMetrics,      "TemplateAppGetMetrics",                    },
{0}
};
MSG_INFO clsFooMethods[] = {
{msgNewDefaults,            "FooNewDefaults",       objCallAncestorBefore   },
#ifdef DEBUG
{msgDump,                   "FooDump",              objCallAncestorBefore   },
#endif

{msgInit,                   "FooInit",              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}
};
```

FOO.C

```
/****************************************************************************
 File: foo.c
 (C) Copyright 1992 by 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:   1.9  $
   $Author:   aloomis  $
     $Date:   09 Oct 1992 19:44:34  $
 This file contains the class definition and methods for clsFoo.
****************************************************************************/
#ifndef FOO_INCLUDED
#include <foo.h>
#endif
#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif
#ifndef RESFILE_INCLUDED
#include <resfile.h>
#endif
#ifndef INTL_INCLUDED
#include <intl.h>
#endif
#ifndef _STRING_H_INCLUDED
#include <string.h>
#endif
#include <methods.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
#ifdef DEBUG
/****************************************************************************
    FooDump

    Respond to msgDump.
****************************************************************************/
MsgHandlerArgType(FooDump, P_ARGS)
{
    Debugf(U_L("foo: msgDump"));
    return stsOK;
    MsgHandlerParametersNoWarning;
}   // FooDump
#endif
/****************************************************************************
    FooInit

    Respond to msgInit.  Create a new object.
****************************************************************************/
MsgHandlerArgType(FooInit, P_FOO_NEW)
{
    FOO_INST    inst;
    memset(&inst, 0, SizeOf(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.
****************************************************************************/
MsgHandlerWithTypes(FooSave, P_OBJ_SAVE, P_FOO_INST)
{
    STREAM_READ_WRITE   fsWrite;
    FILED_DATA          filed;
    STATUS              s;

    memset(&filed, 0, SizeOf(filed));
    filed.style = pData->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(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.
****************************************************************************/
MsgHandlerWithTypes (FooGetStyle, P_FOO_STYLE, P_FOO_INST)
{
    *pArgs = pData->metrics.style;
    return stsOK;
    MsgHandlerParametersNoWarning;

}   // FooGetStyle
/****************************************************************************
    FooSetStyle

    Set foo style.
****************************************************************************/
MsgHandlerWithTypes (FooSetStyle, P_FOO_STYLE, P_FOO_INST)
{
    P_FOO_INST      pInst;
    pInst = pData;
    // Update instance data.
    pInst->metrics.style = *pArgs;
    ObjectWrite(self, ctx, pInst);
    return stsOK;
    MsgHandlerParametersNoWarning;
}   // FooSetStyle
/****************************************************************************
    FooGetMetrics

    Get foo metrics.
****************************************************************************/
MsgHandlerWithTypes (FooGetMetrics, P_FOO_METRICS, P_FOO_INST)
{
    *pArgs = pData->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.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
```

FOO.H

```
/****************************************************************************
 File: foo.h
 (C) Copyright 1992 by 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:   1.9  $
   $Author:   aloomis  $
     $Date:   16 Sep 1992 16:06:34  $
 This file contains the API definition for clsFoo.
****************************************************************************/
#ifndef FOO_INCLUDED
#define FOO_INCLUDED
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif
typedef OBJECT FOO, *P_FOO;
#define clsFoo                          MakePrivateWKN(1,1)
// RES_IDs for the resource lists used with the TAGs.
#define resFooQHelp             MakeListResId(clsFoo, resGrpQhelp, 0)
#define resFooStdMsgError       MakeListResId(clsFoo, resGrpStdMsg, 0)
#define resFooStdMsgWarning     MakeListResId(clsFoo, resGrpStdMsg, 1)
// Quick Help codes.
#define qhFooQuickHelp1                 MakeTag(clsFoo, 1)
// The error status codes for TEMPLTAP.
#define stsFooError1                    MakeStatus(clsFoo, 1)
// The warning and informational status codes for TEMPLTAP.
#define stsFooWarning1                  MakeWarning(clsFoo, 1)
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
```

TEMPLTAP.C

```
/****************************************************************************
 File: templtap.c
 (C) Copyright 1992 by 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:   1.14  $
   $Author:   aloomis  $
     $Date:   09 Oct 1992 19:44:22  $
 This file contains the templtap application.  Template Application serves
 as a shell of an application, and can be used as a starting point for a
 real application.
 PenPoint applications rely on clsApp to create and display their main
 window, save state, terminate the application instance, and so on.
 There are seventy-some messages that clsApp responds to.  You will never
 have to worry about handling most of these, however, every application
 developer needs to create a descendant of clsApp and have the descendant
 handle several important messages.  Template App illustrates the messages
 that most applications will need to be concerned with. You will find a
 description of these messages and how to handle them in the block comments
 for each of the methods.  In addition, these messages are fully documented
 in the Architectural Reference, and the app.h header file.
 The PenPoint Application Framework defines a number of flags that you can
 use to explore the classes that it contains and the messages that they
 respond to:
      Flag     Description                             Context
    ----------------------------------------------------------------------
        2      AppWin & AppLink debug messages         Traversal
        4      General App Framework debug messages    Document Lifecycle
       *8*     move/copy debug messages                Move/Copy
      *20*     dump filesystem on move/copy/delete     Move/Copy
       40      goto button debug messages              Traversal
      *80*     traverse debug messages                 Traversal
     *400*     trace document lifecycle                Document Lifecycle
     *800*     option sheet protocol                   Option Sheets
    *1000*     clsApp performance debug output         Document Lifecycle
    *2000*     clsAppMgr performance debug output      Document Lifecycle
     10000     Search & Replace                        Traversal
     20000     Memory Cop                              Memory
 (See the Application Writing Guide for information on how to turn on a flag.)

 You can try setting the various flag values and exploring the system, or you
 can turn on specific flags to isolate a problem that you are having with your
 application.  Those flags that you will probably find most useful have been
 marked with the asterixes.  In particular, you may want to trace the document
 lifecycle by setting R to 400.  Turn a page, and you will see the places where
 msgInit, msgAppActivate, msgAppOpen, msgAppClose, msgAppTerminate, msgAppSave,
 etc. are received by clsApp and Template App.
****************************************************************************/
```

```
#ifndef APP_INCLUDED
#include <app.h>
#endif
#ifndef FRAME_INCLUDED
#include <frame.h>
#endif
#ifndef TEMPLTAP_INCLUDED
#include <templtap.h>
#endif
#ifndef APPMGR_INCLDUDED
#include <appmgr.h>
#endif
#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif
#ifndef ICONWIN_INCLUDED
#include <iconwin.h>
#endif
#ifndef INTL_INCLUDED
#include <intl.h>
#endif
#ifndef BRIDGE_INCLUDED
#include <bridge.h>
#endif
#ifndef RESFILE_INCLUDED
#include <resfile.h>
#endif
#ifndef OS_INCLUDED
#include <os.h>
#endif
#ifndef _STRING_H_INCLUDED
#include <string.h>
#endif
#include <methods.h>
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                   Defines, Types, Globals, Etc                          *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
typedef struct TEMPLATE_APP_INST {
    TEMPLATE_APP_METRICS metrics;
} TEMPLATE_APP_INST, *P_TEMPLATE_APP_INST;
typedef struct FILED_DATA {
    TEMPLATE_APP_METRICS metrics;
} FILED_DATA, *P_FILED_DATA;
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                            Message Handlers                             *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#ifdef DEBUG
/****************************************************************************
 *
 *  TemplateAppDump
 *
 *  Respond to msgDump.
 *
 *  msgDump requests an object to format its instance data and send it to
 *  the debugger stream.  While developing your 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.
 *
 */
MsgHandlerArgType (TemplateAppDump, P_ARGS)
{
    Dbg(Debugf(U_L("Template App received: msgDump")));
    return stsOK;
    MsgHandlerParametersNoWarning;
}   // TemplateAppDump
#endif      // DEBUG
/****************************************************************************
 *
 *  TemplateAppInit
 *
 *  Respond to msgInit.
 *
 *  MsgInit is the first message sent to the activated instance of your
 *  application.  When your application recieves msgInit, it should
 *  initialize its instance data and use the function ObjectWrite to save
 *  the initialized instance data to protected memory.
 *
 *  The method table calls the ancestor before handling msgInit.  In
 *  response to msgInit, clsApp allocates storage for the application's
 *  instance data in protected memory.  Included in the instance data are
 *  the document's directory handle, resource file list, floating window
 *  list, and embedded document list.
 *
 */
MsgHandlerArgType (TemplateAppInit, P_APP_NEW)
{
    Dbg(Debugf(U_L("Template App received: msgInit")));
    return stsOK;
    MsgHandlerParametersNoWarning;
}   // TemplateAppInit
```

```
/****************************************************************************
 *
 *  TemplateAppFree
 *
 *  Respond to msgFree.
 *
 *  When your application receieves msgFree, it should destroy any perma-
 *  nent objects that it created and free any allocated memory.  (The
 *  application receives msgSave before it receives msgFree.)
 *
 *  The method table should call the application's ancestor *after* it
 *  handles msgFree; the application's ancestors will in turn destroy
 *  objects and free memory that they allocated.
 *
 */
MsgHandlerArgType (TemplateAppFree, P_ARGS)
{
    Dbg(Debugf(U_L("Template App received: msgFree")));
    return stsOK;
    MsgHandlerParametersNoWarning;
}   // TemplateAppFree
/****************************************************************************
 *
 *  TemplateAppSave
 *
 *  Respond to msgSave.
 *
 *  When your application receives msgSave, it should: (1) Write data
 *  that isn't maintained in objects to the resource file. (2) Save any
 *  permanent objects.  (This will involve sending msgSave to your in-
 *  stance data.)
 *
 *  The method table calls the application's ancestor before handling
 *  msgSave.  In response, the ancestors save data and objects that they
 *  created.  clsApp saves your main window, any data objects observed
 *  by views, and, optionally, its client window.
 *
 */
MsgHandlerWithTypes (TemplateAppSave, P_OBJ_SAVE, P_TEMPLATE_APP_INST)
{
    STREAM_READ_WRITE       fsWrite;
    FILED_DATA              filed;
    STATUS                  s;
    Dbg(Debugf(U_L("Template App received: msgSave")));
    memset (&filed, 0, SizeOf (filed));
    filed.metrics = pData->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.
 *
 *  In the message handler for msgRestore, you should restore any data
 *  that you saved in msgSave.
 *
 *  Your application's method table should call the application's ancestor
 *  before handling msgRestore.  In response to msgRestore, the ancestors
 *  will restore data and objects that they saved earlier.  clsApp restores
 *  your main window and its client window.
 *
 */
MsgHandlerArgType (TemplateAppRestore, P_OBJ_RESTORE)
{
    STREAM_READ_WRITE   fsRead;
    TEMPLATE_APP_INST   inst;
    FILED_DATA          filed;
    STATUS              s;
    Dbg(Debugf (U_L("Template App received: msgRestore")));
    memset(&inst, 0, SizeOf (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
/****************************************************************************
 *
 *  TemplateAppAppInit
 *
 *  Respond to msgAppInit.
 *
 *  MsgAppInit is only sent to each instance *once* during its lifetime.
 *  The document receives this message the first time it is activated.
 *  (Your document will receive msgRestore on all subsequent activations,
 *  where it will restore the objects that are created here, and filed
 *  when the document is saved.) The ancestor should be called before your
 *  message handler for msgApp Init is invoked.  In response, clsApp will
 *  create your document's main window and object resource file.
 *
 *  In msgAppInit you should handle the one-time initializations of stateful
 *  objects that wil later be filed.  This includes initializing your
 *  instance data.  Use the function ObjectWrite to save the initialized
 *  instance data to protected memory.
 *
 *  Here we create and install the client window in the application's
 *  main window.  For a more robust example of handling msgAppInit, see
 *  CNTRAPP/cntrapp.c
 *
 */
MsgHandlerArgType (TemplateAppAppInit, DIR_HANDLE)
{
    APP_METRICS     am;
    OBJECT          win;
    STATUS          s;
    Dbg(Debugf(U_L("Template App received: msgAppInit")));
    // 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;
}   // TemplateAppAppInit
/****************************************************************************
 *
 *  TemplateAppOpen
 *
 *  Respond to msgAppOpen.
 *
 *  msgAppOpen is where you should create the windows to display data,
 *  and any other non-stateful user interface or control objects.  You
 *  should also fill in childAppParentWin - normally with the document's
 *  client window.  The menu bar is also typically created here.  Self
 *  send msgAppCreateMenuBar to create the menu bar, and then send
 *  msgFrameSetMetrics to your main window to insert the menu bar in the
 *  window.  If you can't open the document, you should return stsFailed.
 *  However, if you have already displayed an error message to the user,
 *  then return stsAppOpenFailedSupressError.
 *
 *  The method table calls the application's ancestor *after* the appli-
 *  cation handles msgAppOpen.  In response to msgAppOpen, clsApp then
 *  inserts the document's frame into the main window, which displays
 *  the document on screen.
 *
 */
MsgHandlerArgType (TemplateAppOpen, P_APP_OPEN)
{
    WIN                 clientWin;
    APP_METRICS         am;
    OBJECT              menuBar;
    STATUS              s;
    Dbg(Debugf (U_L("Template App received: msgAppOpen")));
    // 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(msgFrameGetClientWin, am.mainWin, (P_ARGS) &clientWin, s);
    pArgs->childAppParentWin = clientWin;
    return stsOK;
    MsgHandlerParametersNoWarning;
}   // TemplateAppOpen
/****************************************************************************
 *
 *  TemplateAppClose
 *
 *  Respond to msgAppClose.
 *
 *  In msgAppClose, you should destroy the windows and control objects
 *  that you created in msgAppOpen.  If you created the menu bar in your
 *  msgAppOpen handler, then you should send msgFrameDestroyMenuBar to your
 *  main window.
 *
 *  This message is not an indication to terminate the document; it may be
 *  followed by other requests for services such as searching or reopening.
 *  However, you will save memory (always desirable for PenPoint applica-
 *  tions!) if you destroy the user-interface objects while your applica-
 *  tion is off screen.
 *
 *  The method table should (and does!) call the application's ancestor
 *  before handling msgAppClose.  In response to msgAppClose, clsApp ex-
 *  tracts the frame from the main window.
 *
 */
MsgHandlerArgType (TemplateAppClose, P_ARGS)
{
    APP_METRICS     am;
    STATUS          s;
    Dbg(Debugf(U_L("Template App received: msgAppClose")));
    ObjCallRet(msgAppGetMetrics, self, &am, s);
    ObjCallRet(msgFrameDestroyMenuBar, am.mainWin, pNull, s);
    return stsOK;
    MsgHandlerParametersNoWarning;
}   // TemplateAppClose
/****************************************************************************
 *
 *  TemplateAppCreateClientWin
 *
 *  Respond to msgAppCreateClientWin.
 *
 *  This is the place to create your application specific client window.
 *  The Application Framework does not send this message by default.
 *  Instead, you should self send it at the appropriate time (typically
 *  during msgAppInit, since the client window is usually stateful).
 *  Usually you will not need to call your ancestor.
 *
 *  The document creates a default client window of class clsEmbeddedWin
 *  and passes back its uid.
 *
 */
MsgHandlerArgType(TemplateAppCreateClientWin, P_OBJECT)
{
    ICON_WIN_NEW    iwn;
    Dbg(Debugf(U_L("Template App received: msgAppCreateClientWin")));
    // If the client win has already been 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.
 *
 *  You should handle this message by creating the document's menu bar.
 *  If pArgs is non-null when the ancestor is called, clsApp will pre-pend
 *  the Document, Edit, and Option menus to the provided menu bar.  So,
 *  the ancestor should be called after you make the menu bar.  After the
 *  ancestor returns, you can fix up the Document and Edit menus to
 *  remove any buttons that you don't support or to add any new buttons.
 *
 */
MsgHandlerArgType (TemplateAppCreateMenuBar, P_OBJECT)
{
    STATUS          s;
    //MENU_NEW      mn;
    Dbg(Debugf(U_L("Template App received: msgAppCreateMenuBar")));
    // 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.
    ObjCallAncestorCtxRet (ctx, s);
    // Fixup the menu bar here...
    return stsOK;
    MsgHandlerParametersNoWarning;
}   // TemplateAppCreateMenuBar

/****************************************************************************
 *
 *  TemplateAppRevert
 *
 *  Respond to msgAppRevert
 *
 *  The document reverts to its previously saved state.  If true is passed
 *  in, the document displays a note, asking the user to confirm the action
 *  first.  If false is passed in, the document just does the action.  This
 *  is handled by an ancestor's (clsApp's) message handler, which needs to
 *  be called *before* your handler.
 *
 *  If you do not support revert, you should handle this message by return-
 *  ing stsAppRefused.  On the other hand, if you support revert, but
 *  manage your own data files, or use memory mapped files, then it may be
 *  necessary to handle this message by appropriately undoing all data and
 *  modifications since the last save.
 *
 */
MsgHandlerArgType (TemplateAppRevert, P_ARGS)
{
    Dbg(Debugf (U_L("Template App received: msgAppRevert")));
    return stsAppRefused;
    MsgHandlerParametersNoWarning;
}  // TemplateAppRevert
/****************************************************************************
 *
 *  TemplateAppSelectAll
 *
 *  Respond to msgAppSelectAll
 *
 *  When the user taps on Select All in the Standard Application Menu, the
 *  document self sends this message.
 *
 *  clsApp does not do anything in its message handler for this message.
 *  You should handle this message, and select everything in the document.
 *  (You tend not to call the ancestor.)
 *
 */
MsgHandler (TemplateAppSelectAll)
{
    Dbg(Debugf(U_L("Template App received: msgAppSelectAll")));
    return stsOK;
    MsgHandlerParametersNoWarning;
}  // TemplateAppSelectAll
/****************************************************************************
*
*   TemplateAppGetMetrics
*
*   Respond to msgTemplateAppGetMetrics
*
*   This is a message defined for Template App.  It returns information
*   specific to the Template Application, in this case the metrics field
*   of the Template App instance data.
*
*/
MsgHandlerWithTypes (TemplateAppGetMetrics, P_TEMPLATE_APP_METRICS,
        P_TEMPLATE_APP_INST)
{
    Dbg(Debugf(U_L("Template App received: msgTemplateAppGetMetrics")));
    *pArgs = pData->metrics;
    return stsOK;
    MsgHandlerParametersNoWarning;
}   // TemplateAppGetMetrics
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                              Installation                              *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/****************************************************************************
 *
 *  ClsTemplateAppInit
 *
 *  The initialization routine ClsTemplateAppInit creates the clsTemplate
 *  App class when the application is installed.  Application classes are
 *  created by sending msgNew to clsAppMgr.
 *
 *  Your application will need a similar routine.
 */
STATUS PASCAL
ClsTemplateAppInit (void)
{
    APP_MGR_NEW     new;
    STATUS          s;
    ObjectCall(msgNewDefaults, clsAppMgr, &new);
    new.object.uid          = clsTemplateApp;
    new.cls.pMsg            = clsTemplateAppTable;
    new.cls.ancestor        = clsApp;
    new.cls.size            = SizeOf (TEMPLATE_APP_INST);
    new.cls.newArgsSize     = SizeOf (APP_NEW);
#ifdef PP1_0
    Ustrcpy(new.appMgr.defaultDocName, "Template Document");
    Ustrcpy(new.appMgr.company, "GO Corporation");
    // 00A9 is the circle-c copyright symbol
    new.appMgr.copyright = "\x00A9 1992 GO Corporation, All Rights Reserved";
#endif // PP1_0
    RectInit(&new.appMgr.defaultRect, 0, 0, 216, 108);
    ObjCallRet(msgNew, clsAppMgr, &new, s);
    return stsOK;
}
/*
 * Here we declare a function prototype so we can create clsFoo when
 * Template Application is installed below.  ClsFooInit is defined
 * in the foo.c file.
 */
STATUS ClsFooInit (void);
/****************************************************************************
 *  main
 *
 *  The function main() is the entry point to your application.  An
 *  application executable file must have a main() function.
 *
 *  main has two primary purposes: (1) Installing your application class,
 *  and (2) Starting the dispatch loop for each document, or application
 *  instance.
 *
 *  The processCount parameter to main() indicates how many other processes
 *  are currently running this application program.  When process count is
 *  zero, the application is being installed, so you call a routine to
 *  install your application class, and then call AppMonitorMain(), a
 *  PenPoint function that starts the application monitor for your appli-
 *  cation class.
 *
 *  When processCount is greater than zero, the user is opening a document
 *  so you call AppMain(), a PenPoint function that creates an instance of
 *  application class, passes it messages to initialize its data, and
 *  enters a dispatch loop to receive messages.
 *
 */
void CDECL
main (
    S32     argc,
    CHAR    *argv[],
    U32     processCount)
{
    if (processCount == 0) {
        // Create the (global) application class.
        StsWarn (ClsTemplateAppInit ());
        // Start msg dispatching.
        AppMonitorMain (clsTemplateApp, objNull);
    }
    else {
        // Create private classes.
        StsWarn (ClsFooInit());
        // Start msg dispatching.
        AppMain();
    }
    Unused(argc); Unused(argv);
}   // main
```

TEMPLTAP.H

```
/****************************************************************************
 File: templtap.h
 (C) Copyright 1992 by 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:   1.9  $
   $Author:   aloomis  $
     $Date:   09 Oct 1992 19:44:12  $
 This file contains the templateapp application API.
****************************************************************************/
#ifndef TEMPLTAP_INCLUDED
#define TEMPLTAP_INCLUDED
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif
#define clsTemplateApp                      MakeGlobalWKN (3513, 1)
// Resource Identifiers
#define resTemplateAppQHelp             \
        MakeListResId(clsTemplateApp, resGrpQhelp, 0)
#define resTemplateAppStdMsgError       \
        MakeListResId(clsTemplateApp, resGrpStdMsg, 0)
#define resTemplateAppStdMsgWarning     \
        MakeListResId(clsTemplateApp, resGrpStdMsg, 1)
// Quick Help tags
#define qhTemplateAppQuickHelp1             MakeTag(clsTemplateApp, 1)
#define qhTemplateAppQuickHelp2             MakeTag(clsTemplateApp, 2)
// Status codes.
#define stsTemplateAppError1                MakeStatus(clsTemplateApp, 1)
#define stsTemplateAppError2                MakeStatus(clsTemplateApp, 2)
// Warning codes.
#define stsTemplateAppWarning1              MakeWarning(clsTemplateApp, 1)
#define stsTemplateAppWarning2              MakeWarning(clsTemplateApp, 2)
typedef OBJECT TEMPLATE_APP, *P_TEMPLATE_APP;
typedef struct TEMPLATE_APP_METRICS {
    U32     dummy;
    U32     reserved;
} TEMPLATE_APP_METRICS, *P_TEMPLATE_APP_METRICS;
/****************************************************************************
 msgTemplateAppGetMetrics   takes P_TEMPLATE_APP_METRICS, returns STATUS
    Get TemplateApp metrics.
*/
#define msgTemplateAppGetMetrics                    MakeMsg(clsTemplateApp, 1)
#endif  // TEMPLTAP_INCLUDED
```

USA.RC

```
/****************************************************************************
File: usa.rc
 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:   1.8  $
   $Author:   bldmstr  $
     $Date:   19 Aug 1992 22:42:52  $
 usa.rc is the English language resource file for the Template App.
 Template App does not actually use any resources besides the standard
 application resources, however we have included resource strings for
 QuickHelp, standard messages and standard warnings to serve as a template
 for your application.
****************************************************************************/
#ifndef RESCMPLR_INCLUDED
#include <rescmplr.h>       // MUST be included in any resource files
#endif
#ifndef APPTAG_INCLUDED
```

```
#include <apptag.h>         // Resource IDs & TAGs for app framework
#endif
#ifndef INTL_INCLUDED
#include <intl.h>
#endif

#ifndef BRIDGE_INCLUDED     // Bridge from 1_0 to future releases.
#include <bridge.h>
#endif
#ifndef TEMPLTAP_INCLUDED
#include <templtap.h>       // Resource IDs & TAGs for this project.
#endif
#ifndef FOO_INCLUDED
#include <foo.h>            // Resource IDs & TAGs for sub-class
#endif
/***************************************************************************
                A P P   F r a m e w o r k   S t r i n g s
***************************************************************************/
// Define the strings for use with the AppFramework resource for TEMPLTAP.
static RC_TAGGED_STRING     appStrings[] = {
    // The default document name for instances of TEMPLTAP documents.  This
    // should normally match the name the user sees for the application.
    tagAppMgrAppDefaultDocName,     U_L("Template Document"),
    // The company that produced the program.
    tagAppMgrAppCompany,            U_L("GO Corporation"),
    // The copyright string. A9 ( hex ) is the "circle-c" copyright symbol
    tagAppMgrAppCopyright,
    U_L("\x00A9 Copyright 1991-1992 by GO Corporation, All Rights Reserved."),
        // User-visible filename.  32 chars or less.
        tagAppMgrAppFilename,
        U_L("Template Application"),
        // User-visible file type.  32 chars or less.
        tagAppMgrAppClassName,
        U_L("Application"),
    Nil(TAG)            // end of list marker
};
static RC_INPUT     app = {
    resAppMgrAppStrings,            // resource id, in this case pre-defined
    appStrings,                     // pointer to data, a string array
    0,                              // data length, ignored for strings
    resTaggedStringArrayResAgent    // how to interpret data pointer
};
```

```
/***************************************************************************
                          T o o l k i t   S t r i n g s
***************************************************************************/
/***************************************************************************
                       Q u i c k   H e l p   S t r i n g s
***************************************************************************/
// Define the quick help resource for TEMPLTAP.
static RC_TAGGED_STRING     templateAppQHelpStrings[] = {
    // First Quick Help string for TEMPLTAP
    qhTemplateAppQuickHelp1,
    U_L("Sample Quick Help Title||")
    U_L("This is the text for the Sample quick help message."),
    // Second Quick Help string for TEMPLTAP
    qhTemplateAppQuickHelp2,
    U_L("Sample Main window Quick Help||")
    U_L("This is the text for the main window quick help message.  Note ")
    U_L("that we can continue the text on another line using string ")
    U_L("concatenation."),
    Nil(TAG)
};
static RC_INPUT     templateAppQHelp = {
    resTemplateAppQHelp,
    templateAppQHelpStrings,
    0,
    resTaggedStringArrayResAgent
};
// Quick Help for clsFoo
static RC_TAGGED_STRING     fooQHelpStrings[] = {
    // class foo's only quick help item
    qhFooQuickHelp1,
    U_L("Sample Quick Help Title||Sample quick help body."),
    Nil(TAG)
};
static RC_INPUT     fooQHelp = {
    resFooQHelp,
    fooQHelpStrings,
    0,
    resTaggedStringArrayResAgent
};
```

```
/****************************************************************************
                M i s c e l l a n e o u s   S t r i n g s
****************************************************************************/
/****************************************************************************
             S t a n d a r d   M e s s a g e   S t r i n g s
****************************************************************************/
// Define the error message resource for TEMPLTAP.
static RC_TAGGED_STRING     stdMsgTemplateAppErrorStrings[] = {
    // Error message when user does XXXXX.
    stsTemplateAppError1,   U_L("Error 1"),
    // Error message when user does YYYYY.
    stsTemplateAppError2,   U_L("Error 2"),
    Nil(TAG)
};
static RC_INPUT     stdMsgTemplateAppError = {
    resTemplateAppStdMsgError,
    stdMsgTemplateAppErrorStrings,
    0,
    resTaggedStringArrayResAgent
};
// Define the error message resource for FOO.
static RC_TAGGED_STRING     stdMsgFooErrorStrings[] = {
    // Error message when user does ZZZZZ.
    stsFooError1,       U_L("Foo Error"),
    Nil(TAG)
};
static RC_INPUT     stdMsgFooError = {
    resFooStdMsgError,
    stdMsgFooErrorStrings,
    0,
    resTaggedStringArrayResAgent
};
static RC_TAGGED_STRING     stdMsgTemplateAppWarningStrings[] = {
    // Warning message when user does XXX.
    stsTemplateAppWarning1, U_L("First Warning"),
    // Warning message when user does YYY.
    stsTemplateAppWarning2, U_L("Second Warning"),
    Nil(TAG)
};
static RC_INPUT     stdMsgTemplateAppWarning = {
    resTemplateAppStdMsgWarning,
    stdMsgTemplateAppWarningStrings,
    0,
    resTaggedStringArrayResAgent
};
static RC_TAGGED_STRING     stdMsgFooWarningStrings[] = {
    // Warning messgae when user does ZZZ.
    stsFooWarning1,         U_L("Foo Warning"),
    Nil(TAG)
};
static RC_INPUT     stdMsgFooWarning = {
    resFooStdMsgWarning,
    stdMsgFooWarningStrings,
    0,
    resTaggedStringArrayResAgent
};
/****************************************************************************
                     L i s t   o f   R e s o u r c e s
****************************************************************************/
// List all of the resources so that RC can find them.
P_RC_INPUT      resInput [] = {
    &app,                       // the Application Framework strings
    &templateAppQHelp,          // the Quick Help for TEMPLTAP
    &fooQHelp,                  // the Quick Help for FOO
    &stdMsgTemplateAppError,    // the Error Messages for TEMPLTAP
    &stdMsgFooError,            // the Error Messages for FOO
    &stdMsgTemplateAppWarning,  // the Warning Messages for TEMPLTAP
    &stdMsgFooWarning,          // the Warning Messages for FOO
    pNull                       // End of list.
};
```

Adder

Adder is a simple pen-centric 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; "42 + –8 + 3 –2.5" for example.

Objectives

This sample application shows how to:

- Create an insertion pad for handwritten input.
- Create a translator and a custom tmplate 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

Files used

The code for Adder is in PENPOINT\SDK\SAMPLE\ADDER. The files are:

METHODS.TBL	the method tables for the adder classes.
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.
JPN.RC	strings for the Japanese version.
USA.RC	strings for the USA version.

Calculator

The Calculator application implements a typical push-button 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 reusable 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 a single operation.
- Handle button notification messages.
- Change the default window size.
- File data.
- Use **ResUtilGetListString**() to load a string from a resource file.
- Reference Unicode characters in a TK_TABLE_ENTRY.

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**).

Files used

The code for Calc is in PENPOINT\SDK\SAMPLE\CALC. The files are:

CAPPMETH.TBL	method table for the application class.
CENGMETH.TBL	method table for 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.
S_CALC.C	symbol name definitions and call to **ClsSymbolsInit**() (this file is generated automatically).
JPN.RC	strings for the Japanese version.
USA.RC	strings for the USA version.

Clock

Clock is an application that serves two purposes; it is both a digital alarm clock distributed as a part of PenPoint, and a sample application. The end-user 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 end-user 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.

Clock uses the Bridging Package in order to maintain a single code base for both PenPoint 1.0 and PenPoint 2.0 Japanese.

Objectives

This sample application shows how to:

- Observe the system preferences for changes to date and time formats.
- Observe the power switch to refresh on power-up.
- 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.**
- Use **StdError**() to display application-level error messages.
- Use the international routines for formatting.
- Use **theSystemLocale** to do different behavior for different locales.
- Use the Bridging Package to maintain a single code base under PenPoint 1.0 and PenPoint 2.0 Japanese.

Class overview

Clock defines four classes: **clsClockLabel**, **clsClockApp**, **clsClockWin**, and **clsNoteCorkBoardWin.**

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

Clock uses a table layout of several windows. There can be up to four child windows (time digits, a.m./p.m. indicator, alarm indicator, and the date). All of these are labels. **clsLabel** only repaints the right-most characters that change. So, to minimize flashing as time ticks away, Clock displays the time digits in a separate window from the a.m./p.m. indicator. The text for the clock labels is created using **SComposeText**() and international routines for formatting the date and time (**PrefsIntlDateToString**() and **PrefsIntlTimeToString**()) in **GetDateTimeStr**() rather than assuming English formats. These routines query preferences to find the appropriate date and time formats for a given locale. In addition, clock calls **theSystem** to determine the locale and uses a different default format depending on the locale (time followed by date for USA, and date followed by time for JPN).

The labels can be of varying font sizes. In some cases, because of the way that the clock window grows and shrinks, it is possible for the clock window to be off-screen when it comes time to display. To prevent that, code is added in the **CreateClock-Window()** routine to ensure that part of the clock window is always on screen.

clsNoteCorkBoardWin appears as a corkboard on the pop-up 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, **clsNoteCorkBoard-Win** observes objects inserted into its window. **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 **clsNoteCorkBoardWin** corkboard window.

Files used

The code for Clock is in PENPOINT\SDK\SAMPLE\CLOCK. The files are:

METHODS.TBL	The method tables for the four classes.
CLABEL.C	Source code for **clsClockLabel**.
CLABEL.H	Header file for **clsClockLabel**.
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**.
NCBWIN.C	Source code for **clsNoteCorkBoardWin**.
NCBWIN.H	Header file for **clsNoteCorkBoardWin**.
BITMAP.RES	Resource file (compiled) for the clock accessory icon.
JPN.RC	strings for the Japanese version.
USA.RC	strings for the USA version.
BRIDGE.RC	strings that emulate PenPoint 2.0 Japanese functionality in PenPoint 1.0

Notepaper Application

Notepaper Application is a simple note-taking 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 Application defines one class: **clsNotePaperApp**. It makes use of the following classes:

- clsApp
- clsAppMgr
- clsGestureMargin
- clsNotePaper

Files used

The code for Notepaper Application is in PENPOINT\SDK\SAMPLE\NPAPP. The files are:

METHODS.TBL	method table for the notepaper application class.
BITMAP.RES	resource file for the document bitmap.
NPAPP.C	source code for the notepaper application.
JPN.RC	strings for the Japanese version.
USA.RC	strings for the USA version.

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 application-specific menu bar (no SAMs).
- Place a button on a menu bar.
- Use resources for text in toolkit tables.
- Create a scroll win, and have a gray border displayed around its client window.
- Use pixelmaps, drawing contexts, and image devices.
- 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 well-understood 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

Files used

The code for Paint is in PENPOINT\SDK\SAMPLE\PAINT. The files are:

METHODS.TBL	method table for the paint classes.
BITMAP.RES	resource file for the document icon.
CUTIL.C	utility routines for reading and writing data and strings.
CUTIL.H	header file for reading and writing utility routines.
PAPP.H	header file for application and its resource.s
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 clsPixWin class.
PIXWIN.H	header file for the clsPixWin class.
PWIN.C	source code for the clsPaintWin class.
PWIN.H	header file for the clsPaintWin clas.s
JPN.RC	strings for the Japanese version.
USA.RC	strings for the USA version.

Toolkit Demo

Toolkit Demo 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.

Toolkit Demo 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 **clsScrollWin** instance as the application's frame client window.
- Specify an application version number.
- Implement string literals as resources.

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 an instance of **clsScrollWin** as its frame client window. This lets the demonstration's windows be larger than the frame. A **clsScrollWin** window can have many client windows, but it shows only one at a time. So, Toolkit Demo 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. Toolkit Demo then switches to that window by sending **msgScrollWinShowClintWin** to the **clsScrollWin** window.

An interesting point is that, to avoid receiving messages while it is creating windows, Toolkit Demo 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**.

Files used

The code for Toolkit Demo is in PENPOINT\SDK\SAMPLE\TKDEMO. The files are:

METHODS.TBL the method table for the Toolkit Demo application.

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 hand-coded GO logo bitmap.

ICON.RES resource file for a smiling face icon, 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, and so on.

LBOXES.C code for making list boxes (**clsListBox**, **clsStringListBox**, and **clsFontListBox**).

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.**

JPN.RC strings for Japanese version.

USA.RC strings for USA version.

Input Application

Input Application is a simple application that demonstrates pen-based input event handling. As the user drags the pen around, Input Application draws a small square in the window. To provide this functionality, it creates a descendant of **clsWin** (**clsInWin**), which looks for pen input events.

Input Application's window tracks the pen by drawing a small box at the X-Y 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" low-level input event processing from their descendants. These classes send higher-level 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 and move events.
- Handle pen events (**msgPenUp**, **msgPenDown**, and so on) in a window.
- Use **msgBeginPaint** and **msgEndPaint** messages.
- Turn on message tracing for a class.
- Use resource files to associate application-specific strings with the tags provided for standard application resources.

Class overview

Input Application 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.

Files used

The code for Input Application is in PENPOINT\SDK\SAMPLE\INPUTAPP. The files are:

METHODS.TBL	the method tables for the classes.
INPUTAPP.C	the source code for the Input Application classes.
USA.RC	strings for the USA version.

Writer Application

Writer Application provides a ruled sheet for the user to write on. When the user lifts the pen out of proximity, Writer Application translates what the user has written, and places the translated text on the line below the ink. The user can change the translation algorithm from word-based, to text- or number-based.

Objectives

This sample application shows how to:

- Use **clsSPaper** and translation classes.
- Make a translator for words, text, or numbers only.
- Use the **clsXList** instance returned by the translation objects and how to interpret its data.
- Put a choice control in a menu.
- Implement all text strings as resources.

Class overview

Writer Application defines two classes: **clsWriter** and **clsWriterApp**. It makes use of the following classes:

clsApp
clsAppMgr
clsChoice
clsClass
clsMenu
clsSPaper
clsSysDrwCtx
clsXText
clsXWord

Files used

The code for Writer Application 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.
JPN.RC	strings for the Japanese version.
USA.RC	strings for the USA version.

Keisen Table Application

Keisen Table Application demonstrates the creation of a complex layout using toolkit tables, instances of **clsTkTable. clsTableLayout** (hence its subclass **clsTkTable**) positions items according to a grid, with rows and columns. However, Japanese keisen tables do not have a uniform, global structure. So, the table is broken up into a one column, eight row table representing the horizontal lines across the whole table. These rows are nested horizontal tables, each cell representing a line that divides the row. This nesting continues until the nested table has only **clsLabel** descendants as its children.

clsTkTable will create a default instance of a class. As this would be very limiting, it also provides style flags in the TK_TABLE_ENTRY structure to provide anticipated style settings, such as bold labels or word wrapping. However, not all circumstances can be meet with these flags, and so this application does not use them. Instead, subclasses which handle **msgNewDefaults** are created, so their default instance is exactly the style desired.

As well, to prevent having to make a new subclass for every style, **msgTkTableInit** is handled by some of these subclasses. This message is sent by a toolkit table when it is creating an object from a TK_TABLE_ENTRY. It is sent after all other processing (**msgNewDefaults**, altering styles, and so on), but before **msgNew** is sent to the class being created. Subclasses can handle this message by using previously unused fields of the TK_TABLE_ENTRY for their own custom style flags.

The grid around each cell in the table is another special feature. All leaf windows in the window tree—the nested table windows are containers which form nodes in the Keisen Table window tree and the labels and fields are windows which are leaves in the window tree—turn their borders on, and all tables turn their borders off. By growing these leaf windows to the size of their cell, these borders merge to make a grid.

To have the grid lines be a single line unit thick, the gap width is set to be -1 line units. This causes the borders of adjacent cells to overlap, forming a single line instead of one to either side of the cell division.

Objectives

Keisen Table Application demonstrates one way to use existing PenPoint layout facilities (**clsTableLayout** and its subclass **clsTkTable**) to build a very complex table.

This sample application also shows how to:

- Create a grid around table cells by overlapping children borders.
- Change default (new) style of a class by subclassing to handle **msgNewDefaults.**
- Have private style flags in TK_TABLE_ENTRYs for subclasses.
- Nest toolkit tables.
- Have a scroll window as the application frame's client window.
- Create private subclasses.
- Set compiler options to allow for Japanese characters in source code.

Class overview

Keisen Table defines several classes:

clsBoxedField private subclass of **clsField.** Handles only **msgNewDefaults** to set edges to **bsEdgeAll.**

clsBoxedKatakanaField private subclass of **clsBoxedField.** Handles only **msgFieldCreateTranslator** to disable translation of Hiragana, Kanji, Romaji, and numerals.

clsBoxedIntegerField private subclass of **clsIntegerField.** Handles only **msgNewDefaults** to set edges to bsEdgeAll.

clsHorizontalTable private subclass of **clsTkTable.** Sets default styles to be a single row table with children that grow to their cells and overlap by exactly one line unit so their border edges overlap. Handles **msgTkTableInit** (when being created from a toolkit table) to use **arg3** for private style flags.

clsKeisenTable **clsApp** subclass. Basically the same as Template Application, except it responds to fewer messages and handles **msgAppCreateClientWin** differently, creating a table in a scroll window as the client window instead of an icon window.

clsBoxedLabel private subclass of **clsLabel.** Handles **msgNewDefaults** to set edges to **bsEdgeAll** and center text in both dimensions. Handles **msgTkTableInit** (when being created from a toolkit table) to use arg2 for private style flags.

clsVerticalTable private subclass of **clsTkTable.** Sets default styles to be a single column table with children that grow to their cells and overlap by exactly one line unit so their border edges overlap. Handles **msgTkTableInit** (when being created from a toolkit table) to use **arg3** for private style flags.

It makes use of the following classes:

clsApp

clsAppMgr

clsClass

clsField

clsIntegerField

clsLabel

clsScrollWin

clsTkTable

Files used

The code for Keisen Table is in PENPOINT\SDK\SAMPLE\KEISEN. The files are:

METHODS.TBL the list of messages that the classes respond to, and the associated message handlers to call.

BFIELD.C the source code for the boxed field class.

BFIELD.H the header file for the boxed field class.

BINTFLD.C the source code for the boxed integer field class.

BINTFLD.H the header file for the boxed integer field class.

BKFIELD.C the source code for the boxed katakana field class.

BKFIELD.H the header file for the boxed katakana field class.

BLABEL.C the source code for the boxed label class.

BLABEL.H the header file for the boxed label class.

KEISEN.C the source code for the application class.

KEISEN.H the header file for the application class.

TABLES.C the source code for all table subclasses.

TABLES.H the header file for all table subclasses.

List Box Demo

List Box Demo is yet another sample application meant to help developers learn how to use some of the features of PenPoint. This demo does not implement a whole application. Instead, it shows, in an clear way, how to organize and program an application incrementally. In fact, List Box Demo was the starting point for a richer sample application called Video Player.

List Box Demo does not show lots of excess functionality, which makes it hard for developers to reuse and learn from the sample code. This release shows how to set up a window using UI toolkit components. The menu bar supports standard menus. The more interesting code is related to the list box. The list box window supports the copy/move protocol to move (shuffle) and copy rows within itself or any other window that supports the transfer of text. The list box also handles some gestures such as the caret gesture to insert a row and the scratch out or cross gesture to delete a row from the list box.

List Box Demo has the Unicode support required to run under PenPoint 2.0. However, to simplify things, strings are not stored in a resource file.

Objectives

This sample application shows how to:

- Embed data in the list box such as list contents and row state.
- Save and restore the list contents (filing).
- Handle gestures in a list box.
- Support the move and copy protocol.
- Deal with **theSelectionManager**.
- Create a graphical interface with the UI toolkit, including a list box.
- Add application-specific menus to the standard menus.

Class overview

List Box Demo defines two classes: **clsLBdemo** and **clsLBList**. It makes use of the following classes:

clsApp
clsAppMgr
clsClass
clsLabel
clsListBox
clsMenu
clsObject
clsXferList

Files used

The code for List Box Demo is in PENPOINT\SDK\SAMPLE\LBDEMO. The files are:

METHODS.TBL the method tables for the two classes used in List Box Demo.

LBDEMO.C the List Box Demo application class and the code to build the interface from the UI Toolkit.

LBDEMO.H header file for the application class.

LBLIST.C the definition of **clsLBList** and support methods for gesture processing and list manipulation.

LBLIST.H header file for **clsLBList**.

LBXFER.C code for moving and copying rows.

Sample Application Monitor

This sample application shows how to implement a reasonably sophisticated application monitor. The monitor allows for installing and deinstalling optional DLLs, selecting which stationery to load, and saving global options.

This sample looks for a DLL directory in the application directory for the optional DLLs, and looks in the normal STATNRY directory for stationery. It will not create option cards for DLLs or stationery if it doesn't find anything in these directories.

The same global option cards are used in the popup when the application is installed and in the document's option sheet.

This sample's makefile does not create the DLL or STATNRY directories. You can try out the optional DLL and stationery functionality by making these directories in PENPOINT\APP\SAMPLMON, and then putting any DLL or stationery document in the directories.

Objective

This sample application shows how to create a descendant of **clsAppMon**.

Class overview

Sample Application Monitor defines two classes: **clsSampleApp** and **clsSampleApp-Monitor.** It makes use of the following classes:

- **clsApp**
- **clsAppMgr**
- **clsAppMonitor**
- **clsButton**
- **clsDirHandle**
- **clsFileHandle**
- **clsOptionTable**
- **clsToggleTable**

Files used

The code for Sample Application Monitor is in PENPOINT\SDK\SAMPLE\SAMPLMON. The files are:

METHODS.TBL	the method tables for the two classes.
SAMPLMON.C	source for the classes.
SAMPLMON.H	header file for the classes.

Serial Transmission Demo

Serial Transmission Demo demonstrates the use of simple serial I/O from within PenPoint. Serial input is read by a semaphore-controlled subtask which places the received characters in a text view. The user enters text through an insertion pad; the output is then sent by the main task.

Objectives

This sample application shows how to:

- Query a service manager for available instances.
- Open and close a serial service instance.
- Read from and write to a serial service instance.
- Get and set serial metrics.
- Respond to connection messages from the service manager.
- Handle serial service events.
- Create a subtask.
- Create and use a semaphore.
- Create and use a list.
- Create an option sheet and interpret its values.

Class overview

Serial Transmission Demo defines two classes: **clsSX** and **clsSXView** (a subclass of **clsTextView**). It makes use of the following classes:

clsApp
clsAppMgr
clsButton
clsChoice
clsClass
clsCommandBar
clsIP
clsLabel
clsList
clsNote
clsObject
clsOptionTable
clsPopupChoice
clsString
clsTablelayout

Files used

The code for Serial Transmission Demo is in PENPOINT\SDK\SAMPLE\SXDEMO. The files are:

METHODS.TBL the method tables for **clsSX** and **clsSXView.**

SXAPP.C the source code for the application class.

SXAPP.H header file containing all definitions for Serial Transmission Demo; tags, messages, structures, handlers for **clsTextView** messages, and so on.

SXIP.C source code for handlers for **clsIP** messages.

SXOPT.C source code handlers for **clsOption** message.s

SXSER.C source code for all serial I/O handlers.

SXVIEW.C the source code for the **clsTextView** subclas.s

USA.RC strings for the USA version.

Note that you need to enable one or more serial ports in MIL.INI to be able to select a serial port. In the SDK, the spooler is available. When Serial Transmission Demo is started for the first time, it opens the first available service instance. If no serial service instance is available, the spooler service will be used. If no service instance is available at all, the application will exit.

User Interface Companion

The User Interface Companion (brings to life many of the diagrams in the *PenPoint User Interface Design Reference* manual. These diagrams show the various components that you can use to build the user interface for your applications: buttons, lists, menus, pop-up lists, text fields, and other controls. The User Interface Companion sample application focuses specifically on some of the more challenging controls.

The User Interface Companion consists of the application class, **clsUICompApp**, an initialization DLL, and the class definitions and methods for several classes.

Objectives

The UI Companion demonstrates how to:

- Build a multi-page application within the notebook.
- Create your interface objects ahead of time and file them via an INIT.DLL.
- Use the UI toolkit classes to implement some of the more complicated user interface controls available.
- Subclass a UI toolkit class to get specialized behavior.
- Extend the functionality of existing UI Toolkit classes.

Class overview

The UI Companion creates five classes:

clsUICompApp the UI Companion application class. The application is derived from Template Application, and handles many typical messages sent by the application framework. In addition, it implements a multi-page application. The UI Companion application creates an option sheet as its main window; this way the user can select different pages from the option sheet's built in title bar. It also creates its own sub-page control, and displays it at the far right of the application menu bar. The user can navigate through different pages in the application using this control. (See UICOMP.C for more information.)

clsUICompPage defines the behavior for a page in the UI Companion. It is a subclass of **clsTableLayout** which provides the specialized layout used in the UI Companion application. **clsUICompPage** is *not* intended to be a general-purpose class; it has a small set of rigidly defined behaviors and performs the minimum amount of work necessary to accomplish page layout for the UI Companion. It also isolates the layout code from the files which demonstrate how to create the specific toolkit components discussed in the *User Interface Design Reference.*

clsEdStrListBox a subclass of **clsStringListBox** which allows a user to edit the strings displayed in a string list box by making the standard circle gesture to pop up an edit pad. A user can also delete items from the string list box by making the X gesture over an item.

clsOptionCmdBar is a subclass of **clsOption** that allows a client to associate a custom, non-standard command bar with each card that is added to the option sheet.

clsScrollPopup a subclass of **clsPopupChoice** which implements a scrolling popup choice. The client can specify the maximum number of items to display. If the number of items in the choice is greater than the number of items to display, the items will appear in a scrolling window inside of the popup choice. This results in a substantial savings of screen real estate for choices with many options.

The UI Companion also makes use of the following classes:

clsApp
clsIconChoice
clsOptionTable
clsAppMgr
clsIconTable
clsPopupChoice
clsBorder
clsIconToggle
clsScrollWin
clsButton
clsIntegerField
clsStringListBox
clsChoice
clsIP

clsTableLayout

clsCommandBar

clsLabel

clsTextField

clsCounter

clsListBox

clsTkTable

clsField

clsMenu

clsToggleTable

clsFrame

clsMenuButton

clsIcon

clsOption

Files used

The code for the UI Companion is in PENPOINT\SDK\SAMPLE\UICOMP. The files are:

METHODS.TBL the method tables for the UI Companion application and **clsUIPage**, **clsEditStrListBox**, **clsOpCmdBar** and **clsScrollPopup.**

INIT.C function containing **InitMain**() entry point, which creates and files the user interface elements before the application is run, so they can be restored quickly at run-time.

UICOMP.C class definition and methods for **clsUICompApp**, the UI Companion application class.

UICOMP.H global defines, resource lists and tags used in the UI Companion application.

EDSTRLB.C class definition and methods for **clsEditStrListBox.**

EDSTRLB.H API definition for **clsEditStrListBox.**

OPCMDBAR.C class definition and methods for **clsOptionCmdBar.**

OPCMDBAR.H API definition for **clsOptionCmdBar.**

SCRPOPUP.C class definition and methods for **clsScrollPopup.**

SCRPOPUP.H API definition for **clsScrollPopup.**

UIPAGE.C class definition and methods for **clsUICompPage.**

UIPAGE.H API definition for **clsUICompPage.**

BUTTONS.C creates tools palette with different styles of buttons, square buttons, command buttons.

LISTS.C creates checklist, boxed lists, toggle switch, editable scrolling list.

MENUS.C creates menu with fill-in fields, multi-column menu, and a menu with a white gap between controls.

POPUPS.C creates popup with pictures as choices and a scrolling popup list.

TEXT.C creates overwrite field, scrolling field and insertion pad.

ALIGN.RES bitmaps illustrating different alignment options (used in popup choice variations).

ICONS.RES bitmaps for tool icons, and line and fill styles.

MISC.RES frog and prince bitmaps (used in toggle switch).

PATTERNS.RES different fill patterns (used in boxed list).

USA.RC United States English strings for USA version.

Basic Service

Basic Service is the absolute minimum code required to implement a service. This is the "Hello World" for service writers. Basic Service handles only one message: msgNewDefaults. Modifying Basic Service will help to get your service up and running in the shortest possible time.

For more complex services, see the Test Service and MIL Service 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

Files Used

The code for Basic Service is in PENPOINT\SDK\SAMPLE\BASICSVC. The files are:

METHOD.TBL	method table for **clsBasicSvc**.
BASICSVC.C	**clsBasicSvc**'s code and initialization.
BASICSVC.H	header file 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 Basic Service and MIL Service 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

Files Used

The code for Test Service is in PENPOINT\SDK\SAMPLE\TESTSVC. The files are:

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**.
USA.RC	strings for USA version.

MIL Service

MIL Service is a full implementation of a MIL (Machine Interface Layer) service. This MIL service implementation shows how a client will interface with the MIL service, how to create and use a ring 0 DLL to interface with a MIL device used, and how to perform connection detection. The MIL device used is the parallel printer MIL device. This sample code is a version of the PPORT MIL service code shipped with PenPoint.

For other examples of services, see the Basic Service and Test Service samples.

Objectives

This sample service shows how to make a service (the makefile differs from application makefiles).

Class Overview

MIL Service defines one class: **clsTestMILService**. It makes use of the following classes:

clsClass

clsMILService

clsStream

Files Used

The code for Test MIL Service is in PENPOINT\SDK\SAMPLE\MILSVC. The files are:

METHOD.TBL	method table for **clsTestMILService**.
MILSVC.C	**clsTestMILService**'s ring 3 code and initialization.
MILSVC.H	header file for **clsTestMILService**.
MILSVC.BAT	batch file used to build both ring 0 and ring 3 DLLs.
MILSVC0.C	**clsTestMILService**'s ring 0 code.
MILSVC0.H	private header file for **clsTestMILService**.

INDEX

In addition to page references within this book, this index also contain references to files and directories in the sample code (in \...\PENPOINT\SDK\SAMPLE). These are identified by the directory, or in some cases, the directory and file where you will find the information. The sample code for the While You Were Out (WYWO) application is not shipped with the SDK, but is available on CompuServe. We remind you of this by adding CS: before WYWO.

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:

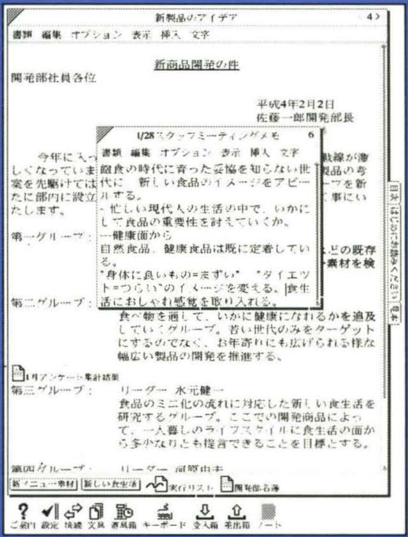
Team Manager, Developer Documentation
GO Corporation
919 E. Hillsdale Blvd., Suite 400
Foster City, CA 94404–2128

Or fax it to: (415) 345-9833

Operating Systems

PenPoint® Application Writing Guide

The *PenPoint Application Writing Guide, Expanded Second Edition* teaches you how to write an application for the PenPoint operating system. The information applies equally well to versions 1.0 and 2.0 of the PenPoint operating system. The book 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 and GO's coding standards, the book shows 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.


新製品のアイデア
4
書類 編集 オプション 表示 挿入 文字
1/28スタッフミーティングメモ
6
書類 編集 オプション 表示 挿入 文字
目次
はじめにお読みください
巻末
1月アンケート集計結果
新メニュー素材
新しい食生活
実行リスト
開発部名簿
ご案内 設定 接続 文具 書類箱 キーボード 受入箱 差出箱 ノート

新商品開発の件

開発部社員各位

平成4年2月2日
佐藤一郎開発部長

今年に入っ … 戦線が激
しくなっていま … 製品の考
案を先駆けては … ープを新
たに部内に設立 … く事にい
たします。

第一グループ： … などの既存
… 素材を検

第二グループ：
食べ物を通して、いかに健康になれるかを追及していくグループ。若い世代のみをターゲットにするのでなく、お年寄りにも広げられる様な幅広い製品の開発を推進する。

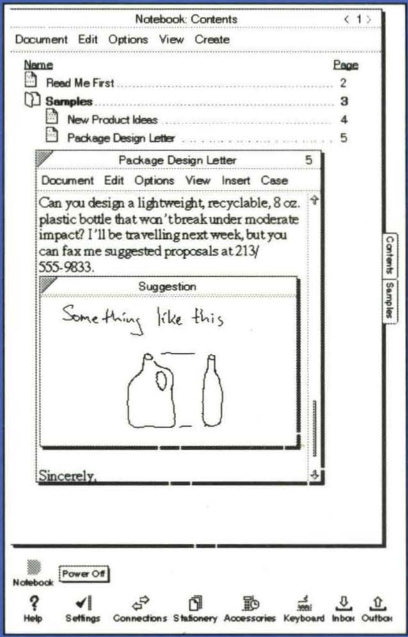
第三グループ： リーダー 水元健一
食品のミニ化の流れに対応した新しい食生活を研究するグループ。ここでの開発商品によって、一人暮しのライフスタイルに食生活の面から多少なりとも提言できることを目標とする。

第四グループ： リーダー 前原由圭

飽食の時代に育った妥協を知らない世代に　新しい食品のイメージをアピールする。
- 忙しい現代人の生活の中で、いかにして食品の重要性を訴えていくか。
—健康面から
自然食品、健康食品は既に定着している。
"身体に良いもの=まずい" "ダイエット=つらい"のイメージを変える。食生活におしゃれ感覚を取り入れる。

Two new sections in the book describe how to write applications that can be easily adapted to international markets and how to localize applications to the Japanese market. Three additional new sections supplement the PenPoint SDK documentation for version 2.0.

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.


Notebook: Contents
< 1 >
Document Edit Options View Create
Package Design Letter
5
Document Edit Options View Insert Case
Suggestion
Something like this
Contents
Samples
Notebook
Power Off
Help Settings Connections Stationery Accessories Keyboard Inbox Outbox

Name	Page
Read Me First	2
Samples	3
New Product Ideas	4
Package Design Letter	5

Can you design a lightweight, recyclable, 8 oz. plastic bottle that won't break under moderate impact? I'll be travelling next week, but you can fax me suggested proposals at 213/555-9833.

Sincerely,

Other volumes in the GO Technical Library are:

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

53295
9 780201 622997
ISBN 0-201-62299-8

USA $32.95
CANADA $42.95