IBM

International Technical Support Centers

PenPoint Operating System Overview and Application Development

GG24-3978-00

PenPoint Operating System
Overview and Application Development

Document Number GG24-3978-00

February 1993

International Technical Support Center
Boca Raton

Take Note!

Before using this information and the product it supports, be sure to read the general information under "Special Notices" on page xiii.

First Edition (February 1993)

This edition applies to PenPoint IBM version, Release Number 1.0 of for use with the IBM 2521 ThinkPad.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

An ITSC Technical Bulletin Evaluation Form for readers' feedback appears facing Chapter 1. If the form has been removed, comments may be addressed to:

IBM Corporation, International Technical Support Center
Dept. 91J, Building 235-2 Internal Zip 4423
901 NW 51st Street
Boca Raton, Florida 33431-1328

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Abstract

This document describes the architecture of the PenPoint IBM version, operating system providing sample code for application developers. The PenPoint IBM version executes on IBM ThinkPad 700T and special bid machine IBM ThinkPad 2521. This document provides an overview of the functional capabilities of the operating system and the process of application development.

This document is intended for system engineering personnel and application developers who need to know how to implement pen-based systems. A knowledge of C programming and object oriented programming techniques is assumed.

PS (122 pages)

Contents

Figures

Tables

Special Notices

This publication is intended to assist system engineering personnel and application developers in understanding and developing applications for using the PenPoint operating system and the IBM ThinkPad. The information in this publication is not intended as the specification of any programming interfaces that are provided by PenPoint and the associated Software Development Toolkit. See the PUBLICATIONS section of the IBM Programming Announcement for PenPoint IBM version for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM (VENDOR) products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The following terms, which are denoted by an asterisk (*) in this publication, are trademarks of the International Business Machines Corporation in the United States and/or other countries:

IBM
Operating System/2
OS/2
Personal System/2
PS/2

The following terms, which are denoted by a double asterisk (**) in this publication, are trademarks of other companies:

HAYES is a registered trademark of Hayes Microcomputer Products, Inc.
GO is a trademark of GO Corporation

GO logo is a trademark of GO Corporation
PenPoint is a trademark of GO Corporation
PenPoint logo is a trademark of GO Corporation
ImagePoint is a trademark of GO Corporation
EDA is a trademark of GO Corporation
GOWrite is a trademark of GO Corporation
MiniText is a trademark of GO Corporation
MiniNote is a trademark of GO Corporation
PenTOPS is a trademark of Sitka Corporation
PenCentral is a trademark of Sitka Corporation

Preface

This document is intended to provide the reader with information about the architecture of the PenPoint operating system and provide the application developer with guidelines and sample code to assist in writing applications to this operating system. It contains a description of the operating system architecture and functions as well as sample code.

This document is intended for persons requiring an understanding of the operating system and the programming techniques involved in developing applications for PenPoint.

How This Document Is Organized

The document is organized as follows:

- Chapter 1, "Introduction" provides an overview of the general concepts and capabilities of the PenPoint operating system.
- Chapter 2, "PenPoint User Interface" examines the Notebook User Interface.
- Chapter 3, "PenPoint Kernel" describes the operating system kernel and associated services.
- Chapter 4, "Application Framework" describes the function of the Application Framework which permits the interaction between the operating system and installed applications, supporting common application behavior.
- Chapter 5, "Application Embedding" describes the process of embedding, or nesting documents inside one another, discussing the correspondence between applications and their associated documents.
- Chapter 6, "ImagePoint" examines ImagePoint and the use of drawing contexts, clipping and graphics primitives.
- Chapter 7, "File System" describes the hierarchical file system and the role of the file system within the application framework.
- Chapter 8, "Input and Handwriting Recognition" describes the process of handwriting recognition and translation.
- Chapter 9, "The Windowing System" examines the windowing system, describing the concepts employed in using windows.
- Chapter 10, "Service Manager" examines the operating system component that coordinates the operations of applications that facilitate communication with hardware devices.
- Chapter 11, "Connectivity" examines the connectivity features available with PenPoint and describes the transport and link layers.
- Chapter 12, "Software Installation" discusses the software installation process.
- Chapter 13, "Application Development" provides an overview of the Software Developer's Toolkit.
- Chapter 14, "Sample PenPoint Application" provides a brief overview of the sample PenPoint application. The source code for this sample is included in Appendix A.

Related Publications

The following publications are considered particularly suitable for a more detailed discussion of the topics covered in this document.

- *ThinkPad Getting Started with PenPoint*, S41G-3122-00
- *ThinkPad Using PenPoint*, S41G-3111-00
- *PenPoint Architectural Reference Vol 1*, ISBN 0-201-60859-6
- *PenPoint Architectural Reference Vol 2*, ISBN 0-201-60860-X
- *PenPoint User Interface Design Ref*, ISBN 0-201-60858-8
- *PenPoint Application Writing Guide*, ISBN 0-201-60857-X
- *PenPoint Development Tools*, ISBN 0-201-60861-8
- *PenPoint API Reference Vol 1*, ISBN 0-201-60862-6
- *PenPoint API Reference Vol 2*, ISBN 0-201-60863-4
- *The Power of PenPoint*, ISBN 0-201-57763-1
- *PenPoint Programming*, ISBN 0-201-60833-2
- *WATCOM C Library Ref for PenPoint*, ISBN 1-55094-035-X
- *WATCOM C Language Ref*, ISBN 1-55094-033-3
- *WATCOM C/386 Optimizing Compiler and Tools*, ISBN 1-55094-0 ISBN 1-55094-034-1

International Technical Support Center Publications

A complete list of International Technical Support Center publications, with a brief description of each, may be found in:

- *Bibliography of International Technical Support Centers Technical Bulletins*, GG24-3070.

Acknowledgments

The advisor for this project was:

Alex Gregor
International Technical Support Center, Boca Raton

The authors of this document are:

Dwight Ronquest
ISM South Africa

Gert Ehing
IBM Germany

This publication is the result of a residency conducted at the International Technical Support Center, Boca Raton.

Thanks to the following people for the invaluable advice and guidance provided in the production of this document:

Maura Oehler
Experimental Development Software IBM Boca Raton

Rick Abbott
Pen Planning IBM Boca Raton

Shirley Tomasi
Pen Planning IBM Boca Raton

Thomas Loeffler
International Technical Support Center Boca Raton

Frank Cook
Software Development Relations IBM Boca Raton

David Lybrand
Experimental Development Software IBM Boca Raton

Chapter 1. Introduction

This chapter provides an overview of the general concepts and capabilities of pen-based systems concentrating on the PenPoint IBM version operating system.

1.1 The Pen-Based Environment

Pen-based computing with the associated tablet systems hardware introduces a significant shift from existing computing paradigms. The pen-based paradigm satisfies a number of key user requirements:

1. The user interface must provide support for the use of a pen, not as a pointing device, but as an input device.
2. The applications developed for pen-based systems must support both handwriting translation and pen gestures.
3. Detachable networking and deferred data transfer whereby the user may make or break networking connections at will, without impacting the performance of the operating system.
4. Both operating system and applications must be developed to run on lightweight mobile computers.

Key hardware design elements for a tablet system to support the pen-based operating environment include:

- Portable and battery powered
- A 32-bit processor complex
- Upgradeable DRAM memory packages
- Upgradeable SSF (Solid State Files) - A SSF card is a lightweight removable storage device with a 10MB capacity
- Integrated digitizing subsystem to support handwriting input
- System I/O support for:
  - Diskette drive ports
  - Parallel port
  - Serial port
  - Data/facsimile modem support

1.2 Pen Operating Systems

Pen-based operating system design has followed two distinct routes: either as an extension to existing operating systems such as DOS, or employing a new architecture based on 386 processor complex instruction sets, using object oriented programming techniques.

Extensions to existing operating system environments essentially replace the standard pointing device, the mouse, with a pen. The standard Microsoft Windows** Graphical User Interface (GUI) is used and many many existing Windows applications can be modified to run in this new environment.

Handwriting is not the primary input mode and tends to be restricted to specified edit environments.

PenPoint is a 32-bit, object-oriented, multitasking operating system, specifically designed for pen-based computing. The change in the nature of the input device, from traditional keyboard to pen, requires that an enhanced GUI be presented to the user.

The Notebook User Interface (NUI) adopts the same concepts used in traditional GUIs in terms of pull-down menus and icons plus a number of new elements to support a pen as the primary input source, such as:

- Notebook metaphor
- Gestures
- Handwriting recognition and translation

1.2.1 Notebook Metaphor

Information is organized and presented as a collection of pages and sections. Individual objects, sections and pages may be readily selected.

The bottom of the display is reserved for the ***Bookshelf*** which contains systemwide objects and resources:

- Notebook
- Online Help
- Stationery
- Accessories
- Keyboard
- In/Out box
- Connection services
- Settings
- Shutdown

1.2.2 Application Framework

PenPoint's NUI provides a standard set of pen gestures that work consistently across all applications. Pattern recognition is performed by the operating system, while the application controls the translation process.

Applications written for PenPoint must adhere to PenPoint's Application Framework, which is a set of protocols that define application structure and common behavior:

- Gesture recognition and response
- Copy and move data transfers
- Live embedding of other applications
- View-data model
- Installation and configuration
- Creation of application instances
- Online Help

- Document properties
- Spell checking
- Search and replace
- Printing
- Import/export file formats

A Service Manager supports background server applications such as databases and networking connections. Applications interrogate PenPoint as to the presence of service, establishing message-passing connections to these services. Applications, not the user, save their internal state in a directory in the file system.

The Application Framework implements an Embedded Document Architecture (EDA) that enhances the NUI, making many of the traditional operating system tasks transparent to the user. Key elements of this architecture are:

- Document model
- Live application embedding
- Hyperlinks

1.2.3 Document Model

The user is relieved of the task of launching applications and loading/saving application data. The user simply moves from page to page, viewing the data in the state it was left, as if the application was still running. With the exception of data transfer, the user does not work with separate files and applications.

1.2.4 Live Application Embedding

EDA provides the facility to embed a live instance of one application inside another application. All PenPoint applications provide compound document capability without special programming. The receiving application simply embeds an instance of an application that is capable of editing and displaying the particular piece of data. The user may mix and match applications seamlessly.

1.2.5 Hyperlinks

PenPoint gestures will create hyperlinks that turn pages and scroll documents to the location selected when the hyperlink was created. Hyperlink buttons may be placed anywhere inside the Notebook, documents and in the Bookshelf area.

1.3 PenPoint Applications

PenPoint provides a single built-in application, the MiniText editor that is a pen-aware formatted text editor.

The standard for PenPoint application distribution is 1.44MB, 3.5-inch DOS diskettes.

PenPoint senses the application distribution diskette in the diskette drive and will display an application installation dialog. Upon confirmation, the application code and resources will be installed.

1.4 Connectivity

PenPoint supports multiple auto-configuring network protocol stacks that may be dynamically installed without rebooting the system. Networking connections may be established and broken at will.

Connections to physical devices are detected automatically. Once the connection is complete operations will be initiated; for example, when connected to a network, PenPoint sends a message to all services that utilize network connections. Documents waiting to be printed to a network printer will begin transmission. The ***Out Box*** facility permits the initiation of file transfers and print requests. It is a central, extensible queueing service for all connection dependent transfer operations. Destination addressing is managed via PenPoint's address book APIs.

The ***In Box*** facility supports download of mail and facsimile.

PenPoint's Connections Notebook provides the NUI for connection management, supporting the following functions:

- Disk management - interrogate drives and manage files
- Eject or dismount disks
- Format disks
- Browse networks and enable network resources
- Create and edit instances of printers
- Initiate document import/export to create:
  - PenPoint documents from non-PenPoint files on disk
  - Non-PenPoint format files of PenPoint documents

PenPoint's file system provides support for reading and writing DOS formatted disks. All PenPoint specific information is stored as a DOS file, in a DOS directory. This approach will be used when mapping to other file systems.

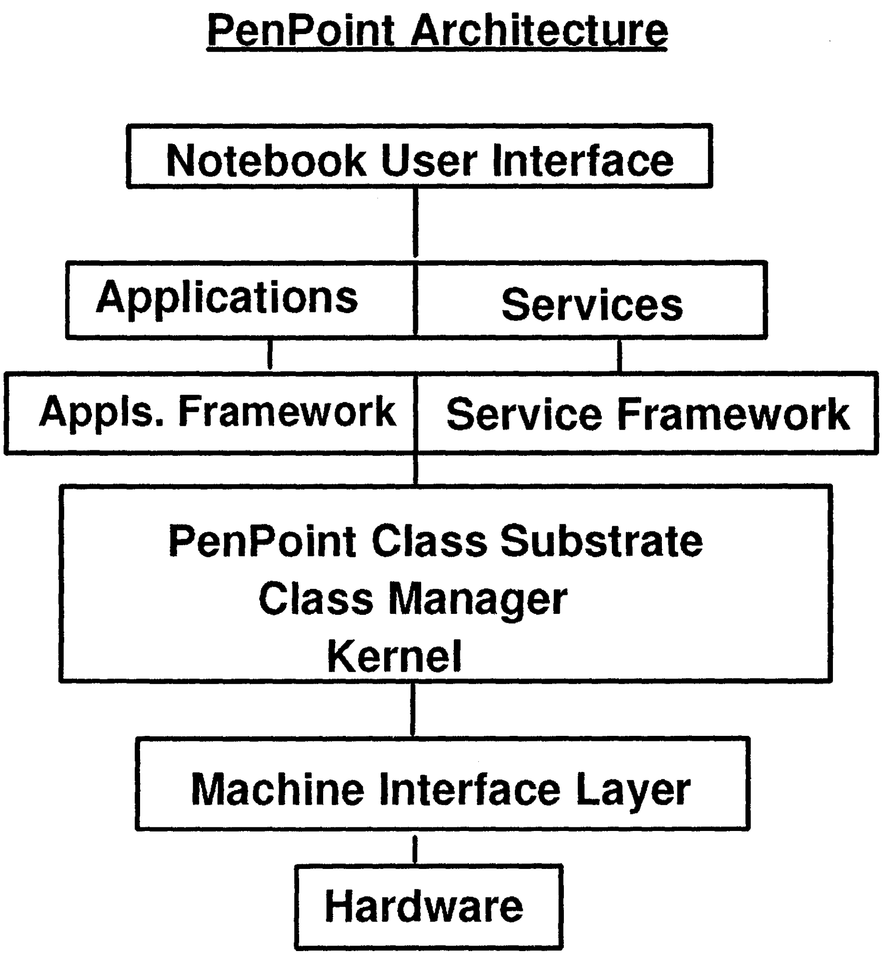


Figure 1. Schematic of PentPoint Architecture

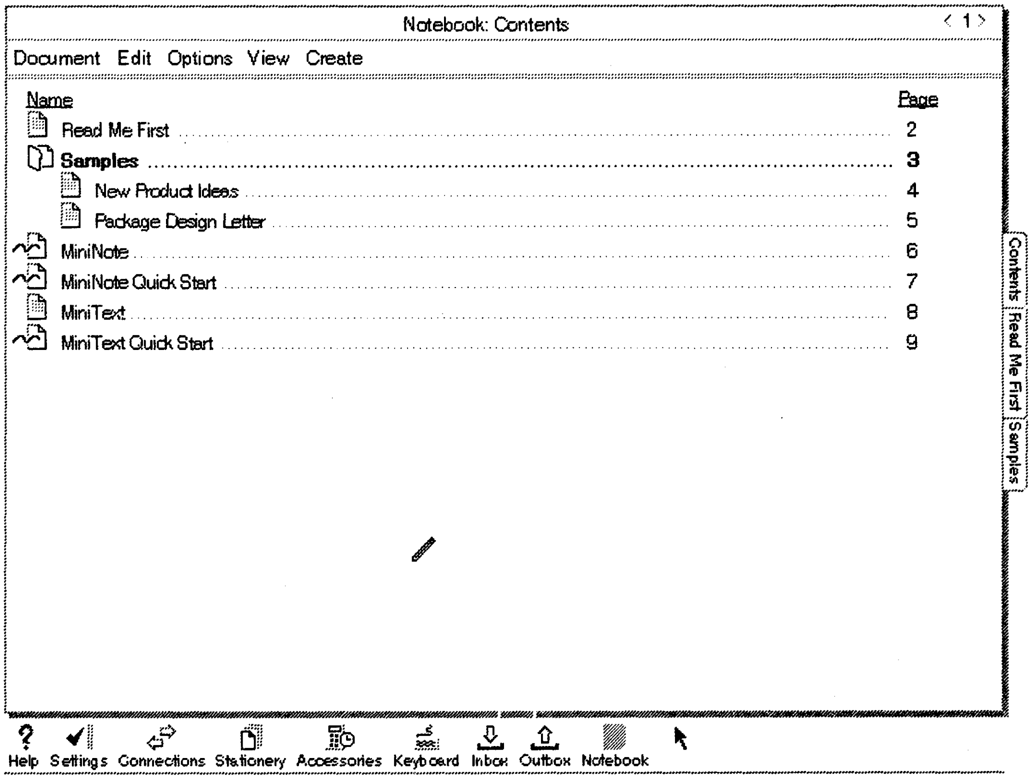
Chapter 2. PenPoint User Interface

This chapter explores PenPoint's Notebook Metaphor and the organizational principles employed in delivering the Notebook User Interface (NUI).

2.1 User Interface

The NUI follows many of the principles used in traditional Graphical User Interfaces (GUI).

PenPoint applications run inside a window and can share the screen with other applications. The windows are referred to as **Document Frames** and may be resized and repositioned; Notebook pages are an exception to this principle.



Name	Page
Read Me First	2
Samples	**3**
New Product Ideas	4
Package Design Letter	5
MiniNote	6
MiniNote Quick Start	7
MiniText	8
MiniText Quick Start	9

Figure 2. PenPoint's Notebook User Interface

In addition to pull-down menus, **Option Sheets** are used to specify global type options:

- Orientation
- Paper size
- Margins
- Fonts - style and size

Options are applied whereas commands are executed within pull-down menus.

PenPoint introduces two additional items, the **Tab** and the **Writing Pad.**

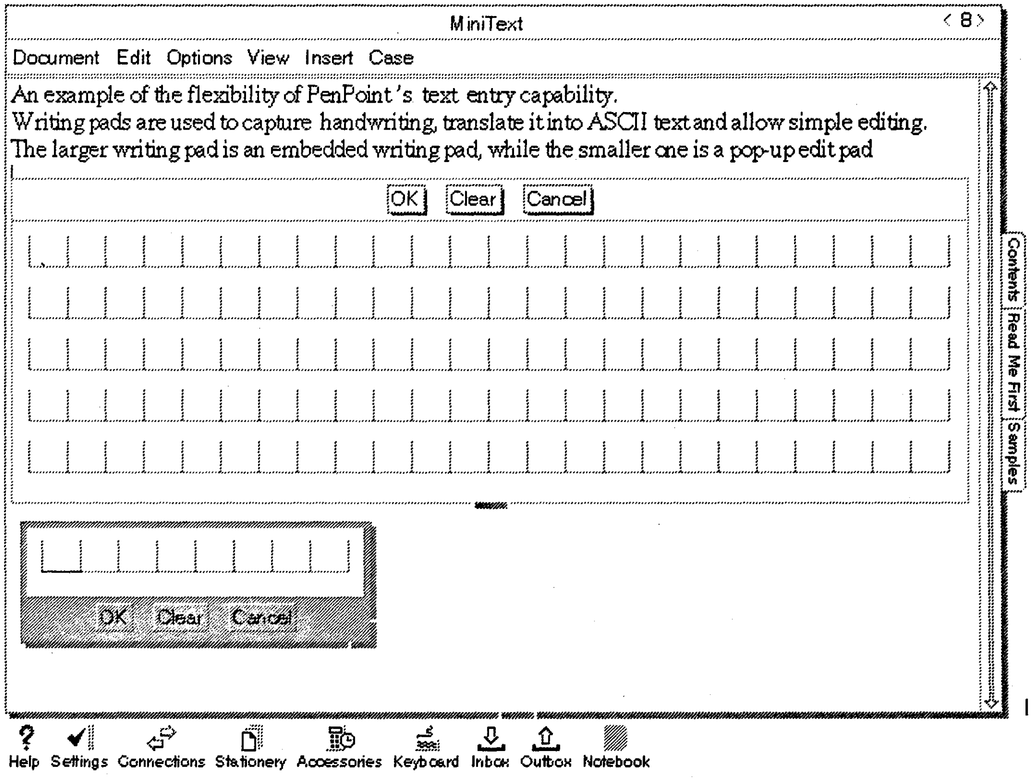
The Tab is used as a navigational tool within an application. The user selects a tab to switch between screens or sheets.

Writing Pads are used to capture and translate handwriting and to perform simple editing. System preference settings provide a choice of either boxed, or ruled styles of pad. Boxes require separation of characters and consequently yield higher recognition rates. Ruled lines permit the user to write characters closer together which may pose recognition problems where characters are not clearly written.

Two forms of writing pads are available, **Embedded** and **Pop-Up.**

The Embedded Pad is used for large amounts of text. The application provides space around the pad so that preceding and succeeding context is visible to the user while writing on the pad.

Pop-Up Pads are optimized for small amounts of text and typically float at or near the location where the pad was requested. The application does not shift its display as with the Embedded Pad.



An example of the flexibility of PenPoint's text entry capability.
Writing pads are used to capture handwriting, translate it into ASCII text and allow simple editing.
The larger writing pad is an embedded writing pad, while the smaller one is a pop-up edit pad

Figure 3. PenPoint's Writing Pads

All writing pads are the same object, merely appearing with various default sizes in response to user commands. Pressing the **OK** button causes the entered text to be translated; the user may also edit the text, making corrections and insertions. A second depression of the **OK** button causes the text to be placed in the underlying application.

2.2 Notebook Metaphor

The metaphor is based on an organizing principle of a table of contents, sections, pages and tabs in a notebook. User data exists as pages.

Pages are numbered in the top-right corner. The page is turned in either direction by tapping the direction indicators with the pen. Notebook tabs are located on the right-hand side of the notebook and may be attached to any page or section, selection of the tab results in the specific section/page being displayed.

There are no file load or file save commands. From the user perspective, the concept of programs and data existing as separate entities does not apply. Each page of the notebook is a **Document** and is viewed as a "running" application at the point where it was left by the user. At a processing level, the Application Framework associates data files with application code and operating system processes. At this level ***documents*** are synonymous with ***application instances.***

The act of turning a page in the Notebook, causes the following operating system instructions to be executed:

1. Clear the screen.
2. Create a process and application object for the destination page.
3. Send a message to the destination application object to restore its saved state from the file system.
4. Send a message to the destination application to display itself.

The original application files its data and this process is terminated to reduce memory consumption.

2.3 Bookshelf

The Bookshelf is situated at the bottom of the screen and contains systemwide objects and resources that are displayed as icons. The standard PenPoint Bookshelf contains the following:

Note

The default PenPoint operating system has been enhanced to include IBM specific facilities including hardware diagnostics. Different release/version levels of the product may include extra or changed resources.

This document is based on PenPoint IBM Version 1.0a, HWX revision 32 Mil 51.05

- Online Help
- System settings
- Accessories
- Stationery notebook
- Connections
- Software keyboard

- In/Out box
- Selected notebook
- Shutdown

The stationery, online help and in/out boxes use floating instances of notebooks as a user interface.

System settings provide a number of configuration options:

- Writing style
- Pen alignment
- Fonts and layout
- Float and zoom
- Date and time
- Sound
- Power conservation parameters

Each of the options selected must be applied before they take effect.

Accessories provides a pop-up window with a number of icons:

- Thinkpad Diagnostics
- System Log
- Corrective Service Facility
- Clock
- Keyboard
- Connections

The Stationery Notebook contains copies of templates for installed applications.

The Connections resource provides various views on connected disks, directly attached or networked, and printers.

The software keyboard is a pop-up image of the keyboard that may be tapped with the pen tip to insert characters.

Chapter 3. PenPoint Kernel

This chapter describes PenPoint's multitasking kernel, resource ownership and allocation.

3.1 Multitasking

PenPoint is a 32-bit, preemptive multitasking operating system similar in function to OS/2*. The basic role of the kernel is management of resource allocation and ownership. The kernel arbitrates over two general types of resources:

- Time resources - CPU execution time
- Space resources - Memory and I/O ports.

The kernel's interface consists exclusively of functions and is the least object-oriented component of the operating system. However the kernel has a Class Manager which provides the object oriented interface of classes and messaging. Together these two components provide the Application Programming Interface (API) structure for the operating system.

3.1.1 Task Management

A task in PenPoint is defined as any executing thread of control. Software tasks are subdivided into processes and subtasks that are scheduled and run by a software scheduler based on a priority scheme. The only hardware tasks available to PenPoint are interrupts. A process is the first task that runs when an application is instantiated and requests local memory. Processes own all the resources used by the application, including memory, subtasks and the semaphores used in locking and interrupt management. When the process terminates, all its resources are returned to the system.

A subtask is a thread of execution started by a process and is owned by the process. Subtasks have the following characteristics:

- Shares local memory with the parent process
- Owns no resources
- Has separate registers and stack
- Subtasks can lock semaphores and send/receive messages.

The software task scheduler manages the initiation and execution of the processes and subtasks. To start a process the kernel creates a new execution context consisting of local memory, a local instance pointer to the executable code and a new stack; the data values are then initialized.

A process may be started by another process or subtask and there is no hierarchical relationship between processes; that is, a process that creates another process does not own the created process:

- The created process will not terminate when the "creator" process is terminated.
- The created process can be associated with other processes at any time.

3.1.2 Memory Management

The key distinction between PenPoint and operating systems such as DOS and OS/2 is that all the components of the operating system, all applications and all the application data are kept in RAM.

The kernel uses privilege settings to determine which of the various tasks and processes has access to which memory and other space-related resources.

Memory may be private to a process, or may be global. Global memory is shared by all processes and any task can allocate memory in the global area of memory. The Memory Manager manages global memory usage through identifiers and counters that track the number of instances of which application processes are sharing a given piece of global memory. PenPoint exploits the 80386 processor complex linear memory using a flat memory model in which heaps may be created and memory may be allocated within the heaps.

3.1.3 Multitasking

PenPoint employs a preemptive multitasking approach. Preemptive multitasking is transparent to the application. The kernel switches CPU time among a number of processes and can regain control of the CPU even if the application crashes.

Approaches to Multitasking

There are two approaches to multitasking:

- Yield-based
- Preemptive

In yield-based multitasking the applications must follow a defined set of processing rules that requires the application to periodically yield control back to the kernel. However if the application crashes while in control of the CPU, the operating system and all other applications will also crash because the kernel cannot regain control of the CPU. This is the approach adopted by Microsoft Windows 3.0.

In a preemptive multitasking environment the operating system is able to preempt the execution of a task and regain control of the CPU.

PenPoint always gives a higher priority to on-screen applications compared with off-screen pages and applications. Tasks of the same priority share the processor, (time-slicing).

Most PenPoint applications are a single process. The applications do not typically contain separate subtasks and do not use the operating system's task management scheme. PenPoint single-threads all of the applications with the operating system, input and other executing applications. There is no true concurrency between two live applications.

Where applications do require separate subtasks, the application must use the kernel's task management and intertask communication routines to avoid deadlock. PenPoint supplies the semaphore architecture to support this requirement.

3.1.4 Operating System Reliability

The reliability of the operating system revolves around the following elements:

- Protection of the kernel
- Enabling the operating system to survive an application crash
- Enabling the operating system to recover from a crash

PenPoint's protection scheme concentrates on **inadvertent misbehavior,** rather than on malevolent software (viruses). Hardware-level protection schemes are employed to protect the operating system core objects from accidental alteration.

The preemptive multitasking approach allows the operating system to regain control of the processor thereby permitting an orderly shut down of the crashed application while maintaining overall system integrity.

If the operating system itself crashes a warm-boot is required to recover the system. All running processes are shut down, and resources, including dynamic memory are cleared.

In PenPoint, executable code exists only in the system's memory. This single copy of code is shared by all instances of an application. Each of the documents owns a pointer to the executable code and keeps track of where in the execution process it last stopped. All instances of the application are preserved by the operating system.

3.1.5 Date and Time Services

The kernel includes an alarm subsystem that maintains a queue of alarm dates and times that will be active even if the hardware is switched off, as long as the batteries are charged and installed.

3.1.6 General Kernel Services

The following general functions are included in the kernel:

- Addition and subtraction
- Multiplication and division
- Trigonometric and logarithmic functions
- Conversion between floating-point and fixed numbers

3.1.7 Class Manager

PenPoint is an object-oriented operating system, using a class manager to support object-oriented programming. The class manager is used to create classes and class hierarchies, to create and destroy objects or class instances, to inherit functions from other objects and to define and send messages between objects. The APIs are based on class manager messages and objects.

PenPoint however does not use an object-oriented programming language. Applications are typically developed under C and are therefore portable between C compilers.

3.1.8 Machine Interface Layer (MIL)

The Machine Interface Layer provides PenPoint with hardware platform independence. It is that portion of the operating system that is specific to a particular hardware platform.

The MIL roughly corresponds to BIOS in a traditional personal computer. Whereas BIOS supports a fixed number of a known collection of device types, for example ports, or disk drives, the MIL can support an unlimited number of devices and extensions to the MIL can be supported in either RAM (Random Access Memory), or ROM (Read Only Memory).

Each MIL implementation supports a number of devices, with a minimum set required by PenPoint. All devices support a set of common functions and a number of device-specific functions via which PenPoint and the MIL communicate.

Each type of device is assigned a constant, the **Device ID.** There can be more than one device for a given device ID. During initialization, all devices are enumerated and assigned a **Logical ID.** This ID is arbitrary and will vary between machines and between machine configurations and each device can support one or more units.

Requests from PenPoint to the MIL are sent by the kernel, or the MIL Services. MIL requests which are implemented as device functions, fall into two categories, those requests that ***complete,*** for example, reading a block of data from a disk, and ***continuous*** requests, such as reading keystrokes from the keyboard.

Continuous requests are associated with asynchronous input events. The result of this event is returned to an event handler or call-back function with PenPoint.

A request to the MIL progresses through one or more stages. Upon completion of each stage, the MIL returns the request to PenPoint, indicating when to return to the MIL for further processing. The processing stages are determined by the particular implementation of the MIL. All continuous functions are multi-staged processes.

Multi-stage requests are driven to completion by the following events:

- Specific interrupts
- Time delay
- Completion of an 80386 real mode (virtual 8086) task.

3.1.8.1 PenPoint - MIL Communication

At power on, PenPoint via queries to the MIL builds data structures which are used for communication with MIL devices. These data structures include **Function Transfer Tables (FTT)** that contain:

- An array of function descriptors, one for each function that the device supports
- Device blocks containing the public and private variables for each device
- A common data structure that holds pointers to the device block and FTT of a particular logical device
- Common data used by both the MIL and PenPoint

Chapter 4. Application Framework

This chapter describes PenPoint's Application Framework layer that provides a set of classes defining the protocols that make up an application.

4.1 Function of the Application Framework

The Application Framework defines the protocols to implement common application behavior:

- Installation of the application
- Creation of application instances
- Activation of an instance of an application
- Saving and restoring application data
- Deleting application instances
- Removal of applications

Applications running under PenPoint may be viewed from the following separate but related elements:

- Application display
- Application file directory
- Application process
- Application object

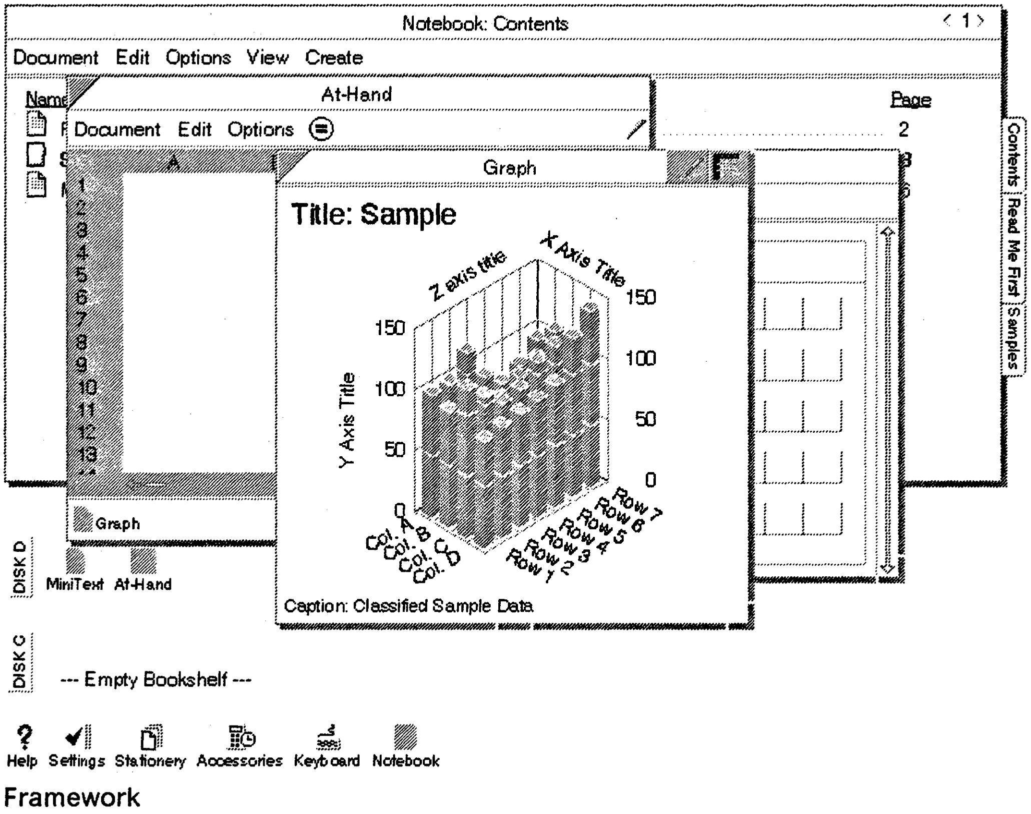


Figure 4. Example of Multiple Live Applications

Figure 4 shows a number of live applications:

- Notebook
- Notebook Contents
- A text-editing application
- A graphics application
- Bookshelf

The Notebook uses the file system to organize its documents so that they parallel the structure of the Notebook Table of Contents. Each section and document has its own directory in the file system. If a document is contained in a section, the document's file entry is a subdirectory of the section directory. If a document has an embedded document, the embedded document's directory is a subdirectory of the enclosing document's directory. The Bookshelf is similar to a section, acting as a repository for all of the top-level subdirectories and documents in the Notebook.

A process is associated with each running application. The Application Framework manages the processes in accordance with the application life cycle. The Application Framework creates the process and sets up the application object to receive messages. When the user launches another application or closes the application, the Application Framework destroys the process and saves the data.

4.1.1 Application Elements

All Penpoint applications contain a number of standard elements:

- Application code
- Document directory
- Document process
- Application object
- Resource files
- Main window

4.1.1.1 Application Code

Application code does not share memory with the PenPoint file system where instances of applications and related data are stored. The operating system and applications share a special area of RAM that is protected against accidental erasure. Application instance data is linked to the application by a global unique identifier (UID).

Note

Disk-based operating systems require two copies of code:

- The unrelocated executable file on disk
- At execution time - a relocated copy of the executable code in memory.

PenPoint installs the application in memory; there is no requirement for a disk-based copy. All application code is reentrant; therefore a single copy of code supports a number of application instances.

4.1.1.2 Document Directory

Documents are instances of the applications that created them. Each document has a corresponding directory. When a document is selected, PenPoint determines from the directory information which application created the document. An instance of the application is then created and activated.

4.1.1.3 Document Process

The Application Framework manages the application processes for each active application. Each process has a number of attributes:

- A message queue that stores messages for the application instance, until they can be forwarded to the appropriate object within the process
- Entry point defining the means by which process startup takes place
- A main routine, which is the event loop within which the program starts the application life cycle, waiting for a user event to which it should respond
- The method table that maps message names to method handlers, that is where the names of messages to which the application responds locally are related to the names of the procedures that contain the responses

4.1.1.4 Application Object

The application object responds to messages sent to it via processes associated with the object. The application's function is contained in the object's structure and processing.

All application instances are objects and because of their inheritance, all application instances receive and process messages from the Application Framework.

4.1.1.5 Resource Files

A resource file is a general purpose storage mechanism, the format and content of which are application dependent. All application instances have at least one associated resource file that is the repository for all objects created by the application. The Resource Manager manages the location of objects on request, taking into consideration both space allocation and compaction.

4.1.1.6 Application Window

All visible applications must have at least one window, the main window that displays the data relevant to the application and provides the user with an input environment.

4.1.2 Standard Behavior and Inheritance

The Application Framework ensures user interface consistency across all applications. This consistency is a direct spin-off from the object-oriented approach and inheritance. The following elements are common to all Penpoint applications:

- Installation behavior
- Creation of new application instances
- Online help
- Document properties
- Move/copy
- Gesture recognition
- Hyperlinks
- Standard application menu support
- File import/export
- Printing support
- Spell checking
- Search/replace
- Application stationery

Gesture recognition is discussed in chapter Chapter 8, "Input and Handwriting Recognition" on page 41.

4.1.2.1 Application Installation

PenPoint provides a set of installation routines that are consistent for all applications. Penpoint detects installable software in attached diskette drives. The installation options are selected from the Settings Notebook. Applications may be installed, deinstalled, or deactivated.

Deactivation is used in a constrained disk environment, and the application is temporarily deinstalled. The application will be automatically reinstalled when the user selects an instance of the application, assuming that an external diskette drive containing the application is attached.

4.1.2.2 Creation of New Application Instances

PenPoint creates a new instance of the application when the user launches the application. The Application Framework sends the application a message to create an instance of itself. The application creates a subdirectory entry at the appropriate place in the file system. If the new instance is created in the Notebook Table of Contents, it will not run automatically upon creation. However if the instance is created within another document, the application will launch immediately.

4.1.2.3 Online Help

PenPoint provides two approaches in providing online help:

- Quick Help (context sensitive help) for individual objects
- Additions to PenPoint's Help Notebook (reference help).

Quick Help is provided by defining resources for each type of object for which assistance is required. Protocols interpret the user's help gesture within the menu, decode the object and display the appropriate help resource.

Reference help is added to the Help Notebook in the form of text files that are managed by the application, or in the form of help applications that are embedded in the Help Notebook.

Text files are generally in Rich Text Format (RTF), placed in the appropriate subdirectories in the application distribution diskette. The default help application is used to display the online help and manages user interaction within the help files.

Help applications are placed in the appropriate subdirectories in the application distribution diskette and are detected and installed by the installation manager.

4.1.2.4 Document Properties

All documents have associated properties:

- Title
- Author
- Comments
- Date created
- Fonts used

The user selects an Option Sheet describing the current document and may change selective attributes. Option Sheets are defined for an application using messages; this option is included in the PenPoint User Interface Toolkit.

4.1.2.5 Move/Copy

Users may initiate a move/copy process between embedded windows by selecting information to be moved or copied and then issuing the move or copy command via menu or gesture. Once initiated, a move/copy icon, (an elastic box), will surround the source; the user then indicates the destination location. There are now two locations for the object. The source object sends a message to the destination object instructing it to move or copy the selected data. The destination determines the type of data and if the data type is understood, the source is requested to send the data. The source now knows the type of the destination object and determines whether the data is be copied or moved into an instance of that type of application or not. (A graphics application might reject the movement of its data into a word processing application.) The source determines the location of the destination object in the file system.

4.1.2.6 Hyperlinks

Goto buttons or hyperlinks may be defined within documents and are used to create cross references to other documents, or sections of the same document.

4.1.2.7 Standard Application Menu Support

Standard menus, menu commands and Option Sheets are provided by PenPoint as a default. These are collectively referred to as PenPoint Standard Application Menus (SAMS). The User Interface Design Guidelines require that all applications provide SAMS related commands. The Application Framework implements SAMS.

When an application is launched, it creates a user interface including the menu bar. SAMS will either merge application unique menus, or menu commands, or the application performs the merging. There are default responses for all SAMS commands. The following menus are fully implemented to display standard dialogs and Option Sheets:

- Document menu
- Print
- Print Setup
- About

If a selection option is not available within a menu, it is automatically grayed out. For example, in the Edit menu the **Delete** command will not be active until a deletion area has been selected.

4.1.2.8 File Import and Export

Data import from operating systems such as DOS requires that the user identify the application that will deal with the imported data. PenPoint creates an instance of that application and sends a message instructing the application to translate the imported data into its own file format.

When exporting files, the user selects an export file format that can be understood by the receiving application under a different operating system.

4.1.2.9 Printing

Printer support is furnished at a system level, rather than at an application level. In PenPoint, printing is a process of drawing a document's image on a hardcopy device. There is a single API which renders output to both the screen and printers. Printed pages are a collection of windows and the application merely displays itself to a different image device; the printer image device is provided by the operating system.

PenPoint provides two commonly used printer drivers, a Printer Control Language (PCL) driver for laser printers and a standard dot matrix driver. The printer objects must be created using the Connections Notebook. This notebook lists the printer description, including printer type, model and port. Multiple printer objects for the same printer may be created to exploit any special print features.

The Document menu for every application contains standard Print and Print Setup commands, that bring up Option Sheets. The Option Sheet controls which printers to use, page size, number of copies, fonts etc.

Print commands may be issued at any time, even if there is no printer attached. The print command will copy the document into the Out Box.

When the targeted printer is available:

1. PenPoint creates an image device for the specified printer.
2. The document in the Out Box receives a message with parameters set to indicate that the application is being opened for printing.
3. The printer image device root window is laid out with optional headers and footers, and with the applications first page of data.
4. If a dot matrix printer has been selected, the fully rendered page image is sent to the printer in bands.
5. If a PCL printer has been selected, ImagePoint** downloads outline fonts to the printer.
6. The page layout and print process repeats for as many pages as the application has data to print.
7. Upon completion of the print job, the Out Box deletes its copy of the document.

The Application Framework prints a document exactly as it appears on screen by default. When a document is selected for printing, all embedded documents are printed as well. Only the visible portion of the embedded document is printed. Applications may provide print-specific formatting and layout.

4.1.2.10 Spell Checking

Spell checking is part of PenPoint's SAMS. New dictionaries may be defined. Spell checking is part of the handwriting component of the operating system.

4.1.2.11 Search and Replace

The operating system provides support for this function. The process however may be complicated through embedding of documents. The user is given the option of excluding embedded documents when setting up the search.

4.1.2.12 Application Stationery

The Stationery Notebook contains the stationery provided by the installed applications. This supports the creation of blank application instances along with default templates.

Users may define new templates and file them as stationery documents. Upon selection of the document, PenPoint will create an instance of the associated application.

4.1.3 Application Life Cycle

All PenPoint applications follow the same life cycle each time an instance is created. The application manages the transitions between the various states in the life cycle in response to the Application Framework messages it receives. The messages are caused by user actions but may also be programmatically generated.

The application life cycle consists of the following states:

1. Non-existent
2. Created
3. Activated
4. Opened and interactive
5. Dormant

PenPoint Application Life Cycle

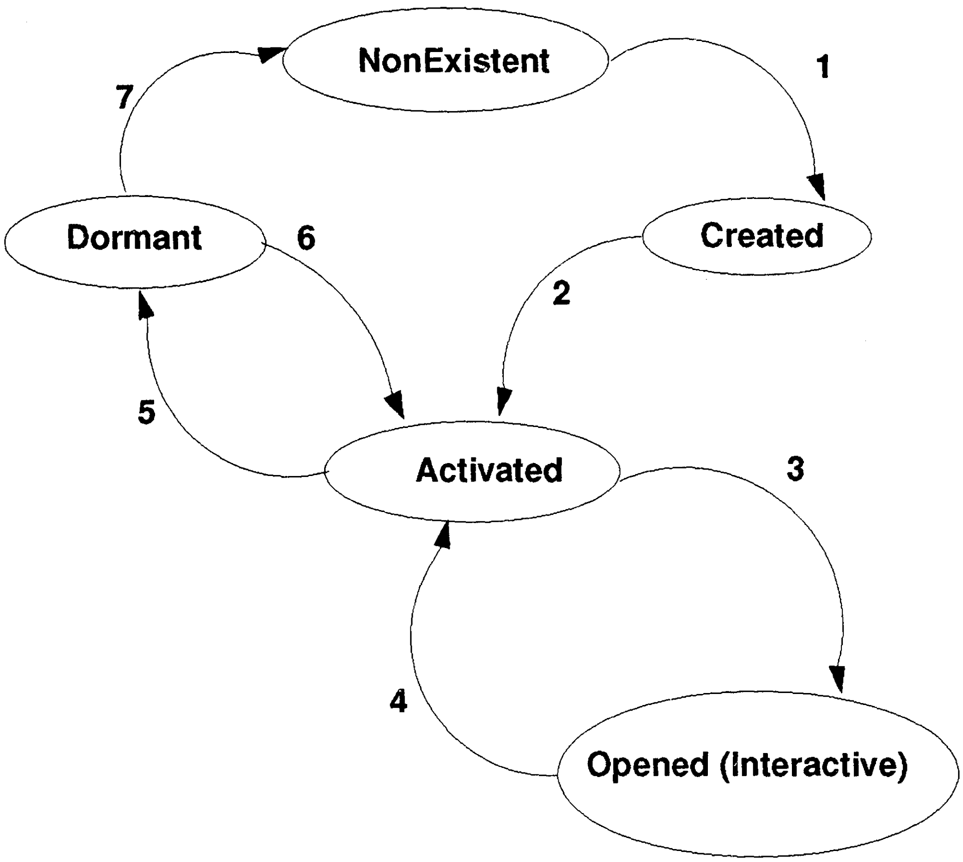


Figure 5. Schematic of a PenPoint Application Life Cycle

4.1.3.1 Instance Creation

The user creates a new instance by selecting the application, or by copying a blank stationery template associated with the application to the Notebook Table of Contents. PenPoint sends a message which is defined to the application class.

4.1.3.2 Application Activation

Selecting the page containing an instance of the application indicates to the operating system that the instance should be activated. Once activated, the document is opened.

4.1.3.3 Application Termination

PenPoint may terminate the application instances when they are closed, except where the application has been defined as a **Hot Mode** application and off-screen processing will continue, for example, file transfer.

When a document is deleted, PenPoint will remove the application instance associated with the document.

Chapter 5. Application Embedding

This chapter describes the Application Framework's support for application embedding and the concepts employed in embedding documents.

5.1 Application Embedding Concepts

Every document has a unique identifier that indicates to the operating system which application to launch when the document is selected. This embedding capability is recursive; the user may embed an application within an application the only limiting factor being memory.

Document embedding takes place dynamically while the application in which another application is about to be embedded continues to run. The embedding process is transparent to the user.

Embedded applications execute inside one another, and the embedded document's application continues to run, that is any application can host another application. This approach is in contrast with traditional Windows, multi-application processing, where compound document support is delivered via clipboard metaphors.

In PenPoint, the user can combine documents from two or more applications in either of two ways:

- Create the documents on separate Notebook pages, then copy or move them to a destination document, with no loss of content or formatting.
- Embed the applications, creating new documents within existing documents as required to build the compound document.

In both approaches, the applications appear in embedded windows that can be inline, overlapping, or take up the entire application frame.

Each embedded document is stored in a separate directory. Embedded documents are therefore stored in a subdirectory of the parent's document file system subdirectory. PenPoint treats every embedded application as a cohesive whole containing all its embedded windows, regardless of the application responsible for the content of the windows.

5.1.1 Basic Concepts of Document Embedding

Document embedding involves three basic dimensions:

- File system
- Process space
- Window hierarchy

The file system provides a hierarchy of document directories, with one document directory for every embedded application, regardless of whether it is running or opened. It is the only dimension that captures the entire hierarchy of embedding at all times.

Process-space consists of processes corresponding to running application instances. This set of running processes is driven by the embedding hierarchy, in which every running, embedded application has its own process.

Terminated applications do not have a process associated with them. Processes do not have an hierarchical relationship with other processes. Processes are created in response to transversals of the hierarchy in the file system.

The window hierarchy captures the hierarchical visual relationship of the "embedding" and "embedded" applications.

5.1.1.1 File System Hierarchy

The file system contains a document directory for every embedded application; each document directory contains the files belonging directly to the application instance - data and resource files. The attributes record the class of the application, associated with the directory, the Unique Identifier of the application (UID), if it is running, the state of the application and the Universal Unique Identifier (UUID) of the document itself that allows other objects to point uniquely to this document.

5.1.1.2 Process Space

Every running application must have a process. The windows within each process are owned by the process. Off-screen processes are shut down by the operating system to free up memory and processor time.

5.1.1.3 Embedded Windows

Windows are the objects that support the cooperative sharing of the display space. The embedding window communicates with the embedded window via messages to determine the location and size of the embedded window and will intercept, approve, or modify messages to the embedded window.

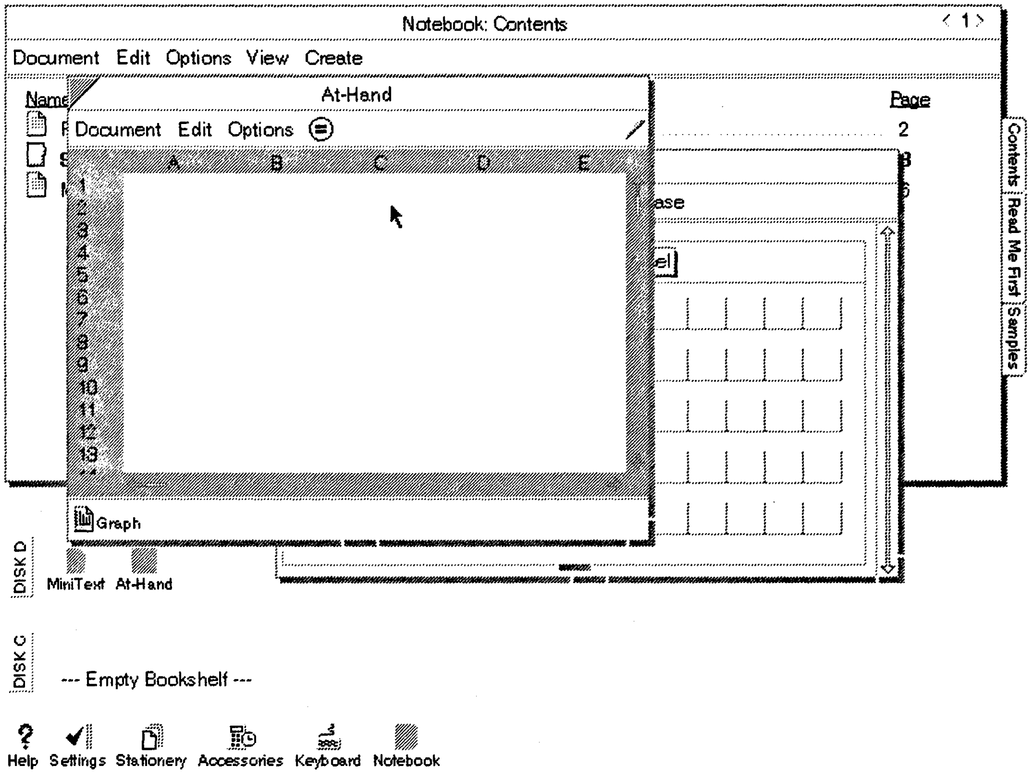


Figure 6. Example of Embedded Applications

In Figure 6, a graphics application is embedded within a text editing application. The graphics application is closed and is represented by an icon. When the user selects the Graph icon application, the text editing application intercepts the message and cooperates with the graphics application in terms of sharing resources. Processes define the class of application - embedding or embedded, and the behavior of the application. Default behavior is to run all embedded applications whenever the embedding application is run, that is, when the user turns to a page, not only is the document on that page run, but the corresponding processes for all embedded documents are also opened.

Note

Embedded applications are costly in terms of memory and processor cycles because each embedded application requires a separate process and document directory.

PenPoint components however, (Goto buttons and Signature Pads), execute within the host application's process space; filing data in the host application's data file does not consume as much resource.

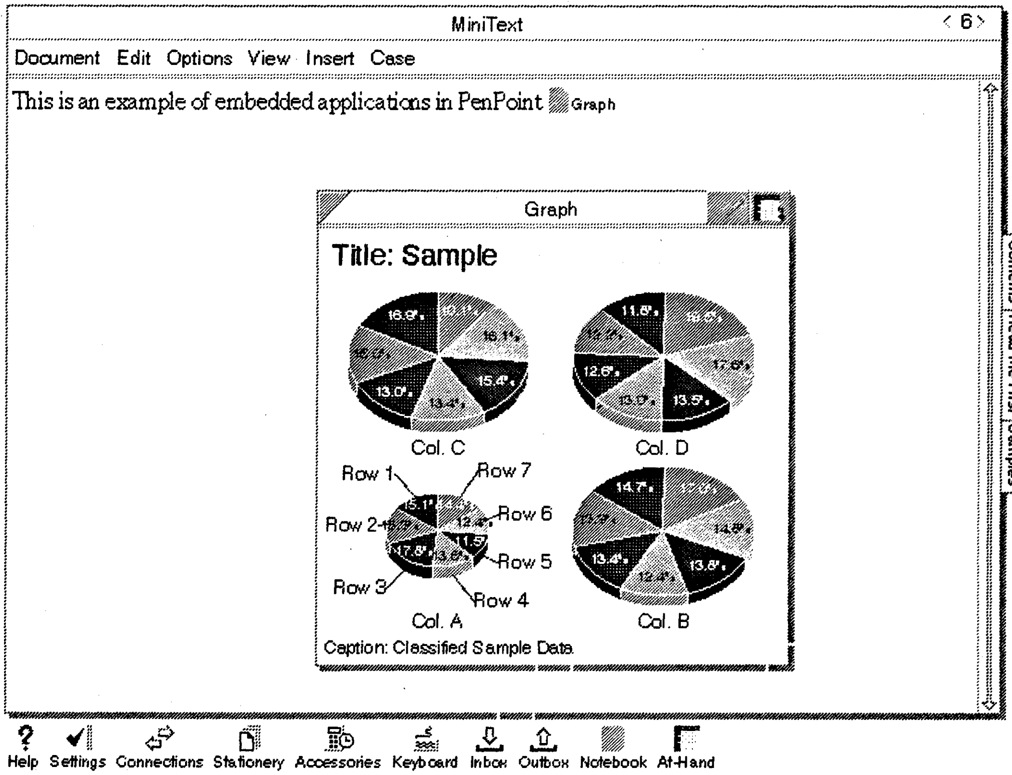
PenPoint keeps track of embedded documents using embedded window marks that contain:

- The UUID of the document containing the mark
- The UUID of the component within the document
- A component-specific token that specifies a location within the component

- A label of the mark

5.1.2 Document Embedding - Example

The Notebook application is embedded in the Bookshelf, and acts as the organizing vehicle for all PenPoint applications.



This is an example of embedded applications in PenPoint Graph

Title: Sample

Caption: Classified Sample Data.

Figure 7. Notebook Displaying Embedded Applications

Figure 7 represents a number of applications visible on the screen, making up a compound document:

- Bookshelf
- Notebook
- Notebook Contents
- Text Editing
- Graphics

The compound document may also be viewed from a file directory structure. The Bookshelf is the root directory. The Notebook application is a subdirectory within Bookshelf, and Notebook Contents is in turn a subdirectory of Notebook. Each document within each directory has at least two associated files, one is the contents of the file, the other records the display state of the file.

5.1.2.1 Window Placement

The placement of embedded windows (documents) may be performed in one of two ways:

1. Unconstrained placement
2. Constrained placement

The default approach is unconstrained placement. The new window floats atop its main, or parent document window.

When using constrained placement, the application determines where to place the embedded window, based on the kind of display item it represents.

5.1.2.2 Traversal

The parent application must be aware of the contents of the windows created and embedded by other applications, as the contents of the windows determine the behavior of that window in response to document operations. In PenPoint this process is known as traversal. The most common operations are:

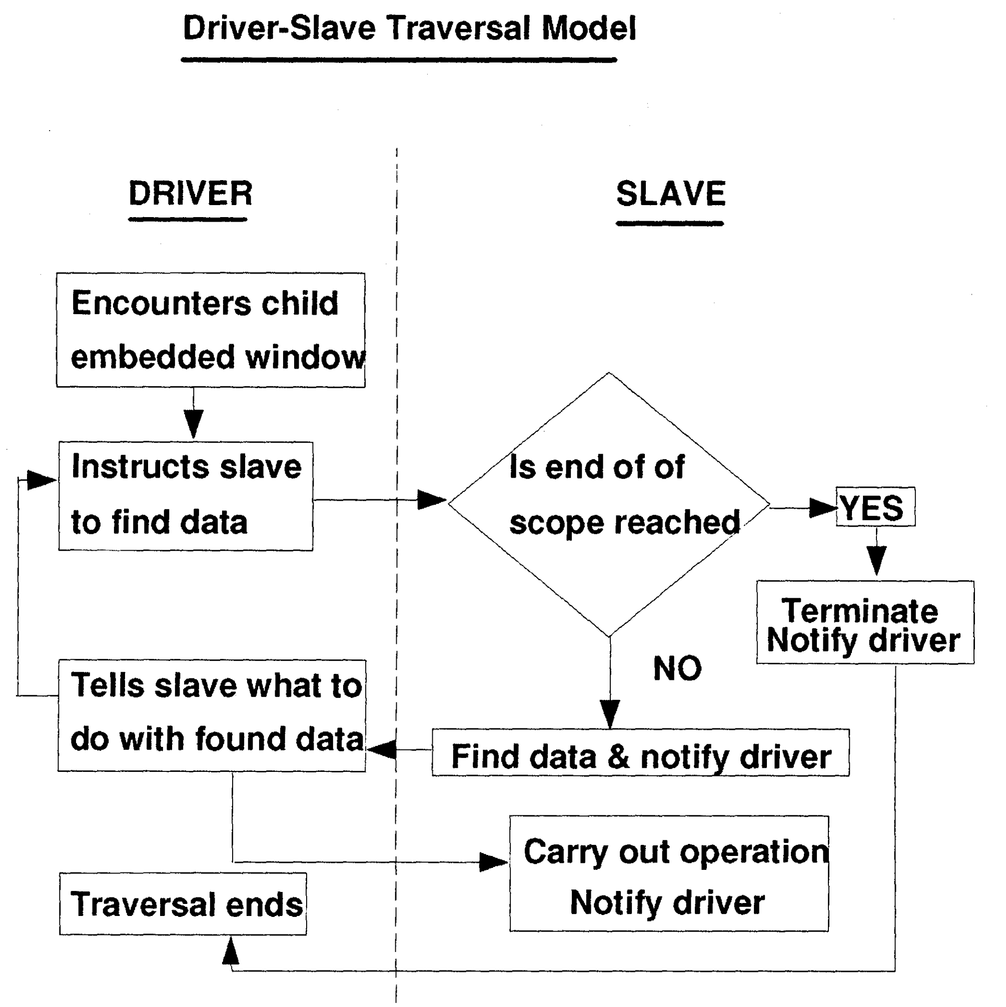
- Search and replace
- Spell check
- Print

Traversal determines whether embedded windows will be scanned for data or not. PenPoint uses a driver-slave model to implement traversal. The object, (application) requesting traversal is termed the ***driver*** and interacts with embedded documents, the ***slave,*** to scan all data within a specified range. The driver-slave model keeps the traversal process in synchronization through a mechanism called the traversal context. This context is a protocol between the driver and all the slaves it encounters within the scope of the traversal, the direction and the current location within the scope of the traversal.

The user determines the scope or method of traversal by choices or commands. The driver sends messages to each slave it encounters, the slaves respond to the messages, in accordance with the traverse style defined in the traverse context.

The following types of behavior can be defined by an application for its instances:

- Embedded windows are not scanned.
- Open embedded windows are scanned.
- All embedded windows are scanned.
- Invoke a call-back routine.



Driver-Slave Traversal Model
DRIVER
SLAVE
Encounters child embedded window
Instructs slave to find data
Is end of of scope reached
YES
Terminate Notify driver
NO
Tells slave what to do with found data
Find data & notify driver
Carry out operation Notify driver
Traversal ends

Figure 8. Driver-Slave Traverse Model

Chapter 6. ImagePoint

This chapter describes the implementation of graphics and imaging in the operating system. PenPoint is based on a graphical user interface tuned for pen input.

In PenPoint, text is unified with graphics and all images can be scaled, rotated, translated and used for both display and printing.

6.1 Graphics and Imaging System

All drawing takes place in a window. Drawing messages are sent to a special object, the ***Drawing context.*** The Drawing context renders the drawing in the window to which it is connected. Clipping is enforced to ensure that the drawing affects only the target window.

A drawing context defines the characteristics of a graphic environment:

- Units of measurement - pixels, points, millimeters
- Matrix to define scaling and rotation
- Type and extent of clipping. The default is to clip to the window boundaries
- Plane Mask - does the window draw on the acetate layer where the pen ink gets dribbled
- Line characteristics - joining of lines and line thickness
- Radius value for round-cornered rectangles
- Foreground/background colors
- Fill patterns
- Line patterns
- Fonts

PenPoint supports the drawing of a number of specified shapes. Both closed and open shapes are supported and the closed shapes may be filled with a solid color or shape.

A closed shape is any shape that starts and ends at the same point, enclosing an area. The following closed shapes are supported:

- Rectangles, including rounded corners
- Ellipses and circles
- Polygons with an arbitrary number of sides
- Sectors
- Chords

An open shape is essentially a line; the following shapes are supported:

- Multi-segment lines
- Bezier curves
- Arcs

Drawing becomes visible to the user when the application responds to a PenPoint message to paint in a specified window. Applications make no distinction between painting and repainting a window. The application must however store the contents of the windows and keep track of what is displayed.

6.2 System Drawing Context

All drawing in PenPoint is performed by sending messages to the System Drawing Context (SysDC). The SysDC is bound to one window at a time. If the SysDC is bound to another window, the drawing in the first window is not cleared, it is simply not updated.

6.2.1 Creating a System Drawing Context

The SysDC is created in the same manner as any other objects in PenPoint by sending messages to the class. The SysDC may be considered as the set of values that describe the state of the environment of the window to which the SysDC is attached. The operating system provides a number of defaults for the state of the window, and messages are available to change each of the values.

Table 1. Default SysDC Elements

Element	Default Value
Units	Unit=point (1/72 of an inch)
Drawing mode	Keep narrow lines visible
Plane mask	Do not draw into acetate layer where ink is dribbled
Line cap	Square off ends of lines
Line join	Use miters for line joins
Line thickness	One point
Foreground color	Black ink
Background color	White
Fill pattern	White
Line pattern	Black
Font scale	One unit
Default font	None

6.2.2 Binding a SysDC to a Window

The SysDC is sent a message containing the ID of the window to which it should be bound. PenPoint will also return the ID of the window, if any, with which the SysDC was formerly associated.

The SysDC must first be bound to a window before most of its associated messages will have a meaning.

A single window may be bound to multiple SysDCs if a complex picture is to be created.

6.2.3 Drawing and Storing with a SysDC

The application process follows two steps to draw in a window:

1. Confirms that the graphics state of the window is correct, that is, the SysDC has the values required to perform the drawing.
2. Sends the SysDC the drawing messages to create the shapes.

The window must be repainted for the drawing to become visible to the user.

When the window is filed, only the information required to re-create the window is stored. Three approaches are available to store the window display contexts:

- Capture a bitmap of the window. This is a memory intensive and device dependent approach.
- Store the application data structures from which the display can be regenerated.
- Store the images as a sequence of SysDC drawing instructions.

PenPoint's preferred method of storing drawings is as a sequence of SysDC drawing instructions. A special object, **PicSeg** (Picture Segment) records all the drawing commands issued and stores them in a compressed format - a **Grafic.**

Each grafic in a PicSeg contains the information to reproduce a single drawing action.

PicSeg is a subclass of a SysDC and is created and bound to a window in the same manner in which the SysDC is setup.

6.2.4 Clipping and Repainting Windows

The SysDC only draws inside a window; any instructions to draw outside the window boundaries are automatically clipped. The PicSeg captures all drawings; even the clipped portion which is not displayed is captured.

Where there are overlapping windows, clipping will ensure that a drawing does not disrupt the contents of the overlapping windows.

The clipping region may be altered to allow a number of embedded windows to share a common drawing area, or a subset of the window's total area may be defined as a drawing area and all drawings will be clipped to the defined area.

In summary, all drawing takes place through the SysDC and appears within a clip region that can be:

- An entire window
- A defined family of windows that cooperate to create a common drawing area
- A portion of a window

PenPoint notifies an application when a window requires repainting. This usually occurs when windows are moved and overlap the drawing area. The operating system sends a message to the application to repaint the window and the application initiates the repaint process. Thereafter, only the affected (dirty) area of the window is repainted, until the application issues an "end repaint" message.

This approach of confining repaints to specific target areas is the most efficient in a multiple overlapped windows environment and reduces screen flicker to a minimum. Windows may be repainted at any time during the application process, via routines in the application without having to wait for the operating system to issue the instruction.

6.2.5 Graphics Primitives

PenPoint defines:

- Open and closed graphics primitives
- Displaying text
- Copying rectangles of bits.

Open shape Polylines are drawn by sending a message that takes as an argument, a pointer to an array containing the points through which the line is drawn and a number defining the number of points in the array.

Bezier curves are drawn by passing a pointer to an array of four points that act as the control points for the curve.

PenPoint treats an arc as a portion of an ellipse, defined by a rectangle enclosing the ellipse of which the curve is a part and the two points that define the end points of the arc.

Six basic PenPoint graphics are defined that produce closed shapes. All closed shapes are filled; a transparent color fill is supported to generate "hollow" shapes. By setting the line width to zero, only the filled area, not the border, will be drawn.

Rectangles are drawn by pointing to a data structure that specifies the origin and size of the rectangle. A value other than zero for the radius produces rectangles with rounded corners.

Ellipses and circles are drawn by passing an argument in which the ellipse is drawn within a rectangle. If the rectangle is a square, a circle is produced.

6.2.6 Text Primitives

All text is drawn using a font and the process of drawing text consists of the following:

1. Load the required font.
2. Scale the font.
3. Initialize the structure that defines the parameters to draw a single text string. The structure defines:
    - Alignment
    - Underlining
    - The pointer to the text
    - Length of the text
    - The coordinates to place the text string (x and y axes)
    - Justification metrics - width of a normal space
4. Drawing the text

6.2.7 Copying Pixels

ImagePoint** supports two forms of pixel copy operations. Rectangular pixel images may be moved between image devices and within a window.

Image devices are in-memory windows into which the drawing operation is rendered. This image may then be copied to the screen.

A portion of the on-screen image may be relocated to another screen position. This technique is employed when the application window has been relocated on the screen, or the contents of the window scrolled.

6.3 Color Graphics Interface

PenPoint supports the use of color on appropriate hardware. The PenPoint color metaphor uses the concepts of foreground and background. Drawing takes place in the foreground, as the pen draws lines, using a color that contrasts with the background color and therefore produces visible output. A single drawing operation can use both the foreground and background colors at the same time.

Color values may be set and described in the following manner:

1. Use a palette of colors and index the selected colors into it. colors
2. Use red, green and blue (RGB) color combinations. This is probably the most effective approach:
   - Ensures printer compatibility. If the first method is used, a printer may not recognize the palette.
   - PenPoint automatically manages color translation from RGB color values to the appropriate colors on a minimum color hardware configuration.
   - Ensures device independence for both printers and displays.

6.4 Prestored Images

PenPoint includes high level support for managing and displaying prestored images, for example facsimile (FAX), in the form of a Sampled Image Operator (SIO). The SIO handles simulated analog image processing, by mapping the source pixels into destination pixels. The SIO also supports run length compressed sources, permitting easy scaling and rotation of pixel-based images.

Note

Pixel based images are generally not editable

6.5 Fonts

PenPoint stores fonts as outlines which is far more memory efficient than using bitmapped images of the font characters and are scalable to any point size.

When the font is specified in the SysDC, PenPoint searches for the closest match using font metrics. If the application has a bitmapped font with the same name and/or ID, the operating system will use that font.

All fonts have standard registered 16-bit IDs that are valid across PenPoint configured systems; therefore, moving applications between systems should not result in font display issues. The font attributes must defined before the font can be used. The font is then selected based on the ID. If the font ID is not available, the the font group will be used to find the "best fit".

Table 2. Font Attributes

Attribute	Description
Typeface	Name of the font family
Character weight	Bold, normal, light
Aspect	Condensed, normal, or extended
Italic	Yes/No

6.6 Drawing Text

All text drawn in a window is drawn using the current foreground color and cannot be color filled without setting the foreground color. Text is treated as graphics content in the window and therefore the text unites with the rest of the images in the window and will scale and rotate along with the rest. All characters appear on the screen as bitmapped images, but are stored in outline form.

When a character is displayed in a particular font, PenPoint will look up the character in an internal bitmap character cache. If the character is not present, PenPoint will render the character into the bitmap character cache. If the installed font is an outline font, the requested character is rendered as a bitmap at the requested point size. Characters are rendered into cache with all associated attributes and rotation.

Chapter 7. File System

This chapter describes the file system of PenPoint. The file system has been designed for compatibility with DOS and includes full support for reading and writing DOS formatted disks.

The following features are included in the file system:

- Hierarchical directories
- 32-character file names
- Memory mapped files
- OOPs APIs

Both the process of remote file transfer, automatic installation and the interface to device drivers is based on the architecture of a hierarchical system of directories and files.

PenPoint's file system is based of the concept of a volume; three types of volume are supported:

- Memory resident
- Local disks
- Remote disks and servers

Memory resident volumes are naturally stored in RAM; RAM is always available to the application because it cannot be disconnected by the user.

Local and remote disks are available to PenPoint when an external diskette drive is attached to the pen-based hardware, or when attached to a network or communications link.

All volumes have root directories. The operating system and applications make extensive use of the subdirectory tree structure of the hierarchical file system to store and retrieve specific files.

7.1.1 File System Activities

The following file activities are shared with other operating systems:

- Creating, opening and deleting files on a volume
- Copying and/or renaming files and directories
- Moving files and or directories
- Moving the read pointer to a new location with a seek operation
- Modifying file and directory attributes.

PenPoint however also supports a number of unique file activities. Every file and directory can have application defined attributes. Pen-based hardware is designed to be highly portable; therefore the operating system must manage random disconnection and reconnection to external volumes. Activities requiring access to external volumes are stored until connection is established.

PenPoint automatically performs file compression and decompression. The user may select the type of compression to be performed and even elect not to perform file compression.

7.1.2 Application Installation

The standard for application distribution is a 3.5-inch DOS formatted diskette. Applications usually include an installation routine which automatically installs the application within the desired file structure for the user.

Drag and drop routines are also supported whereby the user may drag application objects over an installer.

7.1.3 Interaction with other File Systems

The file system contains information that is incompatible with the DOS File Allocation Table (FAT) system and this superset information must be managed by PenPoint to maintain cross file system compatibility. The following information is not supported by the DOS FAT system:

- Long file names
- PenPoint specific attributes
- Application defined attributes

If the files to be stored are to be retrieved and reused by PenPoint, the superset information must be retained; this data is therefore stored in an extra file in each directory for which superset information exists in the PenPoint system.

PenPoint will detect and recognize this information when the external volume is mounted.

PenPoint files are only stored on external volumes without the superset information, if those files are to be subsequently manipulated by DOS applications.

Note

Compound documents (multiple documents composed in and managed by two or more applications), are managed via the file system. PenPoint stores embedded documents in subdirectories of the "containing" documents. The compound document is therefore stored and maintained as a single directory, which permits copying and moving of documents without having to be aware of the contents of the document.

7.1.4 File Import and Export

Applications generally include a set of filters (routines to process and convert data from one format to another format), to facilitate file import or export.

PenPoint supports data compatibility in the following manner:

- Use of shareable filters for data so that applications do not need to provide unique filters
- Use of a standard user interface for controlling file formats and interactions.

7.1.4.1 File Import

Files imported into PenPoint must be associated with an application. When a file is selected and copied/moved from a Disk Viewer window to the Notebook Table of Contents, PenPoint will query every application running on the system whether the particular file format being imported can be supported. The user is presented with a selection list of the appropriate applications.

The application checks the file import type, which is passed as a parameter, against a list of known file types; a positive response places the application on the selection list presented to the user. If the application is selected, the import process is initiated and a new document is created.

7.1.4.2 File Export

When the user selects **Export,** PenPoint queries the document to determine the file formats it can write. A selection list is then presented to the user. The user selects a file format and destination, then initiates the export process.

Each application is aware of the file formats supported for export and presents a list of formats supported together with control information that will be used during the translation process. The application will also provide a suggested file name for the user, which may be overridden.

The selected exporting application receives, along with the instruction to export, information about the source file, destination file and translator to be used.

Chapter 8. Input and Handwriting Recognition

Note

IBM has replaced the standard handwriting recognition module supplied with PenPoint. the following IBM DLLs have been included in the IBM version of PenPoint:

- XLATE.DLL
- PLI.DLL
- IBMSHAPE.DLL
- SPELL.DLL
- US Dictionary

The corresponding text and handwriting classes have also been been replaced.

The general concepts and techniques employed in handwriting recognition are described in this chapter.

The primary input and pointing device in PenPoint is the pen, or stylus. Unlike a keyboard and mouse, that are one-way communication devices, the pen requires a continuous two-way communication process through which handwriting is first recognized, interpreted and acted upon by the operating system, then fed back to the user. The user interface will, whenever possible, provide the context to guide the gesture and handwriting recognition module through the appropriate recognition process.

Applications do not pass text or numbers as input to the input subsystem. Input operations, called ***scribbles*** are passed to the application's user interface code that determines whether and how to translate the scribble, including the context within which the scribble should be translated. The location of a gesture determines its intended meaning; for example, depending on where a circular gesture is made, it will be interpreted differently:

- In text input the gesture "O" represents the alphabetic character "O"
- A circle drawn over text issues the command to edit the selected text
- A circle drawn in a graphics document represents a circle.

An operation is triggered by a pen action and subsequent processes are determined by the window context in which the pen gesture was made. The scribble is passed to the user interface owning the window that controls the translation. The application controls the translation by passing the scribble and control parameters (window context) to the handwriting recognition (HWX) module of the input subsystem. The input subsystem passes the recognition results back to the user interface, where the user may view the results of the operation.

While the pen must be able to dribble ink anywhere on the tablet, the system must not only support recognition, but also rapidly repaint the tablet.

8.1 Pen Input Terminology

The terminology used in pen-based computing is unique to this environment and some description is necessary:

8.1.1 Stroke

A stroke is both a pen action that leads to the appearance of ink on the tablet, and a data structure containing information about the action. Collections of related strokes are called scribbles; scribbles are also data structures that can be stored and manipulated.

8.1.2 Scribbles

Scribbles that have a meaning in a particular context may be gestures, characters, or shapes. Scribbles are interpreted by the application, consistent with the PenPoint User Interface Style Guide recommendations.

8.1.3 Dribbling

Dribbling is the appearance of "ink" on the tablet as the user moves the pen across the screen.

8.1.4 Input Focus

Input focus refers to where input from the keyboard will be directed. Keyboard strokes are always sent to the current window selection. Gestures on the other hand always apply to the data directly beneath where they are made.

8.2 Optimizing Pen Input

Direct input via pen strokes presents a number of challenges in terms of optimizing the performance of this form of input:

- Eliminating flicker and slow response when processing pen input
- Managing dribbles and windows in the user interface
- Providing a level of flexibility in handwriting recognition

The pen is an input mechanism; when dragged across the display, this movement must be "echoed" by the ink that traces the pen's path. As soon as the pen leaves the screen, the ink must be passed to the user interface and erased from the screen. If the echoed "ink" was displayed on the screen directly, screen flicker would result from the screen repaint process.

PenPoint's window system maintains a global, screen wide display plane, called an ***Acetate layer,*** where the ink is dribbled and strokes are collected into scribbles. The operating system ignores intermediate movements of the pen between the time a scribble is started and the time it ends. The data points are collected into a scribble data structure. The ink on the acetate layer can be erased without analyzing the effect of such an erasure, or refreshing the underlying display.

Ink must be confined by the windowing system to the window in which it originates. While the windowing system contains/clips display activities within a window's boundaries, the ink must be allowed to flow wherever the pen moves, to permit writing input a little larger than the size of the input field and gestures that might overlap boundaries.

Pen scribbles are processed to the owning window as an initial step. While the acetate layer displays the dribbles, the internal stroke and scribble data objects are generated from sampling points. The system passes the scribble data to the application when the acetate layer is erased. Strokes drawn outside the window boundaries are echoed back to the user; they are not incorporated in the sampling points.

PenPoint's handwriting recognition subsystem is totally replaceable to permit the inclusion of new technologies and to accommodate the Cyrillic and Kanji alphabets.

8.3 Handwriting Translation - Concepts

The input subsystem must generate input messages for all pen activity on the screen. This input is grouped together into scribbles, the scribbles are passed to a Handwriting Recognition subsystem (HWX) for translation into either text characters, or command gestures.

8.3.1 Characteristics of an HWX Subsystem

Each implemented HWX subsystem should include the following elements:

- Recognize both upper and lowercase characters, numerals, symbols and punctuation.
- Support both "boxed" (one character per box), and "lined" (characters written next to each other on the same line) handwriting.
- Operate in real time based on the clock speed of the processor complex in the hardware.
- The HWX subsystem plus a dictionary executes directly from memory and must be both efficient and compact to avoid excessive memory consumption.
- The HWX subsystem must achieve a high level of translation accuracy, supporting multiple users with minimum "training".
- Tolerate handwriting inconsistencies by the same user.
- Support non-unique character forms, that is, context sensitive character recognition, distinguishing between the character "O" and a numeric "0".
- Access context sensitive translation aids provided by applications.
- Run in background mode.

In essence strokes are received and examined by the HWX subsystem, character recognition is performed by comparing character shapes with a set of character prototypes. New prototypes are added by "teaching" the HWX subsystem to recognize unique styles in shaping the characters, through the handwriting training sessions provided with the subsystem.

8.3.2 Input Processing Concepts

Pen input is generally processed in the following sequence:

1. The input subsystem notifies the application of a pen event (the stylus tip has touched the screen, or some form of stroke has been made).
2. The input subsystem will analyze the application's window data structure, determining an appropriate response, such as echoing the ink on the acetate layer or not.
3. Once the event has completed, which is determined by a combination of time and distance thresholds, the input subsystem passes the resulting scribble to the application.
4. The application determines what to do with the scribble; it may be stored or translated.
5. If translation is required, the application packages the scribble with control parameters that describe the context in which the scribble is to be translated and requests translation.
6. The translation process results in a ranked list of translations that are passed back to the application. The application determines from the list what should be displayed to the user.

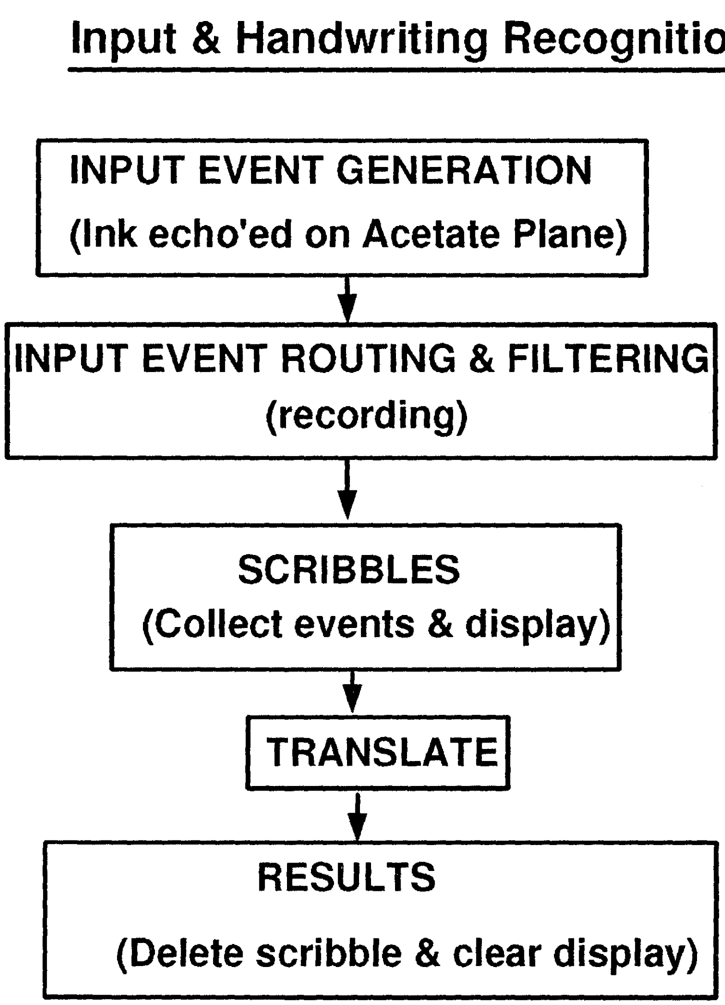

Input & Handwriting Recognition
INPUT EVENT GENERATION
(Ink echo'ed on Acetate Plane)
INPUT EVENT ROUTING & FILTERING
(recording)
SCRIBBLES
(Collect events & display)
TRANSLATE
RESULTS
(Delete scribble & clear display)

Figure 9. The Input Processing Pipeline

8.3.3 Application - HWX Dialog

PenPoint applications, not the operating system, control raw input. The application may process the input directly, or via APIs pass the input to the HWX subsystem. These APIs are a feature of PenPoint and its Context Management Subsystem, therefore the same dialog will be supported even if a new HWX subsystem is integrated into the operating system.

Applications can provide the following information to the HWX subsystem to aid in the recognition process:

- Choice of input, the application user interface (UI) may be boxed input or line input. Using only one, instead of both of these input approaches within a document facilitates translation.
- Choice of context rules which aid the translation process:
  - Spelling dictionary
  - List of acceptable characters
  - List of acceptable words

  - Templates
  - Punctuation rules
- Level of influence that context aids and rules should have in the recognition process:
  - Enable
  - Propose
  - Veto
  - Coerce
- Choice of post-processing aids:
  - Spelling correction
  - Case correction
  - Space correction
- List of acceptable gestures to aid in gesture recognition
- Choice of where to send strokes, the gesture recognition subsystem, or the handwriting translation subsystem

PenPoint applications can also manipulate strokes independently of the handwriting recognition system. The application can:

- Filter the strokes before sending them to the recognition subsystem
- Analyze and/or recognize strokes
- Perform post-processing on the output from the recognition subsystem

These functions may be performed in any combination; it is these functions in a graphics application that determine whether a circle should represent either a gesture, the character "O", or a circular drawing.

The HWX subsystem can also provide the application with information to assist the application in its interpretation of input:

- List of possible characters for single character input
- List of possible words for word input
- Size, boundary information and hot points for gestures

Chapter 9. The Windowing System

Windows are the most visible component of PenPoint. In user terms, a window is a ***document frame,*** the rectangular border, document title, scrollbars and menu surrounding a document. In application development terms, a window is a rectangular region of the screen with a capability for customized display and input behaviors. The document frame uses one or more windows.

Every window has a defined relationship with all other windows in terms of:

- Position
- Overlap border
- Transparency

Most pen activity, and all text display, occurs in a window and windows can execute the following types of operations:

- Input and detection
- Painting and repainting
- Obscuring that is, overlapping windows
- Clipping

9.1 Working with Windows

Windows include multiple coordinate systems, clipping and protection, and are integrated with the input system. All pen input events are automatically directed to the appropriate window.

Windows can contain embedded windows that may belong to other applications. PenPoint structures windows into a tree hierarchy, described as a parent-child relationship, beginning at the root window that corresponds to the physical screen. A child window is always clipped to the parent window and is never visible unless the parent window is visible.

A document frame consists of many components, each of which is at least one window. Within an application, there may be several windows, each of which uses different elements of a typical window.

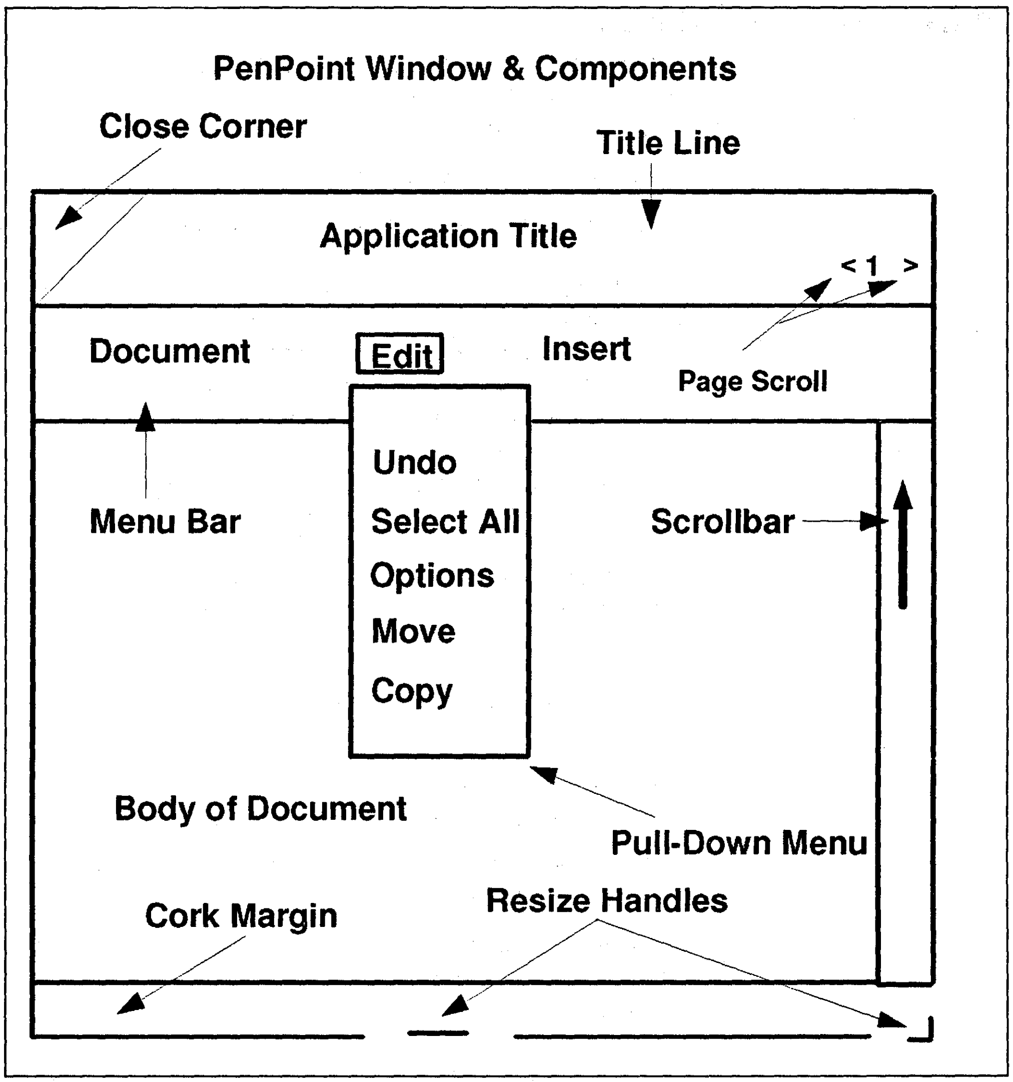


Figure 10. A PenPoint Window

Windows are normally thought of in relation to a display; however PenPoint window trees can be rooted to any image device and can be used to create virtual, in-memory displays. This means that window trees can be rooted on printing devices. The printed image is constructed in memory as a window tree with graphics in each window. The entire page image is then sent to the printer.

New windows are created by providing the operating system with the following information:

- The new window's parent or device
- The size and location relative to the parent window
- Flag settings that determine the layout, clipping and repainting characteristics of the window, together with what kind of input may be received

9.1.1 Displaying Windows

A child window is always clipped to the parent window and is never visible unless the parent window is visible. This is done in accordance with the following principles:

- Child windows are always placed on top of parent windows.
- Drawing in a child window is always clipped by the parent.

PenPoint requires this consistent window hierarchy to provide effective window management; for example, if a window is inserted into a hierarchy and then removed (closed by a user), an underlying window may be uncovered. This now unobscured window must be repainted. In order to avoid the time delays inherent in repainting windows, each time a window is inserted into the hierarchy, it creates a copy of the physical screen region beneath the screen area where it will be displayed. When the window is closed, PenPoint simply copies the stored bit image to the screen.

This concept in PenPoint means that the application specifies the contents of a window and the appearance of the window. PenPoint manages the positioning of all child windows.

Parent window design can adopt one of three approaches in positioning child windows:

1. Permissive. The parent window is set up so that child windows can display themselves anywhere, even completely covering the parent window.
2. Strict. The parent window intercepts and can veto all messages to its child windows that could affect layout.
3. Flexible. The parent window will attempt to accommodate child window layout requests, but will override them to avoid layout conflicts that the parent window has been designed to prevent.

When the user turns a page in the Notebook, or closes a floating window, the parent window files the state and contents of the child windows.

At any given time, an application's display state reflects the results of recent user commands and actions; therefore when an application is closed it files the current window hierarchy.

PenPoint keeps track of each application's window environment:

- Orientation
- Pixel size
- Default system font

When the application is subsequently re-started this window hierarchy is retrieved and the application uses the window environmental information to restore the window hierarchy to its previous display state.

Chapter 10. Service Manager

A service may be defined as a program that enables applications to communicate with a hardware device, or to access a software function. (Software functions do not require user intervention and typically run as a background task.)

PenPoint unifies each of these services under a Service Manager that generalizes common application operations such as finding, observing, binding to and opening services. These operations work in an environment in which services may be dynamically installed and deinstalled and in which the underlying hardware connections can be made and broken at will. Examples of services include:

- Device drivers
- In/Out Box
- Network protocol stacks
- Databases

The Service Manager provides a common architecture and implementation to allow a variety of services to be accessed by applications and adopts a layered approach. The result is a class of services that do not represent hardware devices; for example, the most basic services are those that communicate with a hardware device, such as a serial port. PenPoint's Service Manager generalizes such operations by providing a software service that accesses the hardware service (target service), adding function and abstraction on top of the targeted service. This provides the layered approach.

Services may target other services to any depth and the targeting relationships are viewed as ***pipelining.*** Pipelining is the vehicle used to implement the layered approach. For example, an application might open a service designed to interact with a bulletin board and via pipelining, the serial port is accessed.

The Service Manager consists of two classes:

1. The class that defines the service
2. The class that provides access to the service

Services in turn may belong to one or more service managers. Internally these services are implemented as non-application dynamic link libraries (DLLs). A single Service Manager can manage a group of services, for example, a number of serial ports.

10.1 Standard Service Managers

A number of predefined service managers are provided by PenPoint:

- Apple-Talk** devices
- Serial devices
- Printer devices
- Printers
- Send services - facsimiles and electronic mail

- Transport handlers - component of the networking API
- Link handlers - component of the networking API

The following basic functions are provided by these service managers:

- Locating a particular service (specified printer, or serial port)
- Binding to the service - allowing the client to receive notification messages from the service
- Establishing exclusive ownership of the service
- Opening the service for data transfer
- Closing the service

10.2 Installing and Using Services

Services are dynamically installable and deinstallable, by a user, application, or another service. Only one copy of the service is installed. The operating system maintains a record of the number of clients requesting installation of a service, the service is only deinstalled when the last client is deinstalled.

The application must be bound to the service before it can be used, this is performed by the application sending a message to the appropriate Service Manager. Once the connection has been established, the service adds the application to the list of objects to be notified upon a change in status. The application therefore is constantly aware of the availability of the service.

The Service Manager supports multiple clients sharing the same service. Shared access is supported where the service can support it and is arbitrated where services cannot be simultaneously accessed.

In the case of exclusive ownership, the client must gain the ownership rights to the service before it can be used, for example, a physical serial port. The Service Manager provides protocols for clients to transfer ownership cooperatively.

As with all installable PenPoint objects, services can be deactivated, or deinstalled whenever they are not in use. This destroys all the service's objects and removes all of the code.

10.3 Connecting and Disconnecting Services

The connection status (presence or absence of a physical hardware connection), is managed by the Service Manager. Non-hardware services automatically change their connection status when the target service status changes. Therefore the connection status propagates upwards from the hardware to all the services that are bound to that hardware.

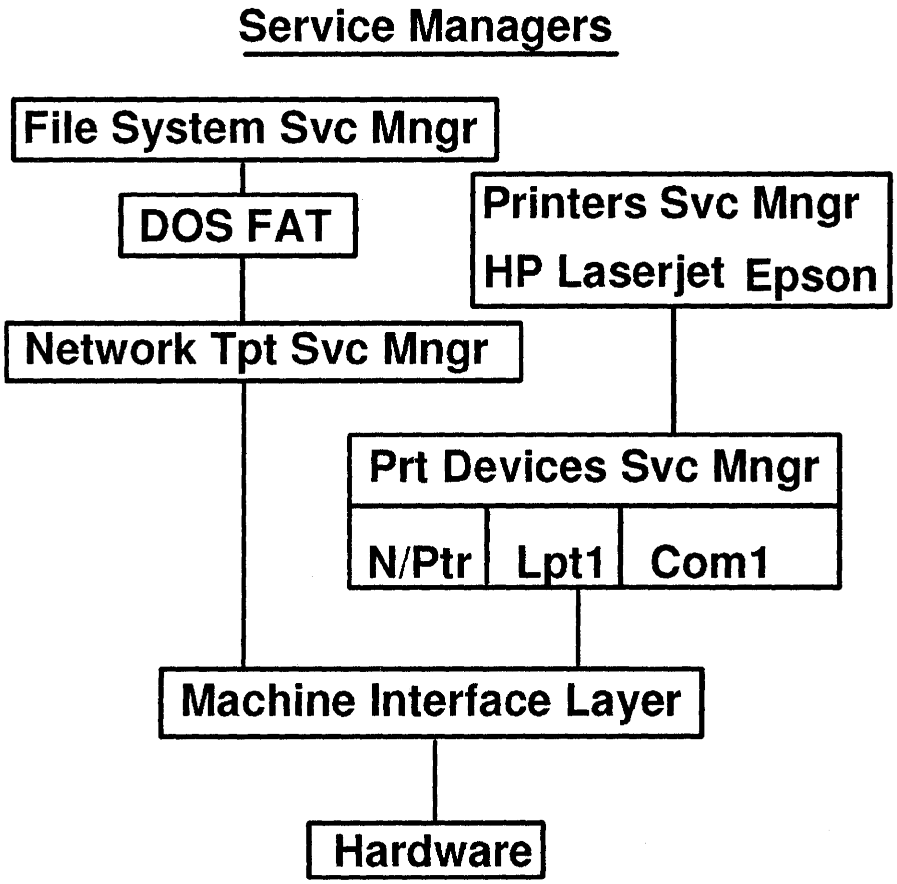


Figure 11.. Schematic of PenPoint Service Managers

Chapter 11. Connectivity

The operating system is designed for ***mobile connectivity.*** Connectivity is provided via three layers:

- Remote file systems
- Transport interfaces
- Link interfaces

Deferred connectivity is supported via the In/Out Box interfaces, which provide mobility.

Connectivity is accomplished via direct serial connection, or via serial connection between modems.

11.1 Remote File System

All documents in the Notebook are stored in the file system. Movement and access of documents are performed through the file system and these operations may extend to remote environments. PenPoint file systems may reside on the pen-based systems, locally attached disks and on remote devices, linked via a network.

The remote file system is accessed via APIs running under PenPoint using networking transport interfaces to communicate with the remote file system. The remote file system behaves in the same manner as the local file system and is transparent to the user because a single Notebook Table of Contents is used to access documents regardless of their location.

Remote printing employs a similar concept to that of remote file systems. The remote printing interface removes the need for clients to know the exact location of the printer.

Program-to-program communication used to establish a live connection, is supported via remote procedure calls; for example, a PenPoint system may send SQL queries to another system and receive data back.

The principles involved in connectivity include:

- Local and remote file systems and volume connectivity, enabling users to access documents on local and remote systems
- The facilities offered by the Service Manager enabling users to connect and disconnect to devices and remote systems on the fly
- The general purpose document import/export architecture, including file format conversion
- In/Out Box support permitting deferred I/O

11.2 Transport Layer

The transport API provides access to layers three and four in the standard Open Systems Interface (OSI) network model.

11.3 Link Layer

Link protocols are the software layer closest to the physical networking hardware, residing at Layer 2 of the OSI model.

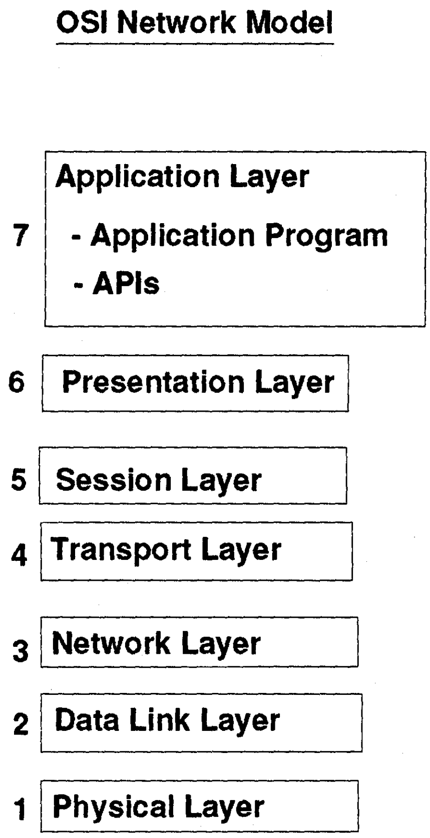


Figure 12. Sven Layer OSI Model

11.4 Send User Interface

Standard application windows provide a Send command on every document's Document menu. The command invokes the Send User Interface, placing the documents in the Out Box. This is the standard user interface for addressing documents, regardless of which transfer protocol is used.

The interface is built around a Send List that in effect, is a database, containing address information and installed transmission services.

11.5 In/Out Boxes

The In/Out Boxes provide support for deferred data transfer and work with all data in PenPoint.

The In/Out Boxes are specialized floating notebooks that act as queues for incoming and outgoing documents. They do not perform any transfer operation, but do provide a common user interface and architecture in which application specific transfer services are grouped.

The user interface component is a section in the In/Out Boxes Notebook, termed a service section. Service sections equate to transfer services, for example, printing, electronic mail and facsimile applications. Service sections queue documents awaiting a transfer operation. The transfer application is not aware that document queuing is occurring; this happens at a file system level using a copy of the document.

The user need not be aware of whether a service is available or not, the Send or Print commands will cause a copy of the document to be placed in the appropriate Out Box service section. As soon as the connection is available the transfer agent (application), is notified of the connection and the documents are processed.

11.6 PenCentral - PenTOPS

PenCentral ** is communications software, installed on a PS/2* that communicates with with PenTOPS** installed on a PenPoint system.

PenCentral is a DOS application requiring at least 512KB memory and a minimum of one parallel, or serial port. Serial connection is via a null modem serial cable. Parallel connection is via a parallel cable. If a 9-pin mini-parallel port is not available, a converter pigtail is used to convert a 25-pin port.

PenCentral supports Hayes** compatible dumb modems

PenCentral is compatible with IBM OS/2 LAN Server 2.0 (Entry and Advanced) and Novell Netware**.

Several PenPoint systems may be attached to a single PS/2, but only one pen-based system may access the PS/2 at a time.

11.6.1 PenCentral Files

PenCentral is comprised of the following files:

- PENTALK.EXE - AppleTalk drivers
- PENSERV.EXE - Server code
- PENMENU.EXE - User interface code

The following files that store directory information are created when PenCentral is used:

- PL.DID - Created in the root directory
- DRIVEA.DID - Created in the PenCentral directory

PenCentral creates a temporary file ***PENMENU.PS$*** at startup, in the PenCentral directory. The file is deleted upon exiting the application.

When a document is printed through PenCentral spool files are created in the spool subdirectory of PenCentral.

Excluding spool files PenCentral requires a minimum 700KB of disk.

11.6.2 Installation and Configuration

Default installation stores the PenCentral files in C:\PCENTRAL. All of the PenCentral files must be located in a single directory as the application will not search a path for required files.

The PenCentral Server Configuration is displayed upon completion of successful installation. The user may select serial, parallel and/or modem links to the PenPoint system. Shared (networked) printers may also be configured.

On invocation, PenCentral displays an ***Activity Status Line*** indicating the PS/2 port with the link that is being served.

Users may query the PenCentral print queue, pause an active printer and delete print jobs.

PS/2 drives that are available to PenCentral are termed **Volumes.** Volumes include:

- Actual physical drives
- RAM disks
- Drives available via an installed network redirector
- Drive created through the ***SUBST*** command

The PenCentral configuration file, PENINFO.DAT, determines what information is presented in the configuration dialogs and is created during installation. The configuration file is divided into the following sections:

- PenCentral server information:
  - Number of configurable serial and parallel ports
  - Last configurable disk drive
  - PenCentral system directory
  - Inactivity timeout value
  - Diskette drive polling frequency
- Modem configuration information
- Volumes configuration:
  - Type of drive (exclude, network, diskette, or hard disk)
  - Network name of published drive
  - Password for drive
  - Directories in root to exclude

    - Read/write access for drive
- Serial (COM) port configuration:
    - Type of port (disabled, cable connect, printer, or modem)
    - Interrupt number
    - I/O base address
    - Printer name
    - Printer type (network or local)
    - Printer baud rate
    - Printer parity
    - Printer data bits
    - Printer stop bits
- Parallel (LPT) port configuration:
    - Type of port
    - Interrupt number
    - I/O base address
    - Printer name
    - Printer type (networked or local)

The configuration file may be modified using any text editor.

PenCentral under OS/2

PenCentral may be installed in multiple Virtual DOS Machines (VDMs) under OS/2 2.0. CONFIG.SYS requires modification and the following statements must be removed/commented out:

- DEVICE=C:\OS2\COM.SYS
- DEVICE=C:\OS2\MDOS\VCOM.SYS
- BASEDEV=C:\OS2\PRINT02.SYS

The standard OS/2 driver VLPT.SYS is replaced by the PenCentral driver. The net result of these changes is that native OS/2 applications can no longer access the parallel and/or serial ports, which are now dedicated to PenCentral, a separate port per PenCentral VDM.

The following services are available from PenCentral:

- Reconfigure PenCentral
- Manage print jobs:
    - View print jobs
    - Start printing
    - Pause printing
    - Delete print jobs

The PenCentral user cannot:

- Copy or move files to/from the PenPoint system
- Access or use files on the PenPoint system
- Use devices locally attached to the PenPoint system

11.6.3 PenTOPS

PenTOPS is the client component running under PenPoint, permitting access to remote resources. Once a connection has been made, the PenPoint user can:

- Access and use all volumes available to the attached PS/2:
  - Access both the local PS/2 disks and network disks.
  - Access data files.
  - Use notebooks and documents.
- Transfer files to/from the attached PS/2:
  - Back up/Restore copies of PenPoint documents.
  - Store both the PenPoint applications and documents.
  - Export DOS format files for further processing by DOS applications.
- Print documents on printers available to the attached PS/2.
- Access diskette drives available to the attached PS/2.

PenTOPS is preinstalled on the IBM ThinkPad*; to confirm installation:

1. Select the **Settings Notebook** on the Bookshelf.
2. Select **Services** in the **Installed Software** section.
3. The PenTOPS listing is displayed.

The Connections notebook is used to set up and modify network connections, disks and printers. This notebook is divided into two sections, ***Disks*** and ***Printers.*** Both sections contain a ***Network View*** and a ***Connected Page.***

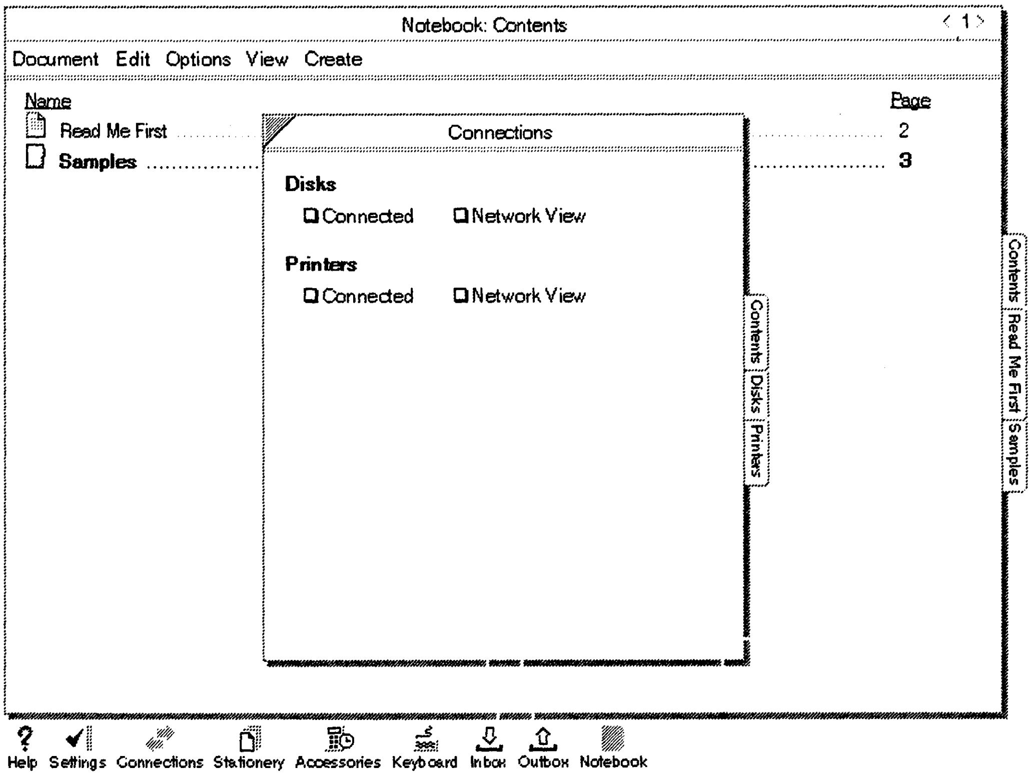
The Connected pages are used to perform PenPoint tasks for disk or printer, either networked, or directly attached.

The Network View pages display:

- Network disks available
- Contents of network disks
- Connection to network disks
- Available networked printers
- Connection to networked printers

Note

- The Connections notebook does not open at a contents page.
- There are no page numbers.
- There are no contents pages for each section.



Name	Page
Read Me First	2
Samples	**3**

Disks

□ Connected □ Network View

Printers

□ Connected □ Network View

Figure 13. The Connections Notebook

Chapter 12. Software Installation

PenPoint is pre-installed on the IBM 2521 ThinkPad, but may be re-installed or refreshed at any stage. The operating system and applications are installed from an attached diskette drive, hard disk, or network disk connected to the IBM 2521 ThinkPad. Applications include fonts, handwriting recognition modules and services. Services include device drivers for printers, plotters and modems, together with software for electronic mail and information services.

PenPoint applications should automatically display the ***Installable Software Sheet,*** when connected to a disk. If the Installable Software Sheet is not displayed, either the Settings, or Connections Notebook may be used to install software.

12.1 PenPoint Installation

Note

Prior to installing the operating system, the IBM 2521 ThinkPad must be reset. (Refer to the reference manual provided with the hardware.)

Check whether similar procedures apply if installing on OEM hardware.

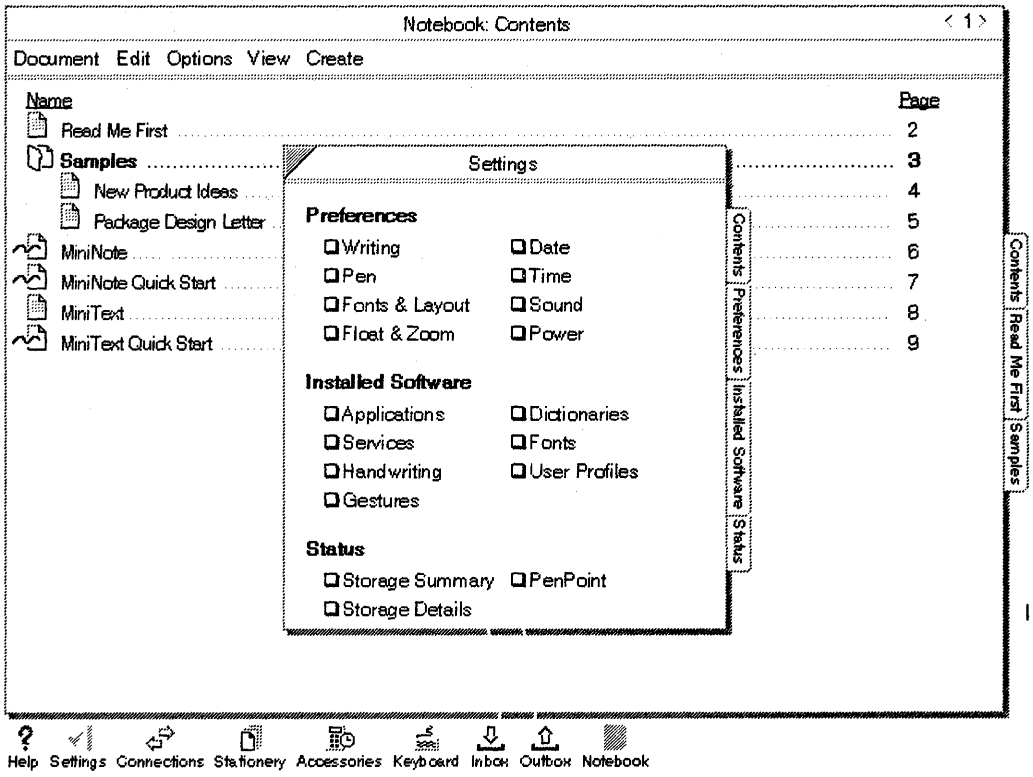
The following procedure should be used to install the operating system:

1. Attach an external 3.5-inch diskette drive, minimum density 1.44MB to the pen-based system's hardware.
2. Insert the PenPoint boot diskette and power on the hardware.
3. If installing on an IBM 2521 ThinkPad, reset the hardware.
4. Select **Begin Hard Disk Installation.** You will be prompted to format the hard disk.
5. Once installation is complete, select **Start PenPoint** to start using the system.

12.2 Automatic Software installation

Note

All software installed on a PenPoint system is listed in the Settings Notebook.



Name	Page
Read Me First	2
Samples	3
New Product Ideas	4
Package Design Letter	5
MiniNote	6
MiniNote Quick Start	7
MiniText	8
MiniText Quick Start	9

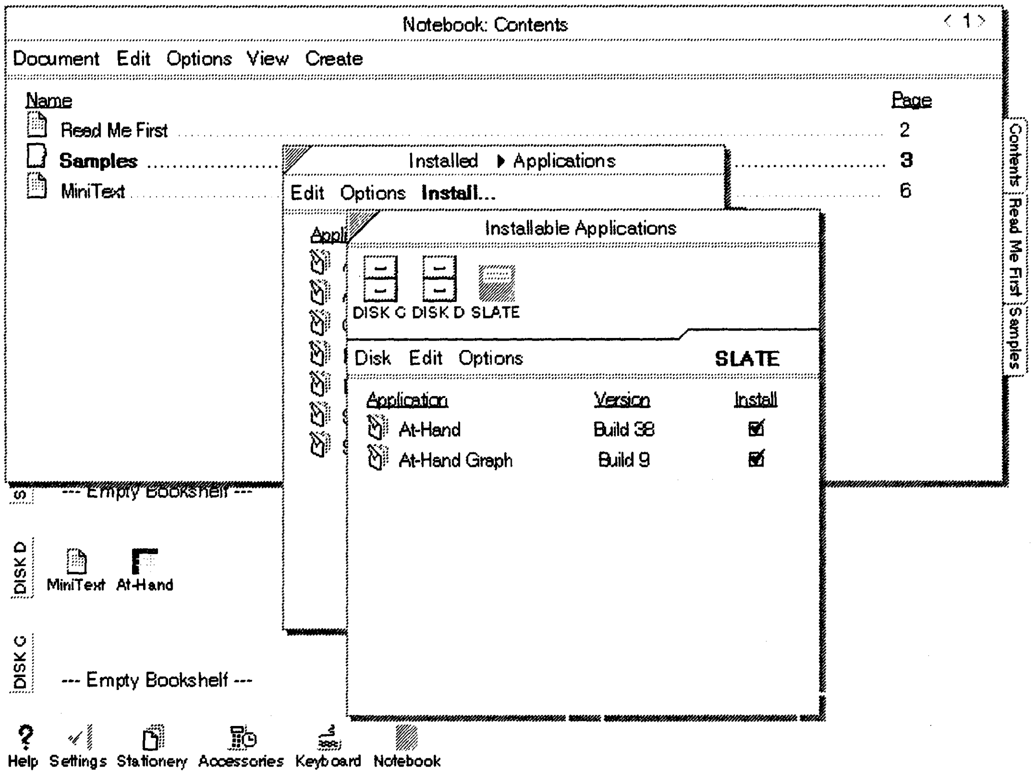
Figure 14. The Settings Notebook

Most PenPoint software installs automatically when the application's installation diskette is inserted in the diskette drive. The Installable Software Sheet is displayed, providing a number of selectable options.

Upon completion of the installation, the application is available for use. The application is placed in the Notebook Table of Contents and a new document may be created using this application.

Use the following procedure for automatic software installation:

1. Connect the PenPoint system to disk drive that contains the application to be installed
2. The Installable Software Sheet is displayed
3. Select the items to be installed.



Name	Page
Read Me First	2
Samples	**3**
MiniText	6

Application	Version	Install
At-Hand	Build 38	☑
At-Hand Graph	Build 9	☑

Figure 15. The Installable Applications Software Sheet

12.3 Manual Software Installation

If an automatic application installer is not provided, software may be installed via the Settings, or Connections Notebooks.

The Settings Notebook is used when:

- Software preference settings are required.
- Software is deinstalled.
- Changes to software settings have been made.

The Connections Notebook is used when:

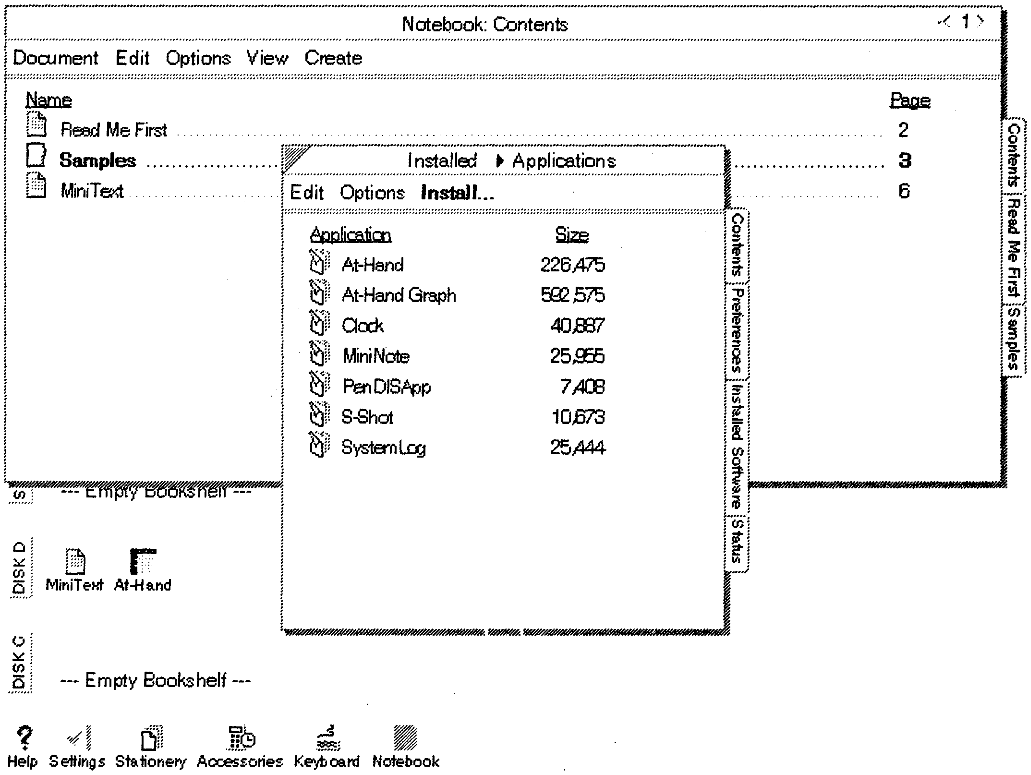
- Diskettes are to be formatted.
- Printers are to be set up.
- Files are to be transferred.
- Network resources are required.

12.3.1 Settings Notebook

The Settings Notebook lists all the installed software within a number of categories:

- Applications
- Fonts

- Services
- Handwriting
- Software preference settings



Name	Page
Read Me First	2
Samples	**3**
MiniText	6

Application	Size
At-Hand	226,475
At-Hand Graph	592,575
Clock	40,887
MiniNote	25,955
PenDISApp	7,408
S-Shot	10,673
SystemLog	25,444

Figure 16. The Installed Applications Notebook

Use the following procedure to install software via the Settings Notebook:

1. Attach an external diskette drive to the PenPoint system.
2. Insert the software diskette.
3. Select the Settings Notebook from the Bookshelf.
4. Select the Applications page.
5. Select Install.

Note

Fonts, services, personal dictionaries and preferences are installed in the same manner by selecting the appropriate notebook tab.

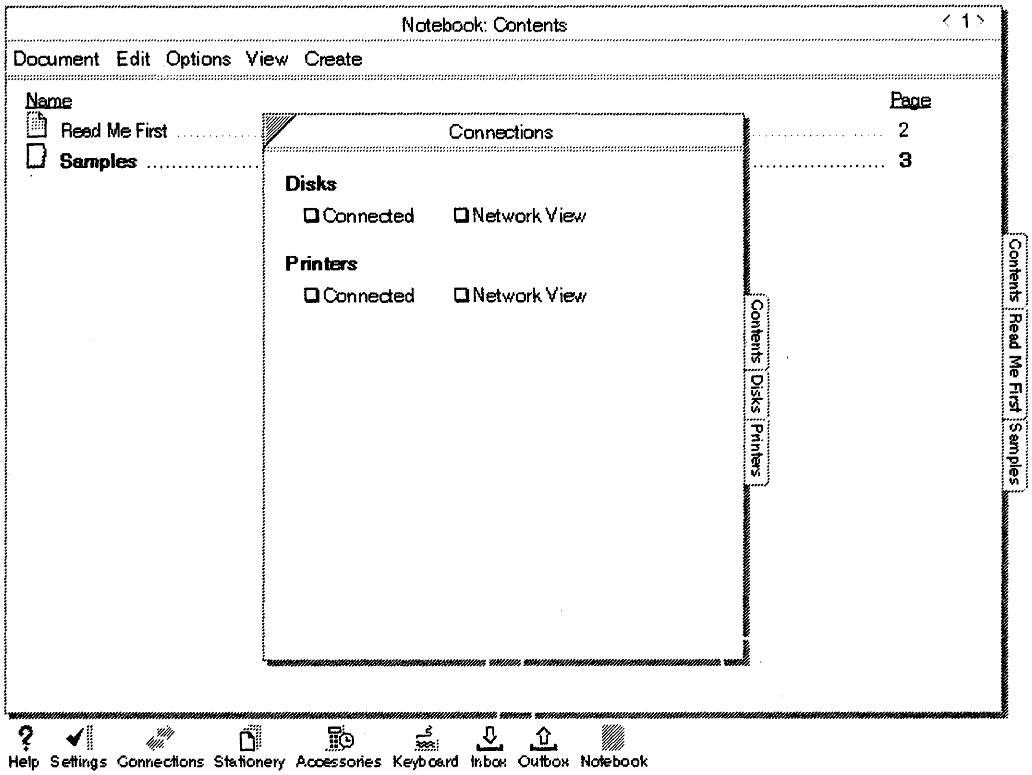
12.3.2 Connections Notebook

The Connections Notebook displays the disks connected to the PenPoint system.

Use the following procedure to install software via the Connections Notebook:

1. Select the Connections Notebook.
2. Select the Disks Connected page.

3. Select the appropriate disk icon.
4. Select the View menu.
5. Select the appropriate software category.



Notebook: Contents

Name	Page
Read Me First	2
Samples	**3**

Connections

Disks

□ Connected □ Network View

Printers

□ Connected □ Network View

Figure 17. The Connections Notebook

Chapter 13. Application Development

This chapter describes the process and tools available to develop a PenPoint application. The following topics are covered:

- Overview of object-oriented terminology
- PenPoint Class Manager
- Resources
- PenPoint Software Developer's Kit (SDK)

13.1 Object-Oriented Terminology and Techniques

A PenPoint program employs functional units called **objects.** Objects communicate with each other by sending and receiving **messages.** The way in which the object responds to a message is determined by the **class** to which the object belongs.

Classes are the mechanism by which objects are created and it is the class that contains the code that determines the response of an object to a message. The code that an object executes in response to the message is called a **message handler.**

When an object is created by a class, that object is an **instance** of the class. Classes moreover may inherit behavior from other classes and subclasses inherit the behavior from all of their ancestors.

When an object receives a message, the class that created the object handles the message; the class may pass the message all the way up the ancestral inheritance hierarchy to determine the appropriate object behavior.

PenPoint provides a wide range of built-in classes that generate the instances an application requires:

- Windows
- Scrollbars
- Lists
- Data views
- Text objects

These functions, macros and support classes used to implement the PenPoint object model, are collectively known as the **Class Manager.**

PenPoint's class hierarchy consists of approximately 180 classes, divided into 6 functional entities:

- Application classes
- Installation classes
- Windows and User Interface (UI) Toolkit Control classes
- Remote Interfaces and File System classes
- Text and Handwriting classes

- Miscellaneous classes

13.1.1 Application Classes

PenPoint's Application Framework (refer Chapter 4, "Application Framework" on page 15), provides a methodology for building applications that ensures that all applications work in a similar manner. The Application Framework implements an application class hierarchy that includes the superclass of all application classes, ***clsApp*** and the Class Manager ***clsClass*** itself.

13.1.2 Installation Classes

The installation classes are used to implement behavior for managing the installation of system resources:

- Fonts
- Handwriting
- Applications
- Services
- User preferences

13.1.3 Windows and UI Toolkit Control Classes

The largest of the PenPoint class hierarchies are dedicated to the implementation and control of the Notebook User Interface (NUI). The windows class, ***clsWin*** is included, which is the superclass to all displayable items in the NUI.

13.1.4 Remote Interfaces and File System Classes

This class hierarchy provides support for network-based computing, file management, hardcopy printing and fax/modem support.

13.1.5 Text and Handwriting Classes

This class hierarchy provides support for managing input to applications, including support for gestures, scribbles, keys and spelling.

Note

The following classes have been replaced in the PenPoint IBM version:

- clsXGesture
- clsXTeach
- clsXText
  - clsXTract
- clsPDict
- clsProof
- clsSpellManager

13.1.6 Miscellaneous Classes

This hierarchy of classes provides support for entities such as the battery monitor, timer and string manager.

13.2 Class Manager

> **Note**
>
> PenPoint does not support current object-oriented programming languages, but implements a set of function calls and macros for managing objects in the PenPoint environment based on ANSI-C.
>
> Current object-oriented languages tend to have been designed to support a single application on a disk-based, procedural operating system.

PenPoint's Class Manager is a collection of functions, macros and support classes used to implement the PenPoint object model. The Class Manager is an integral part of the kernel which means that many of the functional elements of an application are extensions of the facilities provided by the operating system.

The Class Manager provides the object functionality to:

- Create classes and class hierarchies.
- Create or destroy objects or class instances.
- Inherit functionality from other objects.
- Define and send messages between objects.

PenPoint has two root classes in its class hierarchy. Objects descend from **clsObject.** Classes descend from **clsClass.** clsClass is a meta-class, and for each class in the system there is a corresponding object that stores information about the class, including the code that implements its methods and implements class level operations. Objects encapsulate data and behavior, the code (behavior) is not duplicated with every object instance, because clsClass supports classes as a type of object that provides for shared behavior and information for a type, or class of objects.

All PenPoint application programming interfaces (APIs) are based on Class Manager messages and objects. The implications of adopting this approach are that system code may be reused and modified at many levels, applications are generally compact and provide a consistent user interface.

13.2.1 Unique Identifiers

A fundamental process of any program is to reference some entity. These entities include references to memory locations, using pointers, and files, using names. The entities are either dynamic or static.

Dynamic references are either created, then passed into application code, or received from other code; these references are generally pointers to memory addresses.

Static references are placed into the code at compile time. If the reference is a memory address, the code will not be portable. If the reference is expressed as a string, uniqueness cannot be guaranteed and conflicts may occur.

PenPoint unifies dynamic and static references into a single naming convention - Unique Identifiers (UIDs).

The UID is a unique 32-bit identifier, used to identify and keep track of all classes and objects. The UIDs are not data pointers; they contain encoded information indicating whether the object referenced is ***well known*** or ***dynamic*** and include an administered value from GO&astersik. &astersik. Corporation.

13.2.1.1 Well Known UIDs

Well known UIDs identify classes and are permanently defined at compile time. The assigned UID must be unique to avoid conflicts when applications are embedded by other applications. There are a number of types of well known UIDs in addition to the ones used for objects. These UIDs include:

- Management of unique values for status information
- Message identifiers
- Tags

Tags are 32-bit values used to identify well known constants within an application including:

- Option sheets
- Option cards
- Quick Help strings

Well known UIDs contain flags that specify the **scope** of the UID. Global UIDs are known to all tasks in PenPoint. All processes in the system are allowed to access the same object using the same identifier.

A ***process-global*** well-known UID allows each process to reference different objects with a single identifier. This is useful for objects that exist in each process, but the object must have the same identifier. For example, ***WorkingDir*** is a process-global well-known UID identifying the process "working directory". A process that refers to this UID will reference its own working directory object. Other processes that refer to this UID reference other working directory objects.

A ***private*** well-known UID is used by the application developer; a component used only by that application is identified as a private well-known identifier.

13.2.1.2 Dynamic UIDs

Dynamic UIDs identify instances created by the application and are created by the Class Manager at run time. All dynamic UIDs have global scope. After the object referenced by the dynamic UID is released, that UID may refer to a different dynamic object at a later time.

UIDs within filed data are also supported, these UIDs are persistent, that is, unique across all time and space. This is accomplished through the use of Universal UIDs (UUIDs). UUIDs include a unique machine ID from the hardware on which PenPoint is running. UUIDs may be used to point to PenPoint objects even when filed to external media and then loaded back into PenPoint.

13.2.2 Class Manager - Programming Tasks

During application development, the following programming tasks typically involve the Class Manager:

- Setting up message arguments
- Sending messages
- Creating instances
- Controlling object access and capabilities
- Creating new classes
- Setting up observer objects

13.2.2.1 Message Arguments

Sending messages to objects is the primary mechanism for control and data flow in PenPoint. Messages are sent to instruct instances to perform some form of operation, for example, instruct a table to send back data from a specified row/column address.

All processing in PenPoint takes places as a result of one object sending another object a message and responding to the message. The Class Manager provides a set of C functions and macros that send messages to objects. These functions take arguments that describe the target object, the message being sent and a pointer to a structure that may contain additional argument data.

Like objects, messages are identified by 32-bit constants. Message identifiers share the administered portion of the UID of the class that defines the message. Each message requires a specific argument structure, and the message description in the header files specifies the argument structure for each message.

Objects respond to messages in one of two ways:

1. Return a status token indicating the success or failure of the operation requested by the message.
2. Return data in the argument structure supplied by the message sender.

13.2.2.2 Sending Messages

Messages are sent to objects to elicit some form of behavior from the receiving object. The behavior is either part of the object's class definition, or contained in a parent class. The application need not know where the behavior is defined, merely that the receiving object is able to respond to a specific message.

PenPoint is a multi-tasking operating system and therefore supports a number of different tasks, each task getting a share of CPU cycles. Every active document is a separate task. Embedded tasks run in separate tasks from their parent documents. The Class Manager provides separate functions for synchronous processing or asynchronous processing.

In synchronous processing, objects can only send messages to objects that reside in the same application instance; all processing stops until the receiving object responds.

In asynchronous (multitasking) processing, the caller and responder execute concurrently because the processes have separate memory address spaces and the Class Manager will copy the argument data structures from the caller's task space into the address space of the called task.

13.2.2.3 Creating New Instances

Object creation (an instance of a class) is a two-step process involving the initialization of a default data structure and then the creation of the object.

Each class in PenPoint defines the structure that contains the information necessary to initialize a new object. The process may be summarized as follows:

1. Send the class to be instantiated ***msgNewDefaults,*** passing a pointer to an appropriate argument structure.
2. The class initializes the argument structure appropriate for the specified class.
3. Default fields may be overridden.
4. Send ***msgNew*** to the class; instantiation of the object occurs in this step.

13.2.2.4 Object Access and Capability

A major challenge in an object-oriented operating system is to protect objects from unintentional alteration. PenPoint implements this protection through the use of keys and locks. All objects can have an associated key that limits access to specific operations, to the applications that have the key. Messages that request object operations such as freeing, or removing the object require the use of a key, unless that object has specific capability flags set to permit the operation without a key.

Capability flags include:

- A sending object may change the class of the receiving object.
- Free or remove an object.
- Designate the object as the ancestor for a new class.
- Classify an object as observable and control the messages this object may respond to.

Capability flags can be changed dynamically with the appropriate key.

13.2.2.5 Class Creation

New classes are created when an application requires a behavior not available within an existing PenPoint class. Each PenPoint application must have its own subclass within the Application Class, in order to run.

The following steps are used to create a new class:

- Provide a set of functions that defines the behavior for the new class. This behavior must distinguish the new class from other classes.
- Provide a translation mechanism, called a **Method Table** that translates a message into a UID that is used to bind the behavior request (message) with the implementer.
- Provide the function to send a message to the Class Manager to add the new class. Consumers of this class use this function to register the class with the Class Manager when the class is used.
- Provide an interface file containing message definitions required by consumers of the class.

Note

The Class Manager maintains a method table for each class, in which message UIDs index into the table containing the memory addresses of the associated C routine.

During application development, the table is created that associates each message's UID with a C function call. The table is compiled using the Method Table Compiler provided with the PenPoint SDK. At run time the Class Manager binds the class to its method table.

13.2.3 Observer Objects

A unique capability of all PenPoint objects is the ability of an object to register itself as an observer of another object that is capable of being observed.

An observer is notified of any change in the state of an observed object.

The Observer Notification Architecture is the foundation of automatic notifications and updates in PenPoint, for example, automatically notifying applications that a new service has been installed.

Note

There are two ways of establishing whether an event has occurred in a system, polling and notification.

Polling requires that the program or user must periodically inquire whether an event has occurred, such as querying whether a diskette has been inserted in a diskette drive.

13.3 Resources and Resource Management

A PenPoint resource is defined as a collection of data identified with a UID. Programs use resources to maintain information such as string tables, persistent objects, and component descriptions for option sheets.

Resources are special files managed by a **Resource Manager** that is used to create, find, access and modify resource files.

13.3.1 Resource Types

PenPoint has two types of resources, objects and data. A resource file may contain both objects and data resources. Different messages are used to read and write the two types of resources.

An object resource contains information needed to create or restore a PenPoint object. The objects and all its ancestor classes must be able to unite and read the object's instance data to and from a resource file.

Every class created must be able to read and write its object instance data in response to Class Manager messages. The Application Framework maintains an instance data resource file for each application instance or document.

Each resource file has a unique 64-bit resource ID that is used by the application to locate the resource.

Data resources contain information saved as a stream of bytes and are generally used for default Option Sheet settings and default prompt strings. These objects are then portable and facilitate internationalization of the code.

All resources are read and written through resource agents. PenPoint includes a number of resource agents designed to manage specific objects and data structures, unpack and interpret the formats of the data.

Resources may be created at program compile-time, or at run time. Static resources, created at compile time, change infrequently and can be fully defined during application development. This resource is part of the application and not part of the document and defines the non-code part of the application such as the user interface elements and icons. Static resources are identified with a pre-defined ID and declared at compile-time. The application resources reside in a file called **App.Res.**

Dynamic resources are only created at run time. these resources are stored in resource files created through the Resource Manager. Dynamic resources are identified with resource IDs allocated at run time.

The Application Framework provides a default file for dynamically created resources called **DocState.Res,** that contains all the objects belonging to a particular document (application instance).

13.3.2 Resource Location

The Resource Manager uses Resource Lists to locate a resource within a given file. This procedure insulates the application from having to know the file in which the resource is located and the location of the resource within the file.

Entries in a Resource File List can be resource file handles, or other Resource File Lists. When a message is sent to a list object, the message is sent to each object in the list until the message returns a value indicating that the instruction has been carried out.

Every document class has a default Resource File List that contains the following elements:

- The PenPoint system resource file - ***PenPoint.Res.***
- The application resource file - ***App.Res.*** This file is common to all application instances.
- The document resource file - ***Doc.Res.*** This file is unique to one instance of the application.

PenPoint applications have flexibility in providing and sharing resources in that an application can use system resources such as standard fonts and error messages, without having to provide these resources in the application resource file. The application may override system resources, or create application wide resources, placing such resources in the application resource file. Applications can even allow the user to attach specific versions of resources to individual documents.

13.3.3 Resource File Formats

The Resource Manager supports the notion of maintaining many resources in a single file, laid out as a single data stream. The operating system keeps track of where each resource begins and the length of the record, preventing accidental overwrites.

The Resource Manager permits non-linear retrieval of resources. Querying the location of a resource results in a message being returned indicating the file name and the location of the specified resource in that file; this information is then used to retrieve the resource.

13.4 Software Developer's Kit SDK

The SDK contains the documentation and software required to build PenPoint applications and consists of:

- Application developer's guides
- Architectural reference describing all classes and messages in the PenPoint class library
- The API reference that is a reproduction of all header files, formal messages and parameter definitions and data structures
- The header and include files
- An object-aware, source code debugger
- A database-driven class browser
- PenPoint User Interface Style Guide
- A selection of prototyping tools
- An application development environment version of PenPoint that runs under DOS on a PC

PenPoint development requires an ANSI C compiler. The Class Manager provides the object functionality and because the Class Manager is a subsysytem of the operating system rather than a language extension, this functionality is available via standard C syntax.

The debugging tools allow the programmer to set and monitor debug flags in a separate window in the PC-based development environment. The source level symbolic debugger permits:

- clsMgr objects and messages to be examined
- Break points to be set in the source code
- Multiple thread management

The application development platform for PenPoint is an 80386 processor complex, DOS-based PC with a VGA display and a digitizing tablet with stylus to simulate pen activity.

The original source code is created and tested on the PC. The tested application is downloaded to the PenPoint system either via diskette, or via PenCentral/PenTOPS. If a communications link has been established between the PenPoint system and the development PC, the debugger will function in a remote debugging mode; the the debugger user interface and symbol table reside on the PC, while the application on the PenPoint system is debugged.

13.5 User Interface Toolkit UI

The UI Toolkit is the largest API in the SDK and provides the ability to manage the user interface through layout windows via behavior supplied in ***clsTableLayout*** and ***clsCustomLayout.*** Twelve types of controls are provided, each of which is represented by a class with related behaviors.

The classes that layout the windows form the user interface elements such as buttons, tabs, handwriting fields, labels, icons, menus, frames and option sheets. Elements are called **UI components;** UI components send messages among themselves and to their clients when the user interacts with the UI components.

The UI Toolkit implements the middle layer of the appearance and functionality of the user interface architecture in PenPoint. The UI Toolkit calls on the Windows and Graphics subsystem to draw the windows. The Application Framework and the internal classes implementing the NUI use UI Toolkit objects.

The basic principle invovled in the PenPoint user interface is that all of the elements that appear in a window are themselves windows. Therefore all the UI Toolkit based design elements placed into a client window of the application are child windows to that window.

Laying out a window involves arranging the windows in such a way that when the window is displayed, all its child windows appear and are usable. The application developer only provides the high-level directives that arrange the windows and instruct the parent window to lay itself out; the system then manages the hierarchical window layout automatically (child windows first laying out their child windows).

13.5.1 User Interface Controls

All controls in PenPoint respond to gestures by the user, by sending themselves messages. Behavior is implemented for the messages that describes how the application should respond when a control is activated.

Controls are created by storing descriptive information in two data structures:

1. CONTROL_METRICS
2. CONTROL_STYLE

CONTROL_METRICS defines the object that will receive all notifications from the control when user input, such as a gesture, causes the object to provide such notification.

CONTROL_STYLE defines the appearance and behavior of the control.

Each control can have only one client to which it reports user interaction taking place within its borders.

Chapter 14. Sample PenPoint Application

This chapter describes a sample PenPoint application.

The sample application was designed to provide:

- A PenPoint Data Entry document
- Data transmission via serial port to an OS/2 2.0 system, using a null modem cable
- Update an existing OS/2 2.0 database with the transmitted data

This application will form the basis of further applications to be developed for other pen-based systems, including data transmission via modem.

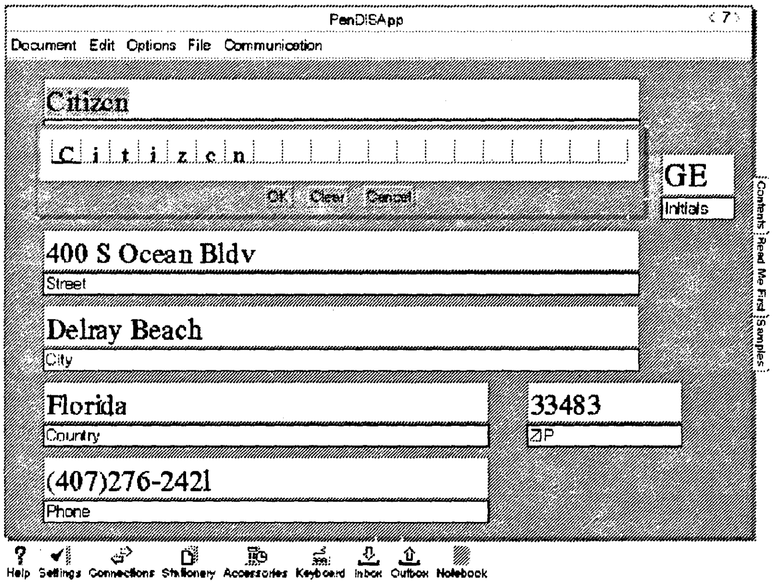
Application design included the following elements:

1. Provide a PenPoint Data Entry application (PenDISApp) using standard PenPoint SDK objects and classes.
2. Manually create an OS/2 2.0 database and Database table, through OS/2 2.0 Query Manager.
3. Access the serial port of an IBM 2521 ThinkPad and IBM PS/2.
4. Transmit the data from the PenPoint system to an OS/2 2.0 system.
5. Update the database with the data transmitted from the PenPoint system via the OS/2 2.0 SQL API.

14.1 User Interface

The following sequence of PenPoint panels graphically represents the input procedure the user would follow to:

- Create a document (Input data in PenDISApp).
- Confirm and/or update communications setup.
- Transmit data to the OS/2 2.0 system.
- File data on PenPoint system if required.


PenDISApp
Document Edit Options File Communication
OK Clear Cancel
Contents Read Me First Samples
Help Settings Connections Stationery Accessories Keyboard Inbox Outbox Notebook

Citizen

C i t i z e n

GE
Initials

400 S Ocean Bldv
Street

Delray Beach
City

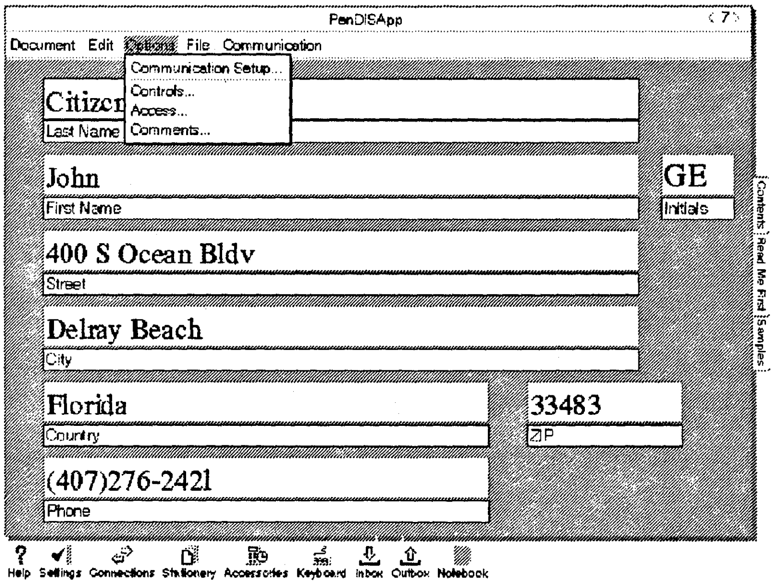
Florida
Country

33483
ZIP

(407)276-2421
Phone

Figure 18. Data Entry Panel

Figure 18 shows the PenDISApp Data Entry document with the pop-up text entry window. The user would typically tap (gesture) with the pen on an input field. The pop-up writing pad would be displayed, where the data would be entered.


PenDISApp
Document Edit Options File Communication
Communication Setup...
Controls...
Access...
Comments...
Contents Read Me First Samples
Help Settings Connections Stationery Accessories Keyboard Inbox Outbox Notebook

Citizen
Last Name

John
First Name

GE
Initials

400 S Ocean Bldv
Street

Delray Beach
City

Florida
Country

33483
ZIP

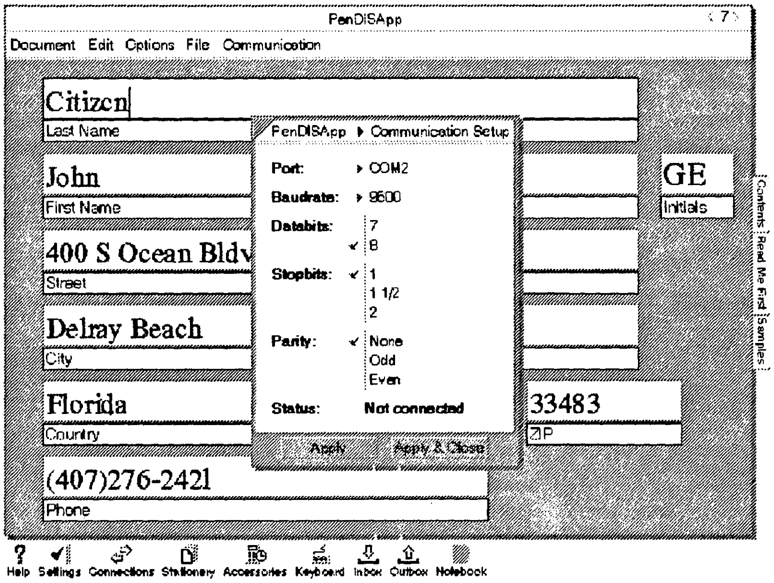
(407)276-2421
Phone

Figure 19. Options Pull-Down Menu

Figure 19 shows the Options pull-down menu. This menu reflects one application specific option, ***Communications Setup*** and three PenPoint default options:

- Controls
- Access
- Comments

Selecting Communications Setup causes the application specific Communications Option Sheet to be displayed; refer to figure Figure 20 on page 81.



Citizcn
Last Name

John
First Name

GE
Initials

400 S Ocean Bldv
Street

Delray Beach
City

Florida
Country

33483
ZIP

(407)276-2421
Phone

Port: ▸ COM2

Baudrate: ▸ 9600

Databits: 7 / ✓ 8

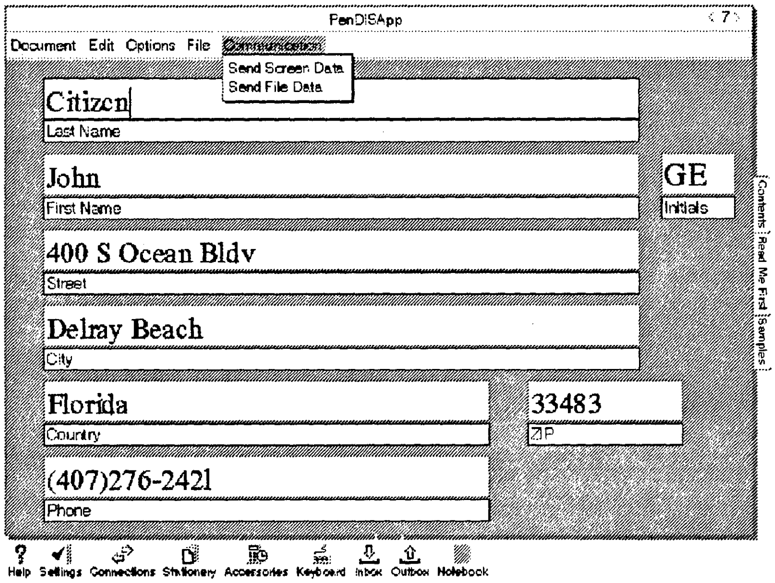
Stopbits: ✓ 1 / 1 1/2 / 2

Parity: ✓ None / Odd / Even

Status: Not connected

Figure 20. Communications Setup Options Sheet

This Option Sheet provides the user with the ability to set/reset communications options.



Citizcn
Last Name

John
First Name

GE
Initials

400 S Ocean Bldv
Street

Delray Beach
City

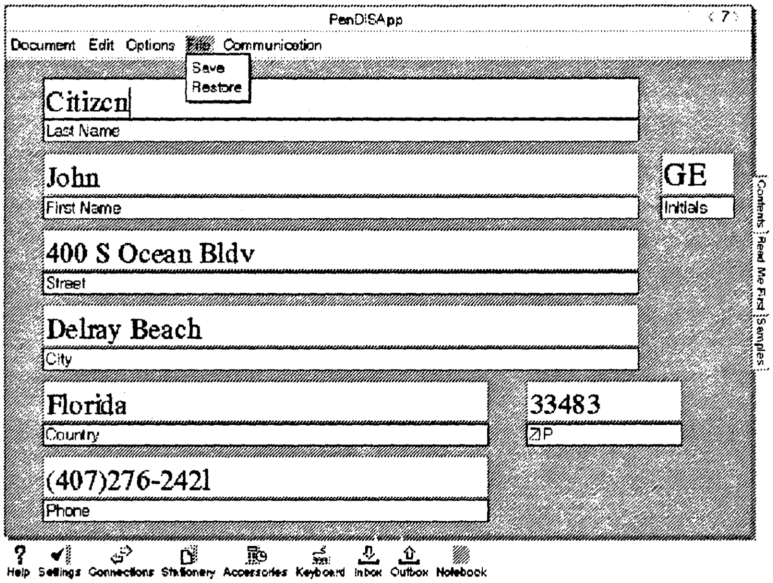
Florida
Country

33483
ZIP

(407)276-2421
Phone

Figure 21. Communication Option Menu

Figure 21 displays the communications options available to the user; both screen data, and file data may be transmitted.


PenDISApp
< 7 >
Document Edit Options File Communication
Save
Restore
Contents
Read Me First
Samples
Help Settings Connections Stationery Accessories Keyboard Inbox Outbox Notebook

Citizen
Last Name

John
First Name

GE
Initials

400 S Ocean Bldv
Street

Delray Beach
City

Florida
Country

33483
ZIP

(407)276-2421
Phone

Figure 22. File Option Menu

Figure 22 displays the options available for filing documents. The user has the choice of filing a completed data entry document, or retrieving a completed document.

14.2 Application Design Flow

Figure 23 is a graphic representation of the application design flow

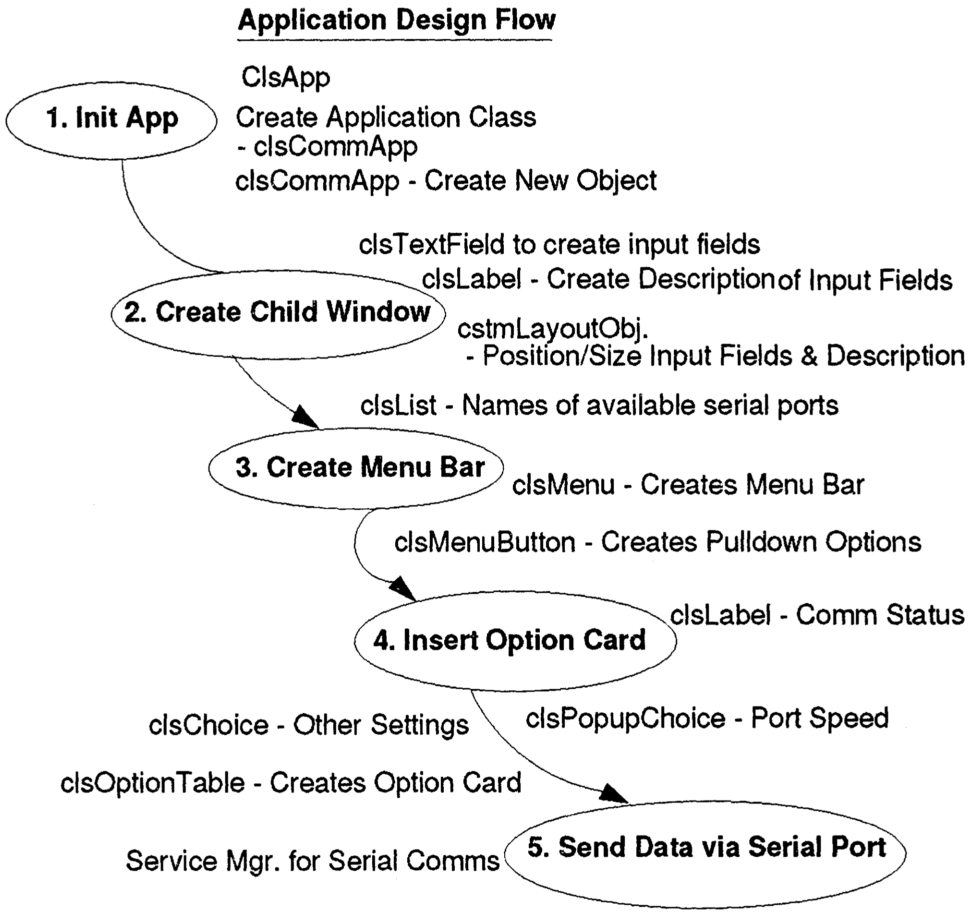

Application Design Flow
ClsApp
1. Init App
Create Application Class
- clsCommApp
clsCommApp - Create New Object
clsTextField to create input fields
clsLabel - Create Descriptionof Input Fields
2. Create Child Window
cstmLayoutObj.
- Position/Size Input Fields & Description
clsList - Names of available serial ports
3. Create Menu Bar
clsMenu - Creates Menu Bar
clsMenuButton - Creates Pulldown Options
clsLabel - Comm Status
4. Insert Option Card
clsChoice - Other Settings
clsPopupChoice - Port Speed
clsOptionTable - Creates Option Card
Service Mgr. for Serial Comms
5. Send Data via Serial Port

Figure 23. Application Design Flow

Stage 1: (Application initialization)

- Initialize application by creating a new application class, ***clsCommApp,*** using ***clsApp.***
- Generate an object from ***clsCommApp.***

Stage 2: (Child windows)

- Using ***clsTextField,*** create the input fields for the Data Entry document.
- ***clsLabel*** is used to create the input field descriptions.
- ***cstmLayoutObj*** is used to position and size both the input fields and descriptions.

- ***clsList*** is used to hold the names of the available serial ports.

Stage 3: (Document pull-down menus)

- ***clsMenu*** is used to create the document Menu Bar.
- ***clsMenuButton*** is used to create the pull-down menus.

Stage 4: (The Communications Option Sheet)

- ***clsOptionTable*** is used to create the Option Sheet.
- ***PopupChoice*** is used to create the optional settings for the port and baud rate.
- ***clsChoice*** is used to create the optional settings for data bits, stop bits and parity.
- ***clsLabel*** is used to create the communications status.

Stage 5: (Data Transmission)

The PenPoint Service Manager is used to access the serial port.

14.3 Directory Structure Distribution Diskette

```
A:\
├──PENPOINT                        Directory structure created from PenPoint
│   │  PENPOINT.DIR                Directory file for PenPoint
│   │
│   └──AP
│       │  PENPOINT.DIR            Directory file for PenPoint
│       │
│       └──PP
│              COMMAPP.MPE         File created from the Linker
│              PENPOINT.DIR        Directory file for PenPoint
│              PENDISAP            Application program
│
├──REC_PEN                         Source directory for OS/2 2.0 program
│      REC_PEN.OBJ                 Object file
│      REC_PEN.C                   Generated C source file (from SQLPREP)
│      REC_PEN.EXE                 Executable OS/2 2.0 program
│      M.CMD                       CMD file for compilation
│      REC_PEN.MAP                 Map file
│      REC_PEN.H                   C header file
│      REC_PEN.SQC                 C source file with SQL
│
└──COMM                            Source directory for PenPoint program
       COMM.C                      C source file
       COMMAPP.C                   C source file
       COMMFILE.C                  C source file
       COMMSET.C                   C source file
       COMMSEND.C                  C source file
       COMM.H                      C header file
       COMMAPP.H                   C header file
       METHOD.H                    Generated C header file (from MT)
       COMM.OBJ                    Object file
       COMMAPP.OBJ                 Object file
       COMMFILE.OBJ                Object file
       COMMSEND.OBJ                Object file
       COMMSET.OBJ                 Object file
       METHOD.OBJ                  Object file
       METHOD.TBL                  Method table
       MAKEFILE                    Makefile for compiling
```

Figure 24. Directory Structure

A.1 Make File for COMMAPP C Routines

Compiler control file used when compiling and linking COMMAPP.EXE. The source for this program is listed in section A.6, "C Source for COMMAPP.C" on page 92.

```
###############################################
#
#  WMake Makefile for CounterApp
#
# Copyright 1990, 1991,1992 GO Corporation. All Rights Reserved.
#
# You may use this Sample Code any way you please provided you
# do not resell the code and that this notice (including the above
# copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
# IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
# EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
# LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
# PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
# FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
# THE USE OR INABILITY TO USE THIS SAMPLE CODE.
#
#
# $Revision:   1.0  $
#   $Author:   gbarg $
#     $Date:   21-Jan-92 $
#
###############################################

PENPOINT_PATH = \penpoint

#  The DOS name of your project directory.
PROJ = commapp

#  Standard defines for sample code (needs the PROJ) definition
&excl.INCLUDE $(PENPOINT_PATH)\sdk\sample\sdefines.mif

#  The PenPoint name of your application
EXE_NAME        = PenDISApp

#  The linker name for your executable : company-name-V<major>(<minor>)
EXE_LNAME       = IBM-commapp-V1(0)

#  Object files needed to build your app
EXE_OBJS = method.obj comm.obj commapp.obj commfile.obj commset.obj commsend.obj

#  Libs needed to build your app
EXE_LIBS = penpoint app

# Targets

all: $(APP_DIR)\$(PROJ).exe .SYMBOLIC

#  The clean rule must be :: because it is also defined in srules
clean :: .SYMBOLIC
 -del method.h
 -del method.tc

#  Dependencies

commsend.obj: commsend.c method.h commapp.h comm.h

commset.obj: commset.c method.h commapp.h comm.h

commfile.obj: commfile.c method.h commapp.h comm.h

commapp.obj: commapp.c method.h commapp.h comm.h

comm.obj: comm.c method.h comm.h

#  Standard rules for sample code
&excl.INCLUDE $(PENPOINT_PATH)\sdk\sample\srules.mif
```

A.2 COMM.H C Header

```
/****************************************************************************
 File: comm.h

 Copyright 1990, 1991, 1992 GO Corporation. All Rights Reserved.

 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.
 $Revision:   1.0  $
   $Author:   gbarg $
     $Date:   21-Jan-92 $

 This file contains the API definition for clsComm.
******************************************************************************/
#ifndef COMM_INCLUDED
#define COMM_INCLUDED

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#define clsComm MakeWKN( 1, 1, wknPrivate)

STATUS GLOBAL ClsCommInit (void);

/******************************************************************************
 msgCommChangeu (void), returns STATUS

 Sent to observer when the comm value changes.
******************************************************************************/
#define msgCommChanged MakeMsg(clsComm, 5)

#endif // COMM_INCLUDED
```

A.3 COMMAPP.H C Header

```
/******************************************************************************
 File: commapp.h

 Copyright 1990, 1991, 1992 GO Corporation. All Rights Reserved.

 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.

 $Revision:   1.0  $
   $Author:   gbarg $
     $Date:   21-Jan-92 $

 This file contains definitions for clsCommApp.
******************************************************************************/
#ifndef COMMAPP_INCLUDED
#define COMMAPP_INCLUDED

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#define OBJECT_COUNT 8

typedef  struct
{
         int        length;
         int        x;
         int        y;
         int        width;
         TAG        uTag;
         char      *labelText;
         TAG        uLabel;
}        FIELD_INFO, *P_FIELD_INFO;

typedef  struct
{
         char       DefaultPort[nameBufLength];
         int        BaudRate;
         int        DataBits;
         int        StopBits;
         int        Parity;
}        COMM_SETUP, * P_COMM_SETUP;

typedef  struct
{
         OBJECT     Objects[OBJECT_COUNT];
         OBJECT     Labels[OBJECT_COUNT];
         COMM_SETUP CommSetupData;
         BOOLEAN    SerInstanceOk;
         BOOLEAN    CommSerConnected;
         U16        SerPortIndex;
         OBJECT     SerialNameList;
         OBJECT     commSIOService;
         OBJECT     commSIOHandle;
         OBJECT     commOptWin;
}        COMMAPP_INST, *P_COMMAPP_INST;

/* Define length of the input fields */
#define LASTNAME_LENGTH       20
```

```
#define FIRSTNAME_LENGTH        20
#define INITS_LENGTH             2
#define STREET_LENGTH           20
#define CITY_LENGTH             20
#define COUNTRY_LENGTH          15
#define ZIP_LENGTH               5
#define PHONE_LENGTH            15

typedef  struct
{
         char       LastName[LASTNAME_LENGTH + 1];
         char       FirstName[FIRSTNAME_LENGTH + 1];
         char       Inits[INITS_LENGTH + 1];
         char       Street[STREET_LENGTH + 1];
         char       City[CITY_LENGTH + 1];
         char       Country[COUNTRY_LENGTH + 1];
         char       ZIP[ZIP_LENGTH + 1];
         char       Phone[PHONE_LENGTH + 1];
}        COMM_DATA, *P_COMM_DATA;

#define SAVE_FILE   "\\\\BOOT\\COMMAPP.DAT"

#define COMM_DATA_FILE  "\\\\BOOT\\COMMAPP.PRO"

#define COMM_PORT_COM1           1
#define COMM_PORT_COM2           2

#define COMM_SETBAUD_300         1
#define COMM_SETBAUD_600         2
#define COMM_SETBAUD_1200        3
#define COMM_SETBAUD_2400        4
#define COMM_SETBAUD_4800        5
#define COMM_SETBAUD_9600        6
#define COMM_SETBAUD_19200       7

#define COMM_SETDATABITS_7       1
#define COMM_SETDATABITS_8       2

#define COMM_SETSTOPBITS_1P0     1
#define COMM_SETSTOPBITS_1P5     2
#define COMM_SETSTOPBITS_2P0     3

#define COMM_SETPARITY_NONE      1
#define COMM_SETPARITY_ODD       2
#define COMM_SETPARITY_EVEN      3

// Define a well known UID for the app
#define clsCommApp MakeWKN(1624, 1, wknGlobal)

#define msgCommSave              MakeMsg(clsCommApp, 1)
#define msgCommRestore           MakeMsg(clsCommApp, 2)
#define msgCommSendScreen        MakeMsg(clsCommApp, 3)
#define msgCommSendFile          MakeMsg(clsCommApp, 4)
#define msgCommOpenSerial        MakeMsg(clsCommApp, 5)
#define msgCommCloseSerial       MakeMsg(clsCommApp, 6)
#define msgCommSetSerialMetrics MakeMsg(clsCommApp, 7)
#define msgCommSendSerial        MakeMsg(clsCommApp, 8)
#define msgCommSetConnectStatusId MakeMsg(clsCommApp, 9)

/* define tags for inputfields */
#define LastNameTag     MakeTag(clsCommApp, 1)
#define LastNameLabel   MakeTag(clsCommApp, 2)
#define FirstNameTag    MakeTag(clsCommApp, 3)
#define FirstNameLabel  MakeTag(clsCommApp, 4)
#define InitsTag        MakeTag(clsCommApp, 5)
#define InitsLabel      MakeTag(clsCommApp, 6)
#define StreetTag       MakeTag(clsCommApp, 7)
#define StreetLabel     MakeTag(clsCommApp, 8)
#define CityTag         MakeTag(clsCommApp, 9)
#define CityLabel       MakeTag(clsCommApp, 10)
#define CountryTag      MakeTag(clsCommApp, 11)
#define CountryLabel    MakeTag(clsCommApp, 12)
#define ZIPTag          MakeTag(clsCommApp, 13)
#define ZIPLabel        MakeTag(clsCommApp, 14)
#define PhoneTag        MakeTag(clsCommApp, 15)
#define PhoneLabel      MakeTag(clsCommApp, 16)
#define tagCommMenu     MakeTag(clsCommApp, 17)

#define tagSetupCard    MakeTag(clsCommApp, 18)
#define tagPort         MakeTag(clsCommApp, 19)
#define tagBaudrate     MakeTag(clsCommApp, 20)
#define tagDatabits     MakeTag(clsCommApp, 21)
#define tagStopbits     MakeTag(clsCommApp, 22)
#define tagParity       MakeTag(clsCommApp, 23)
#define tagConnected    MakeTag(clsCommApp, 24)

/********************** Function definitions **************************/
STATUS   LOCAL      CreateInputWin(OBJECT clientObj,
                                   P_COMMAPP_INST InstData,
                                   P_FIELD_INFO Fields);
STATUS   LOCAL      AlignChildren(OBJECT cstmLayoutObj, P_COMMAPP_INST inst,
                                  P_FIELD_INFO Fields);
         void       RestoreDataFromFile(P_COMMAPP_INST inst);
         void       GetCommDataFromFile(P_COMMAPP_INST inst);
         void       GetTextData(P_COMM_DATA CommData, P_COMMAPP_INST pData);
         void       CommSendField(OBJECT self, P_CHAR Field, BOOLEAN flag);

#endif // COMMAPP_INCLUDED
```

A.4 C Source for METHOD.TBL

This is the message table that defines the behavior for the classes used.

```
/****************************************************************************
 File: method.tbl

 Copyright 1990, 1991, 1992 GO Corporation. All Rights Reserved.

 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.

 $Revision:   1.0  $
   $Author:   gbarg $
     $Date:   21-Jan-92 $

 This file contains the method tables for the classes in CommApp.
****************************************************************************/

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef COMM_INCLUDED
#include <comm.h>
#endif

#ifndef COMMAPP_INCLUDED
#include <commapp.h>
#endif

#ifndef BUTTON_INCLUDED
#include <button.h>
#endif

#ifndef OPTION_INCLUDED
#include <option.h>
#endif

#ifndef SERVMGR_INCLUDED
#include <servmgr.h>
#endif

#ifndef SIO_INCLUDED
#include <sio.h>
#endif

MSG_INFO clsCommAppMethods[] = {
 msgAppInit,                 "CommAppAppInit",    objCallAncestorBefore,
 msgAppOpen,                 "CommAppOpen",       objCallAncestorAfter,
 msgAppClose,                "CommAppClose",      objCallAncestorBefore,
 msgSave,                    "CommSave",          objCallAncestorBefore,
 msgRestore,                 "CommRestore",       objCallAncestorBefore,
 msgOptionAddCards,          "CommOptionAddCards", objCallAncestorAfter,
 msgOptionProvideCardWin,    "CommOptionProvideCard", objCallAncestorAfter,
 msgOptionApplyCard,         "CommOptionApplyCard", objCallAncestorAfter,
 msgCommSave,                "CommSaveButton",    0,
 msgCommRestore,             "CommRestoreButton", 0,
 msgCommSendScreen,          "CommSendScreenButton", 0,
 msgCommSendFile,            "CommSendFileButton", 0,
 msgCommOpenSerial,          "CommOpenSerial",    0,
 msgCommCloseSerial,         "CommCloseSerial",   0,
 msgCommSetSerialMetrics,    "CommSetSerialMetrics", 0,
 msgCommSendSerial,          "CommSendSerial",    0,
 msgCommSetConnectStatusId,  "CommSetConnectStatusId", 0,
 msgSMConnectedChanged,      "CommSMConnectedChanged", 0,
 msgSioEventHappened,        "CommSioEventHappened", 0,
 0
};

CLASS_INFO classInfo[] = {
 "clsCommAppTable", clsCommAppMethods, 0,
 0
};
```

A.5 C Source for COMM.C

This module saves the instance data when it goes to dormant state and restores the data when the program is reactivated.

```
/****************************************************************************

 File: comm.c

 Copyright 1990, 1991, 1992 GO Corporation. All Rights Reserved.

 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.

 $Revision:   1.0  $
   $Author:   gbarg $
     $Date:   21-Jan-92 $

 This file contains the class definition and methods for clsComm.

****************************************************************************/

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#ifndef FS_INCLUDED
#include <fs.h>
#endif

#ifndef FRAME_INCLUDED
#include <frame.h>
#endif

#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef COMM_INCLUDED
#include <comm.h>
#endif

#ifndef COMMAPP_INCLUDED
#include <commapp.h>
#endif

#include <method.h>

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Defines, Types, Globals, Etc
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Local Functions
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Message Handlers
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/****************************************************************************
 CommSave

 Respond to msgSave.

****************************************************************************/

        MSG_HANDLER CommSave(const MESSAGE msg,
                             const OBJECT self,
                             const P_OBJ_SAVE pArgs,
                             const CONTEXT ctx,
                             const P_COMMAPP_INST pData)

{
        STREAM_READ_WRITE fsWrite;
        STATUS     s;

 Debugf("Comm:CommSave");

/* Write instance data to the file. */
fsWrite.numBytes= SizeOf(COMMAPP_INST);
fsWrite.pBuf= pData;
ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s);

return stsOK;
MsgHandlerParametersNoWarning;
}
```

```
/****************************************************************************/
/* CommRestore                                                              */
/*                                                                          */
/* Respond to msgRestore.                                                   */
/*                                                                          */
/****************************************************************************/

        MSG_HANDLER CommRestore(const MESSAGE msg,
                                const OBJECT self,
                                const P_OBJ_RESTORE pArgs,
                                const CONTEXT ctx,
                                const P_COMMAPP_INST pData)
{
        COMMAPP_INST inst;
        STREAM_READ_WRITE fsRead;
        APP_METRICS am;
        OBJECT     frmWin;
        STATUS     s;
        int        i;
static const TAG   Tags[] =
                   {
                   LastNameTag,
                   FirstNameTag,
                   InitsTag,
                   StreetTag,
                   CityTag,
                   CountryTag,
                   ZIPTag,
                   PhoneTag,
                   };

Debugf("Comm:CommRestore");

/* Read instance data from the file. */
fsRead.numBytes= SizeOf(COMMAPP_INST);
fsRead.pBuf= &inst;
ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s);

/* Get the proper UIDs of the input fields */
ObjCallWarn(msgAppGetMetrics, self, &am);
ObjCallJmp(msgFrameGetClientWin, am.mainWin, &frmWin, s, Error);
for (i = 0; i < (sizeof(Tags) / sizeof(TAG)); i++)
   {
   inst.Objects[i] = (WIN)ObjectCall(msgWinFindTag, frmWin,
                                    (P_ARGS)Tags[i]);
   }

/* Update instance data. */
ObjectWrite(self, ctx, &inst);

return stsOK;

Error:
return(s);

MsgHandlerParametersNoWarning;
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                           Installation
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
```

A.6 C Source for COMMAPP.C

This module creates the new message class clsCommApp and the child window with the input fields and labels. It also polls available communication ports and holds the information in a list.

```
/****************************************************************************

 File: commapp.c

 Copyright 1990, 1991, 1992 GO Corporation. All Rights Reserved.

 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.

 $Revision:   1.0  $
   $Author:   gbarg $
     $Date:   21-Jan-92 $
 This file contains the implementation of the application class.

****************************************************************************/

#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef APPMGR_INCLUDED
#include <appmgr.h>
#endif

#ifndef SERVMGR_INCLUDED
#include <servmgr.h>
#endif

#ifndef STROBJ_INCLUDED
#include <strobj.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 TKFIELD_INCLUDED
#include <tkfield.h>
#endif

#ifndef MENU_INCLUDED
#include <menu.h>
#endif

#ifndef COMM_INCLUDED
#include <comm.h>
#endif

#ifndef COMMAPP_INCLUDED
#include <commapp.h>
#endif

#ifndef CLAYOUT_INCLUDED
#include <clayout.h>
#endif

#ifndef BUTTON_INCLUDED
#include <button.h>
#endif

#include <method.h>

#include <string.h>
#include <stdio.h>

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                           Defines, Types, Globals, Etc
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                           Local Functions
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/****************************************************************************/
/* CreateInputWin                                                           */
/*                                                                          */
/* Create the input fields in the child window.                             */
/*                                                                          */
/****************************************************************************/
STATUS   LOCAL      CreateInputWin(OBJECT clientObj,
                                   P_COMMAPP_INST InstData,
                                   P_FIELD_INFO Fields)
{
         TEXT_FIELD_NEW tfn;
         LABEL_NEW  ln;
         STATUS     s;
         int        i;

for (i = 0; i < OBJECT_COUNT; i++)
   {
   /* Create the input field */
   ObjCallWarn(msgNewDefaults, clsTextField, &tfn);

   tfn.win.tag = Fields[i].uTag;
   tfn.control.client = clientObj;
   tfn.label.style.numCols = lsNumAbsolute;
   tfn.field.maxLen = tfn.label.cols = Fields[i].length;
   tfn.border.style.edge = bsEdgeAll;
   tfn.field.style.editType = fstPopUp; /* input only through popup window */
   tfn.label.style.xAlignment = lsAlignLeft;
   tfn.label.style.scaleUnits = bsUnitsFitWindowProper;
   ObjCallRet(msgNew, clsTextField, &tfn, s);

   InstData->Objects[i] = tfn.object.uid;

   /* Create the label for the field description */
   ObjCallRet(msgNewDefaults, clsLabel, &ln, s);
   ln.win.tag = Fields[i].uLabel;
   ln.label.style.scaleUnits = bsUnitsFitWindowProper;
   ln.label.style.xAlignment = lsAlignLeft;
   ln.border.style.edge = bsEdgeAll;
   ln.label.pString = Fields[i].labelText;
   ObjCallRet(msgNew, clsLabel, &ln, s);

   InstData->Labels[i] = ln.object.uid;
   }

return(stsOK);
}

/*****************************************************************************/
/* AlignChildren                                                             */
/*                                                                           */
/* Position/size the input fields and labels                                 */
/*                                                                           */
/*****************************************************************************/
STATUS   LOCAL      AlignChildren(OBJECT cstmLayoutObj, P_COMMAPP_INST pInst,
                                   P_FIELD_INFO Fields)
{
         CSTM_LAYOUT_CHILD_SPEC clcs;
         STATUS     s;
         int        i;

for (i = 0; i < OBJECT_COUNT; i++)
   {
   /* Set the size and position for the input fields */
   CstmLayoutSpecInit(&clcs.metrics);
   clcs.metrics.h.constraint = clPctOf;
   clcs.metrics.h.value      = 9;       /* Height of the input field */
   clcs.metrics.w.constraint = clPctOf;
   clcs.metrics.x.constraint = ClAlign(clMinEdge, clPctOf, clMaxEdge);
   clcs.metrics.y.constraint = ClAlign(clMinEdge, clPctOf, clMaxEdge);

   clcs.child                = pInst->Objects[i];
   clcs.metrics.w.value      = Fields[i].width;
   clcs.metrics.x.value      = Fields[i].x;
   clcs.metrics.y.value      = Fields[i].y;
   ObjCallRet(msgCstmLayoutSetChildSpec, cstmLayoutObj, &clcs, s);

   /* Set the size and position for the input fields */
   clcs.child                = pInst->Labels[i];
   clcs.metrics.h.value      = 5;       /* Height of the label */
   clcs.metrics.w.value      = Fields[i].width;
   clcs.metrics.x.value      = Fields[i].x;
   clcs.metrics.y.value      = Fields[i].y - 5;
   ObjCallRet(msgCstmLayoutSetChildSpec, cstmLayoutObj, &clcs, s);
   }

return(stsOK);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                           Message Handlers                                */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************/
/* CommAppAppInit                                                            */
/*                                                                           */
/* Respond to msgAppInit.                                                    */
/*                                                                           */
/*****************************************************************************/
         MSG_HANDLER CommAppAppInit(const MESSAGE msg,
                                    const OBJECT self,
                                    const P_ARGS pArgs,
                                    const CONTEXT ctx,
                                    const P_IDATA pData)
{
         APP_METRICS am;
         STATUS     s;
         int        i;
         COMMAPP_INST inst;
         CSTM_LAYOUT_NEW cln;
         WIN_METRICS wm;
         MENU_NEW   mn;

/* Description of the input fields and the labels */
static const FIELD_INFO Fields[OBJECT_COUNT] =
{
/* Length,         X,  Y, Width, Tag,         Labeltext,    Tag */
LASTNAME_LENGTH,   5, 88, 80, LastNameTag,  "Last Name",  LastNameLabel,
FIRSTNAME_LENGTH,  5, 72, 80, FirstNameTag, "First Name", FirstNameLabel,
INITS_LENGTH,     88, 72, 10, InitsTag,     "Initials",   InitsLabel,
STREET_LENGTH,     5, 56, 80, StreetTag,    "Street",     StreetLabel,
CITY_LENGTH,       5, 40, 80, CityTag,      "City",       CityLabel,
COUNTRY_LENGTH,    5, 24, 60, CountryTag,   "Country",    CountryLabel,
ZIP_LENGTH,       70, 24, 21, ZIPTag,       "ZIP",        ZIPLabel,
PHONE_LENGTH,      5,  0, 60, PhoneTag,     "Phone",      PhoneLabel,
};

/* Description of the menus in the menu bar */
static   TK_TABLE_ENTRY CommAppMenuBar[] =
{
   {"File", 0, 0, 0, tkMenuPullDown, clsMenuButton},
       {"Save", msgCommSave},
       {"Restore", msgCommRestore},
       {pNull},
   {"Communication", 0, 0, 0, tkMenuPullDown, clsMenuButton},
       {"Send Screen Data", msgCommSendScreen},
       {"Send File Data", msgCommSendFile},
       {pNull},
   {pNull}
};

Debugf("CommApp:CommAppAppInit -- received msgAppInit");

/* Initialize instance data */
memset((P_CHAR)&inst, '\0', sizeof(COMMAPP_INST));

/* Create Child Windows */
CreateInputWin(self, &inst, Fields);

ObjCallWarn(msgNewDefaults, clsCustomLayout, &cln);

cln.border.style.backgroundInk = bsInkGray33;
ObjCallWarn(msgNew, clsCustomLayout, &cln);

/* Create the menubar */
ObjCallRet(msgNewDefaults, clsMenu, &mm, s);
mn.tkTable.client = self;
mn.tkTable.pEntries = CommAppMenuBar;
mn.menu.style.type = msTypeMenuBar;
ObjCallRet(msgNew, clsMenu, &mn, s);

/* Insert the menubar */
ObjCallRet(msgAppCreateMenuBar, self, &mn.object.uid, s);
ObjCallRet(msgAppGetMetrics, self, &am, s);
ObjCallRet(msgFrameSetMenuBar, am.mainWin, mn.object.uid, s);

wm.parent = cln.object.uid;
wm.options = wsPosTop;

/* Insert the input windows and labels */
for (i = 0; i < OBJECT_COUNT; i++)
   {
   ObjCallRet(msgWinInsert, inst.Objects[i], &wm, s);
   ObjCallRet(msgWinInsert, inst.Labels[i], &wm, s);
   }

/* Set the size and position of the input fields and labels */
AlignChildren(cln.object.uid, &inst, Fields);

/* Read the saved file and put the contents into the input fields */
RestoreDataFromFile(&inst);

/* Read the setting of the serial communication */
GetCommDataFromFile(&inst);

/* Update instance data */
ObjectWrite(self, ctx, &inst);

ObjCallWarn(msgAppGetMetrics, self, &am);
ObjCallJmp(msgFrameSetClientWin, am.mainWin, cln.object.uid, s, Error);

return(stsOK);
Error:
return(s);
MsgHandlerParametersNoWarning;
}

/****************************************************************************/
/* CommAppOpen                                                              */
/*                                                                          */
/* 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.                                                */
/*                                                                          */
/****************************************************************************/
MSG_HANDLER CommAppOpen(const MESSAGE msg,
                        const OBJECT self,
                        const P_ARGS pArgs,
                        const CONTEXT ctx,
                        const P_COMMAPP_INST pData)
{
        STATUS     s;
        COMMAPP_INST inst;
        LIST_NEW   ln;
        LIST_ENTRY le;
        U16        n;
        OBJECT     serlist;
        CHAR       buffer[nameBufLength];
```

```
         IM_GET_SET_NAME gn;
         STROBJ_NEW sn;
         BOOLEAN    haveName;

Debugf("CommApp:CommAppOpen -- received msgAppOpen");

/* Copy the instance data to local memory */
memcpy((P_CHAR)&inst, (P_CHAR)pData, sizeof(COMMAPP_INST));

/* Create a list to hold the name of the serial port drivers. */
ObjCallJmp(msgNewWithDefaults, clsList, &ln, s, Error);

/* Get the serial port driver list, copy names */
ObjCallJmp(msgIMGetList, theSerialDevices, &serlist, s, Error2);

/* How many entries are in the list? */
ObjCallJmp(msgListNumItems, serlist, &n, s, Error2);

if (n == 0)
   {
   /* there aren't any service instances! Bug out */
   }
else
   {
   /* Get the list of available service instances from theSerialDevices.   */
   /* Walk down the list, get the name of each instance and store it in my */
   /* own list. Check if a service instance should be default (if none has */
   /* been stored as default yet), or compare it with a previously stored  */
   /* service name.                                                        */
   inst.SerInstanceOk = true;
   haveName = false;
   while (!haveName)
      {
      for (le.position = 0; le.position < n; le.position++)
         {
         ObjCallJmp(msgListGetItem, serlist, &le, s, Error2);
         if (le.item != pNull)
            {
            gn.handle = (OBJECT)le.item;
            gn.pName = buffer;
            ObjCallJmp(msgIMGetName, theSerialDevices, &gn, s, Error2);

            /* Copy name */
            ObjCallWarn(msgNewDefaults, clsString, &sn);
            sn.strobj.pString = buffer;
            ObjCallJmp(msgNew, clsString, &sn, s, Error2);

            /* Add it to the end of the list */
            ObjCallJmp(msgListAddItem, ln.object.uid, sn.object.uid, s, Error2);

            /* Check if this is this one is selected or that I  */
            /* should make a default.                           */
            if (inst.CommSetupData.DefaultPort[0] == '\0' && le.position == 0)
               {
               Debugf("Setting default SerialServ to %s", buffer);
               strcpy(inst.CommSetupData.DefaultPort, buffer);
               haveName = true;
               }
            else
               {
               if ((strcmp(inst.CommSetupData.DefaultPort, buffer)) == 0)
                  {
                  Debugf("Setting default SerialServ to %s", buffer);
                  inst.SerPortIndex = le.position;
                  haveName = true;
                  }
               }
            }
         }
      if (!haveName)
         {
         if (inst.CommSetupData.DefaultPort[0] != '\0')
            inst.CommSetupData.DefaultPort[0] = '\0';
         else
            break;
         }

      }
   /* Keep list uid */
   inst.SerialNameList = ln.object.uid;

   /* Update instance data */
   ObjectWrite(self, ctx, &inst);

   /* Open serial port */
   ObjCallJmp(msgCommOpenSerial, self, (P_ARGS)pNull, s, Error);

   }
/* Destroy the service instance list */
ObjCallWarn(msgDestroy, serlist, pNull);

return stsOK;
Error2:
/* Destroy the service instance list */
ObjCallWarn(msgDestroy, serlist, pNull);
Error:
return(s);
```

```
MsgHandlerParametersNoWarning;
}

/****************************************************************************/
/* CommAppClose                                                              */
/*                                                                           */
/* Respond to msgAppClose.                                                   */
/*                                                                           */
/* 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.                                                 */
/*                                                                           */
/****************************************************************************/
MSG_HANDLER CommAppClose(const MESSAGE msg,
                         const OBJECT self,
                         const P_ARGS pArgs,
                         const CONTEXT ctx,
                         const P_COMMAPP_INST pData)
{
        LIST_FREE  lf;

ObjCallWarn(msgCommCloseSerial, self, (P_ARGS)pNull);

lf.key = objWKNKey;
lf.mode = listFreeItemsAsObjects;
ObjCallWarn(msgListFree, pData->SerialNameList, &lf);

/* Destroy option sheet */
if (pData->commOptWin)
   ObjCallWarn(msgDestroy, pData->commOptWin, Nil(P_ARGS));

return stsOK;

MsgHandlerParametersNoWarning;
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                           Installation                                  */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */


/****************************************************************************/
/* ClsCommAppInit                                                            */
/*                                                                           */
/* Install the application.                                                  */
/*                                                                           */
/****************************************************************************/
STATUS   GLOBAL     ClsCommAppInit(void)
{
        APP_MGR_NEW new;
        STATUS     s;

/* Create the new message class clsCommApp */
ObjCallJmp(msgNewDefaults, clsAppMgr, &new, s, Error);

new.object.uid      = clsCommApp;
new.cls.pMsg        = clsCommAppTable;
new.cls.ancestor    = clsApp;
new.cls.size        = SizeOf(COMMAPP_INST);
new.cls.newArgsSize = SizeOf(APP_NEW);

strcpy(new.appMgr.company, "IBM Corporation");
strcpy(new.appMgr.defaultDocName, "PenDISApp");

ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);

return(stsOK);

Error:                                  /* I don't like goto's in C */
return(s);
}

/****************************************************************************/
/* main                                                                      */
/*                                                                           */
/* Main application entry point.                                             */
/*                                                                           */
/****************************************************************************/
        void CDECL main(int  argc,
                        char *argv[],
                        U16  processCount)
{
if (processCount == 0)
   {
   StsWarn(ClsCommAppInit());
   AppMonitorMain(clsCommApp, objNull);
   }
else
   {
   AppMain();
   }

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

A.7 C Source for COMMFILE.C

This module contains the routines to save and restore the data entered in the input fields onto the hard disk or solid state file (SSF).

```
/******************************************************************************

 File: commapp.c

 Copyright 1990, 1991, 1992 GO Corporation. All Rights Reserved.

 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.

 $Revision:   1.0  $
   $Author:   gbarg $
     $Date:   21-Jan-92 $

 This file contains saving to a file and restoring from there.

******************************************************************************/

#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef APPMGR_INCLUDED
#include <appmgr.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 TKFIELD_INCLUDED
#include <tkfield.h>
#endif

#ifndef MENU_INCLUDED
#include <menu.h>
#endif

#ifndef TXTDATA_INCLUDED
#include <txtdata.h>
#endif

#ifndef COMM_INCLUDED
#include <comm.h>
#endif

#ifndef COMMAPP_INCLUDED
#include <commapp.h>
#endif

#ifndef CLAYOUT_INCLUDED
#include <clayout.h>
#endif

#ifndef BUTTON_INCLUDED
#include <button.h>
#endif

#include <method.h>

#include <string.h>
#include <stdio.h>

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                           Defines, Types, Globals, Etc                  */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
static const int    FieldLength[] =
                    {
                    LASTNAME_LENGTH,
                    FIRSTNAME_LENGTH,
                    INITS_LENGTH,
                    STREET_LENGTH,
                    CITY_LENGTH,
                    COUNTRY_LENGTH,
                    ZIP_LENGTH,
                    PHONE_LENGTH,
                    };

static   CHAR       save_file[] = SAVE_FILE;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                        Local Functions                                */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************/
/* Read saved data from COMMAPP.DAT and put the data into the input         */
/* fields.                                                                   */
/*****************************************************************************/
         void       RestoreDataFromFile(P_COMMAPP_INST inst)
{
         FILE      *stream;
         int        i;
         COMM_DATA  CommData;
         P_CHAR     pString;

memset((P_CHAR)&CommData, '\0', sizeof(COMM_DATA));

/* Read data from file COMMAPP.DAT */
if ((stream = fopen(save_file, "rb")) != NULL)
   {
   Debugf("File Open (Read) Ok.");
   i = fread((P_CHAR)&CommData, sizeof(COMM_DATA), 1, stream);
   if (i == 1)
      {
      Debugf("Read Ok.");

      pString = (P_CHAR)&CommData;

      /* Put data to the input fields */
      for (i = 0; i < OBJECT_COUNT; i++)
         {
         ObjCallWarn(msgLabelSetString, inst->Objects[i], pString);
         pString = &pString[FieldLength[i] + 1];
         }
      }
   fclose(stream);
   }
}

/*****************************************************************************/
/* Get Data from fields and stor it in structure.                            */
/*****************************************************************************/
         void       GetTextData(P_COMM_DATA CommData, P_COMMAPP_INST pData)
{
         CONTROL_STRING cs;
         int        i;

memset((P_CHAR)CommData, '\0', sizeof(COMM_DATA));

cs.pString = (P_CHAR)CommData;

for (i = 0; i < OBJECT_COUNT; i++)
   {
   cs.len = FieldLength[i] + 1;
   ObjCallWarn(msgLabelGetString, pData->Objects[i], &cs);
   cs.pString = &cs.pString[FieldLength[i] + 1];
   }
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                       Message Handlers
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*****************************************************************************/
/* CommSaveButton                                                            */
/*                                                                           */
/* Respond to msgCommSaveButton                                              */
/*                                                                           */
/*****************************************************************************/
         MSG_HANDLER CommSaveButton(const MESSAGE msg,
                                const OBJECT self,
                                const P_ARGS pArgs,
                                const CONTEXT ctx,
                                const P_COMMAPP_INST pData)
{
         COMM_DATA  CommData;
         int        i;
         FILE      *stream;

/* Get data from input fields */
GetTextData(&CommData, pData);

/* Save data in file COMMAPP.DAT */
if ((stream = fopen(save_file, "wb")) != NULL)
   {
   Debugf("File Open (Write) Ok.");
   i = fwrite((P_CHAR)&CommData, sizeof(COMM_DATA), 1, stream);
   if (i == 1)
      {
```

```
      Debugf("Write Ok.");
      }
   fclose(stream);
   }

return(stsOK);

MsgHandlerParametersNoWarning;
}

/***************************************************************************/
/* CommRestoreButton                                                       */
/*                                                                         */
/* Respond to msgCommRestoreButton                                         */
/*                                                                         */
/***************************************************************************/
         MSG_HANDLER CommRestoreButton(const MESSAGE msg,
                                       const OBJECT self,
                                       const P_ARGS pArgs,
                                       const CONTEXT ctx,
                                       const P_COMMAPP_INST pData)
{
Debugf("Message msgCommRestoreButton");

/* Get data from file and store it in the input fields */
RestoreDataFromFile(pData);

return(stsOK);

MsgHandlerParametersNoWarning;
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                            Installation
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
```

A.8 C Source for COMMSET.C

This module contains the routines to insert and handle the option card for setting the serial port.

```
/****************************************************************************
 File: commset.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:   05 Feb 1992 09:07:10  $

 This file contains the clsOptionTable demoing code of the tkdemo application.
****************************************************************************/

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef SEL_INCLUDED
#include <sel.h>
#endif

#ifndef WIN_INCLUDED
#include <win.h>
#endif

#ifndef TLAYOUT_INCLUDED
#include <tlayout.h>
#endif

#ifndef TKTABLE_INCLUDED
#include <tktable.h>
#endif

#ifndef SWIN_INCLUDED
#include <swin.h>
#endif
```

```
#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef APPTAG_INCLUDED
#include <apptag.h>
#endif

#ifndef CHOICE_INCLUDED
#include <choice.h>
#endif

#ifndef POPUPCH_INCLUDED
#include <popupch.h>
#endif

#ifndef STROBJ_INCLUDED
#include <strobj.h>
#endif

#ifndef OPTTABLE_INCLUDED
#include <opttable.h>
#endif

#ifndef OPTION_INCLUDED
#include <option.h>
#endif

#ifndef PREFS_INCLUDED
#include <prefs.h>
#endif

#ifndef BUSY_INCLUDED
#include <busy.h>
#endif

#ifndef COMM_INCLUDED
#include <comm.h>
#endif

#ifndef COMMAPP_INCLUDED
#include <commapp.h>
#endif

#include <method.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static const CHAR   comm_data_file[] = COMM_DATA_FILE;

/***************************************************************************/
/* Read the communication setup file                                       */
/***************************************************************************/
         void       GetCommDataFromFile(P_COMMAPP_INST inst)
{
         FILE      *stream;
         int        ReadOk = 0;

static const COMM_SETUP CommSetupDefaults =
                    {
                    "",
                    COMM_SETBAUD_9600,
                    COMM_SETDATABITS_8,
                    COMM_SETSTOPBITS_1P0,
                    COMM_SETPARITY_NONE
                    };

/* Read data from file COMMAPP.PRO */
if ((stream = fopen(comm_data_file, "rb")) != NULL)
   {
   Debugf("Comm File Open (Read) Ok.");
   ReadOk = fread((P_CHAR)&inst->CommSetupData, sizeof(COMM_SETUP), 1, stream);
   fclose(stream);
   }

if (ReadOk != 1)
   memcpy((P_CHAR)&inst->CommSetupData, (P_CHAR)&CommSetupDefaults,
          sizeof(COMM_SETUP));
}

/***************************************************************************/
/* CommOptionAddCards                                                      */
/*                                                                         */
/* Handles msgOptionAddCards.                                              */
/*                                                                         */
/* Note on error handling: Once a card has been added to the sheet,       */
/* destroying the sheet will destroy the card.                             */
/***************************************************************************/
MsgHandlerWithTypes(CommOptionAddCards, P_OPTION_TAG, P_COMMAPP_INST)
{
         OPTION_CARD card;
         STATUS     s;

/*********************************************************************/
/* Determine which sheet is requesting the cards. Only create       */
/* the cards if it is the Document option sheet. This test is       */
/* only needed by application subclasses. Other subclasses (such    */
/* as Tic-Tac-Toe's view class) don't need to perform such a test.   */
/*********************************************************************/

Debugf("commOptionsAddCard, p1=%08.8lX, p2=%08.8lX",
pArgs->tag, (U32)tagAppDocOptSheet);

if (pArgs->tag == tagAppDocOptSheet)
   {
   Debugf("pData->SerInstanceOk = %d", (U16)pData->SerInstanceOk);
   if (pData->SerInstanceOk)
      {
      /* Create the card. */
      card.tag = tagSetupCard;
      card.win = objNull;
      card.pName = "Communication Setup";
      card.client = self;
      ObjCallJmp(msgOptionAddLastCard, pArgs->option, &card, s, Error);
      }
   }

return stsOK;
MsgHandlerParametersNoWarning;

Error:
        return s;
} /* End CommOptionAddCards */

/****************************************************************************/
/* CommOptionProvideCard                                                    */
/*                                                                          */
/* Handles msgOptionProvideCardWin                                          */
/****************************************************************************/
MsgHandlerWithTypes(CommOptionProvideCard, P_OPTION_CARD, P_COMMAPP_INST)
{
         STATUS     s;
         LIST_ENTRY le;
         OPTION_TABLE_NEW new;
         BUTTON_NEW bn;
         WIN_METRICS wm;
         P_CHAR     sername;
         WIN        control;
         WIN        choice;
         U16        n;
         COMMAPP_INST inst;

/* Description of the option card */
static const TK_TABLE_ENTRY SetupCardEntries[] =
{
        {"Port:"},
            {0, 0, 0, tagPort, tkNoClient, clsPopupChoice},
            {pNull},
        {"Baudrate:"},
            {0, 0, 0, tagBaudrate, tkNoClient, clsPopupChoice},
            {"300", 0, 0, 1},
            {"600", 0, 0, 2},
            {"1200", 0, 0, 3},
            {"2400", 0, 0, 4},
            {"4800", 0, 0, 5},
            {"9600", 0, 0, 6},
            {"19200", 0, 0, 7},
            {pNull},
        {"Databits:"},
            {0, 0, 0, tagDatabits, tkNoClient, clsChoice},
            {"7", 0, 0, 1},
            {"8", 0, 0, 2},
            {pNull},
        {"Stopbits:"},
            {0, 0, 0, tagStopbits, tkNoClient, clsChoice},
            {"1", 0, 0, 1},
            {"1 1/2", 0, 0, 2},
            {"2", 0, 0, 3},
            {pNull},
        {"Parity:"},
            {0, 0, 0, tagParity, tkNoClient, clsChoice},
            {"None", 0, 0, 1},
            {"Odd", 0, 0, 2},
            {"Even", 0, 0, 3},
            {pNull},
        {"Status:"},
            {0, 0, 0, tagConnected, tkLabelStringId | tkNoClient |
                                    tkInputDisable | tkBorderMarginNone,
                                    clsLabel},
        {pNull}
};

pArgs->win = objNull;

if (pArgs->tag == tagSetupCard)
   {
   /* Create the option card */
   memcpy((P_CHAR)&inst, (P_CHAR)pData, sizeof(COMMAPP_INST));

   ObjCallRet(msgNewDefaults, clsOptionTable, &new, s);
   new.tkTable.pEntries = SetupCardEntries;
   new.win.tag = tagSetupCard;
   ObjCallRet(msgNew, clsOptionTable, &new, s);
   inst.commOptWin = pArgs->win = new.object.uid;
   /* Copy data back to protected memory */
   ObjectWrite(self, ctx, &inst);

   /* Check the items in the card */
   control = (WIN)ObjectCall(msgWinFindTag, pArgs->win,
                             (P_ARGS)tagPort);

   /* Get the choice of the popup choice to insert buttons in,  */
   /* representing the various serial port devices available.   */

   ObjCallRet(msgPopupChoiceGetChoice, control, &choice, s);

   /* How many items in list? */
   ObjCallRet(msgListNumItems, pData->SerialNameList, (P_ARGS)&n, s);

   Debugf("Creating %ld choice items", n);

   for (le.position = 0; le.position < n; le.position++)
      {
      /* Get item at this position in the list */
      ObjCallRet(msgListGetItem, pData->SerialNameList, &le, s);

      /* Get string for this object */
      ObjCallRet(msgStrObjGetStr, le.item, &sername, s);

      /* Create a button for the popup menu */
      ObjCallWarn(msgNewDefaults, clsButton, &bn);

      /* Get defaults for this tk type */
      ObjCallWarn(msgTkTableChildDefaults, choice, &bn);

      /* Make it look like standard popup item */
      bn.label.style.decoration = lsDecorationPopup;
      bn.label.pString = sername;

      /* Give it the index in the list as tag so I can  */
      /* link it easily to the string list              */
      bn.win.tag = le.position;

      ObjCallRet(msgNew, clsButton, &bn, s);

      /* Insert it in the choice */
      Debugf("Inserting %s", sername);
      wm.parent = choice;
      wm.options = wsPosTop;
      ObjCallRet(msgWinInsert, bn.object.uid, &wm, s);
      }

   /* Set communication port */
   ObjCallWarn(msgControlSetValue, control,
               (P_ARGS)pData->SerPortIndex);

   /* Set baudrate */
   control = (WIN)ObjectCall(msgWinFindTag, pArgs->win,
                             (P_ARGS)tagBaudrate);
   ObjCallWarn(msgControlSetValue, control,
               (P_ARGS)pData->CommSetupData.BaudRate);

   /* Set 7 oder 8 databits */
   control = (WIN)ObjectCall(msgWinFindTag, pArgs->win,
                             (P_ARGS)tagDatabits);
   ObjCallWarn(msgControlSetValue, control,
               (P_ARGS)pData->CommSetupData.DataBits);

   /* Set 1, 1 1/2 or 2 stopbits */
   control = (WIN)ObjectCall(msgWinFindTag, pArgs->win,
                             (P_ARGS)tagStopbits);
   ObjCallWarn(msgControlSetValue, control,
               (P_ARGS)pData->CommSetupData.StopBits);

   /* Set parity (odd, even or none) */
   control = (WIN)ObjectCall(msgWinFindTag, pArgs->win, (P_ARGS)tagParity);
   ObjCallWarn(msgControlSetValue, control,
               (P_ARGS)pData->CommSetupData.Parity);

   /* Set the correct resource id for the status label; */
   /* connected or not.                                 */

   ObjCallWarn(msgCommSetConnectStatusId, self,
               (pData->CommSerConnected) ? (P_ARGS)"Connected" :
                                           (P_ARGS)"Not connected");

   /* Mark card as clean */
   ObjCallRet(msgControlSetDirty, pArgs->win, (P_ARGS)false, s);
   }

return(stsOK);
MsgHandlerParametersNoWarning;
} /* End CommOptionProvideCard */

/******************************************************************************/
/* CommOptionApplyCard                                                        */
/*                                                                            */
/* Handles msgOptionApplyCard                                                 */
/******************************************************************************/
MsgHandlerWithTypes(CommOptionApplyCard, P_OPTION_CARD, P_COMMAPP_INST)
{
         WIN        control;
         TAG        Tag;
         STATUS     s;
         COMMAPP_INST inst;
         FILE       *stream;
         int        i;
         U32        value;
         LIST_ENTRY le;
         P_CHAR     sername;

if (pArgs->tag == tagSetupCard)
   {
   Debugf("msgOptionApplyCard");

   memcpy((P_CHAR)&inst, (P_CHAR)pData, sizeof(COMMAPP_INST));

   /* Get the data */
   control = (WIN)ObjectCall(msgWinFindTag, pArgs->win, (P_ARGS)tagPort);
   ObjCallRet(msgControlGetValue, control, (P_ARGS)&value, s);
   inst.SerPortIndex = le.position = value;
   ObjCallRet(msgListGetItem, pData->SerialNameList, &le, s);
   ObjCallRet(msgStrObjGetStr, le.item, &sername, s);
   strncpy(inst.CommSetupData.DefaultPort, sername,
           nameBufLength * SizeOf(CHAR));

   control = (WIN)ObjectCall(msgWinFindTag, pArgs->win, (P_ARGS)tagBaudrate);
   ObjCallRet(msgControlGetValue, control, (P_ARGS)&Tag, s);
   inst.CommSetupData.BaudRate = (int)Tag;

   control = (WIN)ObjectCall(msgWinFindTag, pArgs->win, (P_ARGS)tagDatabits);
   ObjCallRet(msgControlGetValue, control, (P_ARGS)&Tag, s);
   inst.CommSetupData.DataBits = (int)Tag;

   control = (WIN)ObjectCall(msgWinFindTag, pArgs->win, (P_ARGS)tagStopbits);
   ObjCallRet(msgControlGetValue, control, (P_ARGS)&Tag, s);
   inst.CommSetupData.StopBits = (int)Tag;

   control = (WIN)ObjectCall(msgWinFindTag, pArgs->win, (P_ARGS)tagParity);
   ObjCallRet(msgControlGetValue, control, (P_ARGS)&Tag, s);
   inst.CommSetupData.Parity = (int)Tag;

   /* Check, if data changed */
   if (memcmp((P_CHAR)&inst.CommSetupData, (P_CHAR)&pData->CommSetupData,
              sizeof(COMM_SETUP)))
      {
      /* Copy data back to protected memory */
      ObjectWrite(self, ctx, &inst);

      /* Close and reopen port */
      ObjCallWarn(msgCommCloseSerial, self, pNull);
      ObjCallRet(msgCommOpenSerial, self, pNull, s);

      /* Initialize again */
      ObjCallWarn(msgCommSetSerialMetrics, self, (P_ARGS)pNull);

      /* Write data to file */
      if ((stream = fopen(comm_data_file, "wb")) != NULL)
         {
         Debugf("Comm File Open (Write) Ok.");
         i = fwrite((P_CHAR)&inst.CommSetupData, sizeof(COMM_SETUP),
                    1, stream);
         if (i == 1)
            {
            Debugf("Write Ok.");
            }
         fclose(stream);
         }
      }
   }
return(stsOK);

MsgHandlerParametersNoWarning;
}

/****************************************************************************/
/* Respond to msgCommSetConnectStatusId.                                   */
/*                                                                          */
/* Set new resource Id for connection status label.                         */
/* A custom handler is provided for this (instead of just calling          */
/* msgLabelSetStringid) because the toolkit accidently sets the            */
/* infoType to zero.                                                        */
/****************************************************************************/
MsgHandlerWithTypes(CommSetConnectStatusId, P_ARGS, P_COMMAPP_INST)
{
         OBJECT     label;

Debugf("msgCommSetConnectStatusId");

/* Only update label if card has been created */
if (pData->commOptWin)
   {
   if (label = (WIN)ObjectCall(msgWinFindTag, pData->commOptWin,
                               (P_ARGS)tagConnected))
      {
      ObjCallWarn(msgLabelSetString, label, pArgs);
      }
   }

return stsOK;
```

```
MsgHandlerParametersNoWarning;
} /* SXSetConnectStatusId */
```

A.9 C Source for COMMSEND.C

This module contains the code for sending the data via the serial port to an OS/2 2.0 system.

```
/****************************************************************************

 File: commsend.c

 Copyright 1990, 1991, 1992 GO Corporation. All Rights Reserved.

 You may use this Sample Code any way you please provided you
 do not resell the code and that this notice (including the above
 copyright notice) is reproduced on all copies.  THIS SAMPLE CODE
 IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
 EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT
 LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU
 FOR ANY CONSEQUENTIAL,INCIDENTAL,OR INDIRECT DAMAGES ARISING OUT OF
 THE USE OR INABILITY TO USE THIS SAMPLE CODE.

 $Revision:   1.0  $
   $Author:   gbarg $
     $Date:   21-Jan-92 $

 This file contains the class definition and methods for clsComm.

****************************************************************************/

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#ifndef FS_INCLUDED
#include <fs.h>
#endif

#ifndef SIO_INCLUDED
#include <sio.h>
#endif

#ifndef SERVMGR_INCLUDED
#include <servmgr.h>
#endif

#ifndef FRAME_INCLUDED
#include <frame.h>
#endif

#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef NOTE_INCLUDED
#include <note.h>
#endif

#ifndef COMM_INCLUDED
#include <comm.h>
#endif

#ifndef COMMAPP_INCLUDED
#include <commapp.h>
#endif

#include <method.h>

#include <stdlib.h>
#include <stdio.h>

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Defines, Types, Globals, Etc
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* Identification sent before the data */
static const char   Id[] = "  \x01Start\x02";

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Local Functions
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/***************************************************************************/
/* Convert a field and send it to the serial port                          */
/***************************************************************************/
        void        CommSendField(OBJECT self, P_CHAR Field, BOOLEAN flag)
{
        CHAR        Temp[60];
        P_CHAR      s1, s2;
```

```
/*                                                              */
/*  Field points to one data field. The data is put into        */
/*  quotation marks. Possible quotation marks within the text   */
/*  are doubled. If flag is true, a komma will be sent after    */
/*  the field.                                                  */
/*                                                              */
/*  Example:                                                    */
/*                                                              */
/*  Field contents              ===> Data sent                  */
/*                                                              */
/*  Peter                       ===> "Peter"                    */
/*  Robert "Bob" Miller         ===> "Robert ""Bob"" Miller"    */
/*                                                              */

s2 = Temp;
*s2 = '"';
s2++;
for (s1 = Field; *s1; s1++, s2++)
   {
   if (*s1 == '"')
      {
      s2[0] = '"';
      s2[1] = '"';
      s2++;
      }
   else
      *s2 = *s1;
   }
*s2 = '"';
s2++;

/* Append komma if flag=true */
if (flag)
   {
   *s2 = ',';
   s2++;
   }
*s2 = '\0';

/* Send the data to the serial port */
ObjCallWarn(msgCommSendSerial, self, (P_ARGS)Temp);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                           Message Handlers
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/****************************************************************************/
/* CommSendScreenButton                                                     */
/*                                                                          */
/* Respond to msgCommSendScreenButton                                       */
/****************************************************************************/
         MsgHandlerWithTypes(CommSendScreenButton, P_ARGS, P_COMMAPP_INST)
{
         COMM_DATA  Data;
static const CHAR   CrLf[] = "\x0D\x0A";

Debugf("Message msgCommSendButton");

GetTextData(&Data, pData);

/* Send the data, if Lastname is given, any serial port is available    */
/* and connection established                                           */
if (Data.LastName[0] && pData->SerInstanceOk && pData->CommSerConnected)
   {
   /* First send the Id */
   ObjCallWarn(msgCommSendSerial, self, (P_ARGS)Id);
   /* Send the fields */
   CommSendField(self, Data.LastName, true);
   CommSendField(self, Data.FirstName, true);
   CommSendField(self, Data.Inits, true);
   CommSendField(self, Data.Street, true);
   CommSendField(self, Data.City, true);
   CommSendField(self, Data.Country, true);
   CommSendField(self, Data.ZIP, true);
   CommSendField(self, Data.Phone, false);
   /* Send CR/LF */
   ObjCallWarn(msgCommSendSerial, self, (P_ARGS)CrLf);
   }

return(stsOK);

MsgHandlerParametersNoWarning;
}

/****************************************************************************/
/* CommSendFileButton                                                       */
/*                                                                          */
/* Respond to msgCommSendFileButton                                         */
/****************************************************************************/

         MsgHandlerWithTypes(CommSendFileButton, P_ARGS, P_COMMAPP_INST)
{
         COMM_DATA  Data;
         FILE       *stream;

static const CHAR   CrLf[] = "\x0D\x0A";
```

```
Debugf("Message msgCommSendButton");

memset((P_CHAR)&Data, '\0', sizeof(COMM_DATA));

/* Read data from file COMMAPP.DAT */
if ((stream = fopen(SAVE_FILE, "rb")) != NULL)
   {
   Debugf("File Open (Read) Ok.");
   fread((P_CHAR)&Data, sizeof(COMM_DATA), 1, stream);
   fclose(stream);
   }

/* Send the data, if Lastname is given, any serial port is available    */
/* and connection established                                           */
if (Data.LastName[0] && pData->SerInstanceOk && pData->CommSerConnected)
   {
   /* First send the Id */
   ObjCallWarn(msgCommSendSerial, self, (P_ARGS)Id);
   /* Send the fields */
   CommSendField(self, Data.LastName, true);
   CommSendField(self, Data.FirstName, true);
   CommSendField(self, Data.Inits, true);
   CommSendField(self, Data.Street, true);
   CommSendField(self, Data.City, true);
   CommSendField(self, Data.Country, true);
   CommSendField(self, Data.ZIP, true);
   CommSendField(self, Data.Phone, false);
   /* Send CR/LF */
   ObjCallWarn(msgCommSendSerial, self, (P_ARGS)CrLf);
   }

return(stsOK);

MsgHandlerParametersNoWarning;
}

/***************************************************************************/
/* CommOpenSerial                                                          */
/*                                                                         */
/* Respond to msgCommOpenSerial                                            */
/***************************************************************************/

         MsgHandlerWithTypes(CommOpenSerial, P_UNKNOWN, P_COMMAPP_INST)
{
         SM_ACCESS  saccess;
         STATUS     s;
         OBJECT     sio;
         SIO_INIT   sinit;
         SIO_EVENT_SET ses;
         COMMAPP_INST inst;
         SM_GET_STATE servstate;

Debugf("Message msgCommOpenSerial");

memcpy((P_CHAR)&inst, (P_CHAR)pData, sizeof(COMMAPP_INST));

/* Since I grab the port and won't let go, I can use SMAccess etc..    */
/* If you don't need exclusive access to the serial port, bind the     */
/* server and only open it if you need it. <servmgr.h> shows how to do */
/* that.                                                               */

saccess.pServiceName = inst.CommSetupData.DefaultPort;
saccess.caller = self;

ObjCallWarn(msgSMAccessDefaults, theSerialDevices, &saccess);
ObjCallRet(msgSMAccess, theSerialDevices, &saccess, s);

sio = saccess.service;

/* Initialize to default state, use small buffers */
sinit.inputSize = 512;
sinit.outputSize = 512;
ObjCallWarn(msgSioInit, sio, &sinit);

/* I'm only interested in transmission errors (well not really, */
/* but just to show how).                                       */
ses.eventMask = sioEventRxError;
ses.client = self;
ObjCallWarn(msgSioEventSet, sio, &ses);

/********************************************************************/
/* The Service Manager will keep me updated about the connection   */
/* status, however I want to find out the inital status.           */
/********************************************************************/

servstate.handle = (OBJECT) saccess.handle;
servstate.connected = false;
ObjCallWarn(msgSMGetState, theSerialDevices, &servstate);

/* Update instance data */
inst.CommSerConnected = servstate.connected;
inst.commSIOService = saccess.service;
inst.commSIOHandle = saccess.handle;

/* Copy data back to protected memory */
ObjectWrite(self, ctx, &inst);
```

```
/* Initialize port */
ObjCallWarn(msgCommSetSerialMetrics, self, (P_ARGS)pNull);

return(stsOK);

MsgHandlerParametersNoWarning;
}

/****************************************************************************/
/* CommCloseSerial                                                          */
/*                                                                          */
/* Respond to msgCommCloseSerial                                            */
/****************************************************************************/

        MsgHandlerWithTypes(CommCloseSerial, P_UNKNOWN, P_COMMAPP_INST)
{
        COMMAPP_INST inst;
        SM_RELEASE srelease;
        SIO_CONTROL_OUT_SET sco;
        STATUS     s;

Debugf("Message msgCommCloseSerial");

memcpy((P_CHAR)&inst, (P_CHAR)pData, sizeof(COMMAPP_INST));

/* pull dtr and rts low to physically disconnect */
sco.dtr = false;
sco.rts = false;
sco.out1 = false;
sco.out2 = false;
ObjCallWarn(msgSioControlOutSet, inst.commSIOService, &sco);

/* release the serial port */
srelease.caller = self;
srelease.service = inst.commSIOService;
srelease.handle = inst.commSIOHandle;
ObjCallRet(msgSMRelease, theSerialDevices, &srelease, s);

/* Update instance data */
inst.commSIOService = objNull;
inst.commSIOHandle = objNull;

/* Copy data back to protected memory */
ObjectWrite(self, ctx, &inst);

return(stsOK);

MsgHandlerParametersNoWarning;
}

/****************************************************************************/
/* CommSetSerialMetrics                                                     */
/*                                                                          */
/* Respond to msgCommSetSerialMetrics                                       */
/****************************************************************************/

        MsgHandlerWithTypes(CommSetSerialMetrics, P_ARGS, P_COMMAPP_INST)
{
        SIO_METRICS smetrics;

static const U32    BaudRateTab[] =
                    {
                    300,
                    600,
                    1200,
                    2400,
                    4800,
                    9600,
                    19200,
                    };

static const SIO_PARITY ParityTab[] =
                    {
                    sioNoParity,
                    sioOddParity,
                    sioEvenParity,
                    };

static const SIO_DATA_BITS DataBitsTab[] =
                    {
                    sioSevenBits,
                    sioEightBits,
                    };

static const SIO_STOP_BITS StopBitsTab[] =
                    {
                    sioOneStopBit,
                    sioOneAndAHalfStopBits,
                    sioTwoStopBits,
                    };

Debugf("Message msgCommSetSerialMetrics");

/* Initialize serial port to preferences */
ObjCallWarn(msgSioGetMetrics, pData->commSIOService, &smetrics);
smetrics.baud = BaudRateTab[pData->CommSetupData.BaudRate - 1];
smetrics.line.dataBits = DataBitsTab[pData->CommSetupData.DataBits - 1];
smetrics.line.stopBits = StopBitsTab[pData->CommSetupData.StopBits - 1];
smetrics.line.parity = ParityTab[pData->CommSetupData.Parity - 1];
smetrics.flowType.flowControl = sioXonXoffFlowControl;
ObjCallWarn(msgSioSetMetrics, pData->commSIOService, &smetrics);

return(stsOK);

MsgHandlerParametersNoWarning;
}

/****************************************************************************/
/* CommSendSerial                                                           */
/*                                                                          */
/* Respond to msgCommSendSerial                                             */
/****************************************************************************/

        MsgHandlerWithTypes(CommSendSerial, P_CHAR, P_COMMAPP_INST)
{
        STREAM_READ_WRITE_TIMEOUT srw;
        STATUS     s;

Debugf("Message msgCommSendSerial");

/* Do nothing if I'm not connected (or open for that matter) */
if (pData->CommSerConnected)
   {
   /* Support unicode... */
   srw.numBytes = strlen(pArgs) * SizeOf(CHAR);

   /* Do nothing if there were no characters entered in the IP */
   if (srw.numBytes > 0)
      {
      srw.pBuf = pArgs;
      srw.timeOut = 750;

      ObjCallRet(msgStreamWriteTimeOut, pData->commSIOService, &srw, s);

      /* Flush stream */
      ObjCallRet(msgStreamFlush, pData->commSIOService, pNull, s);
      }
   }
else
   Debugf("Not Open/Connected");

return(stsOK);

MsgHandlerParametersNoWarning;
}


/****************************************************************************/
/* Respond to msgSMConnectedChanged.                                        */
/*                                                                          */
/* Send by service manager when a change in the connection status has      */
/* occured. Update the serial card Status label to reflect current         */
/* connection status.                                                       */
/****************************************************************************/

MsgHandlerWithTypes(CommSMConnectedChanged, P_SM_CONNECTED_NOTIFY,
                    P_COMMAPP_INST)
{
static const CHAR   Connected[] = "Connected";
static const CHAR   NotConnected[] = "Not connected";
        COMMAPP_INST inst;

Debugf("msgSMConnectedChanged");

Debugf("Connected %ld", pArgs->connected);

memcpy((P_CHAR)&inst, (P_CHAR)pData, sizeof(COMMAPP_INST));

inst.CommSerConnected = pArgs->connected;

/* Copy data back to protected memory */
ObjectWrite(self, ctx, &inst);

ObjCallWarn(msgCommSetConnectStatusId, self,
            pData->CommSerConnected ? (P_ARGS)Connected:
                                      (P_ARGS)NotConnected);

return stsOK;

MsgHandlerParametersNoWarning;
} /* CommSMConnectedChanged */


/******************************************************************************/
/* Respond to msgSioEventHappened.                                            */
/*                                                                            */
/* Respond to an event, in this case only EventRxError. Note that more events */
/* bits may be set in the mask, even those I didn't express interest in.      */
/* Don't do much about the apparent transmission error. Could put up a note   */
/* or something. You can also be informed when the serial input buffer is no  */
/* longer empty. That is most suited for non-continious serial I/O, since it  */
/* has some overhead.                                                         */
/******************************************************************************/

MsgHandlerWithTypes(CommSioEventHappened, P_SIO_EVENT_HAPPENED,
                    P_COMMAPP_INST)
{
        NOTE_NEW   nn;
        MESSAGE    m;
        STATUS     s;
/* Tables for transmission error note */
static const TK_TABLE_ENTRY commNoteContent[] =
                    {
                    {"A transmission error occured", 0, 0, 0, tkLabelStringId},
                    {pNull}
                    };

const TK_TABLE_ENTRY commNoteButton[] =
                    {
                    {"Ok", 0, 0, 0, tkLabelStringId},
                    {pNull}
                    };

Debugf("msgSioEventHappened");

if (pArgs->eventMask & sioEventRxError)
   {
   Debugf("sioEventRxError");

   /* Show a simple note */
   ObjCallWarn(msgNewDefaults, clsNote, &nn);
   nn.note.metrics.flags = nfDefaultAppFlags | nfAutoDestroy;
   nn.note.pContentEntries = commNoteContent;
   nn.note.pCmdBarEntries  = commNoteButton;
   ObjCallRet(msgNew, clsNote, &nn, s);

   ObjCallWarn(msgNoteShow, nn.object.uid, &m);

   }

return(stsOK);

MsgHandlerParametersNoWarning;
} /* CommSioEventHappened */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Installation
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
```

A.10 Make File for REC_PEN C Routines

Compiler control file used when compiling and linking COMMAPP.EXE. The source for this program is listed later.

```
@echo off
sqlprep rec_pen.sqc pen
if errorlevel 1 goto ende
icc /c /W3 /DLINT_ARGS /DES32TO16 rec_pen.c
if errorlevel 1 goto ende
link386 rec_pen,,,sql_dyn;
:ende
```

A.11 REC_PEN.H C Header

```
/******************************************************************************
 File: rec_pen.h

******************************************************************************/

/* Define length of the input fields */
#define LASTNAME_LENGTH        20
#define FIRSTNAME_LENGTH       20
#define INITS_LENGTH            2
#define STREET_LENGTH          20
#define CITY_LENGTH            20
#define COUNTRY_LENGTH         15
#define ZIP_LENGTH              5
#define PHONE_LENGTH           15

typedef  struct
{
         char       LastName[LASTNAME_LENGTH + 1];
         char       FirstName[FIRSTNAME_LENGTH + 1];
         char       Inits[INITS_LENGTH + 1];
         char       Street[STREET_LENGTH + 1];
         char       City[CITY_LENGTH + 1];
         char       Country[COUNTRY_LENGTH + 1];
         char       ZIP[ZIP_LENGTH + 1];
         char       Phone[PHONE_LENGTH + 1];
}        COMM_DATA, *P_COMM_DATA;

         void       InsertData2DB(P_COMM_DATA CommData);
```

A.12 C Source for REC_PEN.C

This module receives the data from the serial port and updates or adds a record in the database PEN.

```
static unsigned char sqla_program_id[40] =
{111,65,65,66,65,68,67,67,85,83,69,82,73,68,32,32,82,69,67,95,
80,69,78,32,84,65,52,69,80,76,76,73,48,32,32,32,32,32,32,32};


/* Operating System Control Parameters */
#ifdef ES32TO16
#include "sqlca.h"
#include "sqlda.h"
#endif

#ifndef SQL_API_RC
#define SQL_STRUCTURE struct
#ifdef ES32TO16
#define SQL_API_RC short
#define SQL_API_FN
#define SQL_POINTER _Seg16
#else
#define SQL_API_RC int
#ifndef SQL_API_FN
#define SQL_API_FN far pascal _loadds
#endif
#define SQL_POINTER
#endif
#endif

SQL_API_RC SQL_API_FN sqlaaloc(unsigned short,
                               unsigned short,
                               unsigned short,
                               void *);
SQL_API_RC SQL_API_FN sqlacall(unsigned short,
                               unsigned short,
                               unsigned short,
                               unsigned short,
                               void *);
SQL_API_RC SQL_API_FN sqladloc(unsigned short,
                               void *);
SQL_API_RC SQL_API_FN sqlasets(unsigned short,
                               unsigned char *,
                               void *);
SQL_API_RC SQL_API_FN sqlasetv(unsigned short,
                               unsigned short,
                               unsigned short,
                               unsigned short,
                               void *,
                               void *,
                               void *);
SQL_API_RC SQL_API_FN sqlastop(void *);
SQL_API_RC SQL_API_FN sqlastrt(void *,
                               void *,
                               struct sqlca *);
SQL_API_RC SQL_API_FN sqlausda(unsigned short,
                               struct sqlda *,
                               void *);

#ifdef ES32TO16
#pragma linkage (sqlaaloc, far16 pascal)
#pragma linkage (sqlacall, far16 pascal)
#pragma linkage (sqladloc, far16 pascal)
#pragma linkage (sqlasets, far16 pascal)
#pragma linkage (sqlasetv, far16 pascal)
#pragma linkage (sqlastop, far16 pascal)
#pragma linkage (sqlastrt, far16 pascal)
#pragma linkage (sqlausda, far16 pascal)
#endif

#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <string.h>
#include <memory.h>
#include <sql.h>
#include <sqlenv.h>

#include "rec_pen.h"

#define INCL_DOSFILEMGR
#include <os2.h>

#define RESTART     -1015
#define BUFSIZE     512


/*
EXEC SQL INCLUDE sqlca;
*/

/* SQL Communication Area - SQLCA - structures and constants */
```

```
#ifndef SQLCODE

/* SQL Communication Area - SQLCA */
SQL_STRUCTURE sqlca
{
   unsigned char  sqlcaid[8];           /* Eyecatcher = 'SQLCA   ' */
   long           sqlcabc;              /* SQLCA size in bytes = 136 */
   long           sqlcode;              /* SQL return code */
   short          sqlerrml;             /* Length for SQLERRMC */
   unsigned char  sqlerrmc[70];         /* Error message tokens */
   unsigned char  sqlerrp[8];           /* Diagnostic information */
   long           sqlerrd[6];           /* Diagnostic information */
   unsigned char  sqlwarn[11];          /* Warning flags */
   unsigned char  sqlstate[5];          /* SQLSTATE */
};

/* Size of SQLCA */
#define   SQLCA_SIZE     sizeof(struct sqlca)

#define   SQLCODE        sqlca.sqlcode
#define   SQLWARN0       sqlca.sqlwarn[0]
#define   SQLWARN1       sqlca.sqlwarn[1]
#define   SQLWARN2       sqlca.sqlwarn[2]
#define   SQLWARN3       sqlca.sqlwarn[3]
#define   SQLWARN4       sqlca.sqlwarn[4]
#define   SQLWARN5       sqlca.sqlwarn[5]
#define   SQLWARN6       sqlca.sqlwarn[6]
#define   SQLWARN7       sqlca.sqlwarn[7]
#define   SQLWARN8       sqlca.sqlwarn[8]
#define   SQLWARN9       sqlca.sqlwarn[9]
#define   SQLWARNA       sqlca.sqlwarn[10]

#endif

struct sqlca sqlca;

                      /* SQL communications area    */

/****************************************************************************/
/* Error handling for SQL statements                                        */
/****************************************************************************/

/*
EXEC SQL
     WHENEVER SQLERROR GOTO exit_error;
*/


/*
EXEC SQL
     WHENEVER SQLWARNING CONTINUE;
*/


/*
EXEC SQL
     WHENEVER NOT FOUND CONTINUE;
*/


/****************************************************************************/
/* SQL declaration section                                                  */
/****************************************************************************/

/*
EXEC SQL BEGIN DECLARE SECTION;
*/


         char       LastName[21];
         char       FirstName[21];
         char       Inits[3];
         char       Street[21];
         char       City[21];
         char       Country[16];
         char       ZIP[6];
         char       Phone[16];

         char       DBKey[21];


/*
EXEC SQL END DECLARE SECTION;
*/



         void       main(argc, argv)

         int        argc;
         char      *argv[];
{
         int        i1, i2, i3;
         USHORT     rc;
         ULONG      ulAction, BytesRead;
         HFILE      FileHandle;
         PEAOP2     peaop2 = (PEAOP2)NULL;
         CHAR       Buffer[200];
         PCHAR      Ptr, Ptr2;

static   CHAR       Id[] = "\x01Start\x02";

static   USHORT     LengthTab[] =
                    {
                    LASTNAME_LENGTH + 1,
                    FIRSTNAME_LENGTH + 1,
                    INITS_LENGTH + 1,
                    STREET_LENGTH + 1,
                    CITY_LENGTH + 1,
                    COUNTRY_LENGTH + 1,
                    ZIP_LENGTH + 1,
                    PHONE_LENGTH + 1,
                    };

static   COMM_DATA  CommData;

#define TAB_SIZE    (sizeof(LengthTab) / sizeof(USHORT))

if (argc < 2)
   {
   printf("No Com-Port specified\n");
   exit(1);
   }

i1 = 0;
if (strlen(argv[1]) == 4)
   {
   if (memicmp(argv[1], "COM", 3) == 0)
      {
      i2 = atoi(&argv[1][3]);
      if ((i2 >= 1) && (i2 <= 3))
         {
         i1 = 1;
         }
      }
   }

if (!i1)
   {
   printf("Invalid Com-Port \"%s\" specified.\n", argv[1]);
   exit(1);
   }

rc = DosOpen(argv[1],
             &FileHandle,
             &ulAction,
             0L,
             FILE_NORMAL,
             FILE_OPEN,
             OPEN_ACCESS_READONLY | OPEN_SHARE_DENYREAD | OPEN_FLAGS_NOINHERIT |
             OPEN_FLAGS_SEQUENTIAL,
             (PEAOP2)NULL);

if (rc)
   {
   printf("Error %u while opening \"%s\"\n", rc, argv[1]);
   exit(1);
   }

while (1)
   {


   printf("Waiting for Data from %s, to end programm press Ctrl-Break...\n", argv[1]);

   Ptr = Buffer;

   i1 = 1;
   i2 = 0;

   while (i1)
      {
      if ((rc = DosRead(FileHandle, Ptr, 1L, &BytesRead)) == 0)
         {
         if (i1 == 1)
            {
            /* Test for Id */
            if (i2 < (sizeof(Id) - 1))
               {
               if (*Ptr == Id[i2])
                  i2++;
               else
                  i2 = 0;
               continue;
               }

            if (*Ptr != '"')
               {
               continue;
               }
            i1 = 2;
            }
         if (*Ptr == '\x0a')
            {
            i1 = 0;
            }
         Ptr++;
         }
      else
         {
         printf("\nError %d while reading file %s\n", rc, argv[1]);
         i1 = 0;
         }
      }

   *Ptr = '\0';


   memset((PCHAR)&CommData, '\0', sizeof(COMM_DATA));

   Ptr2 = (PCHAR)&CommData;

   for (Ptr = Buffer, i1 = i2 = i3 = 0; *Ptr; Ptr++)
      {
      if (*Ptr == '"')
         {
         if (i1 < 2)
            {
            i1++;
            continue;
            }
         else
            {
            i1--;
            Ptr2[i3] = *Ptr;
            i3++;
            continue;
            }
         }
      else
      if (*Ptr == ',')
         {
         if (i1 == 1)
            {
            Ptr2[i3] = *Ptr;
            i3++;
            }
         else
         if (i1 == 2)
            {
            Ptr2 = &Ptr2[LengthTab[i2]];
            i2++;
            i1 = i3 = 0;
            if (i2 >= TAB_SIZE)
               {
               break;
               }
            }
         }
      else
         {
         if (i1 == 1)
            {
            Ptr2[i3] = *Ptr;
            i3++;
            }
         }
      }

   InsertData2DB(&CommData);

   }

DosClose(FileHandle);

exit(0);
}

/**************************************************************************/
/* InsertData2DB                                                          */
/*                                                                        */
/* Insert data into the database                                          */
/**************************************************************************/
         void       InsertData2DB(P_COMM_DATA CommData)
{
         BOOL       Update;
         USHORT     rc;
         struct sqlca sqlca;
static   char       dbase[] = "PEN";
         char       msgbuf[BUFSIZE];

sqlestrd(dbase, 'S', &sqlca);                    /* start database      */

if (sqlca.sqlcode == RESTART)                    /* if start db func    */
   {                                             /* fails, call restart */
   sqlerest(dbase, &sqlca);                      /* restart database    */

   if (sqlca.sqlcode != 0)
      {                                          /* restart failed, exit */
      goto exit_error;
      }

   sqlestrd(dbase, 'S', &sqlca);                 /* start database again */
   }

if (sqlca.sqlcode != 0)                    /* check database for  */
   {                                         /* good start         */

   goto exit_error;                       /* exit error           */

   }

strcpy(DBKey,     CommData->LastName);
strcpy(LastName,  CommData->LastName);
strcpy(FirstName, CommData->FirstName);
strcpy(Inits,     CommData->Inits);
strcpy(Street,    CommData->Street);
strcpy(City,      CommData->City);
strcpy(Country,   CommData->Country);
strcpy(ZIP,       CommData->ZIP);
strcpy(Phone,     CommData->Phone);


/*
EXEC SQL
     UPDATE PENDATA
     SET LASTNAME = :LastName,
         FIRSTNAME = :FirstName,
         INITS = :Inits,
         STREET = :Street,
         CITY = :City,
         COUNTRY = :Country,
         ZIP = :ZIP,
         PHONE = :Phone
     WHERE LASTNAME = :DBKey;
*/

{
  sqlastrt(sqla_program_id,0L,&sqlca);
  sqlaaloc(1,9,1,0L);
    sqlasetv(1,0,460,21,LastName,0L,0L);
    sqlasetv(1,1,460,21,FirstName,0L,0L);
    sqlasetv(1,2,460,3,Inits,0L,0L);
    sqlasetv(1,3,460,21,Street,0L,0L);
    sqlasetv(1,4,460,21,City,0L,0L);
    sqlasetv(1,5,460,16,Country,0L,0L);
    sqlasetv(1,6,460,6,ZIP,0L,0L);
    sqlasetv(1,7,460,16,Phone,0L,0L);
    sqlasetv(1,8,460,21,DBKey,0L,0L);
  sqlacall((unsigned short)24,1,1,0,0L);
if (sqlca.sqlcode < 0)
{
   sqlastop(0L);
   goto exit_error;
}

  sqlastop(0L);
}


if (sqlca.sqlcode == 0)
   Update = 1;
else
   if (sqlca.sqlcode == 100)                   /* Record not found? */
      {
      Update = 0;
      }
   else
      goto exit_error;

if (Update)
   {
   printf("Updating database record for \"%s\" with the following data:\n",
          LastName);
   }
else
   {
   printf("Addind database record for \"%s\" with the following data:\n",
          LastName);
   }


printf("\nLast Name:  %s\n", CommData->LastName);
printf("First Name: %s\n", CommData->FirstName);
printf("Initials:   %s\n", CommData->Inits);
printf("Street:     %s\n", CommData->Street);
printf("City:       %s\n", CommData->City);
printf("Country:    %s\n", CommData->Country);
printf("ZIP:        %s\n", CommData->ZIP);
printf("Phone:      %s\n\n", CommData->Phone);

if (!Update)
   {

/*
EXEC SQL
        INSERT INTO PENDATA
           (LASTNAME, FIRSTNAME, INITS, STREET, CITY, COUNTRY, ZIP, PHONE)
        VALUES(:LastName, :FirstName, :Inits, :Street, :City,
               :Country, :ZIP, :Phone);
*/

{
  sqlastrt(sqla_program_id,0L,&sqlca);
  sqlaaloc(1,8,2,0L);
    sqlasetv(1,0,460,21,LastName,0L,0L);
    sqlasetv(1,1,460,21,FirstName,0L,0L);
    sqlasetv(1,2,460,3,Inits,0L,0L);
    sqlasetv(1,3,460,21,Street,0L,0L);
    sqlasetv(1,4,460,21,City,0L,0L);
    sqlasetv(1,5,460,16,Country,0L,0L);
    sqlasetv(1,6,460,6,ZIP,0L,0L);
    sqlasetv(1,7,460,16,Phone,0L,0L);
  sqlacall((unsigned short)24,2,1,0,0L);
if (sqlca.sqlcode < 0)
{
   sqlastop(0L);
   goto exit_error;
}

  sqlastop(0L);
}

   }


/*
EXEC SQL COMMIT WORK;
*/

{
  sqlastrt(sqla_program_id,0L,&sqlca);
  sqlacall((unsigned short)21,0,0,0,0L);
if (sqlca.sqlcode < 0)
{
   sqlastop(0L);
   goto exit_error;
}

  sqlastop(0L);
}


sqlestpd(&sqlca);                            /* stop database       */

return;

/************************************************************************/
/*                                                                      */
/* SQL error routine - retrieve the error message associated with the */
/*     return code and log error information.                           */
/*                                                                      */
/************************************************************************/
 exit_error:

      printf( "\nSQLCODE IS %ld",sqlca.sqlcode );

      rc = sqlaintp( msgbuf,BUFSIZE,0,&sqlca );

      if (rc < 0)                              /* message retrieve err*/
       {
         printf( "\nSQLAINTP ERROR. Return code = %d\n",rc );
       }

      if (rc > 0)                              /* error message return*/
       {
         printf( "\n%s",msgbuf );
       }


/*
EXEC SQL                          /- reset sqlerror to prevent -/
            WHENEVER SQLERROR CONTINUE;
*/
  /*   endless looping if      */
                                         /*   rollback fails         */

/*
EXEC SQL
            ROLLBACK WORK;
*/

}
```

Glossary

Acetate Layer/Plane. The window system's global screen-wide display plane. This is where ink from the pen is dribbled by the pen tracking software.

Activation. The transition of a document to an active state, with a running process, an application instance.

Application class. A PenPoint class that contains the code and initialization data used to create running applications.

Auxiliary Notebook. A Notebook on the Bookshelf such as Stationery, or Connections that is used for specialized tasks.

Behavior. The functionality of an object, the way and object reacts to messages.

Bitmap. An array of pixels, with an optional mask and hot spot.

Bookshelf. An area at the bottom of the screen that contains accessories and auxiliary notebooks. Each item on the Bookshelf is represented by an icon.

Chord. A straight line joining the ends of an arc.

Class. An object that implements a particular style of behavior in response to messages. The method table tells the class which messages sent to objects of that class to respond to.

Class Hierarchy. A hierarchy of classes in which each subclass inherits the the properties of all its ancestors.

Class Manager. Code that supports the object-oriented, message-passing, class-based programming used in PenPoint and PenPoint applications. The Class Manager implements two classes, clsObject and clsClass.

Component layer. The component layer of PenPoint consists of general purpose subsystems offering function that can be shared among applications.

Cork margin. An area at the bottom of the screen on all documents that stores reference buttons, new documents, embedded documents, or accessories.

Current directory entry. Each directory entry maintains a reference to the next directory entry it will use when the directory is read one entry at a time.

Data object. An object that maintains, manipulates and can recursively filled data.

Deactivate. Removes the application from the system, the installer however maintains a record of the application's UID and its location.

Directory handle. An object that references either a new, or existing directory node in the file system.

Document. A filed instance of an application. A document has a directory in the application hierarchy, but at any given point in time, it may not have a running process and a live application instance. Most documents reside in the Notebook; running copies of floating applications such as the Calculator, are also documents.

Dribble. The ink from the pen where the user writes over windows that support gestures and handwriting.

Embed. The PenPoint Framework provides facilities for applications and components to display and operate inside other applications without detailed knowledge of each other.

Embedded document. A document contained within another document.

Encapsulation. Protection of the instance variables of an object from access by methods other than the object's own methods.

File handle. The object with which a file node and its data are accessed. The handle is not a file itself.

Floating. A floating window appears above the Notebook, the user can move and resize a floating window.

Frame. The border surrounding documents and Option Sheets which includes a title bar, resize corner and move box.

Gesture. A shape or figure that the user draws with the pen on the tablet to invoke an action, or execute a command.

Global memory. Memory accessible from all tasks.

Grafic. Individual figure drawing operations stored in a picture segment.

Hot mode. A state in which the PenPoint Application Framework will not terminate an application.

Inheritance. A mechanism by which a class defines only the properties it needs in addition to those of its super-class.

In-line. In-line fields provide full handwriting and gesture recognition allowing the user to write with the pen directly in the field.

In Box. In and Out Box services allow the user to defer and batch data transfer operations for later execution. In/Out Boxes appear as iconic notebooks.

Instance data. Data stored in an object. It is normally only accessible by the object's class, which uses instance data in responding to messages sent to that object. The class defines the format of the instance data. Classes may have instance data include pointers to instance information stored outside the object.

Kernel. That portion of the operating system that interacts directly with the hardware. The core memory and task management code is the first code loaded when the system boots. Most system services are implemented in the kernel.

Local volume. Volumes in hard or floppy disk drives attached to a PenPoint system through the built-in SCSI port.

Main window. A window of an application that the Application Framework inserts on screen in the page location, or as a floating window. An application's main window is usually a frame.

Menu bar. A frame has an optional menu bar below its title bar. The Application Framework defines standard application menu items (SAMS) for the application's main window frame.

Message. A 32-bit value sent to an object requesting the object to perform some action. Messages are constants representing an action that an object can perform. The type of message is a tag that defines the class defining the message and guarantees uniqueness. When a message is sent to an object, if the message is recorded in the class's message table the Class Manager calls a message handler routine in the class's code which responds to the message.

Message argument. The information needed by a class to respond to a message. The message argument parameter may be a pointer to a separate message argument structure. This is the only way a class can pass information back to the sender.

Message handler. A function in the class's code that implements appropriate behavior for a message. It is called by the Class Manager in response to the message associated with it in the class's method table.

Method. The behavior of objects is implemented in their methods. A method may be compared with a traditional programming routine. A message is sent to the object containing the name of the method to be run along with any optional parameters. Methods can read/update the instance variables of the object. The method will return an object to the sender upon completion

Method table. An array of message-function name pairs and flags that determines which message handler function will handle messages sent to the objects of that class.

Node. A location in the file system, can be a directory or a file. PenPoint's file system is organized as a tree of nodes.

Notebook metaphor. The visual paradigm in PenPoint of a physical notebook containing pages, documents and sections with tabs and a page turn effect.

Object. An entity that maintains private data and can receive messages. Each object is an instance of some class, created by sending a message to the class.

Observer. An object that has requested the Class Manager to notify it when changes occur to another object. Objects maintain a list of their observers.

Option Sheet. A floating frame that displays attributes of the selection in one or more card windows.

PenPoint Framework. Both the protocol supporting multiple, embeddable, concurrent applications in the Notebook and the support code that implements most of an application's default responses.

Process. An operating system with its own local memory.

Recognition. Matching a set of user pen strokes with the most likely prototype during handwriting translation.

Resource. A uniquely identified collection of data. Resources allow applications to separate data from code in a structured manner.

SAMS. Standard Application Menus. The Application Framework supplies a standard set of SAMS - the Document and Edit menus, to which applications can add their own menu items.

Tag. A unique 32-bit number that uses the administered value of a well known value UID to ensure uniqueness. An arbitrary 32-bit number that is associated with any window. A window's tag can be checked and searched for.

UID. Unique Identifier. A 32-bit number that is the handle on an object. Messages are sent to an object's UID.

UUID. Universal Unique Identifier. A 64-bit number that is guaranteed to be unique across all PenPoint computers, used to identify resources in resource files.

Volume. A physical medium or a network entity that supports the file system.

List of Abbreviations

ANSI	American National Standards Institute
API	Application Programming Interface
BIOS	Basic Input Output Services
CPU	Central Processing Unit
DLL	Dynamic Link Library
DOS	Disk Operating System
DRAM	Dynamic Random Access Memory
EDA	Embedded Document Architecture
FAT	File Allocation System
FAX	Facsimile
FTT	Function Transfer Tables
GUI	Graphical User Interface
HWX	Handwriting Recognition
I/O	Input/Output
MIL	Machine Interface Library
NUI	Notebook User Interface
OEM	Original Equipment Manufacturer
OOPs	Object-Oriented Programming
OSI	Opens Systems Interface
PAK	PenPoint Adaptation Kit
PCL	Printer Control Language
PCMCIA	Personal Computer Memory Card International Association
RAM	Random Access Memory
RGB	Red Green Blue
ROM	Read Only Memory
SAMS	PenPoint Standard Application Menus
SIO	Sampled Image Operator
SQL	Structured Query Language
RTF	Rich Text Format
SDK	Software Developer's Kit
SSF	Solid State Files
SysDC	System Drawing Contexts
UI	User Interface Toolkit
UID	Unique Identifier
UUID	Universal Unique Identifier
VDM	Virtual DOS Machine

Index

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM International Technical Support Center
Department 91J, Building 235-2
Internal Zip 4423
901 NORTHWEST 51ST STREET
BOCA RATON FL
USA 33431-1328

Fold and Tape **Please do not staple** Fold and Tape

ITSC Technical Bulletin Evaluation RED000

PenPoint Operating System
Overview and Application Development

Publication No. GG24-3978-00

Your feedback is very important to us to maintain the quality of ITSO redbooks. **Please fill out this questionnaire and return it via one of the following methods:**

- Mail it to the address on the back (postage paid in U.S. only)
- Give it to an IBM marketing representative for mailing
- Fax it to: Your International Access Code + 1 914 432 8246

Please rate on a scale of 1 to 5 the subjects below.
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction ____

Organization of the book	____	Grammar/punctuation/spelling	____
Accuracy of the information	____	Ease of reading and understanding	____
Relevance of the information	____	Ease of finding information	____
Completeness of the information	____	Level of technical detail	____
Value of illustrations	____	Print Quality	____

Please answer the following questions:

a) Are you an employee of IBM or its subsidiaries? Yes___ No___

b) Are you working in the USA? Yes___ No___

c) Was the bulletin published in time for your needs? Yes___ No___

d) Did this bulletin meet your needs? Yes___ No___

If no, please explain:

____

____

What other Topics would you like to see in this Bulletin?

____

____

What other Technical Bulletins would you like to see published?

____

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

________________________________ ________________________________
Name Address

________________________________ ________________________________
Company or Organization

________________________________ ________________________________
Phone No.

GG24-3978-00

PenPoint Operating System Overview and Application Development

GG24-3978-00

PRINTED IN THE U.S.A.

GG24-3978-00