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

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

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

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

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

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

PenPoint Development Tools describes the environment for developing, debugging, and testing PenPoint applications. You need this book when you start to implement and test your first PenPoint application.
Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial capital letters.

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

Copyright ©1991–92 GO Corporation. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

The following are trademarks of GO Corporation: GO, PenPoint, the PenPoint logo, the GO logo, ImagePoint, GOWrite, NoteTaker, TableServer, EDA, MiniNote, and MiniText.

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

PenTOPS Copyright © 1990-1992, Sitka Corporation. All Rights Reserved.

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

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

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

ISBN 0–201–60860–X
123456789–AL–9695949392

First printing, June 1992
Preface

The *PenPoint Architectural Reference* provides detailed information on the various subsystems of the *PenPoint™* operating system. Volume I describes the functions and messages that you use to manipulate classes and describes the fundamental classes used by almost all *PenPoint* applications. Volume II describes the supplemental classes and functions that provide many different capabilities to *PenPoint* applications.

**Intended Audience**

The *PenPoint Architectural Reference* is written for people who are designing and developing applications and services for the *PenPoint* operating system. We assume that you are familiar with the C language, understand the basic concepts of object-oriented programming, and have read the *PenPoint Application Writing Guide*.

**What’s Here**

The *PenPoint Architectural Reference* is divided into several parts, which are split across two volumes. Volume I contains these parts:

- **Part 1: Class Manager** describes the *PenPoint* class manager, which supports object-oriented programming in *PenPoint*.
- **Part 2: PenPoint Application Framework** describes the *PenPoint* Application Framework, which provides you the tools you use to allow your application to run under the notebook metaphor.
- **Part 3: Windows and Graphics** describes ImagePoint, the imaging system for the *PenPoint* operating system, and how applications can control the screen (or other output devices).
- **Part 4: UI Toolkit** describes the *PenPoint* classes that implement many of the common features required by the *PenPoint* user interface.
- **Part 5: Input and Handwriting Translation** describes the *PenPoint* input system and programmatic access to the handwriting translation subsystems.

Volume II contains these parts:

- **Part 6: Text Component** describes the *PenPoint* facilities that allow any application to provide text editing and formatting capabilities to its users.
- **Part 7: File System** describes the *PenPoint* file system.
- **Part 8: System Services** describes the function calls that applications can use to access kernel functions, such as memory allocation, timer services, process control, and so on.
• Part 9: Utility Classes describes a wide variety of classes that save application writers from implementing fundamental things such as list manipulation, data transfer, and so on.

• Part 10: Connectivity describes the classes that applications can use to access remote devices.

• Part 11: Resources describes how to read, write, and create PenPoint resource files.

• Part 12: Installation API describes PenPoint support for installing applications, services, fonts, dictionaries, handwriting prototypes, and so on.

• Part 13: Writing PenPoint Services, describes how to write an installable service.

You can quickly navigate between these sections using their margin tabs. Each volume has its own index. The PenPoint Development Tools has a master index for all the manuals in the Software Development Kit.

**Other Sources of Information**

As mentioned above, the PenPoint Application Writing Guide provides a tutorial on writing PenPoint applications. The tutorial is illustrated with several sample applications.

The PenPoint Development Tools describes how to run PenPoint on a PC, how to debug programs, and how to use a number of tools to enhance or debug your applications. This volume also contains a master index to the five volumes included in the PenPoint SDK.

The PenPoint API Reference is a set of “datasheets” that were generated from the PenPoint SDK header files. These datasheets contain information about all the messages defined by the public PenPoint classes. If you own the PenPoint SDK, you can also find the header files in the directory \PENPOINT\SDK\INC.

To learn how to use PenPoint, you should refer to the PenPoint user documentation. The user documentation is included with the PenPoint SDK, and is usually packaged with a PenPoint computer. The user documentation consists of these books:

• Getting Started with PenPoint, a primer on how to use PenPoint.

• Using PenPoint, a detailed book on how to use PenPoint to perform tasks and procedures.
**Type Styles in This Book**

To emphasize or distinguish particular words or text, we use different fonts.

**Computerese**

We use fonts to distinguish two different forms of “computerese”:

- C language keywords and preprocessor directives, such as `switch`, `case`, `#define`, `#ifdef`, and so on.
- Functions, macros, class names, message names, constants, variables, and structures defined by PenPoint, such as `msgListAddItem`, `clsList`, `stsBadParam`, `P_LIST_NEW`, and so on.

Although all these PenPoint terms use the same font, you should note that PenPoint has some fixed rules on the capitalization and spelling of messages, functions, constants, and types. By the spelling and capitalization, you can quickly identify the use of a PenPoint term.

- Classes begin with the letters “cls”; for example, `clsList`.
- Messages begin with the letters “msg”; for example, `msgNew`.
- Status values begin with the letters “sts”; for example, `stsOK`.
- Functions are mixed case with an initial upper case letter and trailing parentheses; for example, `OSMemAvailable()`.
- Constants are mixed case with an initial lower case letter; for example, `wsClipChildren`.
- Structures and types are all upper case (with underscores, when needed, to increase comprehension); for example, `U32` or `LIST_NEW_ONLY`.

**Code Listings**

Code listings and user-PC dialogs appear in a fixed-width font.

```c
// Allocate, initialize, and record instance data.
//
StsJmp(OSHeapBlockAlloc(osProcessHeapId, SizeOf(*pInst), &pInst), s, Error);
pInst->placeHolder = -1L;
ObjectWrite(self, ctx, &pInst);
```
Less significant parts of code listings are grayed out to de-emphasize them. You needn't pay so much attention to these lines, although they are part of the listing.

```c
ObjCallJmp(msgNewDefaults, clsAppMgr, &new, s, Error);
new.object.uid = clsTtlApp;
new.object.key = 0;
new.cls.pMsg = clsTtlAppTable;
new.cls.ancestor = clsApp;
new.cls.size = SizeOf(P_TTL_APP_INST);
new.cls.newArgsSize = SizeOf(APP_NEW);
new.appMgr.flags.stationery = true;
new.appMgr.flags.accessory = false;
strcpy(new.appMgr.company, "GO Corporation");
new.appMgr.copyright = "\213 1992 GO Corporation, All Rights Reserved.";
ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);
```

### Placeholders

Anything you do not have to type in exactly as printed is generally formatted in italics. This includes C variables, suggested filenames in dialogs, and pseudocode in file listings.

### Other Text

The documentation uses *italics* for emphasis. When a Part uses a significant term, it is usually emphasized the first time. If you aren't familiar with the term, you can look it up in the glossary in the *PenPoint Application Writing Guide* or the index of the book.

DOS filenames such as `\BOOT\PENPOINT\APP` are in small capitals. PenPoint filenames can be upper and lower case, such as `\My Disk\Package Design Letter`.

Book names such as *PenPoint Application Writing Guide* are in italics.
# PENPOINT ARCHITECTURAL REFERENCE / VOL II CONTENTS

## Part 6 / Text
- 62 / Introduction 1
- 63 / Text Subsystem Concepts 3
- 64 / Using Text Data Objects 7
- 65 / Using Text Views 11
- 66 / Using Text Insertion Pads 23
- 67 / Sample Code 33
- 68 / Advanced Information 35

## Part 7 / File System
- 69 / Introduction 39
- 70 / File System Principles and Organization 43
- 71 / Accessing the File System 49
- 72 / Using the File System 57

## Part 8 / System Services
- 73 / Introduction 93
- 74 / PenPoint Kernel Overview 95
- 75 / C Run-Time Library 97
- 76 / Math Run-Time Library 109

## Part 9 / Utility Classes
- 77 / Introduction 119
- 78 / The List Class 123
- 79 / Class Stream 127
- 80 / The Browser Class 133
- 81 / File Import and Export 137
- 82 / The Selection Manager 147
- 83 / Transfer Class 155
- 84 / Help 165
- 85 / The Busy Manager 179
- 86 / Search and Replace 193
- 87 / Undo 195
- 88 / Byte Buffer Objects 199
- 89 / String Objects 207
- 90 / Table Class 211
- 91 / The NotePaper Component 213

## Part 10 / Connectivity
- 92 / Introduction 237
- 93 / Concepts and Terminology 241
- 94 / Using Services 243
- 95 / Serial I/O 255
- 96 / Parallel I/O 265
- 97 / Data Modem Interface 275
- 98 / The Transport API 279
- 99 / In Box and Out Box 295
- 100 / The Address Book 305
- 101 / The Sendable Services 317
- 102 / Introduction 331

## Part 11 / Resources
- 103 / Concepts and Terminology 335
- 104 / Using clsResFile 341
- 105 / Defining Resources with the C Language 347
- 106 / Compiling Resources 355
- 107 / System Preferences 359
- 108 / Introduction 361

## Part 12 / Installation API
- 109 / Installation Concepts 369
- 110 / PenPoint File Organization 373
- 111 / Dynamic Link Libraries 375
- 112 / Installation Managers 381
- 113 / The Auxiliary Notebook Manager 399
- 114 / The System Class 405

## Part 13 / Writing PenPoint Services
- 115 / Introduction 421
- 116 / Service Concepts 429
- 117 / Programming Services 433
- 118 / Distributing Your Service 435
- 119 / Test Service Examples 437

## Index
- 507
Part 6 / Text
Chapter 62 / Introduction

The Text subsystem presents text to the user for viewing and editing. It also provides an API to clients to allow programmatic modification of the text and its presentation attributes.

Overview

clsText implements the data object subclass of the Text subsystem. clsTextView implements the viewing subclass of the Text subsystem; it is the user's view onto the data managed by clsText. TXTDATA.H defines the messages used for text data objects; TXTVIEW.H defines the messages used for text view objects.

clsText is a descendent of clsObject. clsTextView is a subclass of clsView. Figure 62-1 shows the class hierarchy for clsText and clsTextView.
The Text subsystem is implemented as a group of related classes. The Text subsystem uses the PenPoint™ operating system class system. The classes allow:

- Your code to display both plain and fancy text to the user in one or more text data objects.
- The user to interact with the text to modify both the characters and their appearance.
- The user to transfer all or part of the text from one text data object to another (possibly non-text) object, and vice versa.
- Your code to file text data objects.
- Your code to observe and direct the user's interactions with the text.
Embedded objects, which are used to implement insertion pads and signature pads, and can include graphics, spreadsheets and other applications in documents.

There is a difference between displaying text through the graphics subsystem and using the Text subsystem. You can use the graphics subsystem to display characters on the screen, but users can't dynamically manipulate the text. Furthermore, the text subsystem includes paragraph and document attributes that define things such as margins and tabs.

**Developer's Quickstart**

If you want to display text on the screen, you can use the Windows and graphics subsystem to quickly display text in a window. If you want to make text available for editing and be able to file text as separate objects, you need to use the Text subsystem.

The simplest way to access the Text subsystem is to create a text view object by sending `msgNewDefaults` and `msgNew` to `clsTextView` (thus specifying no object to view). Like all views, the text view will create an empty text data object automatically. When you insert the text view into a window, the empty text view appears on screen. The user can now make an insertion gesture to bring up an insertion pad.

If you want to open a view to an existing text data object, you specify the object when you send `msgNew` to `clsTextView`. You can create a text data object separately and map it to a view later.

Most of the time the user will modify the text attributes through an option sheet for the application that uses text. You can programmatically change attributes (both default attributes and local attributes).

**Organization of This Part**

This part consists of six chapters. This, the first chapter, presents a brief overview of the Text subsystem.

Chapter 63, Text Subsystem Concepts, describes the organization and structure of the Text subsystem and presents the terminology used in the Text subsystem.

Chapter 64, Using Text Data Objects, describes how to access the text subsystem. It describes the messages defined for text data objects and how you use the messages.

Chapter 65, Using Text Views, describes how to access text views. It describes the messages defined for text view objects and how you use the messages.

Chapter 66, Using Text Insertion Pads, describes the messages used to handle text insertion pads.
Chapter 67, Sample Code, lists the code that is used in the examples to demonstrate features of the Text subsystem. These files are also available on disk in the \PENPOINT\SDK\SAMPLE directory.

Chapter 68, Advanced Information, describes advanced features of the Text subsystem. Although very powerful, these features are also rather complex.
Chapter 63 / Text Subsystem Concepts

This section describes the concepts related to the Text subsystem.

Topics covered in this chapter:

- The Text subsystem classes and objects.
- The types of attributes that text objects can have.
- Units of measurement used by the Text subsystem.

**The Text Subsystem Classes**

Messages for the Text subsystem are described by two separate files. TXTDATA.H provides an interface to clsText, which allows you to manipulate text data objects (adding, modifying, formatting, and so on). TXTVIEW.H provides an interface to clsTextView, a subclass of clsView, which allows you to display text data objects so that users can modify them.

**Text Data Objects**

The fundamental component of the Text subsystem is the text data object. You create a text data object by sending msgNewDefaults and msgNew to clsText. There is no limit to the size of text data objects.

A text data object has default attributes for characters, paragraphs, and the entire document. Additionally, a text data object supports local attributes for characters and paragraphs. Default attributes apply to an entire object; local attributes apply to contiguous ranges of characters or paragraphs. A text data object has a single set of default attributes, but it can have any number of local attributes. You can use the clsText messages to change both the default attributes and the local attributes.

In this description, a block is synonymous with a text data object. Your application might choose to implement documents that have a larger scale than a single text data object.

Text data objects are observable. Any object can add itself to a text data object’s observer list, so that the object will receive notification of changes in the text data object. The changes include text affected and text replaced messages.
**Organization of Text Data Objects**

A text data object is a stream of characters with no embedded formatting information. The formatting information is managed internally by clsText, and is available to advanced users.

The characters within a text data object are indexed with a value of type TEXT_INDEX (which is a U32 value). The text index is zero based; the index 0 indicates the first character in the text data object.

You must always use TEXT_INDEX type variables when indexing text.

The character encodings are similar to those used by the IBM-PC Code Page 850. However, the GO text data objects use some controls that replace codes used by Code Page 850. These codes and their meanings are defined in the file TENCODEN.H; the codes are listed in Table 63-1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>teEmbeddedObject</td>
<td>An object is embedded at this location.</td>
</tr>
<tr>
<td>teSpace</td>
<td>A space.</td>
</tr>
<tr>
<td>teTab</td>
<td>A tab.</td>
</tr>
<tr>
<td>teNewLine</td>
<td>A new line.</td>
</tr>
<tr>
<td>teNewPage</td>
<td>A new page.</td>
</tr>
<tr>
<td>teNewParagraph</td>
<td>A new paragraph.</td>
</tr>
<tr>
<td>teUnrecognized</td>
<td>A character that was not recognized by the handwriting recognition system.</td>
</tr>
</tbody>
</table>

**Attributes**

As mentioned above, attributes exist for characters, paragraphs, and documents. Default attributes are established when the text data object is created. You can change the default attributes programmatically, or the user can use the option sheet to change the attributes.

Your application can programmatically change local or default attributes (for example, the type family or point size). However, most applications don't intervene in formatting, they allow the user to change attributes by way of the option sheet.

**Character Attributes**

Character attributes apply to any range of characters, regardless of paragraph boundaries. The default character attributes apply to the entire text data object.

Characters have two types of attributes, display attributes and font attributes. The display attributes include size, underlining, small caps, capital letters, and so on. Font attributes include type face and weight.
Defined but not used are:

- Display attributes (superscript and subscript).
- Font attribute (aspect).

When you create a new text data object, clsText usually reads the current default character attributes from the system resource file. However, an application can override these defaults by redefining the resource in its own APPRES file. If you define your own character attribute resource, the resource ID must be textResDefaultCharAttrs. These values are defined in TXTDATA.H.

## Paragraph Attributes

Paragraph attributes include: alignment, leading, space before and after, margins, and tabs. Each tab has its own position; alignment and leader characters are in the design but are not implemented.

## Text Views

You usually use text data objects because you want users to create, and modify text on screen. To display the text data objects, you use a text view, which is an instance of clsTextView. clsTextView is a subclass of clsView, the view class.

While clsText provides facilities for storing the text's characters and attributes, clsTextView provides a user interface that allows the user to view and modify those characters and attributes.

When you create a text view, it displays the initial portion of the data object (that is, the characters beginning at index 0 and their associated attributes). The exact formatting of the view depends both on the data object and on the view’s style. You set the view’s style in the tv.style field of the TV_NEW structure (which you send with the msgNew that creates the view). If there is more data than can be completely displayed in the view, it is the client’s responsibility to place the text view within a scrolling window (clsScrollWin) so that it will have scroll bars.

You can use the TextCreateTextScrollWin function to create a text view and a scrolling window, and then insert the text view into a scrolling window.

Because views file their own data, the text view will send filing messages to its data object when it receives msgSave.

## Text Insertion Pads

When you create a text view, clsTextView creates an insertion pad in the text view by default. (You can turn off this behavior by clearing the tvFillWithIP flag in the tv.flags field of TV_NEW.)

This insertion pad is an object of clsTextIP, the text insertion pad class. clsTextIP inherits from clsIP. Most behavior of text insertion pads is identical to that of other insertion pads. For a complete description of insertion pads, see Part 5: Input and Handwriting Translation.
Units of Measurement

Many text data attributes can be expressed as dimensions (such as character size, margin position, tab position, and so on).

In general, paragraph units (including tabs) and font sizes are expressed in twips. Twips are described in Part 3: Windows and Graphics.

Internally, some attributes are converted from Twips to units that have a lower resolution (and are stored that way). If you change one of these attributes and then get the attribute again, you might notice some round off.
Chapter 64 / Using Text Data Objects

This section describes the messages that create and modify text data objects.

Topics covered in this chapter include:

- Creating and destroying text data objects.
- Getting characters from text data objects.
- Scanning for characters in text data objects.
- Modifying text data objects.
- Setting character, paragraph, and block attributes in text data objects.
- Embedding and extracting objects in text data objects.
- The observer messages for text data objects.

Text Data Functions

The following functions are defined in TXTDATA.H. These functions implement common actions that you might perform on text data. The functions are listed in Table 64-1.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextDeleteMany</td>
<td>Deletes one or more bytes from a text data object.</td>
</tr>
<tr>
<td>TextInsertOne</td>
<td>Inserts a single byte in a text data object.</td>
</tr>
</tbody>
</table>

Deleting Many Characters

You use TextDeleteMany to delete a number of characters from a text data object. The function prototype is:

```c
STATUS EXPORTED TextDeleteMany(
    const OBJECT dataObj,
    const TEXT_INDEX pos,
    const TEXT_INDEX length);
```

The function deletes `length` number of characters, starting at `pos`, in the text data object `dataObj`. If there are any characters in `dataObj` beyond `pos + length`, they are moved to `pos`.

This function uses an ObjectCall to msgTextModify to do its work.
Inserting a Character

You use `TextInsertOne()` to insert a single character into a text data object. The function prototype is:

```c
STATUS EXPORTED TextInsertOne(
    const OBJECT dataObj,
    const TEXT_INDEX pos,
    const CHAR toInsert);
```

The function inserts the character `toInsert` at the location `pos` in the text data object `dataObj`. Characters in the data object after `pos` have their indices incremented by one.

If you insert a character between characters with different formatting (such as between a plain text character and a bold text character), the character takes on the attributes of the character to the right (that is, the character with the higher `TEXT_INDEX` value).

This function uses an `ObjectCall()` to `msgTextModify` to do its work.

Text Data Messages

As described above, the messages for `clsText` are defined in `TXTDATA.H`. Most of those messages are described here. In order to better organize the topics, some messages defined by `clsText` are described later in Chapter 67, Advanced Information.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td></td>
<td>Initializes the <code>msgNew</code> arguments.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_TD_NEW</td>
<td>Creates a new text data object.</td>
</tr>
<tr>
<td>msgTextGetMetrics</td>
<td>P_TD_METRICS</td>
<td>Passes back the <code>textData</code>'s metrics.</td>
</tr>
<tr>
<td>msgTextSetMetrics</td>
<td>P_TD_METRICS</td>
<td>Sets a <code>textData</code>'s metrics.</td>
</tr>
<tr>
<td>msgTextChangeCount</td>
<td>S32</td>
<td>Passes back (and optionally sets) the <code>textData</code>'s <code>changeCount</code>.</td>
</tr>
<tr>
<td>msgTextGet</td>
<td>TEXT_INDEX</td>
<td>Returns the character in a <code>textData</code> at the specified position.</td>
</tr>
<tr>
<td>msgTextGetBuffer</td>
<td>P_TEXT_BUFFER</td>
<td>Passes back a contiguous range of characters from a <code>textData</code>.</td>
</tr>
<tr>
<td>msgTextLength</td>
<td>nothing</td>
<td>Returns the number of characters stored in the <code>textData</code>.</td>
</tr>
<tr>
<td>msgTextModify</td>
<td>P_TEXT_BUFFER</td>
<td>Modifies the characters stored in the <code>textData</code>.</td>
</tr>
<tr>
<td>msgTextSpan</td>
<td>P_TEXT_SPAN</td>
<td>Determines the range corresponding to the requested span.</td>
</tr>
</tbody>
</table>

continued
Creating a New Text Data Object

To create a new text data object, send msgNewDefaults and msgNew to clsText. These messages take a TD_NEW structure that contains:

- **metrics**  A set of metrics for the text object. The metrics include a **flags** value that specifies whether the text object is read only (tdmReadOnly) and whether operations on text can be undone (tdmCanUndo).

- **expectedSize**  The expected number of characters held by the text object.

- **expectedTagCount**  The expected number of tags. This field is reserved for future extensions; just use the default value.

The TD_NEW structure returns the UID of the newly created text object.

If you create a text data object this way, you are responsible for making it visible to the user. You can do this either by creating a window and displaying the text by hand or, preferably, creating a text view and specifying the text data object to the view. Like all views, clsTextView can create a text data object automatically. For more information see Chapter 65, Using Text Views.
Getting and Setting Text Metrics

You specify metrics for a text data object when you create it. You can later get and set the text data object's metrics with msgTextGetMetrics and msgTextSetMetrics. Both messages take a pointer to a TD_METRICS structure, which contains a flags field that indicates:

- tdmReadOnly: Whether the text object is read only.
- tdmCanUndo: Whether operations on text can be undone.

Reading Characters in Text Data Objects

You can get a character or characters from a text data object by sending msgTextGet or msgTextGetBuffer to the text data object.

Getting a Single Character

To get a character from the text data object, send msgTextGet to the object, specifying a text index. The message returns either the character at the specified index or a status value. (Status values have the sign bit set.) If the sign bit is clear, use a cast operator to convert the character from STATUS to CHAR. For example:

```c
status = ObjectCall(textData, msgTextGet, (P_ARGS)10);
if (status < stsOK)
{
   // Some kind of error
}
else
   character = (CHAR)status;
```

If the text index is beyond the end of data, the message returns stsEndOfData.

Getting a Range of Characters

To get a series of characters from a text data object, send msgTextGetBuffer to the object. The message takes a pointer to a TEXT_BUFFER structure that contains:

- first: The index of the starting character.
- length: The number of characters to read.
- bufLen: The length of the buffer that will receive the characters.
- buf: The pointer to the buffer that receives the characters.

When msgTextGetBuffer returns, it passes back the number of characters read in the bufUsed field. If the starting position was beyond the end of the text data object, it returns stsEndOfData; otherwise it returns stsOK.

Text Length

You can request the length, in characters, of a particular text data object by sending msgTextLength to the object. The message does not have any arguments, and returns the length of the text data object.
### Altering Text Data Objects

You can alter the contents of a text data object by sending `msgTextModify` to the object. The message takes a `TEXT_BUFFER` structure that contains:

- **first**: The offset of the first character to replace.
- **length**: The number of characters to replace. If this value is 0, `msgTextModify` inserts the characters in `buf` (rather than replacing any characters).
- **bufLen**: The length of the replacement characters.
- **buf**: A pointer to the buffer containing the replacement characters. If this pointer is null, `msgTextModify` deletes the characters identified by `first` and `length`.

When `msgTextModify` returns, the message passes back the number of characters from `buf` that it used in `bufUsed`.

### Scanning Ranges of Characters

The Text subsystem allows you to scan the characters in a text data object for groups of characters. That is, you can search a text data object for characters that belong to, or don't belong to, a certain class of characters.

You indicate the type of data you are searching for by an identifier, called an **atom**. PenPoint predefines a number of atoms in `TXTDATA.H`; the two atoms used most commonly in text are **atomChar** and **atomPara**. For further information, see Chapter 68.

To search for a range of characters, send `msgTextSpan` to the object. The message takes a `TEXT_SPAN` structure that contains:

- **first**: The starting character index.
- **length**: The length of the initial span.
- **type**: An atom that identifies the group to be matched. Usually the type is **atomWord** or **atomPara**.
- **direction**: A direction indicator. If the direction indicator is `tdForward`, the search range includes the starting character (`first+length-1`) through to the last character in the text data object. If the direction indicator is `tdBackward`, the search range includes the first character in the text data object (index 0) to the starting character (`first`). To search the entire string, you can OR the values.
- **needPrefix**: A `BOOLEAN` value that specifies whether the message should return the prefix of the span.
- **needSuffix**: A `BOOLEAN` value that specifies whether the message should return the suffix of the span.

When the message returns, the `TEXT_SPAN` structure contains:

- **first**: The index of the matched character.
- **length**: The number of characters that were matched.
prefixLength A U16 that will receive the length of the prefix. This field is updated only if needPrefix is true.

suffixLength A U16 that will receive the length of the suffix. This field is updated only if needSuffix is true.

### Getting and Setting Attributes

The messages that affect attributes (msgTextInitAttrs, msgTextGetAttrs, msgTextChangeAttrs, and msgTextClearAttrs) require similar arguments. We will describe the messages after describing the arguments that they use.

### Text Attribute Arguments

All text attribute messages include a tag argument that specifies the type of attribute that is to change (character or paragraph). The tag argument uses an atom to specify the different types of attributes. The atoms that apply to text attributes are atomChar, atomPara, atomParaTab, and atomEmbedded.

Most attribute messages require a first argument that indicates the position, or beginning of the position affected by the message. If first contains the symbol textDefaultAttrs, the message pertains to current defaults for that text data object.

Most attribute messages also require arguments that specify new attributes and a mask. The mask specifies the attributes to change.

The mask is most useful when changing attributes on a range of characters or paragraphs. Any range of characters can contain many different formats. Without the mask, it would be necessary to enumerate all of the existing formats within the range, and carefully update each one with a separate call. With the mask, rather than worry about what attributes are set already, you identify the range of characters you want to modify and let clsText do the work. Similarly, the mask allows modification of a subset of the default attributes without having to first fetch the current values.

### Character Attributes

When changing character attributes, applications use the tag atomChar. The mask for character attributes is defined in TA_CHAR_MASK and the attributes are defined in TA_CHAR_ATTRS. The symbols for the attributes and their corresponding mask bits are almost identical, the difference is that the mask defines additional symbols for the font (SYSDC_FONT_SPEC) structure.

Table 64-3 describes the symbols for the character attributes defined in TA_CHAR_ATTRS.
Getting and Setting Attributes

Attribute | Usage
--- | ---
size | Specifies the character size in Twips.
smallCaps | If true, characters are displayed in small capital letters.
upperCase | If true, characters are displayed in all uppercase.
strikeout | If true, characters have a line through them.
underlines | Specifies the underline style. The three possible styles are none (0), single underline (1), or double underline (2).
font | Specifies the font characteristics structure. The font characteristics are defined by SYSDC_FONT_SPEC. These characteristics are described in Part 4: Windows and Graphics and in SYSFONT.H.

As mentioned above, the symbols for the mask defined in TA_CHAR_MASK are similar to the symbols in TA_CHAR_ATTR, with the exception of the font attributes, which are defined by SYSDC_FONT_SPEC. Table 64-4 describes the mask symbols defined by TA_CHAR_MASK for the font attributes.

<table>
<thead>
<tr>
<th>Mask</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Enables use of font.id in the TA_CHAR_ATTRS structure.</td>
</tr>
<tr>
<td>group</td>
<td>Enables use of font.group in the TA_CHAR_ATTRS structure.</td>
</tr>
<tr>
<td>weight</td>
<td>Enables use of font.weight in the TA_CHAR_ATTRS structure.</td>
</tr>
<tr>
<td>aspect</td>
<td>Enables use of font.aspect in the TA_CHAR_ATTRS structure.</td>
</tr>
<tr>
<td>italic</td>
<td>Enables use of font.italic in the TA_CHAR_ATTRS structure.</td>
</tr>
<tr>
<td>monospaced</td>
<td>Enables use of font.monospaced in the TA_CHAR_ATTRS structure.</td>
</tr>
<tr>
<td>encoding</td>
<td>Enables use of font.encoding in the TA_CHAR_ATTRS structure.</td>
</tr>
</tbody>
</table>

**Paragraph Attributes**

When changing paragraph attributes, set **tag** to **atomPara**. The mask for paragraph attributes is defined in **TA_PARA_MASK** and the attributes are defined in **TA_PARA_ATTRS**. The symbols for the attributes and their corresponding mask bits are identical. Table 64-5 describes the symbols for the paragraph attributes and masks defined in **TA_PARA_ATTRS** and **TA_PARA_MASK**.
### Paragraph Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>alignment</td>
<td>Specifies the alignment of the paragraph. The possible values are taParaLeft, taParaCenter, or taParaRight to indicate left, center and right alignment respectively.</td>
</tr>
<tr>
<td>justify</td>
<td>If true, text should be justified. If false, text is ragged right.</td>
</tr>
<tr>
<td>lineHeight</td>
<td>Specifies the line height in Twips. This is usually the same as the character height. The constant useMaxHeightOnLine indicates the the line height should be big enough to accommodate the tallest character in the line. If a line contains any character or embedded object larger than the specified lineHeight, the larger value is used. Thus lineHeight is a minimum, not a maximum.</td>
</tr>
<tr>
<td>interLineHeight</td>
<td>Specifies additional space between lines in Twips.</td>
</tr>
<tr>
<td>beforeSpacing</td>
<td>Specifies the amount of space before the paragraph in Twips. This space is considered to belong to the first line of the paragraph above the ink of the characters in the first line.</td>
</tr>
<tr>
<td>afterSpacing</td>
<td>Specifies the amount of space after the paragraph in Twips. The before and after spacing are additive. This space is considered to belong to the last line of the paragraph below the ink of the characters in the last line.</td>
</tr>
<tr>
<td>firstLineOffset</td>
<td>Specifies the horizontal offset for the first line in Twips. This is a signed value; a negative value will result in a hanging indent.</td>
</tr>
<tr>
<td>leftMargin</td>
<td>Specifies the position of the left margin, relative to the left edge of the view, in Twips.</td>
</tr>
<tr>
<td>rightMargin</td>
<td>The position of the right margin, relative to the right edge, in Twips.</td>
</tr>
</tbody>
</table>

### Paragraph Tabs

The tab stops for a paragraph are set independently of the other paragraph attributes. The tag for tab stops is atomParaTabs; the mask is pNull, and the value is specified by a TA_MANY_TABS structure, which contains:

- **count**: The number of tab stops in the paragraph.
- **repeatAtEnd**: Whether the last explicit stop acts as a prototype for an infinity of implicit stops.
- **tabs**: A fixed-length array of TA_TAB_STOP descriptors. Only the first count number of descriptors are valid. Each TA_TAB_STOP descriptor contains:
  - **x**: The position of the tab stop, relative to the left edge of the paragraph.
  - **type**: The alignment for the tab. The only valid alignment is taTabLeft.
  - **lead**: The leader characters for the tab. The only valid leader character is space (taLeadSpace).

### Getting Attributes

To get the character or paragraph attributes, send `msgTextGetAttrs` to the text object. `msgTextGetAttrs` can retrieve either local or default attributes. The message takes a pointer to a TEXT_GET_ATTRS structure, which specifies:

- **tag**: An atom that indicates the type of attribute to get.
- **first**: A text index to a location in the text data object. If you specify `textDefaultAttrs` rather than a text index, the message returns the default...
attributes for the type of data specified in \texttt{tag}. If you specify a text index, the attributes are initialized to the \texttt{tag} attributes at that location (which allows you to copy attributes from one location in text to another).

\begin{itemize}
  \item \texttt{length} Reserved for future use. Must be zero.
  \item \texttt{pValues} The address of the buffer to receive the attributes. Because \texttt{pValues} is of type \texttt{P\_UNKNOWN}, you must use a cast to render the attributes in their correct type (\texttt{P\_TA\_CHAR\_ATTRS} for character attributes, \texttt{P\_TA\_PARA\_ATTRS} for paragraph attributes, \texttt{P\_TA\_MANY\_TABLES} for tabs).
\end{itemize}

When the message returns, \texttt{pValues} contains the specified attributes.

\section*{Modifying Attributes}

When you change attributes, you can change either local or default attribute sets. The first step in changing attributes is to initialize an attributes structure by sending \texttt{msgTextInitAttrs} to a text data object. After initializing the structure, you can use it in the arguments to \texttt{msgTextChangeAttrs}.

To change all attributes to their defaults, you can send \texttt{msgTextClearAttr} to the text data object.

\section*{Initializing Attributes}

To initialize an attributes structure and mask, send \texttt{msgTextInitAttrs} to the text data object. The message takes a pointer to a \texttt{TEXT\_CHANGE\_ATTRS} structure. The interesting fields in the structure are:

\begin{itemize}
  \item \texttt{tag} An atom that indicates the type of attribute to initialize.
  \item \texttt{pNewMask} The address of the buffer that will receive the initialized mask.
\end{itemize}

When the message returns, all bits in the mask are disabled (that is, the structure won't affect any attributes if used in \texttt{msgTextChangeAttrs}).

If you want to avoid the time required to send another message, you can skip sending \texttt{msgTextInitAttrs}, but you must use \texttt{memset} to zero the mask. If you do not zero the mask, unpredictable results can occur in \texttt{msgTextChangeAttrs}.

\section*{Changing Attributes}

When you have initialized the attributes structure, you can send \texttt{msgTextChangeAttrs} to the text data object.

\texttt{msgTextChangeAttrs} message takes a pointer to the initialized \texttt{TEXT\_CHANGE\_ATTRS} structure, which contains:

\begin{itemize}
  \item \texttt{tag} An atom that indicates the type of attribute to change.
  \item \texttt{first} A text index for a location in the text data object that specifies the beginning of the range to change. If you specify \texttt{textDefaultAttr} for \texttt{first}, the message changes the default attributes.
  \item \texttt{length} A text index that indicates the range of text over which the attributes should be changed. If \texttt{first} is \texttt{textDefaultAttr}, \texttt{length} is ignored.
\end{itemize}
pNewMask  The address of the mask. The mask's initialized state is all attributes disabled. Therefore, you must enable at least one bit in the mask before you send the change message.

pNewValues  The address of the attributes buffer.

If you are changing paragraph or tab attributes for a range of paragraphs, the first and length values identify the affected paragraphs to begin with the paragraph containing first through to the paragraph containing first + length - 1.

Clearing Attributes

To change all attributes of a specific type to their defaults, send msgTextClearAttrs to a text data object. The only argument for the message is an atom that indicates the type of attributes to clear.

Embedding Objects

An embedded object is represented in a text data object by the character teEmbeddedObject and associated attributes. However, you can't embed an object by simply inserting teEmbeddedObject into the text data object. You must send msgTextEmbedObject to the text data object. The message takes a pointer to a TEXT_EMBED_OBJECT structure that contains:

  first  A text index that indicates where the object should be embedded.

  toEmbed  The UID of the object to embed.

  clientFlags  A set of client flags.

  action  A set of flags that specifies the type of embedding. The possible values are: textEmbedInsert, textEmbedCopy, and textEmbedMove. Most clients should only use textEmbedInsert, which specifies that a new object is to be embedded.

Copy and move are used with the transfer protocol and can cause unpredictable results if used incorrectly.

Again, to remove an embedded object, you can't simply delete the embedded object character. To extract an object, send msgTextExtractObject to the text data object. The only argument required by the message is the UID of the object to extract. clsText observes embedded objects; if you free an embedded object, clsText will clean up the embedded object character. To enumerate all the embedded objects, use msgTextEnumEmbeddedObjects (see TXTDATA.H).
Observer Messages

An observer of a text data object will receive the following four messages. The most important fact to observers is that the text has changed.

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgTextReplaced</td>
<td>A client replaced text in the observed object, using msgTextModify. Descendants must pass this message to superclass.</td>
</tr>
<tr>
<td>msgTextAffected</td>
<td>A client changed attributes for the observed object.</td>
</tr>
<tr>
<td>msgTextMarkAllocated</td>
<td>Informs that a msgTextMarkAlloc has happened.</td>
</tr>
<tr>
<td>msgTextFreed</td>
<td>Informs that a msgTextMarkFree has happened.</td>
</tr>
</tbody>
</table>

Clients should ignore the following two messages:

msgTextReplaced Observer Message

msgTextReplaced indicates that text in the observed object has been replaced. The message takes a TEXT_REPLACED structure that consists of two elements: a TEXT_SPAN_AFFECTED structure (span) and a TEXT_INDEX that indicates the number of bytes that replaced the span (bytesTakenFromBuf).

The TEXT_SPAN_AFFECTED structure contains:

- span.object: The object that was altered.
- span.changeCount: The number of changes that the object has experienced since the counter was last reset to zero, including this change.
- span.first: The first character that changed.
- span.length: The number of characters that changed.

msgTextAffected Observer Message

msgTextAffected indicates that attributes in the observed object have been changed. The message takes a TEXT_AFFECTED structure that consists of two elements: a TEXT_SPAN_AFFECTED structure (span) and a U16 value that indicates whether the change affects the size of the characters in a view (remeasure).

The TEXT_SPAN_AFFECTED structure is described above in the msgTextModified observer message.

remeasure is important to views. If it is true, it means that the change affected the size of the characters in the view and that the view must be remeasured before it can be redrawn.
Chapter 65 / Using Text Views

This section describes the messages that display text data objects. When displayed in a text view, users can use the pen to modify the text data. Topics covered include:

- Creating and destroying text views.
- Embedding objects in views.
- Interacting with the input subsystem.
- Scrolling a text view.
- Getting the current selection.

**Text View Messages**

clsTextView is a subclass of clsView, which inherits indirectly from clsEmbeddedWin. Messages for clsTextView are defined in TXTVIEW.H. The text view object messages are:

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_TV_NEW</td>
<td>Creates a new instance of clsTextView.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_TV_NEW</td>
<td>Initializes the NEW structure.</td>
</tr>
<tr>
<td>msgGWinXList</td>
<td>P_XLIST</td>
<td>Defined in GWIN.H.</td>
</tr>
<tr>
<td>msgGWinGesture</td>
<td>P_GWIN_GESTURE</td>
<td>Defined in GWIN.H.</td>
</tr>
<tr>
<td>msgXferGet</td>
<td>lots of things</td>
<td>Sent by a Receiver to get &quot;one-shot&quot; data transfer</td>
</tr>
<tr>
<td>msgXferList</td>
<td>OBJECT</td>
<td>Ask Sender for its list of data transfer types.</td>
</tr>
<tr>
<td>msgXferStreamWrite</td>
<td>STREAM</td>
<td>Asks the Sender to write more data to the stream.</td>
</tr>
<tr>
<td>msgXferStreamConnect</td>
<td>XFER_CONNECT</td>
<td>Sent to the Sender to ask it to link the Sender's and Receiver's pipe.</td>
</tr>
<tr>
<td>msgXferStreamFreed</td>
<td>STREAM</td>
<td>Sent to the Sender when the Receiver's side of the stream has been freed.</td>
</tr>
<tr>
<td>msgSelYield</td>
<td>BOOLEAN</td>
<td>theSelectionManager requires the release of selection.</td>
</tr>
<tr>
<td>msgSelDelete</td>
<td>U32</td>
<td>The selection owner should delete the selection.</td>
</tr>
<tr>
<td>msgTextViewAddIP</td>
<td>P_TV_EMBED_METRICS</td>
<td>Adds an insertion pad to the textView.</td>
</tr>
<tr>
<td>Message</td>
<td>Takes</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>msgTextViewEmbed</td>
<td>P_TV_EMBED_METRICS</td>
<td>Embeds an object in the textView. Makes associated changes in text data.</td>
</tr>
<tr>
<td>msgTextViewGetEmbedMetrics</td>
<td>P_TV_EMBED_METRICS</td>
<td>Passes back the textView-specific metrics for an embedded object.</td>
</tr>
<tr>
<td>msgTextViewResolveXY</td>
<td>P_TV_RESOLVE</td>
<td>Given an point in LWC space, passes back the character at (or near) the point.</td>
</tr>
<tr>
<td>TextCreateTextScrollWin</td>
<td></td>
<td>Utility function that creates a textView (with a data object) placed inside a scroll window. (See swin.h.)</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_TEXTIP_NEW</td>
<td>Initializes the NEW struct.</td>
</tr>
<tr>
<td>msgTextViewScroll</td>
<td>P_TV_SCROLL</td>
<td>Repositions displayed text within the textView.</td>
</tr>
<tr>
<td>msgTextViewGetStyle</td>
<td>P_TV_STYLE</td>
<td>Passes back a textView's style.</td>
</tr>
<tr>
<td>msgTextViewSetSelection</td>
<td>P_TV_SELECT</td>
<td>Selects one or more characters displayed by the textView.</td>
</tr>
<tr>
<td>msgTextViewSetStyle</td>
<td>P_TV_STYLE</td>
<td>Sets a textView's style.</td>
</tr>
<tr>
<td>msgTextViewCheck</td>
<td>P_UNKNOWN</td>
<td>A textView performs a self-consistency check (DEBUG DLLs only).</td>
</tr>
<tr>
<td>msgTextViewRepair</td>
<td>pNull</td>
<td>Forces a delayed paint operation to take place immediately.</td>
</tr>
</tbody>
</table>

**Creating a Text View**

To create a new text view object, send `msgNewDefaults` and `msgNew` to `clsTextView`. Both messages take a TV_NEW structure.

Because text views behave differently from many other views, `msgNewDefaults` in `clsTextView` changes many of the default values returned from `clsView` (its ancestor). Table 65-2 lists the defaults provided by `clsTextView`.

<table>
<thead>
<tr>
<th>Class</th>
<th>Field</th>
<th>New Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>clsWin</td>
<td>flags.style</td>
<td>wsSendLayout = false</td>
</tr>
<tr>
<td>clsWin</td>
<td>flags.style</td>
<td>wsCaptureLayout = false</td>
</tr>
<tr>
<td>clsWin</td>
<td>flags.style</td>
<td>wsGrowBottom = true</td>
</tr>
<tr>
<td>clsWin</td>
<td>flags.style</td>
<td>wsSendFile = true</td>
</tr>
<tr>
<td>clsWin</td>
<td>flags.style</td>
<td>wsSendGeometry = true</td>
</tr>
<tr>
<td>clsWin</td>
<td>flags.style</td>
<td>wsCaptureGeometry = true</td>
</tr>
<tr>
<td>clsWin</td>
<td>flags.input</td>
<td>inputMoveDown = true</td>
</tr>
<tr>
<td>clsWin</td>
<td>flags.input</td>
<td>inputMoveDelta = true</td>
</tr>
<tr>
<td>clsWin</td>
<td>flags.input</td>
<td>inputHoldTimeout = true</td>
</tr>
<tr>
<td>clsWin</td>
<td>flags.input</td>
<td>inputOutProx = true</td>
</tr>
<tr>
<td>clsWin</td>
<td>flags.input</td>
<td>inputTip = true</td>
</tr>
</tbody>
</table>

---

Table 65-2 (continued)
If `view.CreateDataObject` is `true`, `clsTextView` automatically creates a `clsText` data object. To explicitly pass a data object, set `view.dataObject` to the desired data object and set `view.createDataObject` to `false`.

The `TV_NEW` structure also contains:

- **flags** A set of flags that specify properties of a new text view. The only flag currently defined is `fillWithIP`, which specifies that the text view should have a `clsTextIP` insertion pad added to the end of the text. `fillWithIP` is the default; if you don't want an insertion pad, you must turn off this flag. Subclasses of `clsTextView` can change this behavior by intercepting `msgTextAddIP`.

- **dc** A text data object to view. In PenPoint 1.0, you must leave `dc` set to its default.

- **style** Styles for the view. Styles are defined in `TV_STYLE` and include:
  - **style.flags** Indicate the behavior of the text view. These flags are described in the next paragraph. The flags are saved in the text view's instance data.
  - **style.magnification** A character size adjustment.
  - **style.showSpecial** Show special characters. The special characters that are affected by this field are the tab, line break, paragraph break, and page break characters. The possible values are 0 to show none and 3 to show all.
  - **style.printer** Reserved for future expansion. Leave this value set to its default (null).

The `TV_STYLE` flags are:

- **tvsEmbedOnlyComponents** Don't embed applications.
- **tvsEmbedOnlyIPs** Only embed insertion pads.
- **tvsFormatForPrinter** Format text for a printer.
- **tvsQuietWarning** Don't display warning notes.
- **tvsQuietError** Don't display error notes.
- **tvsQuiet** Don't display either warning notes or error notes.
- **tvsReadOnlyChars** Characters are read only.
- **tvsReadOnlyAttrs** Attributes are read only.
Both characters and attributes are read only.

Wrap lines at word breaks.

**Getting the Viewed Object's UID**

To get the object being viewed, send the message `msgViewGetDataObject` to the view. You can read about using `msgViewGetDataObject` in *Part 2: The Application Framework*. You can change the view's object to a new object by sending `msgViewSetDataObject` to the view object.

**Embedding Objects in Views**

There are two messages used to embed objects in a text view: `msgTextViewAddIP` and `msgTextViewEmbed`.

When the user makes a gesture to create an insertion pad, `clsTextView` sends `msgTextViewAddIP` to itself. This allows its subclasses (such as the Writing Paper application) to intercept the message and thereby learn that the user requested an insertion pad.

You use `msgTextViewEmbed` to actually embed objects, such as insertion pads. The message adds the embedded object character and associated attribute to the text data object and inserts the object into the tree of views.

Both messages take a `TV_EMBED_METRICS` structure that contains:

- **pos** A text data object index, that specifies where the object should be embedded. The special value `TEXT_INDEX` means at the end of the document.
- **flags** A flags word that specifies how the view should present the object. The flags indicate that the embedded object:
  - `tvEmbedReplace` Will replace some text.
  - `tvEmbedFloat` Is floating.
  - `tvEmbedOneChar` Will insert a single character.
  - `tvEmbedAddMargin` Has a margin to leave space for gestures.
  - `tvEmbedPreload` Is preloaded with some text.
  - `tvEmbedPara` Will insert or replace a paragraph.
- **embedded** The UID of the object to be embedded.

If you subclass `clsTextView` and override `msgTextViewAddIP`, you must remember that `tvEmbedPreload` directs `clsTextView` to preload the insertion pad with the text under a circle gesture. However, if you intercept a `msgTextViewAddIP` that was caused by a a circle-line gesture, you must ensure that `tvEmbedPreload` is clear. This ensures that the insertion pad is not preloaded with the text under the gesture.
Also if you override the behavior of msgTextViewAddIP, you must free the object specified in the embedded field before adding your own embedded object.

To determine the metrics of an embedded object, send msgTextViewGetEmbedMetrics to the text view object. The message takes a TV_EMBED_METRICS structure. On input, embedded must specify an embedded object; the message returns the index of the object in text and the flags that pertain to the object.

### Interacting with the Input Subsystem

The Text subsystem provides three messages that interact with the Input subsystem: msgGWinGesture, msgTextViewResolveXY, and msgGWinRunXList.

### Obtaining the Text Index from a Tap Position

msgTextViewResolveXY determines a character index (and a number of other values) from the x and y coordinates for a pen event. The message takes a TV_RESOLVE structure that specifies:

- **xy** The x-y coordinates of the pen tap.
- **flags** Flags that specify whether to select the next character beyond the tap (tvrSelLPO) and whether to select relative to the midpoint of a character (tvrBalance).
- **pos** An index location to receive the position of the character directly under xy. If no character is selected by the pen tap, pos receives the value maxTEXT_INDEX.
- **lineStart** An index location to receive the position of the first character on the line that contains xy.
- **xRegion** An S8 value to receive the x region of xy. See Figure 65-1.
- **yRegion** An S8 value to receive the y region of xy. See Figure 65-2.
- **selects** An index location to receive the position of the character closest to xy, when the tap lies outside the text in the view. When the user taps above or to the left of the text area, selects receives the first character in the view; when the user taps below or to the left of the text area, selects receives the last character in the view.
- **offset** The offset to the previous or next character’s ink. When using tvrBalance, offset helps to determine which half of a character was tapped.

If the message returns stsOK, the location was resolved. If the message returns stsNoMatch, no text was found in the view to resolve the x and y position.
Figure 65-1 illustrates the **xRegion** of a block of text.

![Figure 65-1: Text View X-Regions](image)

Most views contain margins, where the user can tap without touching any text. When `msgTextViewResolveXY` evaluates the position of a tap, it first resolves the y (vertical) region. If the tap was above the text, `yRegion` contains a 1; if the tap was in the text area, `yRegion` contains a 0; if the tap was below the text area, `yRegion` contains a −1. If the tap was in the text area, the messages also evaluates which line of text contains the tap and calculates the index for `lineStart`.

Figure 65-2 illustrates the **yRegion** of a block of text.

![Figure 65-2: Text View Y-Regions](image)
The message then uses that text line to evaluate the x (horizontal) region of the tap. If the tap was to the left of the text line, xRegion contains a -1; if the tap was within the text line, xRegion contains a 0; if the tap was to the right of the text line, xRegion contains a 1.

### Processing an Input Xlist or Gesture

To process an input xlist and act on the commands in the xlist, send msgGViewXList to the text view. The only argument to the message is a pointer to an xlist. For further information on xlists and the Input subsystem, see Part 5: Input and Handwriting Translation.

To process just a gesture, send msgGWinGesture to the text view.

### Scrolling a Text View

If your text view window doesn’t display all the text in its text object, you might want to scroll the text programmatically, either by one line, or to the top of the page. To scroll text, send msgTextViewScroll to the text view object. The message takes a TV_SCROLL structure that specifies:

- **pos** The text index of the location in text to display.
- **flags** A set of flags that specifies where the text at pos should be positioned in the view. The flags are:
  - tsAlignAtTop Position text at the top of the view.
  - tsAlignAtBottom Position text at the bottom of the view.
  - tsAlignAtCenter Position the text at the center of the view.
  - tsAlignEdge Force the text to the edge of the view. Currently this happens whether or not you specify this flag.
  - tslfflnvisble Scroll only if the text is not currently visible.
  - textNoScrollNotify Do not notify the scroll bars to update after scrolling. By default, clsTextView notifies the scroll bars that they should update after msgTextViewScroll.

In the future, msgTextViewScroll might scroll so that it leaves some space between pos and the edge of the view, thereby giving the user some context. If you want the text to scroll to the edge specified by flags, OR tsAlignEdge with the position flag.

For example:

```plaintext
flags = tsAlignAtTop | tsAlignEdge
```

If you want the user to be able to scroll the text with the pen, you should insert the text view in a scrolling window (see “Inserting a Text View in a Scrolling Window” below).
### Inserting a Text View in a Scrolling Window

To enable the user to scroll a text view, you must insert the text view in a scrolling window. Because this set of actions is performed so many times, clsTextView defines a function, `TextCreateTextScrollWin`, which creates a text view and inserts it in a scrolling window.

The prototype for the function is:

```c
STATUS TextCreateTextScrollWin(
P_TV_NEW pNew,
P_OBJECT scrollWin);
```

The function takes a pointer to a new arguments structure (`pNew`) and a pointer to an `OBJECT` (`scrollWin`). `pNew` must point to a `TV_NEW` structure or the new arguments structure for a subclass of clsTextView. If `pNew` is `null`, the function uses a `TV_NEW` structure by default.

The function uses the new argument structure to create a new text view and then creates a scrolling window with the text view as its client window. The function configures the scrolling window for text scrolling. The function determines whether to format for printer or format for screen depending on the style flag in the `TV_NEW` structure.

If the function completes successfully, it returns `stsOK` and passes back the `UID` of the scrolling window in `scrollWin`.

The following code fragment illustrates the use of the function:

```c
TV_NEW tvn;
STATUS status;
OBJECT sw;     // scrolling window

// Initialize the text view new structure.
status = ObjectCall(msgNewDefaults, clsTextView, &tvn);

// Modify initialized structure
tvn.flags = FlagClr (tvFillWithIP, tvn.flags) // Turn off insertion pad flag
tvn.showSpecial = 3;  // Show all special characters

// Create the text view and scroll window.
status = TextCreateTextScrollWin(&tvn, &sw);
if (status == stsOK)
...
```

### Getting the Current Selection

Like other views, clsTextView interacts with the selection owner. Using the Xfer mechanism, you can query the text view for the span of the text contained in the current selection.

For more information on the selection mechanism and the Application Framework, see Part 2: PenPoint Application Framework. For more information on the Xfer mechanism, see Part 9: Utility Classes.
The Xfer mechanism requires two steps: first your application must negotiate with the selection owner for the protocols available for data transfer, then you must send the message requesting the transfer using the best available protocol.

Actually, we know that the text view supports the ASCII Metrics protocol (where we pass a pointer to a metrics structure), so the negotiation is not necessary.

To get the current selection for a text view, you must:

- Make sure that your view owns the selection by sending msgSelOwner to theSelectionManager.
- Declare an XFER_ASCII_METRICS structure and set the id field to XferASCIIMetrics.
- Send msgXferGet to the text view.

When msgXferGet completes successfully, the first field of the XFER_ASCII_METRICS structure contains the text index of the first character in the selection; the length field contains the length of the selection, in characters. The level field indicates the units that make up the selection. The possible units and their values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ignore</td>
</tr>
<tr>
<td>1</td>
<td>characters</td>
</tr>
<tr>
<td>2</td>
<td>words</td>
</tr>
<tr>
<td>3</td>
<td>sentences</td>
</tr>
<tr>
<td>4</td>
<td>paragraphs</td>
</tr>
</tbody>
</table>

The following example illustrates how the Writing Paper application gets the current selection.

```c
XFER_ASCII_METRICS xaMetrics;
OBJECT view;

if (!view)
    return(stsFailed);

if (sel == view) {
    xaMetrics.id = XferASCIIMetrics;
    ObjCallRet(ObjCallWarn(msgXferGet, view, &xaMetrics), s);
    *pSelMetrics = xaMetrics;
```
Getting and Setting the Text Style

You can control several aspects about the way that text views display their contents. You can control whether the view responds to editing gestures, how the text is formatted on screen, and whether to display special characters.

To get the style for a text view, send msgTextViewGetStyle to the text view. To set the style, send msgTextViewSetStyle to the text view. Both messages take a TV_STYLE structure that contains:

- **flags** Flags that indicate the behavior of the text view. Flags include:
  - tvsEmbedOnlyComponents Don't embed applications.
  - tvsEmbedOnlyIPs Only embed insertion pads.
  - tvsFormatForPrinter Format text for a printer rather than the screen.
  - tvsReadOnlyChars Characters are read only.
  - tvsReadOnlyAttrs Attributes are read only.
  - tvsReadOnly Both characters and attributes are read only.
  - tvsWordWrap Wrap lines at word breaks.
- **magnification** A magnification value for fonts. The magnification value specifies the number of points to add to the fonts when they are displayed on screen.
- **showSpecial** A value that specifies whether the view shows special characters, such as tab, paragraph, line break, and page break characters. The special characters that are affected by this field are the line break and paragraph break characters. The possible values are 0 to show none and 3 to show all.
- **printer** In PenPoint 1.0 this should be null.

Checking Consistency of Text Views

When debugging a text view application, you can check the consistency of a text view by sending msgTextViewCheck to the text view object. This message performs work only when you use the DEBUG version of the TEXT.DLL file (in \\PENPOINT\BOOT\DLL). msgTextViewCheck takes a 32-bit value that specifies the type of consistency checks to make.

If HighU16(pArgs) is 0, LowU16(pArgs) contains flags that indicate what type of simple consistency checks should be made. Currently, the only check is made on the line table when the LowU16(pArgs) is 0.

If HighU16(pArgs) is not 0, pArgs is a pointer to a structure whose first 32 bits specify more complicated consistency checks.

If the internal check does not detect a problem, the message returns stsOK.

If you do not use the DEBUG version of the TEXT.DLL, msgTextViewCheck does not perform a consistency check and always returns stsOK.
Chapter 66 / Using Text Insertion Pads

This section describes the messages associated with text insertion pads. The text insertion pads incorporate many gestures that are useful for adding text to a text view.

Topics covered in this chapter:

- Creating and destroying text insertion pads.

### Text Insertion Pad Messages

clsTextIP is a subclass of clsIP. Messages for clsTextIP are defined in TXTVIEW.H. Table 66-1 lists the text insertion pad messages.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_TEXTIP_NEW</td>
<td>Initializes text insertion pad arguments.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_TEXTIP_NEW</td>
<td>Creates a new instance of clsTextIP.</td>
</tr>
</tbody>
</table>

#### Creating Text Insertion Pads

Usually clsTextView creates a text insertion pad by default (provided that tvFillWithIP is not clear). To create a text insertion pad, send msgNewDefaults and msgNew to clsTextIP. Both messages take a TEXTIP_NEW structure that contains a flags field. The flags field is reserved for future expansion; leave it at its default value.

After creating the insertion pad, you can insert it in a text view by sending msgTextViewAddIP to the text view, specifying the UID of the text insertion pad in the TV_EMBED_METRICS structure.

#### Destroying Text Insertion Pads

To destroy a text insertion pad, send msgDestroy to the text insertion pad object.
Chapter 67 / Sample Code

The most trivial use of the Text subsystem is to simply create an object of clsTextView and display it on the RootWindow. This can be achieved with the following code fragment.

```
#include <txtview.h>
TV_NEW new;
STATUS s;

s = ObjectCall(msgNewDefaults, clsTextView, &new);
new.win.bounds.size.h = ...;
new.win.bounds.size.w = ...;
new.win.bounds.origin.y = ...;
new.win.bounds.origin.x = ...;
s = ObjectCall(msgNew, clsTextView, &new);
if (s < stsOK) {...}

// insert the text view in the root window
new.win.parent = theRootWindow;
new.win.options = wsPosTop;
s = ObjectCall(msgWinInsert, new.object.id, &new.win);
```

The resulting view is connected to an empty data object (automatically created by the view because the view.dataObject field of the msgNew argument was not filled in). Such a view is far from useless, because the user can now use pen gestures to create an insertion pad inside the view and enter text through the insertion pad.

The client can also pre-load the data object with text by making a call such as the following before inserting the view into the window tree.

```
TEXT_BUFFER tBuf;

tBuf.first = 0;
tBuf.length = 0;
tBuf.buf = "Hello World";
tBuf.bufLen = strlen(tBuf.buf);
s = ObjectCall(msgTextModify, new.view.dataObject, &tBuf);
```

The view displays the text using the data object's default character, paragraph, and document attributes. The user can change these attributes by selecting the desired characters within the view and making a check mark ✓ to bring up the view's option sheet. Alternatively, client code can obtain these attributes by sending msgTextGetAttrs to the text object, and then modify them with msgTextChangeAttrs.
As an example, suppose that the client wanted to change the default font size to 20 points, and then to make the characters in “Hello” be in 12 points.

```c
#include <txtdata.h>
#include <txtview.h>

#define MakeTwips(i) (i*20)

TEXT_CHANGE_ATTRS tca;
TA_CHAR_ATTRS charAttrs;
TA_CHAR_MASK charMask;

tca.tag = atomChar;
tca.first = txtDefaultAttrs;
tca.pNewMask = &charMask;
tca.pNewValues = &charAttrs;
s = ObjectCall(msgTextInitAttrs, new.view.dataObject, &tca);

charAttrs.size = MakeTwips(20); // points
charMask.size = true;
s = ObjectCall(msgTextChangeAttrs, new.view.dataObject, &tca);

tca.tag = atomChar;
tca.first = 0;
tca.length = 5;
charMask.size = true;
charAttr.size = MakeTwips(12); // points
s = ObjectCall(msgTextChangeAttr, new.view.dataObject, &tca);
```
Chapter 68 / Advanced Information

This chapter covers topics that might be useful for sophisticated applications, but which most clients will not need.

Topics covered in this chapter include:

- Counting changes
- Atoms.

Counting the Changes

The text class keeps a count of the number of times msgTextModify has changed the text. You can get and set the change counter by sending msgTextChangeCount to the text data object. The argument is an S32 value. If the value is maxS32, the message increments the counter and returns the new value. If the argument value is greater than zero, the value becomes the new change count. If the value is minS32, the message returns the current counter.

Atoms

The Text subsystem provides a database of unique identifiers through a globally valid, compact, unique identifier, called atoms.

The strings stored in the database have the following properties:

- They are terminated by a zero byte.
- They have at least one character, other than the terminator.
- They do not contain characters in the ASCII ranges 1 through 31 and 127 through 159.
- Case is preserved when the strings are stored, but it is ignored when searching for a string in the database.

The atom for a nil string is Nil(ATOM).
Predefined Atoms

The following nine atoms are predefined:

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomChar</td>
<td>Any character</td>
</tr>
<tr>
<td>atomWord</td>
<td>A word, delimited by a space or the characters: &quot;!&quot;&amp;()*+,.;&lt;==&gt;?@[]^{}-&quot;</td>
</tr>
<tr>
<td>atomWSDelimit</td>
<td>A span of characters delimited by white space.</td>
</tr>
<tr>
<td>atomLine</td>
<td>A line, terminated by a paragraph, newline, or newpage mark.</td>
</tr>
<tr>
<td>atomSentence</td>
<td>A sentence, terminated by the characters: .?!</td>
</tr>
<tr>
<td>atomPara</td>
<td>A paragraph, terminated by a paragraph mark.</td>
</tr>
<tr>
<td>atomParaTabs</td>
<td>The tab stops for a paragraph.</td>
</tr>
<tr>
<td>atomDoc</td>
<td>A document, which is the whole text data object.</td>
</tr>
<tr>
<td>atomEmbedded</td>
<td>An embedded object.</td>
</tr>
</tbody>
</table>
Part 7 /
File System
## Chapter 69 / Introduction

Overview .......................... 43
Developer's Quick Start .......... 44
  Writing Objects and Data      .. 44
  Reading Objects and Data     .. 45
  Opening and Closing Files    .. 46
Comparison with Other File Systems .. 46
Organization of This Part ....... 47

## Chapter 70 / File System Principles and Organization

Volumes ................................ 49
  Volume Concepts .................. 49
  Volume Types .................... 50
Nodes ................................ 52
Directories and Directory Entries .. 54
Attributes ........................... 54
Files ................................ 55
Locators ............................. 55

## Chapter 71 / Accessing the File System

File System Handles .................. 57
  Handles and Locators ............. 58
  Directory Handles ................. 59
  File Handles ...................... 61
File System Messages ............... 62
  clsFileSystem Messages .......... 62
  clsDirHandle Messages .......... 64
  clsFileHandle Messages ......... 64
Using Handles with Temporary Files .. 65
Accessing the File System with stdio .. 65
  Translating Between Handles and FILE Pointers .. 65
  Paths and stdio .................. 66
  Using stdio ...................... 66
Concurrency Considerations ...... 66
  Protecting Your File Data ...... 66
  File Location Considerations ... 67
  Volume Protection Considerations .. 67
Subclassing File System Classes .......... 67
The PENPOINT.DIR File .......... 68
How the Notebook Uses the File System .. 68

## Chapter 72 / Using the File System

Creating Directories and Files .... 69
  Creating Handles ................. 69
  Checking Valid File and Directory Names .. 70
  Creating a Directory Handle ..... 71
  Creating a File Handle .......... 71
  Mapping a File to Memory ..... 73
Closing Files .......................... 74
Deleting Files and Directories .. 75
Forcing Deletion of a File or Directory .. 75
Getting and Setting Attributes ... 76
  Lists of Attributes ............ 76
  Zero Value Attributes .......... 77
  File System Attributes ........ 77
  Client-Defined Attributes .... 77
  Getting Attribute Values ...... 78
  Setting Attribute Values ..... 78
  Getting the Length of Attribute Values .. 79
  Node Attribute Flags ......... 79
  Creating and Using Directory Indexes .. 80
Copying and Moving Nodes ........ 80
  Traversing Nodes ............... 81
    The Traverse Call-Back Routine .. 82
    The Traverse Quicksort Routine .. 82
    Order of Traversal ........... 82
Renaming Nodes ..................... 83
Determining the Existence of a Node .. 83
Reading and Writing Files ........ 83
  File Position and Size ........ 84
    Getting and Setting File Position .. 84
    Getting and Setting File Size .. 84
Flushing Buffers .................... 85
  Getting the Path of a Handle .. 85
Changing the Target Directory .. 86
Comparing Handles .................. 86
Getting and Setting Handle Mode Flags .. 87
Reading Directory Entries .......... 87
  Reading All Directory Entries .. 88
  Sorting Directory Entries ..... 88
Observing Changes .................. 89
Making a Node Native ............... 89
Getting Volume Information ..... 90
Setting or Changing a Volume Name .. 91
Ejecting Floppies ................. 91
Volume Specific Messages ........ 91
List of Figures

70-1 Directory Structure on a Volume 53
70-2 Contents of a PENPOINT:DIR Directory Entry 54
71-1 Using Directory Handles 60
71-2 File Handles and Byte Position 61
72-1 File Attribute Arguments 76

List of Tables

69-1 Common File System Operations 47
71-1 clsFileSystem Messages 63
71-2 Directory Handle Instance Messages 64
71-3 File Handle Instance Messages 64
72-1 Directory Mode Flags 71
72-2 File Mode Flags 72
72-3 File System Attributes 77
72-4 Node Attribute Flags 80
72-5 FS_SEEK Flags 84
72-6 Volume Metrics Information 90
Chapter 69 / Introduction

The PenPoint™ file system enables you to control and access all aspects of files and file organization. This part describes the file system and how you use it.

Overview

The organization of the file system is similar to most hierarchical file systems. The file system has a hierarchical structure of directories and files (similar to those of MS-DOS and UNIX). Each directory or file is called a node.

The file system is divided into volumes. Each volume represents an installable portion of the file system. PenPoint has a boot volume, which is always present, and supports the dynamic connection and disconnection of additional volumes.

Each volume has a root directory, which is the starting point for the hierarchical organization on that volume. The file system maintains a list of all currently known volumes, connected and disconnected.

You can use file system handle objects to access nodes. There are two types of file system handles: directory handles and file handles. To create a directory or file handle, you send msgNew to either clsDirHandle or clsFileHandle.

To perform file system operations, you send messages to file or directory handles. Messages sent to file handles affect the file directly. Messages sent to directory handles usually include other information indicating the specific node that the message affects. Much confusion can result if you don't remember that a handle is not a node, but a means to access a node. You send messages to the handles; the handles in turn indicate which node to modify.

The file system allows you to:

- Create, open, close, and delete files.
- Read and write file data.
- Copy, rename, and move files and directories.
- Seek to a new position within a file or find out the current byte position within a file.
- Modify file and directory attributes.
- Create user-defined attributes for files and directories.

You can also access nodes with the stdio run-time package. The stdio run-time package provides a conventional C-language interface to the file system. stdio is faster than the object-based file system when performing many small reads or writes to a file. On the other hand, stdio calls are no faster at operations such as opening files, and stdio can't provide all the features of the PenPoint file system.
You use the file system to create data files for your applications, to read and write data in those files, and to manipulate file and directory organization. You can also use the file system to get information about volumes and to get and set nodes' attributes. PenPoint itself uses the file system for storing objects and application data. The organization of the Notebook, which is a specialized PenPoint application, is mapped onto the file system. Each section in the Notebook is a directory in the file system. Each page in a section is a directory within that directory. The In box and Out box sections of the Notebook use the file system to store the contents of their queues.

**Developer's Quick Start**

As an application writer, you'll most commonly use the file system to:

- Save your instance data when you receive `msgSave`.
- Restore your instance data when you receive `msgRestore`.
- Open and close files that contain data from other operating systems.

**Writing Objects and Data**

When the user closes your application (turns away from or deletes your application), you'll receive `msgSave`. Upon receiving `msgSave`, you must:


- Write your instance data to the resource file by sending `msgStreamWrite` to the resource file handle passed to you in the `pArgs` for `msgSave`. You can use `msgStreamWrite` more than once.

There are several approaches to writing your instance data. If you have fixed-length data, it is a good idea to copy the data to a single struct and write that struct to the file. However, if you have variable-length instance data, you'll need to use one `msgStreamWrite` message to write the length of the data and another to write the data itself. That way you will know how many bytes to specify in `msgStreamRead` when you read the data back in.

If you do write several small pieces of data, you might consider using `stdio` functions rather than `msgStreamWrite`. “Accessing the File System with `stdio`,” in Chapter 72, Using the File System, describes some of the criteria for determining when this is appropriate and how to access `stdio` using file handles.

The following example shows a typical message handler that responds to `msgSave`:

```c
MSGPROC ...
    case Msg(msgSave):
        return MyAppWrite(self, (P_OBJ_SAVE)pArgs, ctx, pInst);

METHOD MyAppWrite(
    OBJECT class,
    P_OBJ_SAVE pObjSave,
    CONTEXT ctx,
    P_MY_INSTANCE_DATA pInst)
```
READING OBJECTS AND DATA

When the user opens or reopens your application, you'll receive msgRestore.
Upon receiving msgSave, you must:

- Re-create your objects by sending msgResGetObject to the resource file
  handle passed to you by msgRestore. Part 11: Resources describes how to use
  msgResGetObject.
- Read your instance data by sending msgStreamRead to the resource file
  handle.

Be sure to read the same number of bytes that you wrote when you received
msgSave.

The following example shows a typical message handler that responds to
msgRestore:

```c
MSGPROC ...
    case Msg(msgRestore):
        return MyAppRead(self, (P_OBJ_RESTORE)pArgs, ctx, pInst);

METHOD MyAppRead(
    OBJECT class,
    P_OBJ_RESTORE pObjRest,
    CONTEXT ctx,
    P_MY_INSTANCE_DATA pInst)
{
    STATUS s;
    STREAM_READ_WRITE srw;
    OF_GET get;
    MY_FIELD_DATA filedData;
    ObjectCallAncestor(msgRestore, class, (P_ARGS)pObjRest, ctx);
    /* Read in the field instance data */
    srw.pBuf = &filedData;
    srw.numBytes = SizeOf(MY_FIELD_DATA);
    s = ObjectCall(msgStreamRead, pObjRest->file, &srw);
    pInst->foo = filedData.foo;
    pInst->bar = filedData.bar;
    return stsOK;
}
```
Opening and Closing Files

The other common reason for using the file system is for reading or writing files that belong to other operating systems. For example, you might want to read a database file created on an MS-DOS machine so that you could import the data to your PenPoint application.

To open a file:

1. Declare an FS_NEW structure.
2. Fill in the defaults by sending msgNewDefaults to clsFileHandle.
3. Modify the FS_NEW structure to specify the volume, directory, and file that you want to open.
4. Send msgNew to clsFileHandle.

msgNew returns a file handle on the open file. You can read and write data by sending msgStreamRead and msgStreamWrite to the file handle.

To close the file, send msgFree to the file handle.

The following code excerpt shows how to open and close a file:

```c
STATUS s;
FS_NEW fsNew;
FILE_HANDLE myFileHandle;

s = ObjectCall(msgNewDefaults, clsFileHandle, &fsNew);
/* The following filled in by msgNewDefaults */
fsNew.object.key = objWKNKey;
fsNew.object.cap = objCapCall;
fsNew.object.uid = null;
fsNew.fs.locator.uid = theWorkingDir;
fsNew.fs.mode = fsFileNewDefaultMode;
fsNew.fs.exist = fsExistDefault;
fsNew.fs.locator.pPath = "MyDir\\MyFile";
status = ObjectCall(msgNew, clsFileHandle, &fsNew);
myFileHandle = fsNew.object.uid;

/* time to free the handle */
s = ObjectCall(msgFree, myFileHandle, (P_ARGS) objWKNKey);
```

Comparison with Other File Systems

The PenPoint file system is similar to most other file systems in the following ways:

- It has a hierarchical directory structure.
- You locate nodes in the file system hierarchy by using paths.
- Files and directories have attributes, which you can get and set.
- You access files through handles.
The PenPoint file system is different from other file systems in these ways:

- File system handles are instances of file system classes; you can subclass the file system classes to add special capabilities.
- Clients can define their own file and directory attributes. Many file systems have file and directory attributes, but very few allow clients to add their own attributes.
- In some operating systems (including UNIX), nodes can be subordinate to more than one directory; in PenPoint nodes can have only one parent.
- The PenPoint file system is designed to smoothly handle events when volumes are disconnected and reconnected.

Table 69-1 describes how PenPoint implements many common file operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>File System Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get a list of installed volumes</td>
<td>Send msgFSGetInstalledVolumes to theFileSystem.</td>
</tr>
<tr>
<td>Get information on a volume</td>
<td>Send msgFSGetVolMetrics to a handle for a file or directory on that volume.</td>
</tr>
<tr>
<td>Get a node's attributes</td>
<td>Send msgFSGetAttr to a file or directory handle.</td>
</tr>
<tr>
<td>Change a node's attributes</td>
<td>Send msgFSSetAttr to a file or directory handle.</td>
</tr>
<tr>
<td>Open a file, create if necessary</td>
<td>Send msgNew to clsFileHandle. (Create is the default.)</td>
</tr>
<tr>
<td>Open a file, return error if it</td>
<td>Send msgNew to clsFileHandle specifying fsNoExistGenError in exist.</td>
</tr>
<tr>
<td>doesn't exist</td>
<td></td>
</tr>
<tr>
<td>Read from a file</td>
<td>Send msgStreamRead to a file handle.</td>
</tr>
<tr>
<td>Write to a file</td>
<td>Send msgStreamWrite to a file handle.</td>
</tr>
<tr>
<td>Move file pointer within a file</td>
<td>Send msgFSSeek to a file handle.</td>
</tr>
<tr>
<td>Copy a file or directory</td>
<td>Send msgFSCopy to a directory handle, specifying the path to the file and the location for the new file.</td>
</tr>
<tr>
<td>Move a file or directory</td>
<td>Send msgFSMove to a directory handle, specifying the path to the file and the location for the new file.</td>
</tr>
<tr>
<td>Rename a file or directory</td>
<td>Send msgFSSetAttr to a directory handle, specifying a path to a file or directory node and a new name for that node.</td>
</tr>
<tr>
<td>Change a directory handle's</td>
<td>Send msgFSSetTarget to a directory handle.</td>
</tr>
<tr>
<td>target directory</td>
<td></td>
</tr>
<tr>
<td>Delete a file or directory</td>
<td>Send msgFSDelete to a directory handle, specifying a path to the node to delete.</td>
</tr>
<tr>
<td>Close a file</td>
<td>Send msgFSFree to the directory handle.</td>
</tr>
</tbody>
</table>

### Organization of This Part

This part is organized into four chapters.

Chapter 69, Introduction, presents a brief overview of the file system.

Chapter 70, File System Principles and Organization, describes how the file system is organized and presents the terminology used with the file system.
Chapter 71, Accessing the File System, describes how you access the file system. It describes how the notebook and other applications use the file system. This chapter also describes how to subclass file system classes and presents considerations for dealing with remote file systems.

Chapter 72, Using the File System, describes in detail how you use the file system to perform most file system operations.
Chapter 70 / File System Principles and Organization

This chapter discusses general concepts about the file system organization and defines file system terms. Topics covered in this chapter include:

- Volumes supported by the PenPoint™ operating system, including local disk volumes, remote volumes, memory-resident volumes, memory mapped files.
- File and directory nodes.
- Directories and directory entries.
- File and directory attributes.
- Files.
- Locators.

Volumes

Directories and files are grouped together in a volume. When a volume is removed, the files and directories on that volume are no longer accessible to the file system.

Volume Concepts

The file system maintains a list of volume objects. You can get a copy of this list by sending msgGetInstalledVolumes to the well-known object theFileSystem. When a new volume is connected to the PenPoint computer, the UID of the volume's root directory handle is added to the list of volumes.

You can make yourself an observer of the volume list by sending msgAddObserver to theFileSystem. When a volume is added, removed, or changes state, you will receive notification.

Volume Metrics

Each volume has information associated with it. You can retrieve volume information by sending msgFSGetVolMetrics to a volume object. Volume information includes:

- The volume's type.
- The volume's name.
- The volume's root directory.
- The volume's serial number.
- The total number of bytes on the volume.
- The number of free bytes on the volume.
There are three types of volumes: local disk volumes, remote volumes, and memory-resident volumes. These types are explained in detail below.

The possible characters in the volume name, depend on the volume type.

**Duplicate Volume Names**

The file system allows duplicate volume names. When you write application programs, be aware that duplicate volume names might exist. This is especially critical when you use the list of volumes to find a particular volume. You can use `msgFSGetVolMetrics` and examine the `volType`, `serialNum`, and `creationDate` values to distinguish between volumes with the same name.

**Connecting and Disconnecting Volumes**

PenPoint supports the dynamic connection and disconnection of volumes. For example, when the user connects the PenPoint computer to a network, the network volumes become available. The user installs and removes a volume as a single unit; either the file system can access the entire volume or it cannot access it at all.

The user can physically disconnect a volume from the PenPoint computer. For example, the user can disconnect the computer from a network, or remove a floppy disk from its drive. Your application must be prepared to handle this situation.

If any task has active references to nodes on the disconnected volume, the volume remains in the volume list, but is no longer marked connected. Any attempts by your application to read or modify data on a disconnected volume pops up a dialog box asking the user to connect the volume. (You can suppress this behavior when you first access the file; see “Creating a File Handle” in Chapter 72.)

If the user connects the volume and taps the OK button on the dialog, the file system operation completes and the volume can be used normally once again. Optionally, the user can tap the Cancel button, which causes the file system to send the status message `stsFSVolDisconnected` to your application.

If no tasks have active references to nodes on the disconnected volume, that volume is removed from the volume list, effectively removing all traces of the volume from the PenPoint computer.

**Volume Types**

As mentioned above, the file system defines three types of volumes:

- **Local disk volumes**
- **Remote volumes**
- **Memory-resident volumes**.

Local disk volumes are volumes directly attached to the PenPoint computer. Remote volumes are those that are connected through a network or channel controller. Memory resident volumes reside in PenPoint computer RAM.
Local Disk Volumes

Local disk volumes exist on hard or floppy disk drives internal to or attached to the PenPoint computer. The user can connect and disconnect external disks at any time.

PenPoint has no native disk format. Rather, the file system makes use of a volume's normal organization to store additional information. Currently PenPoint uses the MS-DOS FAT disk format.

An MS-DOS volume name is a string containing between 1 and 11 characters. The volume name cannot use the following characters:

/ \ ; = < > [ ]

The name can use spaces, but not tabs.

Here are some examples of MS-DOS volume names:

- A
- MYDISK
- RAM
- MY VOLUME
- BACKUP 3
- ACCNTS RCVL.

The MS-DOS disk format consists of files and directories. In MS-DOS, file names are limited to 8 characters with a 3-character file extension, there are only a few file attributes, and there is only one type of file (data). PenPoint uses MS-DOS format files and directories whenever possible to store its own files and directories. However, when a file or directory has one or more of these characteristics:

- A node name that uses any control characters, lowercase characters, or any of these special characters: * ? / \ . ; : + = < > [ ] " (space).
- A name longer than eight characters, plus a three-character extension.
- A name that uses lower case characters.
- Has client-defined attributes.

PenPoint creates a special file, named PENPOINT.DIR, in that file or directory's parent directory, which contains the additional information. The PENPOINT.DIR file is described in further detail in Chapter 71, Accessing the File System.

In the future, PenPoint may support disk formats other than the MS-DOS FAT format. The file system architecture makes it easy for GO or third parties to develop support for other disk formats.

Remote Volumes

Remote volumes are available over a network or a communication channel. The name for a remote volume is limited only by the network's volume naming conventions.
A computer that responds to a remote file access protocol is called a **remote file server**. A remote file server can be any sort of computer, ranging from a personal computer to a dedicated file server with gigabytes of storage. The administrator of the remote file server decides how much of the file system to make available to the PenPoint computer and what sort of security to enforce.

As with local disk volumes, the file system provides features that might exceed the remote file system’s capabilities. PenPoint supports these extended features in a fashion similar to that for local disk volumes.

To access a remote volume, file system operations must use the **remote file access protocol** that is appropriate to the server. For example, a TOPS remote volume requires a TOPS remote file access protocol. Like file system support for local disk volumes, the file system architecture makes it easy for GO or third parties to develop support for other protocols.

**Memory-Resident Volumes**

Memory-resident volumes reside in the PenPoint computer’s RAM. The memory-resident RAM volume is named RAM. This volume is only available with the SDK version of PenPoint 1.0, and only when the ENVIRON.INI file includes the `Config=DebugRAM` directive. There is only one RAM volume, and PenPoint dynamically allocates RAM memory to the RAM volume as needed.

**Nodes**

Within each volume, the file system maintains a tree of file system **nodes**. There are two types of nodes:

- Directory nodes
- File nodes.

Directories are catalogs of files and directories. Directories can contain both files and other directories. As in MS-DOS, a volume is organized into a strictly hierarchical structure of files and directories. Strictly hierarchical means that each node appears in only one directory; there are no multiple references to nodes.

The top-most node in a volume’s directory structure is the **root directory**. All other nodes in the volume are descendants of the root directory. Because a volume is self-contained, the directories within a given volume contain nodes only in that volume.

Figure 70-1 illustrates an example of the file system on a volume.
The name of a node is a string containing 1 to 31 characters. Any character is valid except backslash (\) and null (character code 0), and a node name is not valid if it begins or ends with a space character. Here are some examples of valid node names:

- Red
- GO Corporation
- A Very, Very, Lengthy Node Name
- Patient X: Cardiac Status
- FILENAME.DOS.

Note that the characters can be uppercase or lowercase. The file system stores the node names in upper and lower case, but ignores the case when it performs comparisons.

Your application can use the FSNameValid function to determine if a user-supplied file name is a valid node name.

All of the node names within a single directory are unique; the file system will reject any request to create a duplicate node name. You can use the fsExistGenUnique or fsNoExistCreateUnique flags, discussed in Chapter.72,
Using the File System, to direct the file system to generate a unique node name when it creates a new node.

**Directories and Directory Entries**

A directory is a container for file system nodes; it contains a directory entry for each node directly subordinate to it. Each directory entry contains the file-system and client-defined attributes for the node it describes.

Figure 70-2 shows the contents of one of the directory entries shown in Figure 70-1.

**Attributes**

Each node in the file system has a set of attributes. The file system stores a node’s attributes in its parent directory (with the directory entry) or in the PENPOINT.DIR file if the attributes are beyond the native file system’s capabilities. There are two kinds of attributes:

- Client-defined attributes
- File-system attributes.

Nodes can have any number of client-defined attributes. Client-defined attributes can contain any sort of data. Clients can explicitly create and destroy client-defined attributes.
Every node has a fixed set of file system attributes. Clients can alter some of these attributes, but cannot remove any of them. Among the file system attributes are:

- The node name.
- Flags (such as access flags).
- Modification date.
- For files, the size of the file.

**Files**

A file is a repository for data. The maximum size of a PenPoint file is limited only by the available disk or memory space.

When an application opens a data file, it must read information from the file into main memory. An alternative to traditional `stdio`-style file access is to memory map the file.

PenPoint support memory mapped files, which allow you to address data in a file as if it were in main memory. PenPoint makes use of several memory mapped files, including:

- Font files.
- Handwriting prototypes.
- The dictionary, which is used to proof handwriting translations.
- The indexes for the Table Server (`clsTable`).

Memory mapped files will work for all volume types.

**Locators**

To find a particular node, a client must be able to specify the location of the node in the file system. The location is specified by a locator, which consists of:

- A starting point.
- A path to a node.

The starting point is a handle on either a directory or file node.

A path is a null-terminated string that defines a traversal of the file system hierarchy. There are five types of paths:

- If the path is null, the location is the starting point node.
- If the path begins with "." or contains a node name, the traversal begins at the starting point directory node.
- If the path begins with "..", the traversal begins at the parent directory of the starting point. If the path contains ".." the target of the path is the directory that contains the starting point node.
If the path begins with a backslash (\), the traversal begins at the root directory of the starting point's volume.

If the path begins with two backslashes (\\), what follows is a volume name. The traversal begins at the root directory of that volume. If the volume name is not recognized as an installed volume, the file system prompts the user to attach the correct volume.

Locators in the file system messages take on two forms. **Explicit locators** use the FS_LOCATOR structure, which contains both a starting point and a path. **Implicit locators** use the file or directory handle to which the message was sent as the starting point and require only a path argument.

Because PenPoint is a multi-tasking operating system, it is possible for another task to change (move, rename, or delete) nodes in the file system tree. A path that successfully located a node at one time might not locate the same node at a later time. Remember that a path is not a direct handle on the node, but more like a road map to it; the location of a node might change.

Another way to locate a directory is to use a directory index. A **directory index** is a unique identifier for a directory. You create a directory index the same way you create an attribute. Directory indexes work only for nodes in the \PENPOINT directory tree.
Chapter 71 / Accessing the File System

This chapter discusses the mechanism by which you access the file system.

Topics covered in this chapter include:
- Directory and file handles.
- Volume root directory handle.
- Working directory handle.
- File access control.
- Summary of file system messages.
- Temporary files.
- Using stdio function calls.
- Concurrency considerations.
- Subclassing the file system classes.
- The PENPOINT.DIR file.

File System Handles

The programmatic interface to the file system provides:
- Access to information in files.
- Inspection and modification of node attributes.
- Alteration of the directory hierarchy.

This is all accomplished through file system handles. Handles provide a uniform method of accessing file system nodes for every type of volume and shield your application from the low-level file system implementation.

You access files and directories on a volume by sending messages to file system handles.

There can only be one file system operation taking place at any one time on a given volume. Each operation on a volume is run to completion before another operation is permitted to start. However, operations taking place on different volumes can take place concurrently. For example, when copying a text file from one volume to another, the file system can read text from one volume while it writes text to another.

Handle objects guarantee consistent and atomic results if simultaneously used by different tasks. If two tasks send messages to the same handle object at the same time, or if two tasks use two handles to access the same file at the same time, the
PenPoint™ operating system will suspend one task until it finishes processing the operation specified by the other task.

PenPoint defines two file system classes:

- clsDirHandle (directory handles), descended from clsObject.
- clsFileHandle (file handles), descended from clsStream.

Although these classes descend from different classes, they are designed to handle a common set of clsFileSystem messages. clsFileSystem messages perform functions such as creating new handles and nodes, destroying handles, and manipulating node attributes. Generally, do not send messages directly to an instance of clsFileSystem. Instead, you send messages to directory and file handles, both of which are written to handle most clsFileSystem messages. A few messages are specific to one of clsFileHandle or clsDirHandle.

To create a directory handle, you send msgNew to clsDirHandle; to create a file handle, send msgNew to clsFileHandle. A directory handle has a target directory node; a file handle points to a file node. In the arguments to msgNew, you specify the location of the node. You can also include a request in the message to create the node if it does not exist already. If the message succeeds, the file system returns a handle.

A process can create any number of handle objects (up to the memory limit of the PenPoint computer). PenPoint may also allocate disk or communication system buffers as a side effect of handle creation. Therefore, you should free handle objects when they are no longer needed.

If a task terminates while it has active handles, the file system frees the handles.

You can create subclasses of the file system handles to implement specialized file access behavior. Creating descendant classes does not change the actual disk or memory layout of files and directories; it only changes the manner in which clients access them.

**Handles and Locators**

When you use handles, it is important to remember that the handle is not the node itself.

The idea of a completely object-oriented programming environment encounters some obstacles when it is applied to a file system. In a completely object-oriented file system, each node would be an object. To perform any operation in such a file system, you would send a message directly to the node. This approach has these disadvantages:

- Each time you send a message to an object, the file system has to locate that object.
- As the number of files and directories grows, it becomes slow and unwieldy.
- Objects are usually short-lived. Files, being a means of permanent storage, are much longer-lived than objects.
Rather than make each node an object, PenPoint uses handles, which add a level of indirection between the object-based messages and the file system.

A handle is all you need in operations that manipulate information within files and handles. For example, you need a handle on a file for msgStreamWrite (which alters data in the file) and msgFSSeek (which alters the current file position in the handle).

A file handle has a one-to-one relationship with its file. On the other hand, directory handles have a target directory, which you can change to point to other directories (with msgFSSetTarget). Some operations (msgFSGetAttr and msgFSGetPath) allow you to indicate a file in two ways: either by sending the message to a directory handle and specifying a path or by sending the message directly to a file handle.

There are file operations that do not manipulate information within files. These operations, such as move or copy, do not need to open nodes. In these operations you use locators to indicate the source and destination nodes.

A locator makes directory handles even more flexible. A locator consists of a file handle or a directory handle and a path. A file handle indicates the node; a directory handle is merged with the path to give the location of a node. Thus, you can use a locator to indicate a specific directory or file, without having a handle on it. Also you can use one directory handle with any number of paths (one at a time) to indicate many nodes.

Under the system of handles and locators, you can still write an application that has a handle for each node (provided the memory will allow it). In a limited way, this might be a desirable thing to do; however, handles do consume memory. If you are concerned about memory limitations (and most PenPoint programmers should be concerned), this is a high amount of overhead.

At the other extreme, you can use current directory or volume root handle and use paths to specify all of the nodes. This scheme has to be balanced against the memory that you use storing all of the path name strings, and the overhead of always checking to be sure that the path still indicates a valid node.

**Directory Handles**

Directory handles support operations that query and manipulate nodes in the file system.

A directory handle is associated with a directory node. You can use a directory handle to designate the location of a directory, to create, copy, move, and delete nodes, and to access the contents of directories.

Each directory handle has a target directory. The target directory is set when you create the directory handle; you can change it at any time thereafter. This allows you to use one directory handle to roam through the file system tree, and is similar to the working directory concept in MS-DOS or UNIX.
You can also create a directory handle with the \texttt{fsUnchangeable} flag set, which disallows changing the target directory. Figure 71-1 shows how directory handles are used in the file system.

![Diagram of directory handles](image)

When you rename or move a node, the file system modifies any other directory handles that reference it, so that they follow the node to its new location.

You can make yourself an observer of directory handles. If you observe a directory handle, you receive notifications of any changes to nodes in the directory (but not to the directory node itself).

Note that unlike files, there is no access control for directories, other than making a directory node hidden.

Applications can pass directory handles between processes. However, the file system destroys directory handles when the process that owns them is destroyed.

PenPoint has several well-known directory handles. Some are defined by the file system, others are defined by the application framework.

- The boot volume’s root directory handle.
- The “selected” volume, PenPoint’s primary operating volume (this is usually, but not always, the same as the boot volume).
- The current working directory handle.
- For the SDK version of PenPoint 1.0, the RAM volume’s root directory handle.

### Volume Root Directory Handle

Each volume has a root directory. A \textbf{root directory handle} is an unchangeable directory handle that points to a volume’s root directory. There are two ways to get a volume’s root directory handle:

- Send \texttt{msgFSGetInstalledVolumes} to \texttt{theFileSystem} to get a list of available volumes, then send \texttt{msgFSGetVolMetrics} to each volume object in the list, until you find the needed volume.
• Specify a locator that has a null UID and a path that contains only the volume name.

A volume's root directory handle is stored in the FS_VOL_METRICS structure.

**The Working Directory Handle**

`theWorkingDir` is a local, well-known directory handle object created by the file system for each task at creation time. `theWorkingDir` is similar to the DOS or UNIX concept of a default directory.

Both `msgNewDefaults` and the `stdio` run-time package uses `theWorkingDir` for its default volume and directory.

**The RAM Volume Handle**

With the SDK version of PenPoint 1.0, it is possible to configure the file system so that there is a volume in RAM. When configured, the RAM volume has a root directory and a handle on that directory.

**File Handles**

You use file handles to access data in a file node. Creating a file handle is analogous to opening a file in MS-DOS or UNIX. Destroying a file handle is like closing a file (however, destroying a handle should not be equated with deleting the file to which the handle refers).

Each file handle has a **current byte position**, which represents the handle's current position in the file. The current byte position points to the next byte to be read or written. It is updated by read, write, and seek messages.

When the current byte position is at the end of a file, it is set one byte beyond the last byte in the file. Writing past end-of-file automatically enlarges the file and sets the current byte position to the new end-of-file. Reading past the end-of-file returns less data than was requested and sets the current byte position to the end-of-file.

Figure 71-2 illustrates the use of file handles.
Applications can pass file handles between processes. However, file handles are destroyed when the process that owns them is destroyed. When a client renames or moves a file node, all handles that referenced that node continue to reference the node.

File Access Control

The file system supports limited access control for files. When you create a file handle, you specify a set of access intentions (that is, what you plan to do with the file) and a set of exclusivity requirements (the limits you want to place on other applications that might attempt to access the same file). The access intentions are:

- Read-only access
- Read/write access.

Exclusivity requirements are:

- Exclusive access
- Deny other writers
- Public access.

Exclusivity requirements apply when the file system is asked to create a file handle on the same file.

Note that access intentions and exclusivity requirements pertain to handles. Each file has its own read-only attribute flag, which you set with msgFSSetAttr. When a file is marked read-only, you must specify read-only access when you create a handle for that file.

When you create a file handle, the file system compares your handle access intentions and exclusivity requirements to the current state of the file (file's attributes and any existing handles on the file). If your request is compatible with the file's state, the file system allows you access to the file and returns you a file handle.

The access intentions, exclusivity requirements, and the file access flags are only compared when you attempt to create a file handle. Once you have a file handle, a change to the access intentions, exclusivity requirements, or file's access flags will have no effect on you or anyone else currently accessing the file.

File System Messages

This section summarizes the file system messages for each of the three classes (clsFileSystem, clsFileHandle, and clsDirHandle). clsFileSystem, a global well-known, is the only instance of clsFileSystem. Although neither clsFileHandle nor clsDirHandle descend from clsFileSystem, they are designed to handle most of the clsFileSystem messages. A few clsFileSystem messages apply only to clsFileHandle or to clsDirHandle.

clsFileSystem Messages

clsFileSystem defines the operations that are common to directory handles and file handles.
Table 71-1 lists the clsFileSystem messages. The class messages are those that you send to clsDirHandle and clsFileHandle to create new instances of these classes; the instance messages are those that you send to clsDirHandle and clsFileHandle to operate on individual instances of these classes.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_FS_NEW</td>
<td>Creates a directory or file handle object on a new or existing dir/file.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_FS_NEW</td>
<td>Initializes the FS_NEW structure to default values.</td>
</tr>
<tr>
<td>msgFSNull</td>
<td>void</td>
<td>Does nothing.</td>
</tr>
<tr>
<td>msgFSGetVolMetrics</td>
<td>P_FS_GET_VOL_METRICS</td>
<td>Returns metrics of the volume.</td>
</tr>
<tr>
<td>msgFSSetVolName</td>
<td>P_STRING</td>
<td>Changes the name of a volume.</td>
</tr>
<tr>
<td>msgFSNodeExists</td>
<td>P_FS_NODE_EXISTS</td>
<td>Tests the existence of a file or directory node.</td>
</tr>
<tr>
<td>msgFSSetHandleMode</td>
<td>P_U16</td>
<td>Returns the “new” mode for the object’s fs handle.</td>
</tr>
<tr>
<td>msgFSSame</td>
<td>OBJECT</td>
<td>Changes the “new” mode for the object’s fs handle.</td>
</tr>
<tr>
<td>msgFSGetPath</td>
<td>P_FS_GET_PATH</td>
<td>Gets the path to (or name of) a directory or file handle node.</td>
</tr>
<tr>
<td>msgFSGetAttr</td>
<td>P_FS_GET_SET_ATTR</td>
<td>Gets an attribute or attributes of a file or directory node.</td>
</tr>
<tr>
<td>msgFSSetAttr</td>
<td>P_FS_GET_SET_ATTR</td>
<td>Sets the attribute or attributes of a file or directory node.</td>
</tr>
<tr>
<td>msgFSMove</td>
<td>P_FS_MOVE_COPY</td>
<td>Moves a node (and any children) to a new destination.</td>
</tr>
<tr>
<td>msgFSCopy</td>
<td>P_FS_MOVE_COPY</td>
<td>Copies a node (and any children) to a new destination.</td>
</tr>
<tr>
<td>msgFSMoveNotify</td>
<td>P_FS_MOVE_COPY_NOTIFY</td>
<td>Same as msgFSMove with notification routine extensions.</td>
</tr>
<tr>
<td>msgFSCopyNotify</td>
<td>P_FS_MOVE_COPY_NOTIFY</td>
<td>Same as msgFSCopy with notification routine extensions.</td>
</tr>
<tr>
<td>msgFSDelete</td>
<td>P_STRING</td>
<td>Deletes a node (and all of its children).</td>
</tr>
<tr>
<td>msgFSFlush</td>
<td>void</td>
<td>Flushes any buffers and attributes associated with the file or directory.</td>
</tr>
<tr>
<td>msgFSMakeNative</td>
<td>P_FS_MAKE_NATIVE</td>
<td>Removes anything not supported by the native file system.</td>
</tr>
<tr>
<td>msgFSEjectMedia</td>
<td>void</td>
<td>Ejects media from an ejectable, removable volume.</td>
</tr>
<tr>
<td>msgFSForceDelete</td>
<td>P_STRING</td>
<td>Forcibly deletes a node (and all of its children).</td>
</tr>
<tr>
<td>msgFSVolSpecific</td>
<td>P_FS_VOL_SPECIFIC</td>
<td>Sends a volume specific message via a dir or file handle.</td>
</tr>
<tr>
<td>msgFSChanged</td>
<td>P_FS_CHANGE_INFO</td>
<td>Notifies observers of directory changes.</td>
</tr>
<tr>
<td>msgFSVolChanged</td>
<td>P_FS_VOL_CHANGE_INFO</td>
<td>Notifies observer of volume changes.</td>
</tr>
<tr>
<td>msgFSGetInstalledVolumes</td>
<td>P_LIST</td>
<td>Returns list of all installed volumes.</td>
</tr>
</tbody>
</table>
clsDirHandle Messages

Directory handle objects support certain operations unique to directories. For example, to open or create a file, you first need a directory handle, to which you send a message. You can always use the root directory handle for a volume, or your own theWorkingDir handle.

In addition to the instance messages listed in Table 71-1 above, a directory handle responds to the instance messages listed in Table 71-2.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgFSSetTarget</td>
<td>P_FS_LOCATOR</td>
<td>Changes the target directory to directory specified by locator.</td>
</tr>
<tr>
<td>msgFSReadDir</td>
<td>P_FS_READ_DIR</td>
<td>Reads the next entry (its attributes) from a directory.</td>
</tr>
<tr>
<td>msgFSReadDirReset</td>
<td>void</td>
<td>Resets the ReadDir position to the beginning.</td>
</tr>
<tr>
<td>msgFSReadDirFull</td>
<td>P_FS_READ_DIR_FULL</td>
<td>Reads all the entries in a directory into a local buffer.</td>
</tr>
<tr>
<td>msgFSTraverse</td>
<td>P_FS_TRAVERSE</td>
<td>Traverse through the nodes of a tree starting with the target of this msg.</td>
</tr>
</tbody>
</table>

Most of these messages take a path argument. The file system uses the directory handle's target directory and the path to determine the location of the node to act upon. You can reference the target directory alone by supplying an empty path. Some messages do not take a path. These messages always operate on the target directory node.

clsFileHandle Messages

In addition to the instance messages listed in Table 71-1 above, a file handle responds to the instance messages listed in Table 71-3.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgStreamRead</td>
<td>P_STREAM_READ_WRITE</td>
<td>Reads data from the file.</td>
</tr>
<tr>
<td>msgStreamWrite</td>
<td>P_STREAM_READ_WRITE</td>
<td>Writes data to the file.</td>
</tr>
<tr>
<td>msgStreamFlush</td>
<td>void</td>
<td>Flushes any buffers associated with the file.</td>
</tr>
<tr>
<td>msgStreamSeek</td>
<td>P_STREAMSEEK</td>
<td>Seeks to new position within the file.</td>
</tr>
<tr>
<td>msgFSSeek</td>
<td>P_FS_SEEK</td>
<td>Seeks to new position within the file.</td>
</tr>
<tr>
<td>msgFSIZEetSize</td>
<td>P_FS_FILE_SIZE</td>
<td>Gets the size of the file.</td>
</tr>
<tr>
<td>msgFSSetSize</td>
<td>P_FS_SET_SIZE</td>
<td>Sets the size of the file.</td>
</tr>
<tr>
<td>msgFSMemoryMap</td>
<td>PP_MEM</td>
<td>Associates the file with a directly accessible memory pointer.</td>
</tr>
<tr>
<td>msgFSMemoryMapFree</td>
<td>void</td>
<td>Frees the memory map pointer currently associated with the file.</td>
</tr>
<tr>
<td>msgFSMemoryMapSetSize</td>
<td>SIZEOF</td>
<td>Sets the size of the file's memory map.</td>
</tr>
<tr>
<td>msgFSMemoryMapGetSize</td>
<td>P_SIZEOF</td>
<td>Gets the size of the file's memory map.</td>
</tr>
<tr>
<td>FNameValid()</td>
<td>P_STRING</td>
<td>Function to check a file or directory name for validity.</td>
</tr>
</tbody>
</table>
Using Handles with Temporary Files

Sometimes you only need a file for a short time during the life of a task. You might need a file that behaves like an object, which you bring into existence, use it, and when you don't need it, you free it. To create a temporary file, specify the fsTempFile flag in your call to msgNew.

When you specify fsTempFile for a non-existent file (and the exist flags specify fsNoExistCreate), the file system creates the handle and the file at the same time. When you free the handle, the file system deletes the file.

If you specify fsTempFile for a file that exists already (and the exist flags specify fsExistGenUnique), the file system deletes the file when you free the handle.

If you don't know whether you will want to keep the file or not when you create it, do not use fsTempFile. If you decide you want to delete the file, you can always use msgFSSetHandleMode to make the handle an fsTempFile handle, or use msgFSDelete to delete the file before you free the handle.

Accessing the File System with stdio

Your applications will usually open and close files and do most directory management with file system handles. However, to facilitate porting an existing code base, PenPoint supports the use of stdio calls to perform file handle. stdio calls are not class-based, and therefore can't be subclassed.

Because stdio calls are buffered, consecutive, small reads and writes are much faster than using msgStreamRead or msgStreamWrite. However, if the block of information is equal to or larger than the stdio buffer size (usually 512 bytes), the speed is roughly equal to that of the class-based operations. You can change the buffer size with the setvbuf() system service.

You must include the PenPoint SDK header files STREAM.H to use the stdio functions (this is in addition to the usual STDIO.H). You can't use stdio operations to change the WorkingDir.

Translating Between Handles and FILE Pointers

If you open a file by creating a handle, but want to use stdio calls to perform reads and writes, you will need to translate the file handle into a file pointer. To do this, use the StdioStreamBind() system service, supplying the handle. The routine returns a file pointer.

This following code excerpt illustrates the conversion:

```c
FS_NEW fsNew;
FILE *fp;
s = ObjectCall(msgNewDefaults, clsFileHandle, &fsNew);
s = ObjectCall(msgNew, clsFileHandle, &fsNew);
if (s < stsOK) ...
/* Build a FILE structure based on the handle. */
fp = StdioStreamBind(fsNew.object.uid);
```
You might also need to translate a file pointer into a handle. This operation is a little easier, because the handle is stored in the uid field of the FILE struct. This following code excerpt illustrates how to translate a file pointer into a file handle.

```c
#include <stdio>
#include <stream.h>
OBJECT newHandle;
FILE *fp;
fp = fopen("MyFile","r");
if (fp == NULL) ...
   newHandle = StdioStreamToObject(fp);
```

**Paths and stdio**

When you use stdio functions that require a path to a file (such as fopen()), you specify the path to the file as you would for any file system message. The stdio functions use your task's current directory handle, theWorkingDir, for the volume and directory defaults.

Thus, if the path begins with a volume name, indicated by two backslashes (\\), the function uses the path to locate the file. If the path begins with a backslash (\), the function uses the root directory of the volume identified by theWorkingDir. If the path doesn't begin with a backslash, the function starts its traversal at theWorkingDir.

**Using stdio**

You can close files using stdio. However, don't open files by creating a handle, then close them with stdio. Because stdio doesn't know about the handle object, this will result in unreclaimed resources.

If you open a file with stdio, close it with stdio. If you create a handle on a file, destroy the handle.

**Concurrency Considerations**

PenPoint is a multitasking operating system. While your application is performing some action, it might yield the processor to another task. The other task might attempt to alter (or even delete) the file you are working on. This means that you have to program more defensively than you would for an ordinary single-tasking operating system. While you program, assume that another task might attempt to access your files or change the file's location.

**Protecting Your File Data**

While you are not accessing a file, you can use msgSetAttr to make the file read-only.

While you have a handle on a file, the file system will not allow anyone to delete the file, unless they use msgFSForceDelete.
When you create a file handle with `msgNew`, you can specify an exclusion mode so that you can limit access to the file while your handle is attached to it.

## File Location Considerations

Be aware that another task might change your file's location as well as its contents. All tasks can change (move, rename, or delete) nodes in the file system tree. A path that successfully located a node at one time might not locate the same node later. Remember that a path is not a direct handle on the node, but more like a road map to the node.

You can programmatically search for files with the message `msgFSTraverse`. Note also that you can use directory indexes to find directories, no matter what their path is.

A handle, on the other hand, will follow its node wherever the node is moved. If a task has a handle on a node, the node can't be deleted by any other task (unless it uses `msgFSForceDelete`).

## Volume Protection Considerations

Remember also that other computers might have access to volumes you are using. Another user can delete a target node by modifying a remote or local disk volume outside of the PenPoint computer's control.

When a handle's target node is deleted or destroyed, the file system marks the handle invalid. If you use an invalid handle, the file system returns `stsFSHandleInvalid`.

The only things you can do with an invalid directory handle are to free it with `msgFree` or change its target directory to a valid directory with `msgFSSetTarget`. The only valid thing you can do with an invalid file handle is to free it with `msgFree`.

## Subclassing File System Classes

In coding your application, you might find that you perform a particular set of file operations many times or you need to supplement the file system messages with your own. At this point you might consider subclassing `dsFileHandle`.

A good example of a subclass of `dsFileHandle` is `dsResFile`, the resource file class. `dsResFile` defines a number of new messages that, when sent by the user or the class manager, handle all the details of tracking all resources in the resource file. `dsResFile` maintains tables that index resources within the file. Another example is `dsAppDir`, which support the application directories used by the PenPoint Application Framework.

Most of the details about creating a subclass are described in the Part 1: The Class Manager. In short, you must define the class and make it known to the class manager.
The PENPOINT.DIR File

As mentioned before, the PenPoint file system attempts to use a volume's native file system whenever possible. When a file or directory has additional information that the native file system cannot contain, the PenPoint file system creates a PENPOINT.DIR entry for that information. The file system still relies on the native file format to carry most of the information. PENPOINT.DIR contains only the information that the native file system cannot support.

On the standard MS-DOS FAT file system, the following characteristics will cause the file system to create a PENPOINT.DIR entry:

- A node name that uses any control characters, lowercase characters, or any of these special characters: * ? / \ . , : = < > [ ] " (space).
- A name longer than eight characters, plus a three-character extension.
- A name that uses lower case characters.
- A node that has client-defined attributes or other file attributes not supported by the MS-DOS file system.

The structure of the PENPOINT.DIR file is quite simple. It consists of a series of variable-length directory entries that contain:

- Entry information, including whether the file is in use, the total size of the entry, the number of user-defined attributes, and offsets within the directory entry.
- The node flags (both PenPoint-specific flags and those duplicated from the native file system).
- The date the node was created.
- The PenPoint file name.
- The native file system file name.
- The user attributes (if any).

For further information on the PENPOINT.DIR file, see the file VOLGODIR.H.

How the Notebook Uses the File System

The organization of sections and pages in the Notebook is a direct map of the file system. Each section in the Notebook is a directory in the file system; each page in the Notebook is a directory within that directory. The names of the sections are the actual names of the directories that contain the page nodes.
Chapter 72 / Using the File System

This chapter describes how to use the file system to perform most file system operations. The section is organized in a rough “life cycle” order, that is, we present the operations in an order that approximates how you will want to use the file system to create, modify, and delete a file.

Most of the structures and typedefs described in this chapter are defined in FS.H.

Topics covered in this chapter include:

- Creating a handle and creating a new node.
- Creating a handle for an existing node.
- Deleting a node.
- Freeing a handle.
- Freeing a handle and deleting a node.
- Getting and setting file and directory attributes.

Creating Directories and Files

Before you can create or access a node, you need to create a handle object that you use to access the node. You create the handle by sending msgNewDefaults and msgNew to clsDirHandle or clsFileHandle. In the call to msgNew, you direct the file system to create the node (file or directory) if it doesn’t exist already. This is the first step in a file’s life cycle.

Creating a file handle is equivalent to opening a file in other file systems. As with most other file systems, you can specify certain open actions, such as what to do if the file or directory does or does not exist.

Creating a directory handle has no equivalent operation in other file systems.

Creating Handles

To create a handle, send msgNewDefaults and msgNew to clsDirHandle or clsFileHandle. Both messages take a FS_NEW structure that contains:

- **object.key** A key value. If you specify this, you (and any other users of the handle) must provide an equivalent key value in order to destroy the handle. (Locks are further explained in Part 1: Class Manager). If you don’t want to use the lock, specify objWKNKey.

- **fs.locator** A LOCATOR structure that indicates a directory handle and a path to the node. If you do not specify the directory handle, the default is the WorkingDir; if you do not specify a path, the default is nil.
**volType**  The type of volume. The file system uses this argument when
locator.path contains a full path. If this argument is anything other than
fsAnyVolType, it is used for “filtering” if a volume is not found. The
available volume types are:

- fsAnyVolType  Any volume type.
- fsVolTypeMemory  A RAM volume (available only on the SDK version
  of PenPoint).
- fsVolTypeDisk  A local disk volume.
- fsVolTypeRemote  A remote (network) disk volume.

**fs.dirlndex**  An optional directory index. The directory index is for directory
handles only. If you use a directory index, the locator must indicate the
volume to use.

**fs.mode**  Flags that indicate handle characteristics. There are different
options for directory and file handles.

**fs.exist**  What to do if the node does or doesn’t exist.

Use msgNewDefaults to initialize the fields to their default values, then modify
any of the fields.

Use the constants defined in FS_EXIST to specify the action to take if the node
does or does not exist.

Two of the existence flags fsExistGenUnique and fsNoExistCreateUnique work
slightly differently from each other. Ostensibly, fsExistGenUnique takes effect if
the requested node exists already; whereas fsNoExistCreateUnique takes effect
when the requested node does not exist.

At a more detailed level, fsExistGenUnique will only generate a new unique name
if that name exists already. For example, if a node named BLUE exists already, fsNew
will create a new node named BLUE 1. fsNoExistCreateUnique uses the input node
to create a unique name, whether or not the file exists already. The Notebook
uses fsNoExistCreateUnique to create unique page numbers for pages.

### Checking Valid File and Directory Names

If you get a file or directory name from the user, you probably will want to check
the validity of the name before creating a file or directory handle. You can use the
file system function FSNameValid() to test whether a name is valid. The prototype
for FSNameValid() is:

```c
STATUS EXPORTED FSNameValid (  
    P_STRING  pName
);  
```

The pName argument is a string pointer to the file or directory name to be
validated.
If the name is valid, the function returns stsOK; if the name is not valid, the function returns stsFailed.

**Creating a Directory Handle**

The next code fragment shows how to create a directory handle and a directory. First the program declares an FS_NEW structure, and uses msgNewDefaults to set the default values. The program then sets the values that it needs to specify and sends msgNew to clsDirHandle.

```c
FS_NEW fsNew;

status = ObjectCall(msgNewDefaults, clsDirHandle, &fsNew);
/* The following filled in by msgNewDefaults
   fsNew.object.key = objWKNKey;
   fsNew.object.cap = objCapCall;
   fsNew.fs.mode = fsDirNewDefaultMode;
   fsNew.fs.exist = fsExistDefault;
   fsNew.fs.locator.uid = theWorkingDir;
*/

fsNew.fs.locator.pPath = "MyDir";
status = ObjectCall(msgNew, clsDirHandle, &fsNew);
if (status < stsOK) {
   Debugf ("Error creating dir = %lx", status);
}
```

When the file system creates the directory handle, it sends back the UID for the handle in object.uid of the FS_NEW structure. When you need to send other messages to the handle (such as msgFSGetAttr or msgFree), you can use this UID.

Use the constants defined in the FS_DIR_NEW_MODE typedef to specify the mode flags for directories. The mode flags indicate the directory characteristics. Table 72-1 describes the mode flags for directories.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning If Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsTempDir</td>
<td>The file system should delete the directory when the handle is destroyed.</td>
</tr>
<tr>
<td>fsUnchangeable</td>
<td>Disallow changing the target directory.</td>
</tr>
<tr>
<td>fsUseDirIndex</td>
<td>Find the directory using the directory index specified in the dirIndex field. Use the locator's UID and path to determine which volume to use.</td>
</tr>
<tr>
<td>fsSystemDir</td>
<td>Directory handle is owned by the system (ring 0).</td>
</tr>
</tbody>
</table>

You can use the constant fsDirNewDefaultMode, which is the same as specifying a permanent and changeable directory (all flags clear).

To get or set the directory mode flags, use msgFSGetHandleMode and msgFSSetHandleMode.

**Creating a File Handle**

The next code fragment shows how to create a file handle and a new file. First the program declares an FS_NEW structure and initializes it with msgNewDefaults.
The program then specifies the value that it needs to change and sends msgNew to clsFileHandle.

```c
STATUS s;
FS_NEW fsNew;
FILE_HANDLE myFileHandle;

s = ObjectCall(msgNewDefaults, clsFileHandle, &fsNew);
/*
The following filled in by msgNewDefaults
fsNew.object.key = objWKNKey;
fsNew.object.cap = objCapCall;
fsNew.object.uid = null;
fsNew.fs.locator.uid = theWorkingDir;
fsNew.fs.mode = fsFileNewDefaultMode;
fsNew.fs.exist = fsExistDefault;
*/
fsNew.fs.locator.pPath = "MyDir\MyFile";
status = ObjectCall(msgNew, clsFileHandle, &fsNew);
myFileHandle = fsNew.object.uid;
```

Again, when the file system creates the file handle, it sends back the UID for the handle in object.uid of the FS_NEW structure. In this example, the program saves the handle in the variable myFileHandle. Now when the program needs to send other messages to the handle (such as msgStreamRead or msgFree), it can use myFileHandle.

Use the constants defined in FS_FILE_NEW_MODE to specify the mode flags for files. The mode flags indicate how the file system is to open the file. The mode includes your access intentions, your exclusivity requirements, and whether memory mapped regions should be in shared memory. Access intentions describe how you intend to access the file (reading only, or writing and reading). Exclusivity requirements describe what you will let other clients do to the file while you have a handle on it (allow other readers and writers, allow readers only, or deny access to all). If a memory mapped file region is shared, more than one client can access the memory mapped file.

Table 72-2 describes the mode flags for files.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning if Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsTempFile</td>
<td>Delete the file when the handle is destroyed.</td>
</tr>
<tr>
<td>fsReadOnly</td>
<td>Open the file with read-only access.</td>
</tr>
<tr>
<td>fsSystemFile</td>
<td>Directory handle is owned by the system (ring 0).</td>
</tr>
<tr>
<td>fsSharedMemoryMap</td>
<td>Shared memory used for memory mapped files.</td>
</tr>
<tr>
<td>fsDisablePrompts</td>
<td>Do not prompt the user if the volume containing the file is disconnected. Always return stsFSVolDisconnected.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Meaning if Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsNoExclusivity</td>
<td>No exclusive access.</td>
</tr>
<tr>
<td>fsDenyWriters</td>
<td>Deny access to readers.</td>
</tr>
<tr>
<td>fsExclusiveOnly</td>
<td>Handle owner has exclusive access to the file.</td>
</tr>
</tbody>
</table>
You can use the constant `fsFileNewDefaultMode` to use the file open defaults. The constant is the same as specifying `fsNoExclusivity` (the file is permanent with read/write access).

To get or set the file mode flags, use `msgFSGetHandleMode` and `msgFSSetHandleMode`.

### Mapping a File to Memory

Memory mapped files allow you to address information in a file as if its contents were in main memory. In PenPoint 1.0, you map files to memory by establishing a block of virtual memory to which the file system can swap the file contents.

The message `msgFSMemoryMapSetSize` specifies the amount of virtual memory available to a memory mapped file. You cannot specify a size of zero, less than the file size, or less than the size set by any other client. The memory map size can be larger than its previous size. The memory map size must be set before memory mapping the file.

The memory map size should be set to a reasonable expected maximum. If your file is static, then set the memory map size to the file size. If your file will grow then set the limit to its anticipated size. Setting a file's memory map size to 1MB does not take 1MB of RAM, but does require approximately 1KB of data structures to support in the memory manager and uses 1MB of the virtual memory address space. The minimum memory required is 4KB per file, so memory mapping very small files is not efficient.

If a file is memory mapped, then the memory map size can't change (use `msgFSMemoryMapSetSize` before `msgFSMemoryMap`) and the file size can't grow (via either `msgStreamWrite` or `msgFSSetSize`). All of these error cases return `stsFSNodeBusy`. The single pointer returned by `msgFSMemoryMap` can be used to address the entire memory map. A zero length file can be memory mapped.

### Sharing Memory Mapped Files

By default the memory mapped region is in local memory. If you want to share a memory mapped file with other clients, you must specify `fsSharedMemoryMap` in the `mode` argument of the `FS_FILE_NEW_MODE` structure when you create the file handle.

### Memory Mapped File Life Cycle

This section presents the life cycle of a memory mapped file. Clients need to be particularly aware of the second step.

1. **Open the file.**
   
   Some memory map related options are specified in `pNew->fs.mode` when a file handle is created. Setting `fsReadOnly` will result in a read only memory map. Setting `fsSharedMemoryMap` will result in a memory map allocated from shared memory. Setting `fsSystemFile` will result in a memory map owned by the system (this flag is only accessible to supervisor code).
Set the maximum size of the memory mapped file.
You specify the memory map size using msgFSMemoryMapSetSize. This message takes a single argument, the memory map size. Decide on the maximum size that your memory map will grow to and set that size. The memory map size must be as large or larger than the actual file size. You may want to set the memory map size based on the maximum of msgFSGetSize and the desired memory map size. Setting unnecessarily large sizes will quickly use up all of virtual memory space.

Also be aware that each 4KB of virtual memory map space still requires approximately 16 bytes of real memory in the memory manager. If your memory mapped file is read-only or static (for example fonts or a dictionary) set the memory map size to the file size. The message msgFSMemoryMapGetSize can be used to query the current memory map size.

Memory map the file.
msgFSMemoryMap to the file handle will return a pointer to the memory map. The pointer will always point to the base of the memory mapped region. Sending msgFSMemoryMap to a memory mapped file handle will return the same pointer, not another one. There is only one memory map per file handle.

Flush the memory map (optional).
msgFSFlush to the file handle of the memory mapped file will cause all dirty portions of the memory mapped file to be written to disk.

Free the memory map.
msgFSMemoryMapFree frees the memory map.

Set the file size (optional).
If you want your file to be the size of the memory mapped data structures that you have mapped onto the file, then you need to explicitly set the file size before freeing the file handle. If you do not, then the file will be a multiple of the system page size, and the undefined bytes past the end of the memory mapped file will become part of the file on disk.

Close the file.
msgDestroy to the file handle will close the file handle. All dirty pages in the memory map will be written to disk. The memory map will be freed if you did not free it in step 5.

Closing Files

When you have finished with a file or directory, you should free the handle to deallocate the memory required by the handle. This is equivalent to closing a file.

To free a handle, send the message msgDestroy to the handle that you want to free. The only argument to msgDestroy is the key that was used to create the handle, if any. The following fragment illustrates the use of msgDestroy:
Forcing Deletion of a File or Directory

myFileHandle = fsNew.object.uid;

status = ObjectCall(msgDestroy, myFileHandle, (P_ARGS) objWKNKey);

When the file system created the handle, it returned the UID in fsNew.object.uid. Here the program calls msgDestroy to free that handle. The key value specified here uses objWKNKey, the well-known key, which has a value of 0. Use objWKNKey when you didn’t specify a key value in msgNew. For more information about the keys, see the Part 1: The Class Manager.

Sending msgDestroy to a directory or file handle does not delete the node (unless the file is marked temporary); it merely has the effect of closing the file and freeing the resources used by the handle object. The handle is not the node.

Deleting Files and Directories

Delete a file or directory by sending msgFSDelete to a file or directory handle.

The only argument to the message is a path that specifies the node to delete. If the path is empty, the file system deletes the file or directory handle’s target node. Deleting a directory hierarchically deletes all of the nodes in that directory.

This example illustrates msgFSDelete:

status = ObjectCall(msgFSDelete, theWorkingDir, "MyDir\MyFile")

If you use the temporary file flag with msgNew, the file system deletes the file or directory when you free the file or directory handle.

You can’t delete a file that is marked read-only; you must change the file’s attribute to read/write before you can delete it. If you attempt to delete a node that is the target of another directory handle, the deletion will fail with stsFSNodeBusy.

Forcing Deletion of a File or Directory

To force the deletion of a node (file or directory) that is marked read-only, or that is the target of another directory handle, send msgFSForceDelete to a directory handle.

msgFSForceDelete is a powerful message. It will delete any file or directory without question. Careless use of msgFSForceDelete could result in damage to your installed software, including PenPoint.

msgFSForceDelete takes a pointer to a FS_FORCE_DELETE structure that contains a pointer to a path that indicates the node to delete (pPath).

When the message completes successfully, it returns stsOK.

When a handle’s target node is deleted or destroyed, the file system marks the handle invalid. If you use an invalid handle, the file system returns stsFSHandleInvalid.

The only valid thing that you can do with an invalid file handle is to free it with msgFree.
The only valid things that you can do with an invalid directory handle are to free it with msgFree or change its target directory with msgFSSetTarget.

### Getting and Setting Attributes

You can use the msgFSGetAttr and msgFSSetAttr messages to manipulate file and directory attributes.

When you send an attribute message to a file handle, the message affects that file directly. When you send an attribute message to a directory handle, you can specify a path to any node. There are three types of attribute values:

- Fixed-size values (32 or 64 bits).
- Variable-sized values (up to a little less than 64K).
- Null-terminated strings (up to a little less than 64K characters).

### Lists of Attributes

msgFSGetAttr and msgFSSetAttr both take a FS_GET_SET_ATTR structure that contains pointers to three parallel arrays:

- An array of 32-bit attribute labels (identifiers for particular attributes).
- An array of 32-bit (4-byte) or 64-bit (8-byte) attribute values or pointers to variable-length or string attributes.
- An optional array of attribute sizes (in bytes).
The arrays must contain the same number of elements.

If an attribute is fixed-length, its value is stored in the attribute array; however, if the attribute is variable length, the attribute array contains a pointer to the attribute's buffer. Figure 72-1 shows the FS_GET_SET_ATTR structure for four attributes. The first, third and fourth attributes are fixed-length values. The second attribute is a string. The fourth value is a fixed-length 64-bit value, so its entry in the pAttrValues array uses eight, rather than four, bytes.

**Zero Value Attributes**

When an attribute's value is 0, the attribute is deleted from the node's attribute list. You can still get the attribute value, because the file system always passes back the value 0 for an "undefined" attribute.

This saves memory in attribute lists and in the PENPOINT.DIR files, but at the expense of some confusion. You cannot determine if an attribute with a zero value is defined just by asking for its value. What you can do is request all attributes for a file system node, and then examine the attribute label array for presence or absence of the attribute that you want.

**File System Attributes**

The file system defines the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsAttrName</td>
<td>Node name.</td>
</tr>
<tr>
<td>fsAttrFlags</td>
<td>Node attribute flags. For more information, see the discussion below.</td>
</tr>
<tr>
<td>fsAttrDateCreated</td>
<td>Node creation date/time</td>
</tr>
<tr>
<td>fsAttrDateModified</td>
<td>Last modified date/time.</td>
</tr>
<tr>
<td>fsAttrFileSize</td>
<td>Number of bytes in the file.</td>
</tr>
<tr>
<td>fsAttrDirIndex</td>
<td>A directory index value (directories only).</td>
</tr>
<tr>
<td>fsAttrFileType</td>
<td>A file type TAG, as defined in FILETYPE.H.</td>
</tr>
</tbody>
</table>

The PenPoint file system does not maintain an "archive" attribute that records whether the file has been modified.

**Client-Defined Attributes**

The set of attributes the file system defines is fixed, but you can create any number of client-defined attributes.

The file system header file (FS.H) defines three macros to create attribute labels.

The macros are:

```c
FSMakeFixAttr(class, tag)
FSMakeVarAttr(class, tag)
FSMakeStrAttr(class, tag)
```
where \textit{class} is the class to which the attribute belongs (for example, file system attributes are defined by \texttt{clsFileSystem}) and \textit{tag} is a unique value for the attribute.

In many ways client-defined attributes are no different from file system attributes. You can perform the same operations on client-defined attributes (get size, get value, set value). However, when you send \texttt{msgFSSetAttr} and indicate a previously undefined attribute, the file system creates a client-defined attribute.

When you specify the value 0 (or a null pointer) for a client-defined attribute, the file system deletes the attribute. Note that you can still get the attribute value; the file system will return 0 (which is both the value of the attribute and the indication that the attribute does not exist).

### Getting Attribute Values

To get attribute values, send the message \texttt{msgFSGetAttr} to a file or directory handle. The message takes a pointer to a \texttt{FS\_GET\_SET\_ATTR} structure that contains:

- \texttt{pPath} A pointer to a path. This path, combined with the handle to which the message is sent, forms an implicit locator.
- \texttt{numAttrs} The number of attributes you are requesting.
- \texttt{pAttrLabels} A pointer to the array of attribute labels.
- \texttt{pAttrValues} A pointer to the array of 32-bit values that either receive the fixed length attributes or point to the buffers that receive the variable-length attributes. You must specify the pointers in the array when you call \texttt{msgFSGetAttr}, the message will not return the pointers for you.
- \texttt{pAttrSizes} A pointer to the array that receives attribute sizes. If you don't want to receive attribute sizes, use \texttt{ Nil(FS\_ATTR\_SIZE)} for this pointer.

If the attribute does not exist, the value for that attribute is 0.

If you don't want to allocate the variable-length attribute value and size buffers ahead of time, you can direct \texttt{msgFSGetAttr} to allocate the buffers for you. To automatically allocate the value buffer, specify \texttt{fsAllocAttrValuesBuffer} in \texttt{pAttrValues}; to allocate the sizes buffer, specify \texttt{fsAllocAttrSizesBuffer} in \texttt{pAttrSizes}.

You can direct \texttt{msgFSGetAttr} to return all attributes by specifying the constant \texttt{fsAllocAttrLabelsBuffer} in \texttt{pAttrLabels}. If you specify this value, specify \texttt{maxU16} for \texttt{numAttrs}, and use \texttt{fsAllocAttrValuesBuffer} and \texttt{fsAllocAttrSizesBuffer} in \texttt{pAttrValues} and \texttt{pAttrSizes}. The actual number of attributes is returned in \texttt{numAttrs}.

When you are done with the attributes, you must free the returned memory regions with \texttt{OSHeapBlockFree()}; PenPoint cannot do it for you. \texttt{OSHeapBlockFree()} is documented in the \textit{Part 8: System Services}. 
Setting Attribute Values

Setting attribute values is similar to getting attribute values. Send the message `msgFSSetAttr` to a file or directory handle. The message takes a pointer to a `FS_GET_SET_ATTR` structure that contains:

- `pPath` A pointer to a path, if sending the message to a directory handle.
- `numAttrs` The number of attributes you are setting.
- `pAttrLabels` A pointer to the array of attribute labels.
- `pAttrValues` A pointer to the array of 32-bit values or pointers to variable-length attributes.
- `pAttrSizes` A pointer to the array that specifies the attribute sizes. This is required for variable-length attributes; it is optional for fixed-length and string attributes.

The list of attribute labels and the list of pointers must contain the same number of elements.

Getting the Length of Attribute Values

If you need to get a fixed-length attribute, you usually don't have to worry about allocating space for it, whether you allocate space at compilation or dynamically. However, when you are dealing with variable-length attributes and need to be cautious about memory consumption. You can use `msgFSGetAttr` to obtain only the length of an attribute.

The message attributes are similar to those for `msgFSGetAttr`, but you specify null in the arrays indicated by `pAttrValues`.

Node Attribute Flags

The node attribute flags define information that is common to all file system nodes (such as read-only, hidden, and whether it is a directory or file node). To get the node attribute flags, you must create a `FS_NODE_FLAGS_ATTR` structure, which contains a `FS_NODE_FLAGS` structure and a mask.

To get attribute flags, you use the `FS_NODE_FLAGS` structure to indicate the node attribute flags that you want. The file system ignores the mask on input. When `msgFSGetAttr` sends back the structure, the node flags contains `0xFFFF`.

To set attribute flags, you use both the `FS_NODE_FLAGS` structure and the mask. The mask specifies which node attribute flags you want to change. The mask enables you to alter selected node attribute flags without having to get all the node attribute flags first.
Table 72-4 summarizes the attribute flags defined by `FS_NODE_FLAGS`.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fsNodeReadOnly</code></td>
<td>The node is read-only (applies to files only).</td>
</tr>
<tr>
<td><code>fsNodeHidden</code></td>
<td>The node is hidden.</td>
</tr>
<tr>
<td><code>fsNodeDir</code></td>
<td>The node is a directory. Clients cannot change this attribute.</td>
</tr>
<tr>
<td><code>fsNodeGoFormat</code></td>
<td>The node has additional GO information (such as a long name or client-defined attributes). Clients cannot change this attribute.</td>
</tr>
<tr>
<td><code>fsNodePenPointHidden</code></td>
<td>Node is hidden from user in disk browsers.</td>
</tr>
</tbody>
</table>

### Creating and Using Directory Indexes

A directory index is a fast way of specifying a directory, that is independent of paths. The Notebook uses directory indexes to implement GoTo buttons and to jump to section tabs.

To create a directory index, send `msgFSSetAttr` to a directory handle. Specify `fsAttrDirIndex` as the label. The attribute is a variable-length attribute that contains a 64-bit unique identifier (UUID). You must create the UUID by calling the function `MakeDynUUID()`. `MakeDynUUID()` is described in *Part 2: Application Framework*.

You can use the directory index in a `msgNew` rather than a locator. When you send `msgNew` to `clsDirHandle`, specify the `mode` flag `fsUserDirIndex` and put the directory index in `dirIndex`. The `locator.uid` must specify the volume root.

### Copying and Moving Nodes

You copy nodes by sending `msgFSCopy` to a file or directory handle; you move nodes by sending `msgFSMove` to a file or directory handle. Both messages take a `FS_MOVE_COPY` structure that contains:

- `pSourcePath` A path to the source node.
- `destLocator` A locator that indicates the destination for the new node.
- `mode` A set of flags that specify the type of move or copy operation.
  
  Currently there are three flags defined for `mode`:
  
  - `msMoveCopyIntoDest` Used when copying or moving directories. If you specify this flag, the directory node is moved or copied into the location specified by the locator. If you do not specify this flag, the directory node becomes the location specified by the locator.
  - `fsMoveCopyVerifyOnly` Verifies that the move or copy would succeed if performed, but does not do the actual move or copy.
  - `exist` What to do if the destination node exists already.
- `pNewDestName` Passes back the name of the new node. If you specify NULL for this address, the buffer is not written. This parameter is necessary only if you specify `fsExistGenUnique`, so that you can receive the new name.
The following example copies the file `MYDIR\MYFILE` to `MYDIR\MYCOPY` on the same volume. If the destination file already exists, the exist flag specifies that the file system should overwrite it.

```plaintext
FS_MOVE_COPY dhCopy;
dhCopy.pSourcePath = "MyDir\MyFile";
dhCopy.destLocator.uid = theWorkingDir;
dhCopy.destLocator.pPath = "MyDir\MyCopy";
dhCopy.intoDest = FALSE;
dhCopy.pNewDestName = Nil(P_STRING);
dhCopy.exist = fsCopyExistOverwrite;
status = ObjectCall (msgFSCopy, theWorkingDir, &dhCopy);
```

If you move a node that has other handles it, those handles are updated to reflect the new location of the node.

**Traversing Nodes**

At times, you might need to examine attributes for all nodes that are subordinate to a particular node. For example, you might want to sum the sizes of all nodes subordinate to a directory. To traverse the nodes of a tree, send `msgFSTraverse` to a directory handle, specifying an attribute list (similar to `msgFSGetAttr` or `msgFSReadDir`).

`msgFSTraverse` will traverse all directory entries subordinate to the directory that received the message. For each directory entry that it encounters, the message retrieves the specified attributes and can invoke a call-back routine, wherein you can examine the attributes. A quicksort comparison routine allows you to sort the directory entries for each directory before invoking the call-back routine.

`msgFSTraverse` takes a pointer to an `FS_TRAVERSE` structure that contains:

- **mode** The call-back mode, defined in `FS_TRAVERSE_MODE`. The mode is explained in "The Traverse Call-Back Routine," below.
- **numAttrs** The number of attributes you are requesting.
- **pAttrLabels** An array of `numAttrs` attribute labels. The array must include `fsAttrFlags` and `fsAttrName` at a minimum.
- **pCallBackRtn** The name of a call-back routine. For more information on the call-back routine, see "The Traverse Call-Back Routine," below.
- **pClientData** A pointer to an area of data that can be read and written by the call-back routine.
- **pQuickSortRtn** The name of an optional quicksort routine. If you don’t want the directory sorted, specify `Nil(P_UNKNOWN)`. For more information on the quicksort routine, see "The Traverse Quicksort Routine," below.

When traversing a directory handle, `msgFSTraverse` modifies the target of the handle as it visits each subdirectory. This causes a problem when you want to traverse an entire volume, because root directory handles cannot be modified. This means that `msgFSTraverse` will return `stsFSUnchangeable` if you send it to a root directory handle.
To traverse a volume starting at the root directory node, you must create a
directory handle whose target node is a copy of the volume's root directory. Send
msgFSTraverse to the new handle and it will traverse the entire volume. Be sure to
destroy the new handle when you no longer need it.

**The Traverse Call-Back Routine**

Along with your arguments to msgFSTraverse, you name a call-back routine. The
call-back routine is a user-written routine that acts on attributes found by
msgFSTraverse.

When msgFSTraverse encounters a file node, enters a directory node, or exits a
directory node, it can invoke your call-back routine. The *mode* argument to
msgFSTraverse specifies when it should invoke the call-back routine.

The call-back routine receives these arguments, which are defined in
P_FS_TRAVERSE_CALL_BACK:

- **dir** If the node is a directory, the directory handle to the node. If the node is
  a file, the directory handle to the file's parent directory. The call-back
  routine can use this directory handle along with the file name attribute
  *(fsAttrName)* to open a file.
- **level** The current level in the directory hierarchy, relative to starting
  directory for traversal.
- **pNextEntry** A pointer to a FS_READ_DIR structure that contains the
  requested attributes. See the discussion of msgFSReadDir for more
  information on FS_READ_DIR.
- **pClientData** A pointer to the client data.

You can use this area to send data to the call-back routine and to receive
information gathered by the call-back routine.

**The Traverse Quicksort Routine**

If you specify the name of a quicksort comparison routine, the message sorts each
directory according to the quicksort comparison routine before invoking the
callback routine. The comparison routine must compare two values, and return
-1, 0, or 1, depending on the result of the comparison.

**Order of Traversal**

When searching the directories, the traversal starts with the directory that received
the message. The traversal examines each directory entry in order.

When it encounters an entry for another directory node, the traversal moves to
that directory. The traversal continues recursively until it finishes with the
directory entries in a node. It then returns to the previous directory in the traversal
and continues.
Renaming Nodes

You rename nodes by using msgFSSetAttr to change the node name.

Determining the Existence of a Node

To test whether a file or directory node exists, send msgFSNodeExists to a directory or file handle. The message takes a pointer to an FS_NODE_EXISTS structure that contains a path that specifies the node that might exist (pPath). The path can be null, but that would imply that you are sending the message to a handle on a node whose existence you aren’t sure. If you can send messages to the handle, the node probably exists.

When the message completes, it returns stsOK if the file or directory exits, or stsFSNodeNotFound if it doesn’t exist. The message sends back the FS_NODE_EXISTS structure with a BOOLEAN value that contains true if the node is a directory and false if the node is a file (isDir).

Reading and Writing Files

This section discusses how to read and write files using file handles.

When you have a file handle, you can use msgStreamRead and msgStreamWrite to read and write data. For both messages take a STREAM_READ_WRITE structure that contains:

- numBytes: Number of bytes to read or write. This number can be as large as fsMaxReadWrite (defined in FS.H as 0x40000000).
- pBuf: A pointer to a buffer of data to read or write.

When msgStreamRead or msgStreamWrite complete successfully, they send back the actual number of bytes read or written in count.

This example illustrates the use of msgStreamRead and msgStreamWrite.

```c
STREAM_READ_WRITE srw;
char outbuf[80];
status = ObjectCall(msgNewDefaults, clsFileHandle, &fsNew);
fsNew.fs.locator.pPath = "MyDir\RWFile";
if ((status = ObjectCall(msgNew, clsFileHandle, &fsNew)) < stsOK)
    FSErr("****FileHandle msgNew 1. Object Open failed", status);
fh = fsNew.object.uid;
/* Everyone's typical data... */
srw.pBuf =
    "This is the text to write to the file.\r\n";
srw.numBytes = strlen(srw.pBuf);
if ((status = ObjectCall(msgStreamWrite, fh, &srw)) < stsOK)
    FSErr("****FileHandle msgNew 1. Write failed", status);
srw.numBytes = 80;
srw.pBuf = (P_U8) outbuf;
if ((status = ObjectCall(msgStreamRead, fh, &srw)) < stsOK)
    FSErr("****FileHandle msgNew 1. Read failed", status);
```
File Position and Size

As with other file systems, you can ask the file system for the current position within a file or you can set it to a new location. You can also find or change the size of a file.

Getting and Setting File Position

To get and set the current file position, use the message msgFSSeek. The message takes a pointer to a FS_SEEK structure that contains:

- **mode**  The starting position for the seek. You can start a seek at the beginning of a file, the end of a file, or at the current byte position. The starting position symbols are defined in FS_SEEK_MODE (shown in Table 72-5, below).
- **offset**  The offset in bytes. This offset is a signed value. Positive offsets move the current byte position closer to the end of file; negative offsets move it closer to the beginning of the file.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning if Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsSeekBeginning</td>
<td>Seek is relative to the beginning of the file.</td>
</tr>
<tr>
<td>fsSeekEnd</td>
<td>Seek is relative to the end of the file.</td>
</tr>
<tr>
<td>fsSeekCurrent</td>
<td>Seek is relative to the current position of the file.</td>
</tr>
<tr>
<td>fsSeekDefaultMode</td>
<td>The default mode is fsSeekBeginning.</td>
</tr>
</tbody>
</table>

msgFSSeek sends back the current byte position, the old position, and indicates whether you are at the end-of-file marker.

To find out the current byte position, specify 0 as the offset value, relative to the current position. The following example shows a msgFSSeek call that sends back the current position.

```c
FS_SEEK fhSeek;
fhSeek.offset = 0;
fhSeek.mode = fsSeekCurrent;
status = ObjectCall(msgFSSeek, fh, &fhSeek);
Debugf("Seek: Old Position: %ld, New Position: %ld %s",
      fhSeek.oldPos, fsSeek.curPos, fsSeek.eof ? "(EOF)" : "");
```

The following example shows a call to msgFSSeek that sets the current byte position 80 bytes after the beginning of the file.

```c
FS_SEEK fhSeek;
fhSeek.offset = 80;
fhSeek.mode = fsSeekBeginning;
status = ObjectCall(msgFSSeek, fh, &fhSeek);
Debugf("Seek: Old Position: %ld, New Position: %ld %s",
      fhSeek.oldPos, fsSeek.curPos, fsSeek.eof ? "(EOF)" : "");
```

You can’t set the current byte position before the beginning of the file or beyond the end-of-file. If you seek from the current position and specify a byte offset that
is before the beginning of the file, the new position is the beginning of the file; if the offset is after the end, the new position will be end-of-file. However, if you specify seek relative to the beginning of the file and pass a negative byte offset, or you specify seek relative to the end-of-file end and pass a positive byte offset, msgFSSeek returns stsBadParam.

### Getting and Setting File Size

Use msgFSGetSize to get the file size and msgFSSetSize to set it. msgFSGetSize takes a pointer to a FS_FILE_SIZE value; when the message completes successfully, it stores the size in that location. msgFSSetSize takes a pointer to a FS_SET_SIZE structure that contains the new size of the file (newSize); when the message completes successfully, it sends back the previous size of the file (oldSize).

This example shows calls to msgFSGetSize and msgFSSetSize.

```c
#define MIN_SIZE 2048
FS_FILE_SIZE gs;
FS_SET_SIZE ss;
status = ObjectCall(msgFSGetSize, fh, &gs);
if (gs < MIN_SIZE)
{
    ss.newSize = MIN_SIZE;
    status = ObjectCall(msgFSSetSize, fh, &ss);
}
```

If you set the file size to 0, you can delete information in the file without deleting the file itself.

### Flushing Buffers

You flush buffered data and node attributes with msgFSFlush. You send msgFSFlush to the handle of the file you want to flush. This message does not take any arguments.

You might want to use msgFSFlush to be sure that buffered data is written to a file system buffer before another process writes to the same file system buffer. This example shows a call to msgFSFlush:

```c
status = ObjectCall(msgFSFlush, fh, (P_ARGS) null);
```

Sending msgFSFlush to the root directory of a volume flushes data and attributes for every file on the volume that has buffered information.

### Getting the Path of a Handle

If you need to know the path currently used by a file or directory handle, use msgFSGetPath. The message takes a pointer to a FS_GET_PATH structure that contains:

- **mode** What the returned path is relative to. The possible values are:
  - fsGetPathAbsolute Relative to the volume (the returned path begins with two backslashes (\)).
fsGetPathRoot Relative to the root directory (the returned path begins with a backslash (\)).
fsGetPathRelative Relative to a specified directory (see dir, below).
fsGetPathName The node name only.
dir If you specified fsGetPathRelative for mode, you must specify a directory handle.
bufLength The length of your return buffer.
pPathBuf A pointer to the buffer that receives the path.

This example shows an example of msgFSGetPath:

```
FS_GET_PATH fsGetPath;
char *pPath[fsMaxPathLength];
fsGetPath.mode = fsGetPathRoot;
fsGetPath.dir = objNull;
fsGetPath.pPathBuf = pPath;
fsGetPath.bufLength = fsMaxPathLength;
status = ObjectCall(msgFSGetPath, dh, &fsGetPath);
```

## Changing the Target Directory

You can change the target directory for a directory handle by sending the message msgFSSetTarget to any directory handle. This message takes a pointer to an FS_LOCATOR structure that specifies:
uid A directory handle.
pPath The path to the new target directory.

This example illustrates a call to msgFSSetTarget.

```
FS_LOCATOR loc;
loc.uid = theWorkingDir;
loc.pPath = "MyDir\SubDir";
status = ObjectCall(msgFSSetTarget, dh, &loc);
```

You can also use msgFSSetTarget to change the target of the well-known handle, theWorkingDir.

Remember that the unchangeable attribute might prevent you from changing the target on other well-known or global handles. If you attempt to change a target to a file node, rather than a directory node, msgFSSetTarget returns stsFSNotDir.

## Comparing Handles

Use msgFSSame to find out whether two handles reference the same node. You call msgFSSame by sending it to a handle; the only message argument is another handle. If the two handles reference the same node, msgFSSame returns stsOK; if different, it returns stsFSDifferent.
This example illustrates the use of `msgFSSame`.

```c
FS_LOCATOR loc;
DIR_HANDLE dhl;
... loc.uid = theWorkingDir; loc.pPath = "MyDir\MyFile1";
status = ObjectCall(msgFSSetTarget, dhl, &loc);
... status = ObjectCall(msgFSSame, theWorkingDir, dhl);
```

### Getting and Setting Handle Mode Flags

When you create a directory or file handle, you specify mode flags that indicate options for the different handles. If, at a later time, you want to get or re-set the options for a handle, you can use `msgFSGetHandleMode` and `msgFSSetHandleMode`.

`msgFSGetHandleMode` takes a pointer to a U16 value that will receive the mode flags. When you send the message to a file handle, it interprets the pointer as a pointer to a `FS_FILE_NEW_MODE` value; when you send the message to a directory handle, it interprets the pointer as a pointer to a `FS_DIR_NEW_MODE` value.

The mode flags for directory handles are described in Table 72-1; the mode flags for file handles are described in Table 72-2.

`msgFSSetHandleMode` takes a pointer to a `FS_SETHANDLE_MODE` structure, which contains:

- **mode** A `FS_FILE_NEW_MODE` or `FS_DIR_NEW_MODE` value. The type of value depends on whether you send the message to a file or directory handle (both types are actually U16s).
- **mask** A mask value.

The mask specifies which mode flags you want to change. The mask enables you to alter selected mode flags without having to get all the mode flags first.

`msgFSSetHandleMode` can change only a few of the handle mode fields (fsTempDir or fsTempFile and fsSharedMemoryMap, for example). Very few, if any, network file systems allow you to change the read-write access or exclusivity of a file once you have opened it.

### Reading Directory Entries

Use `msgFSReadDir` to sequentially read selected attributes from all entries in a directory. Each time you send `msgFSReadDir`, the file system sends back the specified attributes, then updates the directory's current directory position. The current directory position is initially set to the beginning of the directory entries, and is advanced each time you send `msgFSReadDir`.

The arguments to `msgFSReadDir` are similar to `msgFSGetAttr`. You define three parallel arrays that contain attribute labels, locations or pointers, and sizes.
msgFSReadDir takes a pointer to a FS_READ_DIR structure that contains:

- **numAttrs** The number of attributes that you want.
- **pAttrLabels** An array of attribute labels, numAttrs long.
- **pAttrValues** An array of P_UNKNOWN, used to store both 32-bit values and
  pointers to buffers that receive variable-length attributes. You must
  specify the pointer values when you send msgFSReadDir.
- **pAttrSizes** An array of FS_ATTR_SIZE, used to store the length of the
  variable-length attributes. If you don’t want the message to return the
  sizes, specify Nil(p_FS_ATTR_SIZE) for the pointer.

After you read the last entry, any subsequent msgFSReadDir messages return
stsEndOfData. You can move the current directory pointer back to the beginning
of the directory with msgFSReadDirReset.

When you change a target node (with msgFSSetTarget), the current directory
position is set to 0.

**Reading All Directory Entries**

You use msgFSReadDirFull to get a copy all directory entries. You tell PenPoint
which directory you want and the attributes you are interested in. PenPoint
allocates a heap block, creates a linked list of blocks containing the attributes you
requested, and sends back a pointer to the first block (in pNext) and the number
of blocks in the list (numEntries).

Each block contains a pointer to next block (NULL for the last block) and a series
of pointers to the requested attribute values. The values are part of the returned
heap block.

When you are done with the attributes, you must free the memory block with
OSHeapBlockFree(); PenPoint cannot do it for you. OSHepBlockFree() is
documented in the Part 8: System Services.

The current directory position used by msgFSReadDir is not affected by
msgFSReadDirFull.

**Sorting Directory Entries**

One reason to use msgFSReadDirFull is to sort a series of directory entries on a
particular attribute. To sort the returned directory entries, use the PenPoint
quicksort routine.

quicksort sorts a linked list of variable length blocks (which is the format of
attributes returned by msgFSReadDirFull). quicksort takes a pointer to the list
and a user-supplied routine for comparing two blocks. It rearranges the sequence
by readdressing the links. quicksort is described in detail in the API Reference
This example uses quicksort to sort directory entries. It was drawn from the PenPoint application framework.

```c
/* Comparison routine for sorting directory entries by sequence number. */
int cdecl AppDirCompSeq ({
P_FS_READ_DIR p,
P_FS_READ_DIR q)
{ 
  U32 a;
  U32 b;
  a = ((P_APP_DIR_NEXT)(p->attrValue.pValues))->attrs.sequence;
  b = ((P_APP_DIR_NEXT)(q->attrValue.pValues))->attrs.sequence;
  if (a < b) return -1;
  else if (a > b) return 1;
  else return 0;
}

/* Read entire directory. */
readDirFull.numAttrs = appDirNumReadDirFullAttrs;
readDirFull.attrLabel.pLabels = appDirAttrLabels;
ObjCallRet(msgFSReadDirFull, dir, &readDirFull, status);
pReadDir = pArgs->handle = readDirFull.pDirBuf;
/* Sort directory snapshot by sequence. */
pReadDir = quicksort(pReadDir, AppDirCompSeq);

/* Deallocate attributes’ segment. */
status = OSHeapBlockFree(pReadDir);
```

### Observing Changes

The file system allows you to make yourself an observer of the FileSystem or a directory. When a volume is added or removed from the FileSystem or when nodes are added or removed from a directory, observers receive msgFSChanged. The message carries an FS_CHANGE_INFO structure that contains:

- **observed** The UID of the observed object that changed.
- **reason** The message that caused the change.

### Making a Node Native

At some point you might need ensure that a node will be fully compatible in a non-PenPoint environment. This means removing all PenPoint-specific information from a file, such as the long file names, PenPoint-specific attributes, and client-specified attributes. To remove the PenPoint-related information, use msgFSMakeNative.
msgFSMakeNative takes a pointer to an FS_MAKE_NATIVE structure that contains:

- **pPath** A path to the node.
- **newName** A pointer to the buffer that will hold the new node name.

This example shows a call to `msgFSMakeNative`.

```c
FS_MAKE_NATIVE make_native;
char nameArray[13];
make_native.pPath = "File with a long name";
make_native.newName = nameArray;
status = ObjectCall(msgFSMakeNative, theWorkingDir, &make_native);
```

## Getting Volume Information

You can get information about a specific volume by sending `msgFSGetVolMetrics` to a volume object. To get a list of volume objects send `msgFSGetInstalledVolumes` to `theFileSystem`.

The message takes a pointer to an FS_GET_VOL_METRICS structure that contains:

- **updateInfo** A BOOLEAN value that indicates that all volume information should be refreshed before it is returned.
- **pVolMetrics** An FS_VOL_METRICS structure that defines the variables that contain the returned information. Table 72-6 lists the members in FS_VOL_METRICS and their meanings.

### Table 72-6 Volume Metrics Information

<table>
<thead>
<tr>
<th>Metric</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Indicates the volume type. The type constants are defined in FS_VOL_TYPE.</td>
</tr>
<tr>
<td>flags</td>
<td>Indicates a number of volume attributes. The flags are defined in FS_VOL_FLAGS and are:</td>
</tr>
<tr>
<td>fsVolReadOnly</td>
<td>The volume is read-only.</td>
</tr>
<tr>
<td>fsVolConnected</td>
<td>The volume is currently connected (must set update=true to keep this current).</td>
</tr>
<tr>
<td>fsVolRemovableMedia</td>
<td>The volume is located on removable media.</td>
</tr>
<tr>
<td>fsVolEjectableMedia</td>
<td>The volume media can be ejected.</td>
</tr>
<tr>
<td>fsVolDirsIndexable</td>
<td>The volume supports directory indexes. Some volumes cannot support directory indexes.</td>
</tr>
<tr>
<td>fsVolFormattable</td>
<td>The volume can be formatted.</td>
</tr>
<tr>
<td>fsVolDuplicatable</td>
<td>The volume can be duplicated by a track-by-track physical duplicator.</td>
</tr>
<tr>
<td>rootDir</td>
<td>A handle on the volume's root directory.</td>
</tr>
<tr>
<td>volObj</td>
<td>The volume object UID.</td>
</tr>
<tr>
<td>serialNum</td>
<td>The volume's serial number.</td>
</tr>
<tr>
<td>optimalSize</td>
<td>The optimal block size for I/O.</td>
</tr>
<tr>
<td>totalBytes</td>
<td>Total number of bytes on the volume.</td>
</tr>
<tr>
<td>freeBytes</td>
<td>The number of free bytes on the volume (must set update=true to keep this current).</td>
</tr>
<tr>
<td>commSpeed</td>
<td>The communication speed (for remote volumes).</td>
</tr>
<tr>
<td>iconResId</td>
<td>The resource ID for the volume's icon.</td>
</tr>
<tr>
<td>pName</td>
<td>A buffer to receive the volume name.</td>
</tr>
</tbody>
</table>
The following example illustrates a call to `msgFSGetVolMetrics` that gets volume information for the current directory.

```
FS_GET_VOL_METRICS volMetrics;
volMetrics.updateInfo = TRUE; // This ensures an update to all fields.
status = ObjectCall(msgFSGetVolMetrics, theWorkingDir, &volMetrics);
```

### Setting or Changing a Volume Name

To set or change the name of a volume, send `msgFSSetVolName` to a file or directory handle on the volume that you want to rename. The message takes a single argument, a pointer to a string that contains the new volume name.

Currently the volume name must conform to MS-DOS volume naming conventions (up to 11 characters; cannot contain `/ \ ; : = < > [ ]`). If the volume name does not conform, the message returns `stsBadParam`.

### Ejecting Floppies

For floppy disk drives that support programmatic control of disk ejection, there is normally no eject button on the drive. Usually the user ejects such a floppy disk volume by making an E, e, or X gesture on the disk icon in the disk browser.

If you need to eject the floppy disk volume under programmatic control, send `msgFSEjectMedia` to a directory handle or file handle. The file system ejects the volume associated with that handle. The message takes no arguments.

### Volume Specific Messages

You can send messages that are specific to a particular device with `msgFSVolSpecific`. You can send this message to either a directory or file handle. Among the reasons for using this message are:

- Getting or setting the Macintosh icon on a TOPS file that resides on a Macintosh.
- Getting a volume-specific error code.
- Low-level device access to device drivers.

The message takes a pointer to a `VOL_SPECIFIC` structure that contains:

- **pPath** A path to a node. This is only meaningful when you send the message to a directory handle; it must be null if you send the message to a file handle.
- **msg** A message. You must define the messages yourself.
- **pArgs** A pointer to the arguments for the message.
Part 8 / System Services
# Chapter 73 / Introduction

Organization of This Part 95
Other Sources of Information 95

# Chapter 74 / PenPoint Kernel Overview

The Machine Interface Layer 98
The Kernel Layer 98
Task Management
  Processes 98
  Subtasks 98
  Software Task Scheduler 99
  Priority Level 99
Intertask Communication
  Messages and Queues 100
  Semaphores 100
Memory Management
  Heaps 101
  80386 Protected Mode 101
  Rings 103
  Privilege Levels 103
Date and Time Services
  Timer Routines 103
  Alarm Services 103
  Current Time 104
  Other Routines 104
  Object-Oriented Timer Interface 104
Sound Routine 105
PenPoint Kernel Summary 105

# Chapter 75 / C Run-Time Library

ANSI Standard C Routines 109
Time and Date Preferences
  System Time 110
  Time Formats 110
  Date and Time Strings 110
16-Bit Character Support
  16-Bit Character Types 111
  16-Bit String Function 111
  String Composition Functions 114

# Chapter 76 / Math Run-Time Library

Introduction 115
Programmatic Interface
  Fixed-Point Numbers 115
  Performance Notes 115
  PenPoint Fixed-Point Summary 116
List of Figures
  74-1  PenPoint System Architecture 97
List of Tables
  74-1  theTimer Messages 104
  74-2  Kernel Functions 105
  74-3  Heap Routines 107
  75-1  WATCOM C Run-Time Library 109
  75-2  16-Bit String Functions 111
  75-3  String Composition Functions 114
  76-1  Fixed-Point Functions 116
Chapter 73 / Introduction

The PenPoint™ system services allow you to enhance memory utilization and performance in your programs by providing access to the PenPoint kernel at the lowest level, and by providing optimized run-time routines.

Simple applications may not need to use system services at all. Instead they rely on the PenPoint Application Framework to install their code, create their processes, and communicate with them. They use the class manager to communicate with other objects.

The most common use of system services is to allocate and free memory using OSHeapBlockAlloc() and OSHeapBlockFree().

Organization of This Part

This part explains the PenPoint system services concisely. The chapters of Part 8: System Services cover the following information:

- Chapter 73, Introduction, is this chapter.
- Chapter 74, PenPoint Kernel Overview, describes the kernel, and the functional elements such as processes, tasks, and task communications that comprise the low-level operation of a PenPoint computer. At its end are tables summarizing the kernel-level APIs.
- Chapter 75, C Run-Time Library, introduces GO’s run-time support for the C programming language.
- Chapter 76, Math Run-Time Library, introduces GO’s fixed-point math libraries.

Other Sources of Information

“Datasheets” for each function are in the PenPoint API Reference, so that you can quickly locate the syntax for an individual kernel or run-time function.

Most datasheets in the PenPoint API Reference are formatted versions of the information in the corresponding header file. However, because the run-time library and other parts of system services are based on WATCOM’s standard header files, the information in the header files is not the same as the information in the datasheets.

The PenPoint Application Writing Guide explains common typedefs, status values, and macros used in PenPoint.
Chapter 74 / PenPoint Kernel Overview

The kernel is the lowest-level application-accessible component of the PenPoint™ operating system. Applications and other programs access the kernel through function calls; in turn, the kernel accesses the computer hardware:

- Directly when performing process management, scheduling, synchronization, and memory management.
- Indirectly through the PenPoint MIL (Machine Interface Layer), when communicating with devices.

These calls are invisible to the application programmer. The kernel provides the following services for PenPoint operations:

- Multitasking, message-passing executive.
- Threads (lightweight tasks sharing the same address space).
- Protected memory management and code execution.
- Semaphores for process synchronization.
- Procedural interface.

Figure 74-1 shows the position of the kernel in the PenPoint software architecture.
The Machine Interface Layer

The PenPoint operating system is designed so that it is not dependent on the hardware on which it runs.

The most obvious hardware dependency is on the central processor. However, even hardware that uses a common central processor can have different peripheral devices, such as screens, stylus and digitizers, storage devices, clock chips, and ports.

To support different types of devices, the PenPoint operating system defines a protocol for communicating with devices, called the Machine Interface Layer (MIL). Only the PenPoint kernel and PenPoint device drivers (called MIL services) make calls to the MIL; applications and non-MIL services never communicate directly with MIL.

The Kernel Layer

The kernel is the portion of the operating system that interacts directly with the processor to manage tasks, memory, and communication between tasks. It allocates memory dynamically as applications run. It manages the sending and receiving of synchronization messages between concurrent processes, and it schedules the resources that these processes need as they execute (access to common data, execution rights for shared code, and so on).

The kernel layer is different in several ways from the portions of the PenPoint operating system that are above the kernel layer:

- The kernel layer is not object-oriented, rather it provides support for the PenPoint object-oriented architecture.
- The kernel layer does not send or receive messages, it uses function calls.

Task Management

In PenPoint, a task is the basic executing entity and refers to any executing thread of control. There are two kinds of tasks: processes and subtasks. Processes and subtasks are scheduled and run by a software scheduler based on a priority scheme that determines which task should run at any given time.

Processes

A process is the first task that runs when an application is instantiated (typically when the user turns to a document of that application). Processes own the resources used by the instance including memory, subtasks, and semaphores (used in locking and interrupts). When a process is terminated, all resources owned by it are returned to the system.

Subtasks

A subtask is a thread of execution started by either a process or a subtask. It uses the same context as its parent process. Subtasks created by other subtasks are called
sibling subtasks; all siblings are considered to be at the same level regardless of the creator. The process that creates a subtask owns that subtask and any sibling subtasks created by it in a child-parent relationship. A process and all its subtasks make up a task family. When a process dies, all its subtasks are cleaned up so that PenPoint can reuse the task family’s local memory.

A subtask has the following characteristics:

- It shares all memory with its parent process and any siblings.
- It owns no resources itself.
- It has its own general registers and stack.
- It can lock semaphores, receive and send messages.

Software Task Scheduler

PenPoint allows tasks to dynamically create and delete other tasks in the system. Kernel functions are provided to start processes and subtasks. In addition, the kernel allows for the termination of a given task by any other software task.

In order to start another process, the executable file that contains the code for that process must have already been loaded into the loader database (see the OSProgramInstall() function for more details). To start a process, the kernel creates a new execution context (that is new local memory context).

Unlike subtasks, which are hierarchically below their parent process, processes started by other processes (or subtasks) are not hierarchically linked. Since there is no process hierarchy, the following is true:

- The process creator has no special impact on the process being created (and vice versa). Thus, if the creator terminates, the newly created process is not notified or terminated. Likewise, the creator is not notified if the newly created process terminates.
- A copy of the process creator's file handles are not passed on to the new process.
- A process is free to associate itself with other processes in whichever way it wants. This more closely resembles the metaphor used by the notebook software (where an application can move from one page to another and applications can be arbitrarily embedded into other applications).

Priority Level

Software tasks have set priority levels to control their order of execution. Task scheduling is by priority only. Processes and subtasks with higher priorities execute before lower priority tasks. Priorities may be altered by the task itself or by any other task in the system. Priorities are defined by their priority class (high, medium high, medium low, and low) and a priority within that class (0-50). The PenPoint operating system arranges task priorities within bands.
Intertask Communication

Intertask communication in PenPoint takes place through messages and semaphores.

Messages and Queues

An intertask message is a pre-defined PenPoint element that contains miscellaneous information for a task along with an optional, variable-length message buffer.

Intertask messages should not be confused with class manager messages. Intertask messages are sent to tasks and begin with the prefix OSITMsg where OS stands for operating system and ITMsg stands for intertask message. Class manager messages are sent to objects and begin with the prefix msg.

For communications between tasks, the kernel relies on intertask messages. Intertask messages are transferred between tasks, using the task identifier to target the message. Messages contain miscellaneous information along with an optional message buffer. Message buffers are variable length and are not interpreted by the kernel. There is only one message queue per task (subtask or process). The message queue is organized using a first in-first out algorithm.

Messages can be transferred in two modes: copy or noCopy. Copy is the normal mode with all of the data associated with the message copied into the workspace of the target task. NoCopy is used for high speed data exchange at the expense of some protection. In noCopy mode, the sender passes a pointer to a shared heap, which contains the message buffer.

The message facility also allows the client to pass a 32-bit token as part of the message. In small transfers of data, this token could contain the entire message data. ObjectSend() uses intertask messages to send a message to an object in a different task. Application developers should use ObjectSend() instead of the kernel call OSITMsgSend().

Normally messages are sent to just one task. However, the kernel will allow messages to be sent to multiple tasks by the use of broadcast messages. When broadcasting, the kernel sends the message to all tasks that are allowed to receive messages. The broadcast message can be sent to all tasks in the system, or to a single task and its task family. Note that the sending task does not receive its own broadcast message.

As described above, messages are normally processed in FIFO order. However, messages can be processed out of order by the use of filters. A task receiving messages can choose to get a message from the queue using two types of filters:

- The task can have a filter that specifies the types of messages that can be placed in its queue.
- The dispatch loop for the task can use a filter when receiving messages from the task’s message queue.
The kernel uses messages to inform tasks of certain important events. As mentioned before, upon task termination, the kernel broadcasts a message to the entire system on a special filter. Those applications that have enabled that filter in their filter mask will be notified.

**Semaphores**

For synchronization (which really is a form of communication), the kernel provides **semaphores**. A semaphore, as in most operating systems, is a lock that allows only one process or subtask access to a resource at one time. Semaphores have two major purposes:

- To lock tasks in order to prevent collisions between contending tasks seeking access.
- To accept interrupts in event handling, usually from device drivers.

The kernel implements a counting semaphore, so that if a particular task calls for a particular number of lock operations, the task must issue an equal number of unlock operations before the semaphore is free. If a task requests access to a semaphore and is forced to wait because another task has already locked it, the requesting task will be put to sleep until the semaphore is cleared. At that time the highest priority task waiting on that semaphore will be given ownership of the semaphore lock and will be put into the ready list.

If a task with a locked semaphore dies, the kernel will unlock that semaphore and notify the next task attempting to lock that semaphore of the forced unlock by the system. It is up to that next task to determine if there is a problem (for example, an inconsistent data structure). If the semaphore is protecting a critical data structure, any task accessing that data structure (and semaphore) should know how to clean up from accidental task termination. The best way to do this is to centralize access to the semaphore. If all tasks access the semaphore using the same code, then handling the exception case need only be done in that one location.

**Memory Management**

In the PenPoint operating system, memory has attributes, such as shared or local, locked or unlocked, access rights, privilege level, length, and so on. Tasks must request the amount of local and shared memory required for execution.

The PenPoint operating system provides a 32-bit flat memory model. Memory is allocated in **heaps**. Allocation of heaps is referred to as **heap management**. The heap manager uses a portion of address space that has already been allocated.

**Heaps**

A heap is a region or list of regions of virtual memory. The heap manager handles the allocation and freeing of smaller blocks within those regions. The heap manager code runs at the privilege level of the code that calls it, so the memory it allocates has the privilege of the code calling it.
The size and characteristics of a heap is defined by the task that creates it. By default, heap regions are 16K bytes long; heaps larger than 16K bytes are allocated in multiples of 4K bytes. Each heap uses a minimum of 4K bytes; its elements can be fixed or variable length. Heaps can be shared (accessible to multiple processes) or local. See OSHEAP.H for more information on heaps.

The system automatically creates two heaps for each process: a local heap and a shared heap. osProcessHeapId is the handle (well-known UID) on the local heap for each process; osProcessSharedHeapId is the handle for the shared heap.

Most applications should be able to use these predefined heaps and should not need to create additional heaps. When necessary, an application can minimize fragmentation within a heap by using different heaps for different functions. This is only efficient when the data in each heap is greater than than 4K bytes.

Because the heap manager allocates memory within a region, no hardware protection is provided.

The PenPoint operating system does not support either heap compaction or garbage collection.

The heap manager provides a number of utility routines to simplify the management of heaps.

80386 Protected Mode

The PenPoint 1.0 runs in the protected mode of the 80386. This means that the kernel utilizes the hardware facilities for task management and memory protection. The hardware will provide a completely separate address space (through virtual memory) for different programs.

A task family shares local memory. Each process has its own local memory, which is accessible by all the subtasks in its task family, but not by other processes. This means that if a task passes the address of some object in local memory to a task in another task family, the pointer will be invalid or point to some random data for the second task. You should be careful to use shared memory for data which needs to be accessed by tasks using different process contexts.

Shared memory is accessible to all processes. When the CPU switches to a task in a different process, it switches to the new process's local memory.

A task can allocate memory that is either local to its task family or shareable among many different tasks. Different tasks can share ownership of shared memory. The memory manager maintains a reference count of the memory. The memory will be freed only when all tasks sharing the memory have freed their pointers to that memory.

The memory model does support memory movement. The memory manager guarantees that the selector pointing to the memory block will always be valid until that block is freed. In addition, the memory manager exports a number of low level memory routines to support externally developed system code such as device drivers.
In addition to memory protection among processes, PenPoint utilizes the ring structure of the 386 to protect system data structures from mischievous applications. The kernel runs in supervisor mode (ring 0). Applications run in user mode (ring 3). Data structures that are allocated in supervisor mode by the kernel are not accessible to application code. Application code can use the function `OSSupervisorCall()` to access the kernel. Applications cannot access system buffers (even those in shared memory).

Shared memory is used for anything that is required across many different processes. Examples include system data structures, application code, and shared objects.

**Privilege Levels**

A task always executes at some privilege level. Memory is accessed through regions. Each region has a privilege level and access permissions (read-only, read-write). All of a task’s accesses to memory are checked against the privilege level and access rights of the region. The PenPoint kernel and device drivers have a higher privilege than application tasks.

A task cannot access more privileged data, nor can it execute code at other privilege levels unless that code has made special arrangements. For example, application code can only call PenPoint kernel routines through routines in `PENPOINT.LIB`.

**Date and Time Services**

**Timer Routines**

Timer notification comes in two flavors: notification by semaphore and notification by message. In the first case, the client calls a function. In the latter case, the client sends a message to a special well-known object, `theTimer`.

If notification is done by semaphore, the client must provide the semaphore; the system will reset that semaphore at the appropriate notification time.

Because the kernel counts time by the system tick interrupt, no timeout values in the range of the system tick interval time will be very accurate. When a timer request occurs and is placed on the transaction queue, the timer subsystem has no way of knowing how much time elapses between the request and the next system tick. As a result, the timer allots the entire systick interval time (roughly 30 milliseconds) to the transaction.

**Alarm Services**

The alarm subsystem uses the clock chip to keep track of the next alarm. A queue of alarms transactions is maintained so that more than one alarm can be set in the system. The alarm code calls the MIL clock device to handle the details of setting an alarm.
**Current Time**

The kernel provides routines to set and get the date and time from the clock chip. The clock chip is updated through the MIL.

**Other Routines**

Chapter 75, C Run-Time Library, describes the run-time routines to manipulate the time and date, and some system preferences that determine the user’s desired time and date formats.

**Object-Oriented Timer Interface**

You can access timer and alarm services in an object-oriented manner. There is a well-known object, theTimer, which responds to messages. You can ask it to notify an object when a timer period expires, or repeatedly, or at a particular alarm time. At the appropriate time, theTimer sends msgTimerNotify or msgTimerAlarmNotify to that object.

theTimer is a well-known instance of clsTimer; it’s the only instance. (They share the same UID, in fact.)

The messages which clsTimer defines are in PENPOINT\SDK\INC\TIMER.H. They are:

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgTimerRegister</td>
<td>P_TIMER_REGISTER_INFO</td>
<td>Registers a request for notification with ObjectPost().</td>
</tr>
<tr>
<td>msgTimerRegisterAsync</td>
<td>P_TIMER_REGISTER_INFO</td>
<td>Registers a request for notification with ObjectPostAsync().</td>
</tr>
<tr>
<td>msgTimerRegisterDirect</td>
<td>P_TIMER_REGISTER_INFO</td>
<td>Registers a request for notification with ObjectPostDirect().</td>
</tr>
<tr>
<td>msgTimerRegisterInterval</td>
<td>P_TIMER_INTERVAL_INFO</td>
<td>Registers a request for interval notification.</td>
</tr>
<tr>
<td>msgTimerAlarmRegister</td>
<td>P_TIMER_ALARM_INFO</td>
<td>Registers a request for alarm notification.</td>
</tr>
<tr>
<td>msgTimerAlarmStop</td>
<td>OS_HANDLE</td>
<td>Stops a pending alarm request.</td>
</tr>
<tr>
<td>msgTimerTransactionValid</td>
<td>OS_HANDLE</td>
<td>Determines if a timer transaction is valid.</td>
</tr>
<tr>
<td>msgTimerStop</td>
<td>OS_HANDLE</td>
<td>Stops a timer transaction.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgTimerNotify</td>
<td>P_TIMER_NOTIFY</td>
<td>Notifies the client that the timer request has elapsed.</td>
</tr>
<tr>
<td>msgTimerAlarmNotify</td>
<td>P_ALARM_NOTIFY</td>
<td>Notifies the client that the alarm request has elapsed.</td>
</tr>
</tbody>
</table>

A timer request can continue to count down after a PenPoint computer is powered-on. An alarm can go off while a PenPoint computer is off, and it will turn the PenPoint computer back on in order to deliver the alarm message.

**Note** This is a hardware feature that may or may not be present on any particular PenPoint computer.
The Clock application uses the Timer extensively. Its code is in PENPOINT\SDK\SAMPLE\CLOCK, and is briefly discussed in the PenPoint Application Writing Guide.

Sound Routine

The kernel provides two basic routines to sound the speaker on a PenPoint computer: OSErrBeep() and OSTone(). In OSErrBeep() you specify the type of error (warning or fatal), and the computer beeps the appropriate tone. OSTone() is more general; it allows you to sound the speaker, specifying a tone, duration, and volume.

PenPoint Kernel Summary

The PenPoint operating system kernel functions are defined in two header files:

OS.H Defines functions for tasking, memory information, inter-task communication, and timer services.

OSHEAP.H Defines functions for memory management.

The corresponding Part 8 in the PenPoint API Reference describes the details of PenPoint Kernel API. Every function is described along with its associated parameters. The functions defined in OS.H are summarized in Table 74-2; the functions defined in OSHEAP.H are summarized in Table 74-3. For more information on each of these functions, see the header files.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSProgramInstall()</td>
<td>Installs a program into the loader database.</td>
</tr>
<tr>
<td>OSProgramDeinstall()</td>
<td>Deinstalls a program already loaded into the loader database.</td>
</tr>
<tr>
<td>OSProgramInstantiate()</td>
<td>Creates an instance of a program.</td>
</tr>
<tr>
<td>OSProgramInfo()</td>
<td>Returns information on the program from the loader.</td>
</tr>
<tr>
<td>OSSubTaskCreate()</td>
<td>Creates a new execution thread in this context.</td>
</tr>
<tr>
<td>OSTaskTerminate()</td>
<td>Terminates a task.</td>
</tr>
<tr>
<td>OThisTask()</td>
<td>Passes back the task identifier of the current running task.</td>
</tr>
<tr>
<td>OSTaskPrioritySet()</td>
<td>Sets the priority of a task or a set of tasks.</td>
</tr>
<tr>
<td>OSTaskPriorityGet()</td>
<td>Passes back the priority of a task.</td>
</tr>
<tr>
<td>OSTaskNameSet()</td>
<td>Sets a 4 character name for the given task.</td>
</tr>
<tr>
<td>OSTaskDelay()</td>
<td>Delays the current task for a specified period of time.</td>
</tr>
<tr>
<td>OSEnterpointFind()</td>
<td>Finds an entrypoint in a loaded module either by name or by ordinal.</td>
</tr>
<tr>
<td>OSMODULELoad()</td>
<td>Loads a module into the loader's database.</td>
</tr>
<tr>
<td>OSEntryPointFind()</td>
<td>Passes back the program handle for the process.</td>
</tr>
<tr>
<td>OSMODULELoad()</td>
<td>Passes back the application object stored with the current process.</td>
</tr>
<tr>
<td>OSTaskTerminate()</td>
<td>Passes back the application object for a given process.</td>
</tr>
<tr>
<td>OSMODULELoad()</td>
<td>Returns the process id for the task specified.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OSTaskInstallTerminate()</td>
<td>Notifies tasks waiting on OSProgramInstall that the instance is finished.</td>
</tr>
<tr>
<td>OSEnvSearch()</td>
<td>Searches the environment for the specified variable and returns its value.</td>
</tr>
<tr>
<td>OSAppObjectPoke()</td>
<td>Stores the application object for the current process.</td>
</tr>
</tbody>
</table>

### Intertask Communications Routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSITMsgSend()</td>
<td>Sends an inter-task message to a task or set of tasks.</td>
</tr>
<tr>
<td>OSITMsgReceive()</td>
<td>Receives a message from the task's message queue.</td>
</tr>
<tr>
<td>OSITMsgPeek()</td>
<td>Gets the next message from the message queue without removing it.</td>
</tr>
<tr>
<td>OSITMsgFilterMask()</td>
<td>Sets the filter mask for this task.</td>
</tr>
<tr>
<td>OSITMsgFromId()</td>
<td>Passes back the message associated with the message identifier.</td>
</tr>
<tr>
<td>OSITMsgQFlush()</td>
<td>Flushes the message queue of all messages matching the message filter.</td>
</tr>
<tr>
<td>OSSemaCreate()</td>
<td>Creates a semaphore.</td>
</tr>
<tr>
<td>OSSemaOpen()</td>
<td>Opens (accesses) an already existing semaphore.</td>
</tr>
<tr>
<td>OSSemaDelete()</td>
<td>Deletes a semaphore.</td>
</tr>
<tr>
<td>OSSemaRequest()</td>
<td>Locks the counting semaphore (increments the count).</td>
</tr>
<tr>
<td>OSSemaClear()</td>
<td>Unlocks the counting semaphore (decrements the count).</td>
</tr>
<tr>
<td>OSSemaReset()</td>
<td>Resets event semaphore (no matter what count).</td>
</tr>
<tr>
<td>OSSemaSet()</td>
<td>Sets the event semaphore to 1.</td>
</tr>
<tr>
<td>OSSemaWait()</td>
<td>Waits for the event semaphore to be reset.</td>
</tr>
<tr>
<td>OSFastSemaInit()</td>
<td>Initialize fast sema.</td>
</tr>
<tr>
<td>OSFastSemaRequest()</td>
<td>Fast version of sema request.</td>
</tr>
<tr>
<td>OSFastSemaClear()</td>
<td>Fast version of sema clear.</td>
</tr>
</tbody>
</table>

### Memory Information Routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSMemInfo()</td>
<td>Returns information on memory usage for a specified task.</td>
</tr>
<tr>
<td>OSMemUseInfo()</td>
<td>Returns information on memory usage for a specified task.</td>
</tr>
<tr>
<td>OSMemAvailable()</td>
<td>Return amount of swappable memory available (to caution zone).</td>
</tr>
</tbody>
</table>

### Date and Timer Routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSTimerAsyncSema()</td>
<td>Reset a semaphore after time milliseconds.</td>
</tr>
<tr>
<td>OSTimerIntervalSema()</td>
<td>Resets a semaphore after each time interval has elapsed.</td>
</tr>
<tr>
<td>OSTimerStop()</td>
<td>Stops a timer request given its transaction handle.</td>
</tr>
<tr>
<td>OSGetTime()</td>
<td>Returns local time.</td>
</tr>
<tr>
<td>OSSetTime()</td>
<td>Sets the time or time zone.</td>
</tr>
<tr>
<td>OSPowerUpTime()</td>
<td>Passes back the number of milliseconds since the last reset.</td>
</tr>
<tr>
<td>OSSetInterrupt()</td>
<td>Sets up an interrupt handler.</td>
</tr>
<tr>
<td>OSTimerTransactionValid()</td>
<td>Checks to see if the timer transaction is valid.</td>
</tr>
</tbody>
</table>

### Debugger Entry Routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debugger()</td>
<td>Enters the debugger.</td>
</tr>
<tr>
<td>OSDdebugger()</td>
<td>Enters the debugger, should only be called in special situations.</td>
</tr>
</tbody>
</table>
### Table 74-2 (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyPressed()</td>
<td>Determines if a key is available.</td>
</tr>
<tr>
<td>KeyIn()</td>
<td>Passes back the next key and the scan code from the keyboard.</td>
</tr>
<tr>
<td>OSErrorBeep()</td>
<td>Outputs a tone based on the type of error encountered.</td>
</tr>
<tr>
<td>OSTone()</td>
<td>Sends a tone for a given duration at the specified volume level.</td>
</tr>
<tr>
<td>ScreenOnlyStringPrint()</td>
<td>Prints a string onto the console.</td>
</tr>
<tr>
<td>OSDisplay()</td>
<td>Changes the display to the console or the graphics screen.</td>
</tr>
<tr>
<td>OSWinDev()</td>
<td>Passes back the windowing device for this application.</td>
</tr>
<tr>
<td>OSWinDevPoke()</td>
<td>Stores the windowing device for the specified process.</td>
</tr>
<tr>
<td>osPrintBufferRoutine()</td>
<td>Function variable print routine.</td>
</tr>
<tr>
<td>OSPowerDown()</td>
<td>Powers down the machine.</td>
</tr>
<tr>
<td>OSSystemInfo()</td>
<td>Passes back information on the system configuration.</td>
</tr>
</tbody>
</table>

### Table 74-3

<table>
<thead>
<tr>
<th>Heap Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
</tr>
<tr>
<td>OSHearpCreate()</td>
</tr>
<tr>
<td>OSHearpDelete()</td>
</tr>
<tr>
<td>OSHearpBlockAlloc()</td>
</tr>
<tr>
<td>OSHearpBlockFree()</td>
</tr>
<tr>
<td>OSHearpBlockResize()</td>
</tr>
<tr>
<td>OSHearpInfo()</td>
</tr>
<tr>
<td>OSHearpId()</td>
</tr>
<tr>
<td>OSHearpBlockSize()</td>
</tr>
<tr>
<td>OSHearpPoke()</td>
</tr>
<tr>
<td>OSHearpPeek()</td>
</tr>
<tr>
<td>OSHearpAllowError()</td>
</tr>
<tr>
<td>OSHearpClear()</td>
</tr>
<tr>
<td>OSHearpOpen()</td>
</tr>
<tr>
<td>OSHearpClose()</td>
</tr>
<tr>
<td>OSHearpEnumerate()</td>
</tr>
<tr>
<td>OSHearpWalk()</td>
</tr>
<tr>
<td>OSHearpMark()</td>
</tr>
<tr>
<td>OSHearpPrint()</td>
</tr>
</tbody>
</table>
Chapter 75 / C Run-Time Library

This chapter lists the C run-time library available to developers of PenPoint applications and services. Many of the C run-time functions are provided by the WATCOM C run-time library. Other functions provided in PenPoint include:

- Time and date preferences.
- 16-bit character support.

**ANSI Standard C Routines**

For information on the WATCOM C run-time library, see the *WATCOM C Library Reference for PenPoint*.

Table 75-1 lists the files in the WATCOM C run-time library.

<table>
<thead>
<tr>
<th>File</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSERT.H</td>
<td>Assertion macros.</td>
</tr>
<tr>
<td>CONIO.H</td>
<td>Port I/O functions.</td>
</tr>
<tr>
<td>CTYPES.H</td>
<td>Character manipulation functions.</td>
</tr>
<tr>
<td>DIRENT.H</td>
<td>Directory functions and declarations.</td>
</tr>
<tr>
<td>ENV.H</td>
<td>Prototypes of environment string functions.</td>
</tr>
<tr>
<td>FCNTL.H</td>
<td>Flags used by open and sopen.</td>
</tr>
<tr>
<td>FLOAT.H</td>
<td>Declarations and constants used with floating point numbers.</td>
</tr>
<tr>
<td>I86.H</td>
<td>Low-level CPU functions.</td>
</tr>
<tr>
<td>LIMITS.H</td>
<td>Constants for limits and boundaries.</td>
</tr>
<tr>
<td>MALLOC.H</td>
<td>Memory allocation and deallocation functions.</td>
</tr>
<tr>
<td>MATH.H</td>
<td>Mathematical functions.</td>
</tr>
<tr>
<td>SEARCH.H</td>
<td>Searching functions (ifind and lsearch).</td>
</tr>
<tr>
<td>SETJMP.H</td>
<td>Declarations for setjmp and longjmp functions.</td>
</tr>
<tr>
<td>SIGNAL.H</td>
<td>Declarations for signal and raise functions.</td>
</tr>
<tr>
<td>STDARG.H</td>
<td>Variable-length argument list functions.</td>
</tr>
<tr>
<td>STDDEF.H</td>
<td>A number of standard constants.</td>
</tr>
<tr>
<td>STDIO.H</td>
<td>Standard input and output functions.</td>
</tr>
<tr>
<td>STDLIB.H</td>
<td>Declarations for standard functions.</td>
</tr>
<tr>
<td>STRING.H</td>
<td>String and memory functions.</td>
</tr>
<tr>
<td>TIME.H</td>
<td>Time and date functions.</td>
</tr>
<tr>
<td>UNISTD.H</td>
<td>System level I/O functions.</td>
</tr>
<tr>
<td>UTIME.H</td>
<td>Declarations for utime function.</td>
</tr>
</tbody>
</table>
Time and Date Preferences
The user (or a program) can specify the desired format for certain system parameters called preferences. These are maintained by the SystemPreferences in a resource file. You can get and set preferences by sending resource messages such as msgResReadData to the SystemPreferences. Resources are explained in Part 11: Resources.

System Time
One of the preferences is the current system date and time, with the well-known resource ID prTime. The time is in an OS_DATE_TIME structure.
Getting this resource is an alternative to calling the OSGetTime() routine.

Time Formats
Other user preferences indicate:
- Whether the user prefers time in military (24-hour) format or regular format (resource ID prTimeFormat).
- Whether the user wants to see seconds in time displays (ID prTimeSeconds).
- How the user prefers to see dates displayed (ID prDateFormat).

Date and Time Strings
Instead of retrieving all these resources, then formatting a string accordingly, you can call functions that create formatted strings for a specified time based on the current user preferences:
- PrefsDateToString() fills in a string you supply with the date. The string should be at least prefsMaxDate long (this may change if additional formats are added).
- PrefsTimeToString() fills in a string you supply with the time. The string should be at least prefsMaxTime long (this may change if additional formats are added).

16-Bit Character Support
PenPoint 2.0 will contain support for applications that are written for more than one language or region. To support languages that have large numbers of characters (such as Japanese), PenPoint 2.0 will support 16-bit characters.
PenPoint 1.0 already includes many features that will be used to support 16-bit character sets. These features include:
- New character types and macro support.
- New run-time library string functions.
- New string composition functions.
**16-Bit Character Types**

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

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

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

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

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

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

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

**16-Bit String Function**

The file INTL.H defines a new set of run-time library string functions that operate on 16-bit characters. The names of the 16-bit functions are similar to the existing 8-bit functions, but the 16-bit function names are preceeded with the letter “U”. For example, strcmp() becomes Ustrcmp().

In PenPoint 1.0, the U... functions are identical to their 8-bit namesakes. In PenPoint 2.0, the U... functions will be true 16-bit functions. In other words, the old functions only work on CHAR8 strings, the U... functions in 1.0 work on CHAR8 strings, in 2.0 the U... functions will work on CHAR16 strings.

Table 75-2 lists the 16-bit string functions and the corresponding 8-bit functions.

<table>
<thead>
<tr>
<th>16-Bit Function</th>
<th>8-Bit Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ustrcat()</td>
<td>strcat()</td>
</tr>
<tr>
<td>Ustrncat()</td>
<td>strncat()</td>
</tr>
<tr>
<td>Ustrcmp()</td>
<td>strcmp()</td>
</tr>
<tr>
<td>Ustrnncmp()</td>
<td>strncmp()</td>
</tr>
<tr>
<td>Ustrcpy()</td>
<td>strcpy()</td>
</tr>
</tbody>
</table>

Table 75-2: 16-Bit String Functions

continued
<table>
<thead>
<tr>
<th>16-Bit Function</th>
<th>8-Bit Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ustrncpy()</td>
<td>strncpy()</td>
</tr>
<tr>
<td>Ustrlen()</td>
<td>strlen()</td>
</tr>
<tr>
<td>Ustrdup()</td>
<td>strdup()</td>
</tr>
<tr>
<td>Ustrrev()</td>
<td>strrev()</td>
</tr>
<tr>
<td>Ustrset()</td>
<td>strset()</td>
</tr>
<tr>
<td>Ustrnset()</td>
<td>strnset()</td>
</tr>
<tr>
<td>Ustrchr()</td>
<td>strchr()</td>
</tr>
<tr>
<td>Ustrrchr()</td>
<td>strrchr()</td>
</tr>
<tr>
<td>Ustrspn()</td>
<td>strspn()</td>
</tr>
<tr>
<td>Ustrcspn()</td>
<td>strcspn()</td>
</tr>
<tr>
<td>Ustrupbrk()</td>
<td>strpbrk()</td>
</tr>
<tr>
<td>Ustrstr()</td>
<td>strstr()</td>
</tr>
<tr>
<td>Ustrtok()</td>
<td>strtok()</td>
</tr>
<tr>
<td>Ustricmp()</td>
<td>stricmp()</td>
</tr>
<tr>
<td>Ustrnicmp()</td>
<td>strnicmp()</td>
</tr>
<tr>
<td>Ustrlwr()</td>
<td>strlwr()</td>
</tr>
<tr>
<td>Ustrupr()</td>
<td>strupr()</td>
</tr>
<tr>
<td>Umemcpy()</td>
<td>memcpy()</td>
</tr>
<tr>
<td>Umemccpy()</td>
<td>memccpy()</td>
</tr>
<tr>
<td>Umemchr()</td>
<td>memchr()</td>
</tr>
<tr>
<td>Umemcmp()</td>
<td>memcmp()</td>
</tr>
<tr>
<td>Umemcmpl()</td>
<td>memcmpl()</td>
</tr>
<tr>
<td>Umemmove()</td>
<td>memmove()</td>
</tr>
<tr>
<td>Umemset()</td>
<td>memset()</td>
</tr>
<tr>
<td>Ustrerror()</td>
<td>strerror()</td>
</tr>
</tbody>
</table>

```c

CTYPE.H
```
<table>
<thead>
<tr>
<th>16-Bit Function</th>
<th>8-Bit Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uatoi()</td>
<td>atoi()</td>
</tr>
<tr>
<td>U atol()</td>
<td>atol()</td>
</tr>
<tr>
<td>Uitoa()</td>
<td>itoa()</td>
</tr>
<tr>
<td>U ltoa()</td>
<td>ltoa()</td>
</tr>
<tr>
<td>Uutoa()</td>
<td>utoa()</td>
</tr>
<tr>
<td>Ustrtol()</td>
<td>strtol()</td>
</tr>
<tr>
<td>Uatof()</td>
<td>atof()</td>
</tr>
<tr>
<td>U strtod()</td>
<td>strtod()</td>
</tr>
<tr>
<td>Ustrtoul()</td>
<td>strtoul()</td>
</tr>
</tbody>
</table>

STDLIB.H

<table>
<thead>
<tr>
<th>16-Bit Function</th>
<th>8-Bit Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ufopen()</td>
<td>fopen()</td>
</tr>
<tr>
<td>Usprintf()</td>
<td>sprintf()</td>
</tr>
<tr>
<td>Uvsprintf()</td>
<td>vsprintf()</td>
</tr>
<tr>
<td>Usscanf()</td>
<td>sscanf()</td>
</tr>
<tr>
<td>Uputc()</td>
<td>putc()</td>
</tr>
<tr>
<td>Ufputc()</td>
<td>fputc()</td>
</tr>
<tr>
<td>Ugetc()</td>
<td>getc()</td>
</tr>
<tr>
<td>Ufgets()</td>
<td>fgets()</td>
</tr>
<tr>
<td>Upgetchar()</td>
<td>getchar()</td>
</tr>
<tr>
<td>Ugets()</td>
<td>gets()</td>
</tr>
<tr>
<td>Ufputchar()</td>
<td>fputchar()</td>
</tr>
<tr>
<td>Ufputchar()</td>
<td>fputchar()</td>
</tr>
<tr>
<td>Ugets()</td>
<td>fgets()</td>
</tr>
<tr>
<td>Utmpnam()</td>
<td>tmpnam()</td>
</tr>
<tr>
<td>Urename()</td>
<td>rename()</td>
</tr>
<tr>
<td>Uremove()</td>
<td>remove()</td>
</tr>
<tr>
<td>Utmpnam()</td>
<td>tmpnam()</td>
</tr>
</tbody>
</table>

STDIO.H

<table>
<thead>
<tr>
<th>16-Bit Function</th>
<th>8-Bit Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ufopen()</td>
<td>fopen()</td>
</tr>
<tr>
<td>Usprintf()</td>
<td>sprintf()</td>
</tr>
<tr>
<td>Uvsprintf()</td>
<td>vsprintf()</td>
</tr>
<tr>
<td>Usscanf()</td>
<td>sscanf()</td>
</tr>
<tr>
<td>Uputc()</td>
<td>putc()</td>
</tr>
<tr>
<td>Ufputc()</td>
<td>fputc()</td>
</tr>
<tr>
<td>Ugetc()</td>
<td>getc()</td>
</tr>
<tr>
<td>Ufgetc()</td>
<td>fgetc()</td>
</tr>
<tr>
<td>Uungetc()</td>
<td>ungetc()</td>
</tr>
<tr>
<td>Ufdopen()</td>
<td>fdopen()</td>
</tr>
<tr>
<td>Ufreopen()</td>
<td>freopen()</td>
</tr>
<tr>
<td>Uprintf()</td>
<td>printf()</td>
</tr>
<tr>
<td>Ufprintf()</td>
<td>fprintf()</td>
</tr>
<tr>
<td>Uvprintf()</td>
<td>vprintf()</td>
</tr>
<tr>
<td>Uscanf()</td>
<td>scanf()</td>
</tr>
<tr>
<td>Ufscanf()</td>
<td>fscanf()</td>
</tr>
<tr>
<td>Uvscanf()</td>
<td>vscanf()</td>
</tr>
<tr>
<td>Utmpnam()</td>
<td>tmpnam()</td>
</tr>
<tr>
<td>Urename()</td>
<td>rename()</td>
</tr>
<tr>
<td>Utmpnam()</td>
<td>tmpnam()</td>
</tr>
</tbody>
</table>

continued
### String Composition Functions

The file CMPSTEXT.H contains **Compose** functions for assembling a composite string out of other pieces. Use these routines to create strings in your UI—*don't use sprintf()*!

The **Compose** routines will also save you effort because you can specify the `resId` of a format string and the code will read it from the resfile for you. You can, of course, give the format string directly to the routines.

Table 75-3 lists the string composition functions.

#### Table 75-3

**String Composition Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SComposeText()</td>
<td>Composes a string from a format and arguments.</td>
</tr>
<tr>
<td>VSCreateText()</td>
<td>Composes a string from a format and a pointer to the argument list.</td>
</tr>
</tbody>
</table>
Chapter 76 / Math Run-Time Library

Introduction

- PenPoint provides a fixed-point math facility as a part of its run-time support for ANSI C compilers. This math package resides permanently in the memory of the computer and is shared among computational applications.

The mathematics library supports fixed-point calculations. Fixed-point arithmetic is ideal for situations in which a fixed-point number with 16 bits of precision for the integer part and 16 bits of precision for the fractional part is acceptable.

Floating point support is provided directly by the C language and a shared library called directly from the compiler.

Programmatic Interface

Application programs invoke the fixed-point functions through procedure calls. For example:

```c
FIXED a, b, c;
STATUS s;
s = FxAdd(b, c, a);
```

not:

```c
a = b + c;
```

We have chosen the simplest procedure names possible in order to enhance readability of programs. The details of the routines are in the PenPoint API Reference.

Fixed-Point Numbers

The FIXED type is an S32. To create a fixed-point number, you use the routine FxMakeFixed(), specifying an S16 whole part and a U16 fractional part. For convenience, \`\`PENPOINT\`\`SDK\`\`INC\`\`GOMATH.H defines GoFx0, GoFx1, and GoFxMinus1.

ImagePoint uses FIXED numbers to specify scale factors. To save including GOMATH.H just to specify a scale factor, the definitions of FIXED and FxMakeFixed() are in \`\`PENPOINT\`\`SDK\`\`INC\`\`GO.H.

Performance Notes

FxAdd(), FxSub(), FxMul(), and FxDiv(), which include rounding and error checking, perform about 5% slower than FxAddSC(), FxSubSC(), FxMulSC(), and FxDivSC(), the truncating, trusting alternatives (which are macros instead of function calls).
PenPoint Fixed-Point Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FxAdd()</td>
<td>Adds two FIXED numbers, producing a FIXED.</td>
</tr>
<tr>
<td>FxAddSC()</td>
<td>Macro form of FxAdd() with no overflow detection.</td>
</tr>
<tr>
<td>FxSub()</td>
<td>Subtracts two FIXED numbers, producing a FIXED.</td>
</tr>
<tr>
<td>FxSubSC()</td>
<td>Macro form of FxSub() with no overflow detection.</td>
</tr>
<tr>
<td>FxMul()</td>
<td>Multiplies two FIXED numbers, producing a FIXED.</td>
</tr>
<tr>
<td>FxMulSC()</td>
<td>Multiplies two FIXED numbers returning the product. No overflow detection.</td>
</tr>
<tr>
<td>FxMulInt()</td>
<td>Multiplies a FIXED number by an S32, producing a FIXED.</td>
</tr>
<tr>
<td>FxMulIntSC()</td>
<td>Multiplies a FIXED number by an S32, returning the FIXED product. No overflow detection.</td>
</tr>
<tr>
<td>FxMulIntToInt()</td>
<td>Multiplies a FIXED number by an S32, producing a rounded S32 product.</td>
</tr>
<tr>
<td>FxMulIntToIntSC()</td>
<td>Multiplies a FIXED number by an S32, returning a rounded S32 product. No overflow detection.</td>
</tr>
<tr>
<td>FxDiv()</td>
<td>Divides two FIXED numbers, producing a FIXED quotient.</td>
</tr>
<tr>
<td>FxDivSC()</td>
<td>Divides two FIXED numbers, returning a FIXED quotient. No overflow detection.</td>
</tr>
<tr>
<td>FxDivInts()</td>
<td>Divides two 32-bit signed integers, producing a FIXED quotient.</td>
</tr>
<tr>
<td>FxDivIntsSC()</td>
<td>Divides two FIXED numbers, returning a FIXED quotient. No overflow detection.</td>
</tr>
<tr>
<td>FxDivIntToInt()</td>
<td>Divides an S32 by a FIXED, producing a rounded S32 quotient.</td>
</tr>
<tr>
<td>FxDivIntToIntSC()</td>
<td>Divides an S32 by a FIXED, producing a rounded S32 quotient. No overflow detection.</td>
</tr>
<tr>
<td>FxSin()</td>
<td>Returns the sine of an angle specified as an integer degree.</td>
</tr>
<tr>
<td>FxCos()</td>
<td>Returns the cosine of an angle specified as an integer degree.</td>
</tr>
<tr>
<td>FxTan()</td>
<td>Returns the tangent of an angle specified in as integer degree.</td>
</tr>
<tr>
<td>FxSinFx()</td>
<td>Returns the sine of an angle specified as a FIXED degree.</td>
</tr>
<tr>
<td>FxCosFx()</td>
<td>Returns the cosine of an angle specified as a FIXED degree.</td>
</tr>
<tr>
<td>FxTanFx()</td>
<td>Returns the tangent of an angle specified as a FIXED degree.</td>
</tr>
<tr>
<td>FxArcTanInt()</td>
<td>Returns an arctangent value as a FIXED angle.</td>
</tr>
<tr>
<td>FxArcTanFx()</td>
<td>Returns an arctangent value as a FIXED angle.</td>
</tr>
</tbody>
</table>

76.2.3
Table 76-1
PenPoint Fixed-Point Functions

Addition and Subtraction

Multiplication

Divison

Trigonometric Functions

continued
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FxCmpO</td>
<td>Compares two FIXED.</td>
</tr>
<tr>
<td>FxRoundToIntO</td>
<td>Rounds a FIXED number to a 32-bit signed integer.</td>
</tr>
<tr>
<td>FxRoundToIntSCO</td>
<td>Rounds a FIXED number to a 16-bit signed integer without overflow detection.</td>
</tr>
<tr>
<td>FxNegateO</td>
<td>Negate a FIXED.</td>
</tr>
<tr>
<td>FxAbsO</td>
<td>Takes the absolute value of a FIXED.</td>
</tr>
<tr>
<td>FxChopO</td>
<td>Returns the 16-bit signed integer part of a FIXED.</td>
</tr>
<tr>
<td>FxChopSC</td>
<td>Returns the 16-bit signed integer part of a FIXED.</td>
</tr>
<tr>
<td></td>
<td>No overflow detection.</td>
</tr>
<tr>
<td>FxFractionO</td>
<td>Returns the 16-bit fractional part of the absolute value a FIXED.</td>
</tr>
<tr>
<td>FxMakeFixedO</td>
<td>Make a FIXED with an S16 (integer) and a U116 (fraction).</td>
</tr>
<tr>
<td>FxIntToFxO</td>
<td>Convert a 16-bit signed integer into a FIXED.</td>
</tr>
<tr>
<td>FxBinToStrO</td>
<td>Converts a FIXED format value into an ASCII string in decimal.</td>
</tr>
<tr>
<td>FxStrToBinO</td>
<td>Converts a null-terminated ASCII string to a FIXED.</td>
</tr>
</tbody>
</table>
Part 9 / Utility Classes
Chapter 77 / Introduction

Overview 77.1 123
Features 77.2 124
Organization of This Part 77.3 125
Other Sources of Information 77.4 125

Chapter 78 / The List Class

List Concepts 78.1 127
Using List Messages 78.2 128
Creating Lists 78.3 129
Positioning Within a List 78.4 129
Adding, Removing, Getting, and Replacing Items 78.5 129
Counting Items 78.6 130
Removing All Items 78.7 130
Enumerating Items 78.8 130
Destroying Lists 78.9 131

Chapter 79 / Class Stream

Overview 79.1 133
Creating a Stream Object 79.2 134
Reading and Writing Streams 79.3 134
Reading and Writing with a Timeout 79.4 134
Setting the Current Byte Position 79.5 135
Flushing the Stream 79.6 136
Examples 79.7 136

Chapter 80 / The Browser Class

Browser Concepts 80.1 137
Browsers and Tables of Contents 80.1.1 137
Integrating a Browser into Your Application 80.1.2 138
Using clsBrowser 80.2 138
Creating a Browser Object 80.2.1 140
Getting the Current Selection 80.2.2 140
Setting the Current Selection 80.2.3 141
Making File System Changes 80.2.4 141
Refreshing the Browser Data 80.2.5 142
Changing Information Displayed 80.2.6 142
Changing the Sort Order 80.2.7 142
Expanding and Collapsing Sections 80.2.8 143
Reading and Writing the Browser State 80.2.9 143
Getting and Setting Browser Metrics 80.2.10 143
Changing the Browser Client 80.2.11 144
Navigating With the Browser 80.2.12 144
Getting the Internal Display Window 80.2.13 144
Browser Notification Messages 80.3 145
The Selection Changed 80.3.1 145
Bookmark Check Box Changed 80.3.2 145
Menu Messages 80.4 145
User Columns 80.5 145

Chapter 81 / File Import and Export

Concepts 81.1 147
Import Overview 81.1.1 148
Export Overview 81.1.2 148
Application Responsibilities 81.1.3 150
Handling the clsImport Messages 81.2 150
Responding to msgImportQuery 81.2.1 150
Responding to msgImport 81.2.2 151
Handling the clsExport Messages 81.3 152
How Export Happens 81.3.1 152
Responding to msgExportGetFormats 81.3.2 152
Responding to msgExportName 81.3.3 153
Responding to msgExport 81.3.4 154

Chapter 82 / The Selection Manager

Concepts 82.1 155
The Selection Manager 82.1.1 155
Selection Owners 82.1.2 156
Preserving the Selection 82.1.3 156
Selection Transitions 82.1.4 157
Determining What is Selected 82.2 157
Classes that Handle Selection 82.3 157
The Selection Class Messages 82.4 157
Messages from Clients to the SelectionManager 82.5 158
Setting the Selection Owner 82.5.1 159
Handling msgSellsSelected 82.5.2 159
Messages Sent to Selection Owners 82.6 159
Handling msgSelYield 82.6.1 160
Handling msgSelDemote and msgSelPromote 82.6.2 160
Handling msgSelDelete 82.6.3 160
Handling msgSelOptions and msgSelOptionTagOK 82.6.4 160
Beginning Move and Copy Operations clsEmbeddedWin Handles Selection Messages 82.6.5 160
Messages Passed to the Selection Manager 82.7 161
Finding the Selection Owners 82.7.1 161
Setting the Selection Owner 82.7.2 162
Observer Notification 82.8 163
Chapter 90 / Table Class

A Distributed DLL 213
90.1  213
Table Concepts 213
90.2  213
    Describing a Table 213
    Table Data Files 214
    Beginning Table Access 214
    Positioning in Tables 215
    Observing Tables 215
90.2.2  214
90.2.3  214
90.2.4  215
90.2.5  215
Shared Tables 215
90.3  215
    Ownership 216
    Access to the Table Object 216
    Concurrency 216
90.3.1  216
90.3.2  216
90.3.3  216
Using Tables in a Database 217
90.4  217
Using Table Messages 217
90.5  217
Defining a Table 218
90.6  218
Creating a Table Object 220
90.7  220
Observing Tables 220
90.8  220
Start Access 221
90.9  221
Using Semaphores 222
90.10  222
Adding Rows to a Table 222
90.11  222
Setting Data 223
90.12  223
Getting Data 223
90.13  223
Deleting a Row 224
90.14  224
Searching a Table 224
90.15  224
Getting Information About a Table 226
90.16  226
    Finding a Column Number 226
    Converting a Row Number to a Row Position 226
    Getting the Number of Columns in a Table 227
70.16.2  226
70.16.3  227
70.16.4  227
70.16.5  227
70.16.6  227
70.16.7  227
70.16.8  227
Getting a Table's State 228
90.17  228
Ending Access 228
90.18  228
Freeing a Table 228

Chapter 91 / The NotePaper Component 229

The clsNotePaper View 229
91.1  229
NotePaper Metrics 230
91.2  230
NotePaper Messages 231
91.3  231
NotePaper Data 232
91.4  232
NotePaper Data Items 234
91.5  234

List of Figures 149
81-1 Export Dialog 149
84-1 A Quick Help Window 186

List of Tables 128
78-1 clsList Messages 128
79-1 clsStream Messages 133
80-1 clsBrowser Messages 138
80-2 Browser Menu Messages 145
81-1 clsImport Messages 150
81-2 clsExport Messages 152
82-1 clsSelection Messages 158
83-1 clsXfer Transfer Types 166
83-2 clsXfer Functions 170
83-3 clsXferStream Messages 171
83-4 Transfer Buffer Types 174
84-1 clsQuickHelp Messages 187
84-2 PenPoint Gesture Font 188
86-1 Search and Replace Messages 198
87-1 clsUndo Messages 202
88-1 clsByteBuf Messages 208
89-1 clsString Messages 212
90-1 clsTable Messages 217
90-2 Table Column Data Types 219
90-3 Table Boolean Operators 225
90-4 Table Information Messages 226
91-1 clsNotePaper Messages 231
91-2 clsNPData Messages 233
91-3 clsNPItem Messages 234

List of Examples 184
84-1 Defining a Quick Help Resource 184
90-1 Creating a Table 221
90-2 Beginning Access to a Table 221
90-3 Using Table Semaphores 222
Chapter 77 / Introduction

The utility classes provide services to applications and other classes. Many of the classes documented here are subclassed to implement special features. The following classes are included in this part:

- The list class.
- The stream class.
- The file system browser.
- The import and export classes.
- The selection manager.
- The transfer class and protocol.
- The Quick Help class.
- The busy manager.
- The search and replace class and protocol.
- The undo manager.
- The byte buffer and string storage classes.
- The table component.
- The system component.
- The NotePaper component.

Overview

There are various functions which many objects in object-oriented systems have a use for, such as:

- Maintaining a list of items or table of data.
- Accessing a stream device.
- Browsing a directory or table of contents.

The PenPoint™ operating system supplies basic classes for each of these toolkit functions. They are each like a function library in ordinary procedural programming. The difference is that not only can your objects employ these utility objects, they can actually be a list, stream, or browser, by inheriting from them. If you look at the GO class diagram you will notice many sophisticated classes which inherit from these utility classes.
Features

- **clsList** provides a fundamental set of tools for creating and managing a list of 32-bit values. It is no coincidence that UIDs and pointers are also 32-bits long. You can use these objects to store lists of UIDs or pointers to larger structures and you can pass these list objects to other objects.

- **clsStream** provides the basic messages used to communicate with a stream device. Many other classes descend from clsStream, such as clsFileSystem (the File System) and clsSio (the Serial Port class).

- The browser allows you to create a browser window or a table of contents on screen so that the user can manipulate the files and directories or documents and sections.

- File import and export uses messages from the browser to import files as PenPoint documents and to export PenPoint documents as files.

- The selection manager provides a central manager that keeps track of the selection owner. The selection manager notifies observers when the selection changes.

- The transfer class provides the messages and functions that implement the PenPoint operating system transfer protocol, which objects can use to exchange data.

- The Quick Help API provides a simple way to provide help to users. When the user makes a question mark gesture on a window, the Quick Help manager locates the Quick Help resources associated with that window and displays the resources on screen.

- The busy manager allows applications to inform the user when a time-consuming operation is taking place, thereby reassuring the user that the machine is still running.

- The search and replace API provides the protocol and traversal driver to search and replace text strings in embedded objects.

- The undo manager enables applications to respond to the Undo command to undo user interface actions.

- **clsByteBuf** and **clsString** implement simple data objects which file byte arrays and null-terminated strings.

- **clsTable** provides a general-purpose table component using a row and column metaphor to implement random and sequential access to data in a file.

- **clsNotePaper, clsNPData, and clsNPItem** together provide most of the function necessary for a small note-taking application, including a generic data item protocol that allows arbitrary items in the notes.
Organization of This Part

This part is organized into 15 chapters:

- Chapter 77, this chapter, provides an introduction to the utility classes.

Each of the following chapters describes one utility class:

- Chapter 78, The List Class, describes the list class and how you use it to maintain lists.
- Chapter 79, Class Stream, describes the stream I/O subsystem.
- Chapter 80, The Browser Class, describes the API for the file browser.
- Chapter 81, File Import and Export, describes the classes that you use to import files as PenPoint documents and to export PenPoint documents as files.
- Chapter 82, The Selection Manager, describes the API to the manager that controls selection ownership.
- Chapter 83, Transfer Class, describes the generalized data transfer mechanism.
- Chapter 84, Help, describes the Help notebook and the Quick Help API.
- Chapter 85, The Busy Manager, describes the interface that you use to indicate to the user that the machine is busy.
- Chapter 86, Search and Replace, describes the messages used to search and replace text in applications.
- Chapter 87, Undo, describes how to use the undo manager so that your application can respond to Undo commands.
- Chapter 88, Byte Buffer Objects, describes the byte buffer data object class.
- Chapter 89, String Objects, describes the string data object class.
- Chapter 90, Table Class, describes the concepts of the table component, how to use a table, sharing tables, and table messages.
- Chapter 91, The NotePaper Component, describes the classes that make up the NotePaper component, a very capable data/view system for note taking applications.

Other Sources of Information

There are hundreds of classes in the PenPoint operating system, many of which will also be of great use to you in developing your application. Other classes are described throughout the PenPoint Architectural Reference.

Not everything in PenPoint is object-oriented. Utility functions in the PenPoint C run-time library are documented in Part Eight: System Services, of this volume of the PenPoint Architectural Reference.

Datasheets for all utility class messages of the utility classes are in the PenPoint API Reference.
Chapter 78 / The List Class

clsList provides fundamental functions for maintaining lists of 4-byte values. Typically, a list contains either the UIDs of related objects or pointers. This chapter covers the following topics:

- The concepts of lists.
- How to use the list messages.

List Concepts

A list is an object that holds a collection of items. Many components in the PenPoint™ operating system use lists for keeping track of objects and exchanging information. Each item in the list is 32 bits in length; often this is a handle on (or a pointer to) a larger item. Lists have no semaphores or other forms of access control mechanisms.

The messages defined by clsList allow you to:

- Create and destroy lists.
- Add items to and remove items from a list.
- Replace items in a list.
- Find an item in a list.
- Count the items in a list.
- Remove all items from a list.
- Copy a part or all of the list to an array.

A simple picture of a list is a series of 4-byte cells that hold data. The list object maintains an index to the current item (position). This index is used by most messages that alter data in the list. To point to the beginning of the list, set position to the value 0; to point to the end of the list, set position to the number of members in the list (which you obtain through msgListNumItems).
### Using List Messages

Table 78-1 summarizes the clsList messages. clsList is a descendent of clsObject.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_LIST_NEW</td>
<td>Creates a new empty list.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_LIST_NEW</td>
<td>Initializes the LIST_NEW structure to default values.</td>
</tr>
<tr>
<td>msgListFree</td>
<td>P_LIST_FREE</td>
<td>Frees a list according to mode.</td>
</tr>
<tr>
<td>msgListAddItem</td>
<td>LIST_ITEM</td>
<td>Adds an item to the end of a list.</td>
</tr>
<tr>
<td>msgListAddItemAt</td>
<td>P_LIST_ENTRY</td>
<td>Adds an item to a list at pArgs-&gt;position.</td>
</tr>
<tr>
<td>msgListRemoveItem</td>
<td>LIST_ITEM</td>
<td>The list searches for pArgs in the list and removes the item if found.</td>
</tr>
<tr>
<td>msgListRemoveItemAt</td>
<td>P_LIST_ENTRY</td>
<td>Removes the item in the list at pArgs-&gt;position.</td>
</tr>
<tr>
<td>msgListReplaceItem</td>
<td>P_LIST_ENTRY</td>
<td>Replaces the item in the list at pArgs-&gt;position.</td>
</tr>
<tr>
<td>msgListGetItem</td>
<td>P_LIST_ENTRY</td>
<td>Gets the item in the list at pArgs-&gt;position.</td>
</tr>
<tr>
<td>msgListFindItem</td>
<td>P_LIST_ENTRY</td>
<td>Searches for pArgs-&gt;item in the list.</td>
</tr>
<tr>
<td>msgListNumItems</td>
<td>P_U16</td>
<td>Passes back the number of items in a list.</td>
</tr>
<tr>
<td>msgListRemoveItems</td>
<td>no arguments</td>
<td>Removes all of the items in a list.</td>
</tr>
<tr>
<td>msgListGetHeap</td>
<td>P_OS_HEAP_ID</td>
<td>Passes back the heap used by the list.</td>
</tr>
<tr>
<td>msgListEnumItems</td>
<td>P_LIST_ENUM</td>
<td>Enumerates the items in a list.</td>
</tr>
<tr>
<td>msgListCall</td>
<td>P_LIST_NOTIFY</td>
<td>Sends a message to each object in the list using ObjectCall().</td>
</tr>
<tr>
<td>msgListSend</td>
<td>P_LIST_NOTIFY</td>
<td>Sends a message to each object in the list using ObjectSend().</td>
</tr>
<tr>
<td>msgListPost</td>
<td>P_LIST_NOTIFY</td>
<td>Sends a message to each object in the list using ObjectPost().</td>
</tr>
<tr>
<td>msgListNotifyAddition</td>
<td>P_LIST_NOTIFY_ADDITION</td>
<td>Notifies observers that an item has been added to the list.</td>
</tr>
<tr>
<td>msgListNotifyDeletion</td>
<td>P_LIST_NOTIFY_DELETION</td>
<td>Notifies observers that an item has been deleted from the list.</td>
</tr>
<tr>
<td>msgListNotifyReplacement</td>
<td>P_LIST_NOTIFY_REPLACEMENT</td>
<td>Notifies observers that an item in the list has been replaced.</td>
</tr>
<tr>
<td>msgListNotifyEmpty</td>
<td>P_LIST_NOTIFY_EMPTY</td>
<td>Notifies observers that a list is now empty.</td>
</tr>
</tbody>
</table>

#### Class Messages

#### List Management Messages

#### Enumeration Messages

#### Observer Notification Messages
Creating Lists

To create a new list, send msgNew to clsList. msgNew takes a LIST_NEW structure that specifies:

- **fileMode**: A filing mode indicator, which specifies how the list object should file items if it receives msgFile. There are three filing modes:
  - `listFileItemsAsData`: File items as U32 data.
  - `listFileItemsAsObjects`: Send filing messages to items.
  - `listDoNotFileItems`: Don't file list items. Upon restore, the list will be empty.

Positioning Within a List

Each list object maintains a current position indicator. You can change the position by sending `msgListFindItem` to the list object. You must declare a LIST_ENTRY structure and specify:

- **position**: The place to start the search.
- **item**: The item you are searching for.

`msgListFindItem` returns the first position where it found the item. If the item was not found, `msgListFindItem` returns `stsNoMatch`.

Adding, Removing, Getting, and Replacing Items

When you have the position of an item, you can do any of the following:

- Add a new item at that position with `msgListAddItemAt`.
- Get the item at that position with `msgListGetItem`.
- Remove the item at that position with `msgListRemoveItemAt`.
- Replace the item at that position with `msgListReplaceItem`.

The four messages related with these tasks are similar. All messages require you to declare a LIST_ENTRY structure, which contains:

- **position**: A position within the list.
- **item**: A 32-bit list item.

To add an item to the list, send `msgListAddItemAt` to the list. You specify the item to add to the list and its position. The list manager adds the item at the specified position (placing it before the item that is currently at that position). The message returns the new item and its position.

To get an item from the list, send `msgListGetItem` to the list. You specify only a position. The message returns the item and its position.
To remove an item from the list, send msgListRemoveItemAt to the list. You specify only the position. The message returns the removed item and its position. The heap memory where the item was stored is freed.

To replace an item in the list, send msgListReplaceItem to the list. You specify the item to add to the list and its position. The message returns the old item and its position.

You can read through a list by starting with the first item (give position the value 0), and increment position for each item.

If you specify a position that is beyond the end of the list, the message uses the last item in the list; if you use a position beyond the end of the list in an add operation, the item is added to the end of the list. A good way to access the end of the list is to use the constant maxU16 for the position value.

### Counting Items

To get the number of items in a list, send msgListNumItems to the list. The message takes only a pointer to a U16 value that will receive the count.

### Removing All Items

Before you destroy a list, it is a good idea to remove all items from the list, thereby removing them from the heap so that you don’t have to clear the items from the heap by hand. To remove all items from the list, send msgListRemoveItems to the list. The message does not require any arguments.

### Enumerating Items

To copy the list, or a portion of the list, to an array, send msgListEnumItems to the list. When the message copies the items to the array, you can perform any operation on the items, such as sorting the list or sending messages to UIDs. You must declare a LIST_ENUM structure and specify:

- **max** The size of the array, if you have allocated one.
- **count** The number of items you are requesting.
- **plitems** A pointer to an array, if you have allocated one.
- **pNext** A pointer to a value that contains the current position in the list. Use the value 0 to start at the beginning of the list.

You can either create the array yourself or you can specify a null pointer for plitems, in which case msgListEnumItems will allocate an array for you. If you create an array that is smaller than count, msgListEnumItems will allocate a new array that can contain count items.

msgListEnumItems uses count to indicate the number of items that it returned.

If msgListEnumItems allocated an array, it returns the size of the array in max and the pointer to the new array in plitems. After each call to msgListEnumItems, it is a good idea to make sure that the pointer sent to the message is the same as...
the pointer returned. If the two are different, a new array was allocated. It is important that you note this, because it is up to you to de-allocate the array.

If the list is long, or you can't allocate a large amount of storage for your array, you can read the list in chunks (by specifying a small count value). When the message returns, pNext contains the index to the next item in the list. Usually pNext points to the end of the list, but if count was less than the size of the list, pNext points to count + 1.

**Destroying Lists**

To free a list, send msgDestroy to the list object as usual. This frees the list data structures but does not affect the items in the list.

If the items in the list are objects that you want to destroy along with the list object, then use msgListFree. The message arguments are in a LIST_FREE structure that contains:

- **key**  The object key for the list object.
- **mode**  A mode that specifies whether to free the items in the list or not.

There are two possible values for mode:

- **listFreeItemsAsData**  The items in the list should be treated like U32 data that doesn't need freeing.
- **listFreeItemsAsObjects**  The items in the list should be freed as objects.

clsList sends the items msgDestroy (with a nil key). All items in the list must be object UIDs.
Chapter 79 / Class Stream

clsStream is an abstract superclass that defines messages for common stream operations. A stream operation is one in which files or data items are treated as a series of individual bytes.

clsStream inherits from clsObject. Many classes descend from clsStream. Structures and PENPOINT\SDK\INC\STREAM.H defines the structures and macros clsStream uses.

Overview

clsStream is an abstract class; it does not implement the methods for its messages. It is up to the individual subclasses to implement the methods.

Any class that subclasses clsStream must implement the “descendant responsibility” messages listed in Table 79-1.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_STREAM_NEW</td>
<td>Initializes defaults for new stream object.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_STREAM_NEW</td>
<td>Creates a new stream object.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgStreamRead</td>
<td>P_STREAM_READ_WRITE</td>
<td>Reads data from stream.</td>
</tr>
<tr>
<td>msgStreamWrite</td>
<td>P_STREAM_READ_WRITE</td>
<td>Writes data to stream.</td>
</tr>
<tr>
<td>msgStreamReadTimeOut</td>
<td>P_STREAM_READ_WRITE_TIMEOUT</td>
<td>Reads data from stream with timeout.</td>
</tr>
<tr>
<td>msgStreamWriteTimeOut</td>
<td>P_STREAM_READ_WRITE_TIMEOUT</td>
<td>Writes to the stream with timeout.</td>
</tr>
<tr>
<td>msgStreamFlush</td>
<td>pNull</td>
<td>The stream flushes any buffered data.</td>
</tr>
<tr>
<td>msgStreamSeek</td>
<td>P_STREAM_SEEK</td>
<td>Sets the stream’s current byte position.</td>
</tr>
<tr>
<td>msgStreamBlockSize</td>
<td>P_STREAM_BLOCK_SIZE</td>
<td>Passes back the most efficient write block size for this stream.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions</th>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StdioStreamBind()</td>
<td>OBJECT</td>
<td>Returns a stdio file pointer bound to a stream object.</td>
</tr>
<tr>
<td>StdioStreamUnbind()</td>
<td>P_UNKNOWN (FILE *)</td>
<td>Frees the stdio file handle bound to a stream object.</td>
</tr>
<tr>
<td>StdioStreamToObject()</td>
<td>P_UNKNOWN (FILE *)</td>
<td>Returns the stream object bound to a stdio file pointer.</td>
</tr>
</tbody>
</table>
You can send these messages to any object whose class has implemented methods for them. The most common stream object is a file handle; clsFileHandle is a subclass of clsStream. For further discussion of the file system, see Part 7: File System.

Other stream objects are selection transfers, serial I/O, and the many kinds of services (printers, Out boxes, etc.).

Creating a Stream Object

Before you can write or read a stream, you must create a stream object. To create a stream object, send msgNewDefaults and msgNew to a clsStream subclass (because clsStream is an abstract class, you should not create instances of clsStream itself).

Reading and Writing Streams

To read or write stream data, send msgStreamRead or msgStreamWrite to a stream object. Both messages require you to declare a STREAM_READ_WRITE structure that specifies:

- numBytes: The number of bytes to read or write.
- pReadBuffer: A pointer to a buffer to receive the data, or containing data to be written. On msgStreamRead, the buffer must hold at least numBytes of data.

The STREAM_READ_WRITE structure passes back in count the number of bytes read or written.

If the number of bytes read or written (count) equals the number of bytes specified (numBytes), the messages return \( \text{stsOK} \).

If you read the end of the stream and the number of bytes read is greater than zero, msgStreamRead returns \( \text{stsOK} \). If you read the end of the stream and the number of bytes read is zero, msgStreamRead returns \( \text{stsEndOfData} \). If you specify zero for numBytes after you receive \( \text{stsEndOfData} \), msgStreamRead returns \( \text{stsOK} \).

Reading and Writing with a Timeout

The stream messages with timeouts are similar to the non-timeout messages. msgStreamRead and msgStreamWrite return when a certain number of bytes have been read or written (or when the end of the stream is reached). The timeout stream messages (msgStreamReadTimeOut and msgStreamWriteTimeOut) return when either of two conditions is met:

- A certain number of bytes have been read or written (or when the end of stream is reached).
- A specified amount of time elapses.
To read or write stream with a timeout, send `msgStreamReadTimeOut` or `msgStreamWriteTimeOut` to a stream object. Both messages take a pointer to a `STREAM_READ_WRITE_TIMEOUT` structure that contains:

- **numBytes**  The number of bytes to read or write.
- **pBuf**  A pointer to a buffer to receive the data, or containing data to be written. On `msgStreamReadTimeOut`, the buffer must hold at least `numBytes` of data.
- **timeOut**  A timeout value in milliseconds.

If the message completes successfully, it returns `stsOK` and passes back the number of bytes read or written in the `count` field of the `STREAM_READ_WRITE_TIMEOUT` structure.

If you read to the end of the stream and the number of bytes read is less than the number of bytes requested, `msgStreamRead` returns the warning status `stsTimeOutWithData`.

If the timeout expires before `numBytes` bytes were read or written, the messages return the warning status `stsTimeOutWithData`.

If the timeout expired and no data was read or written, the messages return the error status `stsEndOfData`.

### Setting the Current Byte Position

Some subclasses of `clsStream` can seek to a specific byte position in the stream. This is true for file operations and some buffered operations, but is obviously not possible when the stream originates from a serial port or a keyboard.

To get and set the current stream position, use the message `msgStreamSeek`. The message takes a `STREAM_SEEK` structure that contains:

- **mode**  The starting position for the seek. This can be any one of the following:
  - `streamSeekBeginning`  Seek is relative to the beginning of the stream.
  - `streamSeekEnd`  Seek is relative to the end of data.
  - `streamSeekCurrent`  Seek is relative to the current position of the stream.
- **offset**  The offset in bytes. This offset is a signed value. Positive offsets move the current byte position closer to the end of stream; negative offsets move it closer to the beginning of the stream.

If you just want to find out the current byte position, specify 0 as the `offset` value, relative to the current position.

If the subclass of `clsStream` does not support seeks, it should return `stsMessageIgnored`. There is no way for a client to find out ahead of time whether the stream supports seeks or not.

If `msgStreamSeek` completes successfully, it passes back a `STREAM_SEEK` structure including the following fields:
curPos  The current position in the stream, relative to the beginning.
oldPos  The old position, relative to the beginning.
eof    A BOOLEAN value that indicates whether the new position is at the end of data.

The following example code fragment shows msgStreamSeek passing back the current position:

```c
STREAM_SEEK ss;
OBJECT stream;
ss.offset = 0;
ss.mode = streamSeekCurrent;
status = ObjectCall(msgStreamSeek, stream, &ss);
Debugf("Seek: Old Position: %ld, New Position: %ld %s",
      ss.oldPos, ss.curPos, ss.eof ? "(EOF)" : ");
```

The following example code fragment shows msgStreamSeek setting the current byte position to 80 bytes after the beginning of the stream:

```c
STREAM_SEEK ss;
OBJECT stream;  // stream handle
ss.offset = 80;
ss.mode = streamSeekBeginning;
status = ObjectCall(msgStreamSeek, stream, &ss);
Debugf("Seek: Old Position: %ld, New Position: %ld %s",
      ss.oldPos, ss.curPos, ss.eof ? "(EOF)" : ");
```

You can't set the current byte position before the beginning of the stream or beyond the end of data. If you seek from the current position and specify a byte offset that is before the beginning of the stream, the new position is the beginning of the stream; if the offset is after the end, the new position will be the end of stream. However, if you specify seek relative to the beginning of the stream and pass a negative byte offset, or you specify seek relative to the end of data and pass a positive byte offset, msgStreamSeek returns stsBadParam.

### Flushing the Stream

Occasionally you need to wait for a buffer to be written out before you can continue processing (or shut down an application). To flush the stream buffer, send msgStreamFlush to the stream object. The message doesn't take any arguments. If the message succeeds in emptying the buffer, it returns stsOK. If the buffers do not empty after a timeout period, the message returns stsFailed. When you subclass clsStream, you must establish a timeout period.

### Examples

All the sample programs which file use msgStreamWrite in response to msgSave to save state, and use msgStreamRead in response to msgRestore to restore state. See the source code under PENPOINT\SDK\SAMPLE for simple examples of using these.
Chapter 80 / The Browser Class

The browser class, clsBrowser, allows you to create a browser window in your application. By making your application a client of a browser, the application receives messages when the user taps on items in the browser. A browser is very useful for file selection dialogs. Since the application hierarchy is part of the file system hierarchy, a browser can also display the state of a notebook or section. The Table of Contents of the Notebook and the Disks page of the Connections notebook are examples of the use of browsers.

**Browser Concepts**

A browser is a window that contains a list of files and directories on a particular volume. As with the Disks page of the Connections notebook, which is documented in the manual Using PenPoint, the user can scroll the browser window. If the user double taps on a directory, the directory expands to display its files and directories; another double tap on the directory collapses it down to its name.

The user can select the criterion for sorting files and directories. The user can also choose what information to show about files and directories.

clsBrowser allows your application to display and control a browser window, just like the Connections notebook. You can send commands to the browser to change the sort order, to display certain information, to go to or bring to the selection, to set the selection to a particular file system node, and so on.

Additionally, if your application is a client of a browser window, it receives notification messages when the user makes a selection, turns off the selection, or taps on a bookmark check box.

clsBrowser inherits from clsScrollWin, a UI Toolkit class that supports scrolling. You insert a browser window object into your application just like you would insert a scrollwin. The object that displays the browser contents is actually the client window inside the scrollwin. Usually you don't have to do anything to this "hidden" object; however if you need to modify it, you can get its UID by sending msgBrowserGetBrowWin to the browser object.

**Browsers and Tables of Contents**

A table of contents is a specialized form of a browser window. While a browser shows you the files and directories in a particular volume, a table of contents displays the sections and documents in a notebook.

When you create an instance of clsBrowser, you can specify whether you want to create a browser or a table of contents (in the tocView field of the BROWSER_NEW structure).
If you create a table of contents, the window displays a list of sections and documents, their page number, and the bookmark checkbox. You can choose whether to sort by name or by page number, and whether or not to display the bookmark check box or the icon for the application.

If you create a browser, whether a standard browser or a table of contents, the window displays a list of files and directories (the table of contents variation of clsBrowser interprets them as PenPoint documents). You can choose to sort the list by name, size, or date and whether or not to display the size, date, type, or icon.

### Integrating a Browser into Your Application

If you incorporate a browser into your application, you should consider the user interface guidelines for dialog boxes.

You can give your browser a menu bar similar to the menu on the Disks page of the Connections notebook by creating a menu bar (an instance of clsMenu) with commands such as Expand and Collapse, and adding the menu bar to your frame. If the browser is a floating frame, to follow the user interface guidelines you should create a command bar (an instance of clsCommandBar), to provide Apply and Apply & Close buttons, and a close corner for your browser dialog.

For more information about clsMenu and clsCommandBar, see Part 4: UI Toolkit. For more information about user interface guidelines, see the PenPoint User Interface Design Reference manual.

### Using clsBrowser

Table 80-1 lists the messages clsBrowser handles.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_BROWSER_NEW</td>
<td>Initializes the BROWSER_NEW structure to default values.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_BROWSER_NEW</td>
<td>Creates a new browser object.</td>
</tr>
<tr>
<td>msgBrowseCreateDir</td>
<td>nothing</td>
<td>Creates a directory at the selection.</td>
</tr>
<tr>
<td>msgBrowseByName</td>
<td>nothing</td>
<td>Sorts by name order.</td>
</tr>
<tr>
<td>msgBrowseByType</td>
<td>nothing</td>
<td>Sorts by type order.</td>
</tr>
<tr>
<td>msgBrowseBySize</td>
<td>nothing</td>
<td>Sorts by size order.</td>
</tr>
<tr>
<td>msgBrowseByDate</td>
<td>nothing</td>
<td>Sorts by date order.</td>
</tr>
<tr>
<td>msgBrowseExpand</td>
<td>nothing or P_FS_FLAT_LOCATOR</td>
<td>Expands the section or directory that the argument identifies. If the argument is pNull, expands all sections or directories.</td>
</tr>
<tr>
<td>msgBrowseCollapse</td>
<td>nothing or P_FS_FLAT_LOCATOR</td>
<td>Collapses the section or directory that the argument identifies. If the argument is pNull, collapses all sections or directories.</td>
</tr>
</tbody>
</table>
### Table 80-1 (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgBrowserRefresh</td>
<td>nothing</td>
<td>Refreshes the disk image the browser is displaying.</td>
</tr>
<tr>
<td>msgBrowserDelete</td>
<td>nothing or P_FS_FLAT_LOCATOR</td>
<td>Deletes selection if arg is pNull, otherwise deletes the file that the argument identifies.</td>
</tr>
<tr>
<td>msgBrowserRename</td>
<td>nothing or P_FS_FLAT_LOCATOR</td>
<td>Renames the selection if argument is pNull. Otherwise, renames the file that the argument identifies.</td>
</tr>
<tr>
<td>msgBrowserConfirmDelete</td>
<td>BOOLEAN</td>
<td>Sets a flag whether to confirm deletions within a browser (OBSOLETE).</td>
</tr>
<tr>
<td>msgBrowserExport</td>
<td>nothing</td>
<td>Puts the selection into export mode (OBSOLETE).</td>
</tr>
<tr>
<td>msgBrowserByPage</td>
<td>nothing</td>
<td>Sorts by page number.</td>
</tr>
<tr>
<td>msgBrowserWriteState</td>
<td>nothing</td>
<td>Writes the current browser expanded/collapsed state to a file.</td>
</tr>
<tr>
<td>msgBrowserReadState</td>
<td>nothing</td>
<td>Reads the browser expanded/collapsed state from a disk file.</td>
</tr>
<tr>
<td>msgBrowserSetSaveFile</td>
<td>P_FS_LOCATOR</td>
<td>Sets the file that the browser will save open/close state to.</td>
</tr>
<tr>
<td>msgBrowserGetMetrics</td>
<td>P_BROWSER_METRICS</td>
<td>Gets browser metrics.</td>
</tr>
<tr>
<td>msgBrowserSetMetrics</td>
<td>P_BROWSER_METRICS</td>
<td>Sets browser metrics.</td>
</tr>
<tr>
<td>msgBrowserUserColumnGetState</td>
<td>P_BROWSER_USER_COLUMN</td>
<td>Does nothing.</td>
</tr>
<tr>
<td>msgBrowserUserColumnSetState</td>
<td>P_BROWSER_USER_COLUMN</td>
<td>Sets the user column states in the browser for columns that are marked changed.</td>
</tr>
<tr>
<td>msgBrowserUserColumnStateChanged</td>
<td>P_BROWSER_USER_COLUMN</td>
<td>Notifies subclass when user checks a user column checkbox.</td>
</tr>
<tr>
<td>msgBrowserUserColumnQueryState</td>
<td>P_BROWSER_USER_COLUMN</td>
<td>Gets the user column state from subclass.</td>
</tr>
<tr>
<td>msgBrowserShowIcon</td>
<td>BOOLEAN</td>
<td>Controls icon field display.</td>
</tr>
<tr>
<td>msgBrowserShowButton</td>
<td>BOOLEAN</td>
<td>Controls button field display.</td>
</tr>
<tr>
<td>msgBrowserShowSize</td>
<td>BOOLEAN</td>
<td>Controls size field display.</td>
</tr>
<tr>
<td>msgBrowserShowDate</td>
<td>BOOLEAN</td>
<td>Controls date field display.</td>
</tr>
<tr>
<td>msgBrowserShowType</td>
<td>BOOLEAN</td>
<td>Controls type field display.</td>
</tr>
<tr>
<td>msgBrowserShowBookmark</td>
<td>BOOLEAN</td>
<td>Controls bookmark field display.</td>
</tr>
<tr>
<td>msgBrowserShowHeader</td>
<td>BOOLEAN</td>
<td>Controls column header display.</td>
</tr>
<tr>
<td>msgBrowserGoto</td>
<td>BOOLEAN</td>
<td>Takes true to goto, false to bring to the selection.</td>
</tr>
<tr>
<td>msgBrowserGotoBringto</td>
<td>P_BROWSER_GOTO</td>
<td>Takes P_BROWSER_GOTO. If pFlat is pNull, applies to selection.</td>
</tr>
<tr>
<td>msgBrowserUndo</td>
<td>nothing</td>
<td>Does nothing yet.</td>
</tr>
<tr>
<td>msgBrowserSetSelection</td>
<td>P_FS_FLAT_LOCATOR</td>
<td>Causes browser/TOC to select and display the given file system item.</td>
</tr>
<tr>
<td>msgBrowserSetClient</td>
<td>OBJECT</td>
<td>Sets the target of the browser client messages.</td>
</tr>
<tr>
<td>msgBrowserGetClient</td>
<td>P_OBJECT</td>
<td>Passes back the target of the browser client messages.</td>
</tr>
<tr>
<td>msgBrowserGetBaseFlatLocator</td>
<td>P_FS_FLAT_LOCATOR</td>
<td>Passes back the directory the browser is looking at.</td>
</tr>
<tr>
<td>msgBrowserSelectionPath</td>
<td>P_BROWSER_PATH</td>
<td>Passes back the full path of the selection.</td>
</tr>
<tr>
<td>msgBrowserSelection</td>
<td>P_FS_FLAT_LOCATOR</td>
<td>Passes back the flat locator of the selection.</td>
</tr>
</tbody>
</table>
**Creating a Browser Object**

To create a browser window, send `msgNewDefaults` and `msgNew` to clsBrowser. The messages take a `BROWSER_NEW` structure that contains:

- **base** A locator that indicates the starting point in the file system to browse.
- **client** The name of the client that is to receive the browser messages. Usually `client` contains `self`.
- **tocView** A BOOLEAN value that indicates whether the browser should take the appearance of a browser or a table of contents. If `tocView` is true, the browser object is a table of contents.

**Getting the Current Selection**

You can use browser messages to get a file system path to the current browser selection.

- To get a flat locator for the current selection, send `msgBrowserSelection` to the browser object.
- To get the full path of the directory that contains the current selection, send `msgBrowserSelectionDir` to the browser object.

The following three messages all take a pointer to a `FS_FLAT_LOCATOR` structure, which they use to return the path or name of current selection. Flat locators are explained in *Part 7: File System*.

- To get the full path of the selection, send `msgBrowserSelectionPath` to the browser object. This takes a pointer to a `BROWSER_PATH` structure.
• To get the name of the selection, send `msgBrowserSelectionName` to the browser object. This takes a pointer to a string.

• To get the UUID of the selection, send `msgBrowserSelectionUUID` to the browser object. This takes a pointer to a UUID.

### Setting the Current Selection

To specify which file system object should be displayed by the browser, send `msgBrowserSetSelection` to the browser object. The message takes a pointer to a `FS_FLAT_LOCATOR` structure that contains the path to the file system object.

When the message completes, the browser display scrolls and displays the specified file system object.

### Making File System Changes

You can use browser messages to create, rename, and delete directories or files.

• To create a new directory, send `msgBrowserCreateDir` to the browser object. `msgBrowserCreateDir` creates a directory in the directory that contains the current selection. The location of the new directory is unimportant, because it will be positioned according to the current sort order.

• To create a new document, send `msgBrowserCreateDoc` to the browser object.

• To rename a directory or file, send `msgBrowserRename` to the browser object. `msgBrowserRename` brings up a dialog box to rename the current selection or a file system node specified by a flat locator.

• To delete a directory or file, send `msgBrowserDelete` to the browser object. `msgBrowserDelete` deletes the current selection or a file system node specified by a flat locator.

`msgBrowserCreateDir` takes no arguments. `msgBrowserRename`, `msgBrowserExport`, and `msgBrowserDelete` take one argument; if that argument is null, the message affects the current selection; if the argument contains a pointer to a `FS_FLAT_LOCATOR` structure, the message affects the file system node indicated by the flat locator.

`msgBrowserCreateDoc` takes a pointer to a `BROWSER_CREATE_DOC` structure that contains:

- `docClass` The class of the document to create.
- `pName` A pointer to a string that contains the name of the new document.
- `xy` An `XY32` value that specifies where to place the new document. This value is meaningful only when `atSelection` is `false`. `clsBrowser` will create the new document as close to the `xy` position as possible.
- `atSelection` A `BOOLEAN` value that specifies whether to create the new document immediately after the current selection or at the location...
specified by xy. If atSelection is true, clsBrowser creates the new
document at the current selection. If false, uses the value in xy.

**Refreshing the Browser Data**

The browser doesn’t monitor the file system for changes. Occasionally you might
need to tell the browser to update its information about the state of the file
system. To do this, send `msgBrowserRefresh` to the browser object.

The message doesn’t take any arguments. When the message completes
successfully, the browser reflects the new file system state.

**Changing Information Displayed**

clsBrowser defines messages that allow you to select the information displayed by
the browser. You can elect to show or not show information by sending these
messages to the browser object:

- Send `msgBrowserShowIcon` to display the icon for the file or section.
- Send `msgBrowserShowSize` to display the size of the file or section.
- Send `msgBrowserShowDate` to display the date that the file or section was
  last modified.
- Send `msgBrowserShowBookmark` to display the bookmark check box for
  the file or section. This message applies only to table of contents windows.
- Send `msgBrowserShowType` to display the type of file.
- Send `msgBrowserShowHeader` to display the column heads at the top of the
  browser display.

All of these messages take a BOOLEAN value. If the value is true, the browser
displays the information; if the value is false, the browser doesn’t display the
information.

**Changing the Sort Order**

You can change the sort order of the information displayed by the browser by
sending these messages to the browser object:

- Send `msgBrowserByName` to sort the information by name.
- Send `msgBrowserBySize` to sort the information by size.
- Send `msgBrowserByDate` to sort the information by date.
- Send `msgBrowserByType` to sort the information by type.
- Send `msgBrowserByPage` to sort a table of contents by page number. (If the
  browser window displays a table of contents.)

These messages take no arguments.
Expanding and Collapsing Sections

When the user double taps on the name of an unexpanded section, the section expands to reveal the documents and sections contained in that section. If the user double taps on the name of an expanded section, the section collapses down to just its name.

You can programmatically expand and collapse the current selection or a specific file system node by sending msgBrowserExpand and msgBrowserCollapse to the browser object.

Both messages take a single argument. If the argument is null, these messages expand and collapse the current selection. If the argument contains a pointer to a FS_FLAT_LOCATOR structure, the message expands or collapses the file system node specified by the path.

By specifying a path, you can expand or collapse a file system node, whether or not that object is currently displayed. To scroll to that path, you must use msgBrowserSetSelection.

Reading and Writing the Browser State

The browser state specifies which directories or sections are expanded in the browser window. When the user turns to another page, you can save the browser state. Then, when the user turns back to the browser, you can display it in the same state as when the user turned away.

For example, the user might expand a section (to see its documents) and can turn to one of those documents. When the user turns back to the table of contents, the notebook restores the browser state so that the same section is expanded.

To save the browser state, send msgBrowserWriteState to the browser object. To restore the browser state, send msgBrowserReadState to the browser object. Neither message takes any arguments.

Usually the read and write state messages save the state in a file named BROWSTAT (in the process directory). However, you can change the file by sending msgBrowserSetSaveFile to the browser object. The message takes a pointer to a file system locator that contains the path to the new browser state file.

Getting and Setting Browser Metrics

The browser metrics save the type of information being displayed by a browser window. That is, whether the browser window includes the icon, size, date, type, or bookmark check box in its display.

You can set these values by sending msgBrowserSetMetrics or one of the msgBrowserShow... messages to the browser object.
To get the metrics, send `msgBrowserGetMetrics` to the browser object. Both `msgBrowserGetMetrics` and `msgBrowserSetMetrics` take a `BROWSER_METRICS` structure that specifies or returns:

- **showIcon**: Whether to display the icon.
- **showType**: Whether to display the item type.
- **showSize**: Whether to display the size.
- **showDate**: Whether to display the date.
- **showBookmark**: Whether to display the bookmark check box.
- **showHeader**: Whether to display the browser column heads.
- **computeRecursive**: Whether to compute recursive size for directories.
- **showIconButton**: Whether to display page turn buttons instead of icons (for TOC only).
- **sortBy**: A `SORT_BY` structure determining the field by which to sort items.
- **userColumn**: A subclass-definable array of `BROWSER_COLUMN` structures.
- **defaultColumn**: The default columns for the class.

The difference between sending `msgBrowserShow...` messages and setting the metrics is that the messages are usually sent by an external UI, such as a browser menu. If you subclass `clsBrowser`, you might want to intercept these messages.

### Changing the Browser Client

You can change the client of the browser by sending `msgBrowserSetClient` to the browser object. The only argument to the message is the UID of the new client.

When the message completes successfully, the browser window is left unchanged, only the new client receives the browser messages.

### Navigating with the Browser

Using a browser in TOC mode, the user can go to a particular document by tapping on its page number or icon, or can float a particular document by double tapping on its page number or icon.

You can do the same thing programmatically by sending `msgBrowserGoto` to the browser object for a table of contents. The message only takes one argument, a `BOOLEAN` value, which specifies whether the operation is a go-to or a bring-to. If value is `true`, the notebook turns to the document currently selected. If the value is `false`, the document currently selected is floated.

If there is no current selection when `msgBrowserGoto` is sent, the message is ignored.

### Getting the Internal Display Window

Usually you don’t have to do anything to a browser’s internal display window; `clsBrowser` creates the window and controls it. However if you need to access the browser window, you can request its UID by sending `msgBrowserGetBrowWin` to the browser object. The only argument to the message is a pointer to an `OBJECT`.
When the message completes successfully it returns \texttt{stsOK} and passes back the UID of the browser window.

### Browser Notification Messages

The client of a browser object receives notification messages when the user performs certain actions within the browser window. If you are a client of the browser, you might need to respond to these messages.

#### The Selection Changed

When the user makes a selection in the browser, it sends \texttt{msgBrowserSelectionOn} to its client. The message doesn't have any arguments, but you can always send \texttt{msgBrowserSelectionPath} (or a related message) to the browser to find out what was selected.

When the user selects something outside the browser (the browser no longer owns the selection), the browser sends \texttt{msgBrowserSelectionOff} to its client. This message has no arguments.

#### Bookmark Check Box Changed

When the user taps the bookmark check box in the table of contents, the browser sends \texttt{msgBrowserBookmark} to its client. The message takes a \texttt{BROWSER_BOOKMARK} structure that contains a locator (\texttt{loc}). The locator indicates the file for which the bookmark check box was set.

### Menu Messages

You can add a menu bar to your browser object. Table 80-2 lists the \texttt{clsBrowser} messages that you are likely to associate with items on a menu or other user interface.

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgBrowserExpand</td>
<td>Expands selection.</td>
</tr>
<tr>
<td>msgBrowserCollapse</td>
<td>Collapses selection.</td>
</tr>
<tr>
<td>msgBrowserGoTo</td>
<td>Go to the selection.</td>
</tr>
<tr>
<td>msgBrowserCreateDir</td>
<td>Pops up create dir dialog box.</td>
</tr>
<tr>
<td>msgBrowserRename</td>
<td>Pops up rename dialog.</td>
</tr>
<tr>
<td>msgBrowserDelete</td>
<td>Deletes selection.</td>
</tr>
<tr>
<td>msgBrowserRefresh</td>
<td>Renews file system data.</td>
</tr>
</tbody>
</table>

### User Columns

\texttt{clsBrowser} supports a column of text or checkboxes in addition to its usual display contents. Subclasses of \texttt{clsBrowser} can control the appearance of the column, the header above the column, and whether or not the checkboxes appear next to sections or documents or both.
This optional user column is enabled by specifying metrics for it in the userColumn field in BROWSER_METRICS. The subclass must respond to msgBrowserUserColumnQueryState messages self-sent by the browser so that clsBrowser can know whether to check the box or what text to display.
Chapter 81 / File Import and Export

File import and export are closely related to the browser. When the user selects a file other than a PenPoint document (for example, a text file on a connected disk) and copies or moves the file to the PenPoint computer, the PenPoint™ operating system prompts the user for the type of document to create from the file. Similarly, when the user copies or moves a document from a PenPoint computer to a connected volume, PenPoint prompts the user for the type of file to write.

This chapter covers the following topics:

- Export and import concepts.
- How to respond to import messages.
- How to respond to export messages.

**Concepts**

Most operating systems are designed so that the users run a single program and open and close files of a specific type from that program. In PenPoint, the user turns to a page that contains a particular document and the operating system finds and runs the correct application.

This reversal of perspective requires some translation when moving files from operating systems that have a program-data orientation to the PenPoint operating system, which has a document-application orientation.

When the user moves a file from a traditional operating system to PenPoint, the user must identify the application that will present the data. The browser’s **file import** mechanism presents the user with a list of available applications. When the user chooses an application, the file import mechanism creates an instance of the application and tells that instance to translate the data.

Similarly, when the user needs to give a document created under PenPoint to someone who uses a different computer, the user must translate the document’s instance data into a form that is understood by the other computer. The browser’s **file export** mechanism presents the user with a choice of file format translators.

Import and export involve the use of the move or copy protocol and user interface between two browsers:

- A disk viewer that presents the directory/file view of a foreign disk.
- A table of contents that presents the PenPoint document view.

The user simply drags an icon from one browser to another. The import and export mechanisms prompt the user to select the appropriate file translation on the fly.
Import Overview

When the user selects a file in a browser acting as a disk viewer and moves or copies the file to a location in the Table of Contents (TOC), the TOC browser examines the file's appAttrClass file system attribute. If it exists, then the TOC browser creates an application instance of that application class in the application hierarchy and copies the data. However, if the file does not have an appAttrClass attribute, then the TOC browser assumes it isn't a PenPoint document and needs to be imported. The user must specify the type of document to use for displaying the data.

The TOC browser imports the file by:

1. Querying all installed application classes to see if they can import the file by sending them msgImportQuery.
2. If an application responds affirmatively to the query, the browser adds it to a list of applications that can import the file.
3. The browser displays the import dialog box to the user, listing all applications that can import the document. The list will always contain at least one application, the Placeholder application, that stores the data as a stream of bytes. (If the user chooses the Placeholder application and later turns to the Imported document, the Placeholder will prompt the user to choose an application with which to activate the document.)
4. The user selects an application and taps Move or Copy, depending on the operation.
5. The browser creates a new instance of the application selected by the user. The new document is at the location in the TOC where the user dragged the file icon.
6. The browser sends msgImport to the new document, telling it the file to import.
7. The selected application class does whatever it has to do to translate the chosen file's data into its representation.

If everything succeeds, the user has created a new document in the TOC holding the converted file contents.

Export Overview

The user exports a document to a file by copying or moving a document from a TOC browser to a disk viewer. Typically, the user moves or copies a document from the Notebook Table of Contents to a the disk viewer directory in the Connections notebook. At this point clsApp takes over and performs the following tasks:

1. Activates the selected document, if it isn't currently active, by sending msgAppMgrActivate to its class.
2. Sends msgExportGetFormats to the document to query the document for the file formats that it can write.
3 The application passes back its export format types and the UID of an object that will translate the data. Usually this object is self, but applications can use this UID to identify a separate translator object that will do the work.

4 clsApp presents the export dialog box to the user, listing the export file formats provided by the application.

5 The user chooses a file format and name for the exported file, then taps the Export button.

6 The disk browser creates the destination file and sends `msgExport` to the translator. `msgExport` tells the translator which format to write and also gives the translator the handle on the destination file.

![Export Dialog](image-url)
Application Responsibilities

The clsBrowser messages that lead to import and export episodes are not really important to most application developers (although you can use them if you want to force import or export to take place). The important thing is that all applications that import or export files must respond to the messages defined by clsImport and clsExport.

clsImport and clsExport are abstract classes that define the messages used to communicate information about importing and exporting files.

There are no instances of clsImport or clsExport.

Handling the clsImport Messages

The messages and defines for clsImport are defined in IMPORT.H. Table 81-1 lists the clsImport messages.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgImportQuery</td>
<td>P_IMPORT_QUERY</td>
<td>Queries each app class to see if it is capable of importing the file.</td>
</tr>
<tr>
<td>msgImport</td>
<td>P_IMPORT_DOC</td>
<td>Initiates the import procedure.</td>
</tr>
</tbody>
</table>

Responding to msgImportQuery

When the user drags an icon from a disk viewing browser to a TOC browser, the TOC browser sends msgImportQuery to each installed application class.

msgImportQuery is sent to your application class, not to an instance of your application. This means that to receive this message, your application class must set the objClassMessage flag in its method table entry for msgImportQuery, as follows:

```c
MSG_INFO myAppClassMethods [] = {
    ...
    msgImportQuery, "MyImportQueryHandler", objClassMessage, 0
};
```

When your application class receives msgImportQuery, it should determine if it can import the file. The message arguments for msgImportQuery are a pointer to an IMPORT_QUERY structure that contains:

- **file**  An open file handle on the file. Your application class can use the file handle to read data from the file to see if import would succeed. The browser resets the file pointer before sending msgImportQuery to the next application class.

- **fileName**  The file’s name.

- **fileType**  The file’s type, a file system attribute of the file. These file types are defined in FILETYPE.H:

  - **fileTypeUndefined**  Indicates that the file type is undefined. (This will be the case for files from foreign operating systems which don’t use PenPoint file system attributes.)
fileTypeASCII  Specifies that the file contains ASCII data; each line in the file ends with a hard line break.

fileTypeASCIIsoftLineBreaks  Specifies that the file contains ASCII data; however, hard line breaks in the file are treated as soft line breaks.

fileTypeRTF  Specifies that the file contains Microsoft's RTF (Rich Text Format) data.

fileTypeTIFF  Specifies that the file contains TIFF (Tagged Image File Format) data.

fileTypePicSeg  Specifies that the file contains data in GO Corporation's picture segment format.

canImport  Passed back, a BOOLEAN value specifying whether the application can import the file format.

suitabilityRating  Passed back, a value from 0 to 100 indicating how suitable the file format is to the application. 0 indicates that the file is not suitable to the application; 100 indicates that the file is very suitable to the application; 50 is the average value.

Developers can register new file types with GO. If you have a special file type that you want to register, please contact GO Developer Support for more information.

Your application class can check the file type, analyze the file name (for example, the extension), and read the file contents to determine whether it can import the file. Your application class should pass back an indication of whether it can import the file in the canImport BOOLEAN field of the IMPORT_QUERY structure. If you set this to true, the browser will add your application to the import dialog.

\section{Responding to msgImport}

When the user chooses an application to import the file, the browser creates a document of that application, starts it up, and sends msgImport to the document. The message argument for msgImport points to an IMPORT_DOC structure which contains:

- file  An open handle on the file.
- fileName  The file's name.
- fileType  The file type, as described in the previous section.
- sequence  The sequence number of the destination.
- destHandle  The directory handle on the destination directory.

When your application receives msgImport, it should use the information in the IMPORT_DOC structure to read the data from the file and attempt to translate it into information that it can understand.

If your application can't import the file contents, then it should return an error. If clsApp gets an error from msgImport then it deletes the newly-created document.
If possible, it's better to detect that import is impossible when your application class receives msgImportQuery so that your application never shows up in the import list. However, you may not discover that the file can't be imported until you try (for example, if the file is corrupt or otherwise unreadable).

### Handling the clsExport Messages

The messages and defines for clsExport are defined in EXPORT.H. Table 81-2 lists the clsExport messages.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgExportGetFormats</td>
<td>P_EXPORT_LIST</td>
<td>Passes back the export format array from from the source of the export.</td>
</tr>
<tr>
<td>msgExport</td>
<td>P_EXPORT_DOC</td>
<td>Initiates export by the translator.</td>
</tr>
<tr>
<td>msgExportName</td>
<td>P_EXPORT_FORMAT</td>
<td>Passes back a possibly modified destination name from the translator.</td>
</tr>
</tbody>
</table>

### How Export Happens

The export protocol comes into play when the user moves or copies an icon from a TOC browser to a disk viewer. Typically, the user moves or copies a document from the Notebook Table of Contents to a disk viewer directory in the Connections notebook.

When the user releases the move or copy icon over a disk viewer, the disk viewer asks the selection owner for supported transfer types: the TOC viewer supplies clsExport as one of the data transfer types. This is how the disk viewer knows that the copy is an export operation and not a simple copy operation to back up a document onto disk.

This interaction with the selection and transfer protocols allows anything that can be moved or copied to invoke export, although only selections from the TOC currently support export.

The disk viewer sends the source of the copy msgExportGetFormats. The source of the copy is the TOC browser, which does not know what export formats the document supports. So the TOC browser returns stsExportActivateSource. This tells the disk viewer to activate the source of the selection. The disk viewer sends msgAppMgrActivate to the application class of the document that the user originally selected.

### Responding to msgExportGetFormats

After activating the selected document, the disk viewer then sends msgExportGetFormats to the document. Your application is responsible for responding to msgExportGetFormats with the file formats that it can write.
msgExportGetFormat's message argument points to an EXPORT_LIST structure that contains:

- **format** A pointer to an array of EXPORT_FORMAT structures. You must allocate this array from global memory. Each EXPORT_FORMAT structure describes one file format that your application can write.

- **numEntries** The number of EXPORT_FORMAT structures in the array.

You should fill in each EXPORT_FORMAT structure with:

- **translator** The UID of a translator that can convert the information from the source document type to the export file type. Usually the translator is self.

- **documentType** The source document type. This field is meaningful only when the translator field specifies a translator other than self. A translator might be able to translate a number of different document types.

- **exportName** A user-visible name for the export type.

- **exportType** The type of file for the export destination. These file types are defined in FILETYPE.H:
  - **fileTypeUndefined** Indicates that the file type is undefined. (This will be the case for files from foreign operating systems which don’t use Pen-Point file system attributes.)
  - **fileTypeASCII** Specifies that the file contains ASCII data; each line in the file ends with a hard line break.
  - **fileTypeASCIISoftLineBreaks** Specifies that the file contains ASCII data; however, hard line breaks in the file are treated as soft line breaks.
  - **fileTypeRTF** Specifies that the file contains RTF data.
  - **fileTypeTIFF** Specifies that the file contains TIFF data.
  - **fileTypePicSeg** Specifies that the file contains data in GO Corporation’s Picture Segment format.

You can use the translator field to identify a separate translator object that will do the work.

clsApp puts up an export dialog showing the export types (using the exportNames in the EXPORT_FORMAT array).

**Responding to msgExportName**

When the user selects an export type, clsApp sends msgExportName to the translator specified in the EXPORT_FORMAT structure. exportName contains the name of the source document; the translator, if necessary, can pass back a suggested destination file name in the same exportName field. For example, it might append .TXT to the end. You can ignore this message. Regardless, the user can write in a different name for the exported file.
The message argument for `msgExportName` is a pointer to an `EXPORT_FORMAT` structure, described above.

**Responding to `msgExport`**

When the user selects a particular export format, `clsApp` sends `msgExport` to that format's translator. The message argument for `msgExport` points to an `EXPORT_DOC` structure that contains:

- `exportType` A tag that indicates the export type in the `EXPORT_FORMAT` structure that the user selected.
- `source` A locator for the source document.
- `destination` An open file handle on the destination.
- `path` The path to the destination document.

The translator should use this information to export the information in the document into the file. It is the exporting application's responsibility to clean up any invalid file handles. If the export fails, this includes the destination file handle provided by `msgExport`.
Chapter 82 / The Selection Manager

The selection manager provides a mechanism for keeping track of what process owns the current selection, the data the user has selected for subsequent action such as copying or deleting. Because the selection can be owned by many different types of objects, the selection manager does not do any formatting.

The selection manager keeps track of the selection owner, even when the object that contains the selection is not on screen.

clsSelection defines messages that allow objects to request ownership of the selection. clsSelection can record the current selection owner before changing ownership to another object.

An instance of clsSelection—theSelectionManager—keeps track of the current selection owner and the preserved selection, if any. theSelectionManager is the only instance of clsSelection.

Topics covered in this chapter:
- Selection concepts.
- Determining the selection.
- Existing PenPoint classes that handle the selection.
- Selection messages that operate on the selection.
- Selection messages sent to selection owners.
- Selection messages passed to the selection manager.

Concepts

No more than one object can own the current selection at any one time. Objects can request ownership of the selection. There doesn’t always have to be a selection. If the user hasn’t selected anything, there is no selection owner.

The Selection Manager

The ownership of the selection is administered by theSelectionManager, the only instance of clsSelection. theSelectionManager has the following responsibilities:
- It handles the transition between the current selection owner and the new selection owner.
- It keeps track of the current selection owner.
- It sends messages to observers of theSelectionManager when selection ownership changes.
The `theSelectionManager` is the only instance of the selection class, `clsSelection`, which defines two categories of messages:

- Messages that provide function for `theSelectionManager`.
- Abstract messages for common operations performed by owners of the selection. These abstract messages fall into two categories:
  - Messages that an object or one of its ancestors must respond to (such as “delete the selection” or “yield the selection”). If these messages reach `clsObject`, it sends `msgNotUnderstood`.
  - Messages that an object and its ancestors have the option of ignoring (such as “make yourself the selection”). If these messages reach `clsObject`, it returns `stsIgnored`.

You must always use `ObjectCall()` when you pass messages to `theSelectionManager`.

### Selection Owners

Just about any object can own the selection, so long as it handles the required `clsSelection` messages. For example, when an object that owns the selection receives a message asking it to yield the selection, it must yield the selection and return.

Certain object types can’t own the selection, either because there is no way for them to receive messages from `theSelectionManager`, or because other objects cannot communicate with them. These object types are:

- Objects that do not have global scope.
- Global objects that cannot receive messages (those for which `objCapSend` is disabled).

Any attempt to give an object of this type ownership of the selection will return a scope violation status code, `stsScopeViolation`.

### Preserving the Selection

There are times when you need to preserve the current selection while allowing the user to make a selection in another window. Generally, the only time this is true is when using an option sheet. For example, the user might select some text in a text editor and then put up an option sheet. `theSelectionManager` preserves the text selection while the option sheet is up. The user can make a selection in the option sheet, do something with the selection. When the user closes the option sheet, the preserved text selection is restored.
Selection Transitions

While theSelectionManager requests one object to yield ownership and gives ownership to another object, there is a time when the owner of the selection is not defined. If another clsSelection message arrives in this time, problems could arise. To avoid these problems, some clsSelection messages return stsSelYieldInProgress while it is in transition from one owner to another. For example, if you request the UID of the current selection owner (by sending msgSelOwner to theSelectionManager) while ownership is in transition, msgSelOwner returns stsSelYieldInProgress.

Certain other messages, rather than return stsSelYieldInProgress, just wait longer than usual to complete, until the new owner is established. These messages are:

- msgSelSetOwner
- msgSelDelete

A deadlock situation can occur if you receive msgSelYield and immediately send msgSelSetOwner to theSelectionManager, with your UID as the object to own the selection. Eventually the block on the requests times out, allowing your object to continue.

Determining What is Selected

If you inherit from clsEmbeddedWin, you will probably receive selection messages at some time. If you present information on screen and do not inherit from a class that handles clsSelection messages (as described below), you or one of your component classes will need to:

- Handle clsInput messages to detect when the user makes a selection.
- Request the selection (by sending msgSelSetOwner to theSelectionManager and specifying self as the new owner).
- Determine the type of selection.
- Show the user what is selected.

It is up to your application to determine what is selected.

Classes that Handle Selection

Some classes already handle selection ownership for you. If you subclass these classes, you inherit their selection behavior. In particular, clsEmbeddedWin and its subclasses all have selection ownership behavior built in.

The Selection Class Messages

Table 82-1 lists the clsSelection messages. The SEL.H header file declares these messages. There are no msgNew or msgNewDefaults messages because theSelectionManager is the only instance of clsSelection ever created.
### clsSelection Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgSelSetOwner</td>
<td>OBJECT</td>
<td>Sets the selection owner.</td>
</tr>
<tr>
<td>msgSelSetOwnerPreserve</td>
<td>OBJECT</td>
<td>Sets the selection owner with the preserve option.</td>
</tr>
<tr>
<td>msgSelOwner</td>
<td>P_OBJECT</td>
<td>Passes back the selection owner.</td>
</tr>
<tr>
<td>msgSelPrimaryOwner</td>
<td>P_OBJECT</td>
<td>Passes back the primary selection owner (the preserved owner if any, else the current owner).</td>
</tr>
<tr>
<td>msgSelOwners</td>
<td>P_SEL_OWNERS</td>
<td>Passes back both selection and preserved owners.</td>
</tr>
<tr>
<td>msgSelChangedOwners</td>
<td>P_SEL_OWNERS</td>
<td>Notifies observers when either of the selection owners changes.</td>
</tr>
<tr>
<td>msgSelPromotedOwner</td>
<td>P_SEL_OWNERS</td>
<td>Notifies observers when the preserved owner has been promoted back to the selection owner.</td>
</tr>
<tr>
<td>msgSelYield</td>
<td>BOOLEAN</td>
<td>theSelectionManager requires the release of the selection.</td>
</tr>
<tr>
<td>msgSelDemote</td>
<td>nothing</td>
<td>Informs the owner that it is becoming the preserved owner.</td>
</tr>
<tr>
<td>msgSelPromote</td>
<td>nothing</td>
<td>Informs the preserved owner that it is becoming the owner.</td>
</tr>
<tr>
<td>msgSelSelect</td>
<td>nothing</td>
<td>Sets self to be the selection owner.</td>
</tr>
<tr>
<td>msgSelIsSelected</td>
<td>nothing</td>
<td>Returns TRUE if self is current selection owner.</td>
</tr>
<tr>
<td>msgSelBeginCopy</td>
<td>P_XY32</td>
<td>Initiate a copy operation.</td>
</tr>
<tr>
<td>msgSelBeginMove</td>
<td>P_XY32</td>
<td>Initiates a move operation.</td>
</tr>
<tr>
<td>msgSelCopySelection</td>
<td>P_XY32</td>
<td>The receiver should copy the selection to self at (x, y).</td>
</tr>
<tr>
<td>msgSelMoveSelection</td>
<td>P_XY32</td>
<td>The receiver should move the selection to self at (x, y).</td>
</tr>
<tr>
<td>msgSelDelete</td>
<td>U32</td>
<td>The selection owner should delete the selection.</td>
</tr>
<tr>
<td>msgSelRememberSelection</td>
<td>P_XY32</td>
<td>The receiver should “remember” the selection and place the “remembrance” at (x, y).</td>
</tr>
</tbody>
</table>

#### Messages from Clients to theSelectionManager

Clients send messages to theSelectionManager to set the selection owner and to get information about the selection owner.
Setting the Selection Owner

To make an object the selection owner, send msgSelSetOwner to the SelectionManager. msgSelSetOwner takes as its sole argument the UID of the object which is to become the selection owner. The SelectionManager may send msgSelSelect (described in “Embedded Window Messages,” below) to an object to ask it to become the selection owner. The object should respond by sending msgSelSetOwner with self as the argument.

When setting the selection owner, you can preserve the previous selection for later restoration by sending msgSelSetOwnerPreserve instead of msgSelSetOwner to set the new selection owner. For example, option sheets use msgSelSetOwnerPreserve when the user makes a selection within the option sheet. This preserves the original selection (the item to which the options apply) so that the option sheet can later restore the original selection and apply the options to it.

Both msgSelSetOwner and msgSelSetOwnerPreserve return stsScopeViolation if the object specified as the argument cannot receive messages from other objects (either because the object is not a global object or because its capability flags prevent it from receiving messages). In this case, the selection owner does not change.

Handling msgSelIsSelected

When your object receives msgSelIsSelected, it should return a BOOLEAN value that indicates whether it owns the selection or not. The message has no arguments.

If you don’t inherit from an ancestor that handles msgSelIsSelected, the easiest way to handle the message is to send msgSelOwner to the SelectionManager. msgSelOwner sends back the UID of the current selection owner. If the returned UID matches self, return true to msgSelIsSelected; otherwise return false.

Messages Sent to Selection Owners

The SelectionManager sends certain messages to the selection owner. A class which can own the selection (or one of its ancestor classes) must handle these messages.

If no one responds to the required messages, clsObject sends msgNotUnderstood to self.

These messages perform the following functions:

- Tell the owner to give up the selection (msgSelYield).
- Tell the owner to demote the selection to the preserved selection or to promote the selection from preserved selection (msgSelDemote and msgSelPromote).
- Tell the owner to delete the selection (msgSelDelete).
- Tell the owner to display the option sheet for the selection (msgSelOptions).
- Tell the owner to begin a move or copy operation (msgSelBeginMove and msgSelBeginCopy).
Handling msgSelYield

The SelectionManager sends you msgSelYield when you no longer own the selection. The message has a single argument, a BOOLEAN value that indicates whether you lost the selection or the preserved selection. If the argument is true, yield the selection; if the argument is false, yield the preserved selection.

You should always return stsOK.

Handling msgSelDemote and msgSelPromote

If you are the selection owner and a client sends msgSelSetOwnerPreserve to the SelectionManager, the SelectionManager sends you msgSelDemote to inform you that your selection has been preserved.

If that client later sends msgSelSetOwnerPreserve to the SelectionManager with a null argument, the SelectionManager sends you msgSelPromote to inform you that your selection has been restored.

You can respond to msgSelDemote by graying the selection (or some other response that shows the user that the selection is preserved, but not current).

These messages are essentially informative. You should always return stsOK.

If you maintain your own selection status, you can use these messages to update your status indicators.

Handling msgSelDelete

If you receive msgSelDelete and you have the selection, delete the current selection. The message argument is a U32 that represents flags that specify the visual behavior for your object after the delete. The flags specify:

SelDeleteReselect Display a selection after deleting the current selection.
SelDeleteNoSelect Show no selection after deleting the current selection.

After you receive and handle msgSelDelete, you still own the selection.

Handling msgSelOptions and msgSelOptionTagOK

If your object receives msgSelOptions, it should activate the option sheet for the current selection. The message has no arguments.

An option sheet sends msgSelOptionTagOK to your object to check if its options can be applied to the current selection. The message passes an option sheet tag. If your object receives msgSelOptionTagOK, it should examine the option sheet tag and see if the options can be applied to the selection.

Beginning Move and Copy Operations

The user can start a move or copy operation by making a selection and either holding the pen on the selection or tapping on the Move or Copy commands in the Edit menu. Either of these actions sends msgSelBeginMove or msgSelBeginCopy to the selection owner.
If you own the selection and receive either of these messages, you should obey the rules of the Embedded Window move/copy protocol. Usually, you send the message to your ancestor (clsEmbeddedWin) which handles the protocol for you. For a detailed explanation of the Embedded Window move/copy protocol, see Chapter 9, Embedded Windows, in Part 2: The Application Framework of volume I.

**clsEmbeddedWin Handles Selection Messages**

While you can handle all of the selection messages yourself, clsEmbeddedWin provides message handlers for most of the selection messages. If your class inherits from clsEmbeddedWin, you can simply pass the messages to your ancestor.

The metrics of an embedded window object contains information about whether it should preserve the selection or not before taking the selection. The client that creates the embedded window sets this style information.

If you inherit from clsEmbeddedWin, you should pass msgSelSelect to your ancestor. The message will trickle up to clsEmbeddedWin, which handles the message. clsEmbeddedWin examines the Embedded Window metrics for the selection style. If the selection style is ewSelect, clsEmbeddedWin sends msgSelSetOwner to the SelectionManager; if the style is ewSelectPreserve, clsEmbeddedWin sends msgSelSetOwnerPreserve to theSelectionManager. If the selection style is ewSelectUnknown, clsEmbeddedWin runs up the window hierarchy to find a style that is not ewSelectUnknown. The messages msgSelSetOwner and msgSelSetOwnerPreserve are described later in this chapter.

**Messages Passed to the Selection Manager**

You pass messages to theSelectionManager for three reasons:

- You want to know who the current and preserved selection owners are.
- You want to set a selection owner.
- You want to set a selection owner and preserve the current selection owner.

**Finding the Selection Owners**

If you receive a message that instructs you to something with the selection, but you don’t know who the current owner is, you can pass msgSelOwner to theSelectionManager. The message takes a pointer to the OBJECT location that receives the UID of the current selection owner.

If the message completes successfully, it returns stsOK. The OBJECT location contains the UID of the selection owner.

To find out the current owner and the owner of the preserved selection, send msgSelOwners to theSelectionManager. The message takes a pointer to a SEL_OWNERS structure, which theSelectionManager uses to send back the UIDs of the owners. The structure contains:

- owner The UID of the selection owner. objNull is a valid value.
preservedOwner   The UID of the owner of the preserved selection. objNull is a valid value.
hasPreservedOwner  A BOOLEAN indicating whether preservedOwner is defined.

If the selection was between owners when you sent msgSelOwner or msgSelOwners, theSelectionManager might return stsSelYieldInProgress. The best thing to do is wait and try again.

### Setting the Selection Owner

To set the selection owner, send msgSelSetOwner to theSelectionManager. The message takes the UID of the object that will become the new selection owner.

### Preserving the Selection Owner

To set the selection owner and preserve the current owner, send msgSelSetOwnerPreserve to theSelectionManager. This message also takes the UID of the object that will become the new selection owner.

When theSelectionManager receives msgSelSetOwnerPreserve with a valid argument, it:

- Sends msgSelDemote to the current owner.
- Sets the current preserved owner to be the current owner.
- Sets the current owner to be the UID received in pArgs.
- Sends msgSelChangedOwners to observers of theSelectionManager.

### Restoring the Selection Owner

To restore the preserved selection owner, send msgSelSetOwnerPreserve to theSelectionManager with a null argument value.

When theSelectionManager receives msgSelSetOwnerPreserve with a null argument, it:

- Sends msgSelYield to the current owner, if one exists.
- Sends msgSelPromote to the current preserved owner.
- Sets the current owner to the current preserved owner.
- Sets the current preserved owner to null.
- Sends msgSelChangedOwners to observers of theSelectionManager.
Observer Notification

An object that wants to watch the current selection can be an observer of the SelectionManager. For example, property sheets need to know when the selection owner changes. If the user selected some text, requested the property sheet for text, and then selected a scribble object in a MiniNote document, the text property sheet should inform the user that it cannot affect the new selection (by making itself gray).

When the owner of the selection or the preserved selection owner changes, the SelectionManager sends msgSelChangedOwners to observers of the SelectionManager. When the preserved selection owner is promoted back to selection owner, the SelectionManager sends msgSelPromotedOwner to observers.

The argument to the message is a pointer to a SEL_OWNERS structure that contains:

- **owner** The UID of the new selection owner.
- **preservedOwner** The UID of the preserved selection owner.
- **hasPreservedOwner** A BOOLEAN indicating whether preservedOwner is defined.
Chapter 83 / Transfer Class

Although most messages transfer data between two objects, usually the objects have a relationship of ownership or inheritance. The transfer class, clsXfer, defines a general mechanism for exchanging information between two unrelated objects. This mechanism becomes particularly useful when responding to move or copy gestures. When two objects participate in a move or copy, they might have never communicated before.

Using clsXfer, the destination can send a message requesting the types of data that the source can transmit. When the source responds with the data types that it supports, the destination can decide whether it wants the data in one of the source's data types. If the destination can handle one of the source's data types, it asks the source to send the data using that data type.

clsXfer provides three different models for exchanging data, which allow for small or large transfers, one-shot or stream transfers, and formatted or unformatted transfers.

Topics covered in this chapter:
- Transfer concepts
- Agreeing on a transfer type
- One-shot transfers
- Stream transfers
- Using transfer messages
- Using transfer functions.

Concepts

The greatest single use of data transfers occur when the user moves or copies data from one location to another.

When an object receives a move or copy gesture, it is responsible for finding the owner of the selection and attempting to move or copy the selection to itself at the hot point of the gesture. The PenPoint™ operating system, through the application framework, allows users to attempt to move or copy any object they can select to any location they can draw a move or copy gesture.

This means that any object that responds correctly to application framework (embedded window) messages can be asked to move or copy information from any object to itself, perhaps from a class of object with which it has never communicated before.
The transfer class, clsXfer, provides protocols that allow unrelated objects to
determine if they both understand the same data types. If the objects can agree on
a data type, they use clsXfer protocols to transfer data using that data type.

General Scenario
In transferring data there are always two participants: the sender and receiver. The
**sender** is an object in a PenPoint task that sends data to the **receiver**, which is
another PenPoint object. In the transfer protocols, the receiver always requests
data from the sender.

When responding to move or copy gestures, the sender is the owner of the
selection. The receiver can find out the selection owner by sending **msgSelOwner**
to the selection manager, which sends back the UID of the selection owner.

All transfers begin the same way. The receiver sends a message to the potential
sender asking it to provide a list of the data transfer types that it can send. A data
**transfer type** identifies a specific data format (such as a string or a structure) and
transfer protocol. The receiver examines the sender’s list and finds the best transfer
type that it can use.

Tags for Data Transfer Types
The data transfer type is identified by a well-known tag that specifies the format of
the data. The data transfer type is associated with a transfer protocol, although the
protocol is not encoded in the tag.

**clsXfer** defines tags for several transfer types that are commonly used by PenPoint
components. The tags and uses for the data transfer types are listed in Table 83-1.
The table also lists the protocol implied by each transfer type. The following
section discusses protocols in more detail.

<table>
<thead>
<tr>
<th>Transfer Type Tag</th>
<th>Protocol</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xferString</td>
<td>one-shot</td>
<td>A string up to 256 bytes.</td>
</tr>
<tr>
<td>xferLongString</td>
<td>one-shot</td>
<td>A variable length string.</td>
</tr>
<tr>
<td>xferName</td>
<td>one-shot</td>
<td>A label (such as a GoTo button).</td>
</tr>
<tr>
<td>xferFullPathName</td>
<td>one-shot</td>
<td>A full path name.</td>
</tr>
<tr>
<td>xferFlatLocator</td>
<td>one-shot</td>
<td>A flat locator file path.</td>
</tr>
<tr>
<td>xferASCIIMetrics</td>
<td>one-shot</td>
<td>Metrics for a block of ASCII text (doesn't return the text).</td>
</tr>
<tr>
<td>xferRTF</td>
<td>stream</td>
<td>A stream of RTF data.</td>
</tr>
<tr>
<td>xferScribbleObject</td>
<td>one-shot</td>
<td>A scribble object.</td>
</tr>
<tr>
<td>xferPicSegObject</td>
<td>one-shot</td>
<td>A picture segment object (see Part 3, Windows and Graphics).</td>
</tr>
</tbody>
</table>

The clsXfer transfer types are defined in XFER.H.
You can use the `MakeTag()` macro (defined in GO.H) to define tags for other transfer types. `clsXfer` is the class for the PenPoint transfer type tags. If you use any other class, it implies that the data transfer type uses a client-defined protocol, defined by that class.

If you define your own transfer types, they must be understood by both the sender and receiver. Thus, you cannot use your own transfer types to transfer data with PenPoint components, unless you subclass a PenPoint component to handle your transfer types.

### Transfer Protocols

When the receiver finds an acceptable data transfer type in the list sent back by the sender, it initiates the data transfer protocol. The data transfer type implies the data transfer protocol that will be used to exchange data. There are three protocols for transferring data:

- **One-shot.** The receiver sends a single block of data to the receiver. One-shot transfers also identify the type of data being transferred (such as string, long string, or path name).
- **Stream.** The sender and receiver create a data stream. Stream transfers are used to transfer large amounts of data or continually arriving data.
- **Client-defined protocol.** The sender and receiver agree on a special protocol for transferring data. The client-defined protocol is usually defined by another class. For example, embedded windows (`clsEmbeddedWin`) establish their own protocol for transferring data.

### One-Shot Transfers

In one-shot transfers, the receiver asks the sender for a block of data by sending `msgXferGet` to the sender. The specific structure used with the message depends on the transfer type. `XFER.H` defines the following transfer buffer structures:

- `XFER_FIXED_BUF` Transfers a fixed length buffer of 256 bytes.
- `XFER_BUF` Transfers a variable length buffer (of up to 64K bytes).
- `XFER_ASCII_METRICS` Transfers ASCII metrics (this type is only used by the `xferASCII_metrics` transfer type).
- `XFER_OBJECT` Transfers an arbitrary object.

Each structure includes a `TAG` field representing the transfer type being used and a buffer or pointer to a buffer for the data to be transferred. All structures also include a `U32` value that the receiver can use to communicate information to the sender, such as more specific information on the requested data.

If you create other transfer types, you can also define other buffer structures. Your transfer types will be known only to your senders and receivers, unless you make them available to other application developers by publishing information about them.
When the sender receives `msgXferGet`, it should move the information into the buffer and return `stsOK`.

### Stream Transfers

In stream transfers, the receiver and sender must create and initialize a stream. When the stream is established, the sender and receiver use `clsStream` messages to read from and write to the stream (usually `msgStreamRead` and `msgStreamWrite`).

Because the steps that the receiver and the sender take to create a stream are so common, `clsXfer` defines a set of functions that do most of the work for you.

Before we describe the functions, we should describe the stream itself.

### Streams

The stream consists of two stream objects; one belonging to the sender and one belonging to the receiver. The stream objects are instances of a private class that inherits from `clsStream`, they respond to `clsStream` messages, but have additional features to support the transfer protocol.

The two stream objects share a common **stream buffer**, a shared storage area. The sender's stream object handles `msgStreamWrite` by copying data from the sender's buffer into the stream buffer. The receiver's stream object handles `msgStreamRead` by copying data from the stream buffer into the receiver's buffer.

### Stream Protocols

There are two stream protocols variations that you can use. The variations are:

- **Blocking protocol**, in which `clsXfer` blocks the sender when its buffer is full and releases the sender when the receiver empties the buffer sufficiently for it to continue. Blocking protocol works only when the sender and receiver are in separate tasks.

- **Producer protocol**, in which `clsXfer` communicates with an object (called a **producer**) that works on behalf of the sender to manage the transmission. When the sender and receiver are in the same task, they must use producer protocol (or limit their transfer to 64K bytes of data).

### Blocking Protocol

In stream transfers, the stream uses an intermediate buffer to store the data being transferred. The sender specifies the size of the stream buffer when it calls `XferStreamAccept`.

Usually, blocking protocol is fairly straightforward. The sender uses `msgStreamWrite` to copy data from its buffer into the stream. `clsXferStream` receives the data and stores it in the stream buffer. When the receiver sends `msgStreamRead` to its stream object, `clsXferStream` copies the data from the stream buffer to the receiver's buffer.
CHAPTER 83 / TRANSFER CLASS
Transfer Protocols

169

However, if the receiver doesn’t read enough data from the stream (either because it didn’t send msgStreamRead or it only read a portion of the data), the stream buffer can become full. Before data gets lost, clsXferStream blocks the sender’s task (by setting a semaphore).

When the receiver sends msgStreamRead and empties the stream buffer, clsXferStream clears the sender’s semaphore, allowing clsStream to send more data.

All this occurs while the sender’s msgStreamWrite is being handled. The sender doesn’t have to know anything about being blocked. When msgStreamWrite returns stsOK to the sender, all data has been transferred.

Blocking Protocol Deadlocks

Thus far we have assumed that the sender and receiver are in different tasks. If the sender and receiver are in the same task, however, blocking protocol presents some synchronization problems. If the sender and receiver are in the same task and clsXfer blocks the sender’s task, the receiver is blocked also. A deadlock exists; the sender is blocked until the receiver removes information from the buffer, but the receiver is blocked because it is in the same task as the sender.

clsXfer does as much as it can to avoid full-buffer deadlocks. If the sender and receiver are in the same task, and there is no producer, and the buffer is smaller than the data to be transferred, clsXfer will allocate additional space for the stream transfer buffer (up to 64K). However, if there is more than 64K to be transferred, this will not work because the maximum size of a stream transfer buffer is 64K.

Producer Protocol

Producer protocol avoids the blocking protocol’s deadlock situation by making the producer keep track of the amount of information that has been transferred. clsXfer communicates with the producer to let it know when it can copy more information into the transfer buffer. The producer works for the sender; it can be a separate object or it can be the sender.

The receiver initiates the transfer by sending msgStreamRead to its stream object. The stream object knows that there is a producer, so it sends msgXferStreamWrite to the producer. The producer uses msgStreamWrite to copy the sender’s data to the stream buffer. If the stream buffer is unable to accommodate all of the data, msgStreamWrite returns to the producer and passes back the number of bytes that it accepted. The producer must remember the location of the last byte accepted.

The producer then returns msgXferStreamWrite and the receiver’s stream object copies the stream data to the receiver’s buffer. If it reaches the end of data, it sends msgXferStreamWrite to the producer again. The producer sends msgStreamWrite to copy more data to the stream buffer.
Note that the sender never issues a message during the transfer. The producer acts as the sender's agent to handle the transfer. Of course the producer has to know what buffer the sender wanted to be transferred. There are two ways to accomplish this:

- When the sender initially creates the producer, part of the instance data can be the address of the send buffer.
- The receiver can use msgXferStreamSetAuxData to store auxiliary data in the stream (such as the address of the sender's buffer). The producer can send msgXferStreamAuxData to the stream to read the data.

Eventually the producer will have sent all the data from the sender's buffer. To terminate the stream, the producer sends msgFree to the stream object. When the stream is freed, the receiver can read the remaining data from the stream buffer, but when there is no more data, the receiver gets an EOF. The receiver should free its end of the stream.

If the receiver needs to abort a transfer that is in progress, it can free its stream object. When it does so, the sender's stream object sends msgXferStreamFreed to the producer. The producer should free its stream object.

### Client-Defined Protocols

Client-defined protocols can encompass a wide range of data transfer schemes, from clones of the one-shot or stream protocols to specialized transfer methods that use a transfer medium other than shared buffers or message data.

As described above, if the sender and receiver agree on a transfer type that is defined by a class other than clsXfer, the receiver should initiate the transfer using the protocol defined by that class.

For one example of a client-defined protocol, see the embedded window move/copy protocol described in Chapter 9, Embedded Documents, in Part 2: Application Framework of volume I.

### The Transfer Functions and Messages

Because the steps used to start and perform a transfer are common, no matter which transfer type you finally agree on, clsXfer defines a set of functions that perform most of the work for you.

Table 83-2 lists the clsXfer functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XferMatch()</td>
<td>The receiver calls XferMatch() to find a mutually acceptable data transfer type.</td>
</tr>
<tr>
<td>XferListSearch()</td>
<td>Searches two sets of data transfer types for a match.</td>
</tr>
<tr>
<td>XferAddIds()</td>
<td>Adds data transfer types to a list of acceptable types.</td>
</tr>
<tr>
<td>XferStreamConnect()</td>
<td>A receiver calls this function to create a stream connection to a sender.</td>
</tr>
<tr>
<td>XferStreamAccept()</td>
<td>Called by sender in response to msgXferStreamConnect.</td>
</tr>
</tbody>
</table>
Table 83-3 lists the clsXferStream messages. Generally, these messages are used by the clsXferStream functions; most clients shouldn’t need to use them.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgXferList</td>
<td>OBJECT</td>
<td>Ask sender for its list of data transfer types.</td>
</tr>
<tr>
<td>msgXferGet</td>
<td>lots of things</td>
<td>Sent by a receiver to get one-shot data transfer information.</td>
</tr>
<tr>
<td>msgXferStreamConnect</td>
<td>XFER_CONNECT</td>
<td>Sent to the sender to ask it to link the sender’s and receiver’s pipes.</td>
</tr>
<tr>
<td>msgXferStreamAuxData</td>
<td>PP_UNKNOWN</td>
<td>Passes back auxiliary information associated with the pipe.</td>
</tr>
<tr>
<td>msgXferStreamSetAuxData</td>
<td>P_UNKNOWN</td>
<td>Stores arbitrary client data with the pipe.</td>
</tr>
<tr>
<td>msgXferStreamWrite</td>
<td>STREAM</td>
<td>Asks the sender to write more data to the stream.</td>
</tr>
<tr>
<td>msgXferStreamFreed</td>
<td>STREAM</td>
<td>Sent to the sender when the receiver’s side of the stream has been freed.</td>
</tr>
</tbody>
</table>

**Establishing a Transfer Type**

The job of establishing a transfer type is made fairly easy by calling the clsXfer functions. However, before describing the functions, let’s take a look at how the receiver and sender interact to establish the transfer type.

1. The receiver calls XferMatch(), which creates two lists:
   - An array of the transfer types that it can use.
   - An empty clsXferList object.
2. XferMatch() sends msgXferList to the potential sender. The messages arguments include the empty clsXferList object.
3. When the sender receives msgXferList, it uses the function XferAddIds() to add its transfer types to the list. The sender then sends msgXferList to its ancestor.
4. When the ancestor returns the message, the sender returns msgXferList to the receiver.
5. XferMatch() uses the XferListSearch() function to compare the list of returned transfer types against the array of acceptable transfer types provided by the receiver.
6. When the first match is found, XferMatch() places the matched transfer type in the location specified by the receiver and returns stsOK.

The order the array of transfer types and the order of the transfer types in the clsXferList object is extremely important. XferAddIds() adds transfer types to the end of the clsXferList object. XferListSearch() starts at the beginning of both the array of transfer types and the clsXferList object.

If the receiver has a preferred transfer type, it should put it at the beginning of the transfer type array.
If the sender (or more importantly one of its ancestors) has a strong preference for a transfer type, it should use clsList messages to insert the transfer type at the beginning of the list object (XferAddIds() adds transfer types to the end of the list).

### Requesting Transfer Types

The process of looking for a transfer type is made fairly simple by the XferMatch() function. The prototype for XferMatch() is:

```c
STATUS EXPORTED XferMatch(
    OBJECT sender,
    TAG ids[],
    SIZEOF idsLen,
    P_TAG pId
);
```

The parameters to XferMatch() are:

- **sender** The UID of the potential sender of data.
- **ids** An array of transfer types familiar to the receiver.
- **idsLen** The number of elements in the ids array.
- **pId** A pointer to a TAG that will receive the returned transfer type.

If XferMatch() does not find a matching transfer type, it returns stsNoMatch. On receiving stsNoMatch, the receiver should call ancestor with the message that caused it to call XferMatch() (usually msgGWinGesture). This enables the receiver's ancestors to attempt to find a transfer type.

For example, clsText intercepts msgGWinGesture and uses XferMatch() to determine if the selection owner can send text data. If the selection is actually an object or application, XferMatch() returns stsNoMatch. clsText sends the msgGWinGesture to its ancestor. Eventually clsEmbeddedWin receives the message, which is able to move an object or application.

### Listing Transfer Types

To get a list of the available transfer types from the receiver, most clients use the XferMatch() function. However, if you need to provide special transfer type handling, you can send msgXferList on your own.

Before you send msgXferList, you need to do two things:

- You must identify the object that is the potential sender.
- You must create a transfer list (by sending msgNew to clsXferList).

Then you can send msgXferList to the sender; the only argument for the message is the UID of the transfer list.

### Adding a Transfer Type to a List

If you are acting as a sender and receive msgXferList, you must add your transfer types to the list indicated by pArgs. To make this job easier, you can call XferAddIds(). The prototype for XferAddIds() is:
The function sends `msgListAddItem` to the list for each transfer type in the array. Remember that the order of the transfer types in `ids` is important. `XferAddIds()` adds the transfer types to the end of the list object as they appear in `ids`.

After adding your transfer types to the list, you must send it to your ancestor. When `msgXferList` returns from your ancestor, the transfer list object contains the transfer types supported by your class and all your ancestors. At this point you can return from `msgXferList`.

### Searching a Transfer Type List

When the sender and its ancestors return `msgXferList`, the `XferMatch()` function calls `XferListSearch()`, which searches the transfer list returned by `msgXferList`, looking for a transfer type that matches one in an array of acceptable transfer types. The first array element is compared to all list elements, then the second array element, until every array element has been tried. Thus, the array should contain the optimal transfer types first.

If your class provides special operations, you can call `XferListSearch()` yourself. The prototype for `XferListSearch()` is:

```c
STATUS EXPORTED XferListSearch(
    OBJECT listObject,
    TAG ids[],
    SIZEOF idsLen,
    p_TAG pId
);
```

The parameters are:

- `listObject`  The transfer list object sent to the sender.
- `ids`  The list of transfer types that the receiver supports.
- `idsLen`  The number of elements in the `ids` array.
- `pId`  A pointer to the location to receive the matched transfer type.

### Performing One-Shot Transfers

If the transfer type is a one-shot type transfer, the receiver sends `msgXferGet` to the sender. The message takes a pointer to a transfer buffer. Transfer buffers can take on a number of forms. XFER.H defines the following types of transfer buffers:

- **XFER_FIXED_BUF** transfers up to 256 bytes in a single transfer.
XFER_BUF transfers a variable length buffer (of up to 64K bytes).

- XFER_ASCII_METRICS is used to transfer ASCII metrics. This type is only used by xferASCIIMetrics transfer type.
- XFER_OBJECT is used to transfer an arbitrary object.

The type of transfer buffer depends on the transfer type. Table 83-4 lists the various one-shot transfer types and the corresponding buffer type. (The information in this table was obtained from XFER.H, which includes the transfer buffer type as a comment when defining the transfer types. If you define your own transfer types, it is a good idea to follow this practice.)

<table>
<thead>
<tr>
<th>Transfer Type</th>
<th>Buffer Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xferString</td>
<td>XFER_FIXED_BUF</td>
</tr>
<tr>
<td>xferLongString</td>
<td>XFER_BUF</td>
</tr>
<tr>
<td>xferName</td>
<td>XFER_FIXED_BUF</td>
</tr>
<tr>
<td>xferFullPathName</td>
<td>XFER_FIXED_BUF</td>
</tr>
<tr>
<td>xferFlatLocator</td>
<td>XFER_FIXED_BUF</td>
</tr>
<tr>
<td>xferASCIIMetrics</td>
<td>XFER_ASCII_METRICS</td>
</tr>
<tr>
<td>xferRTF</td>
<td>(stream protocol)</td>
</tr>
<tr>
<td>xferScribbleObject</td>
<td>XFER_OBJECT</td>
</tr>
<tr>
<td>xferPicSegObject</td>
<td>XFER_OBJECT</td>
</tr>
</tbody>
</table>

If you define your own transfer types, you can use these transfer buffers or you can define your own transfer buffers. Of course, transfer types that you create are known only to your senders and receivers.

**Fixed-Length Buffer Transfers**

If the transfer type is xferString, xferName, xferFullPathName, or xferFlatLocator, the argument to msgXferGet is a pointer to an XFER_FIXED_BUF structure, which contains:

- **id** The transfer type.
- **buf** A 256 byte buffer that contains the data.
- **len** The length of the data in buf.

**Variable-Length Buffer Transfers**

If the transfer type is xferLongString, the argument to msgXferGet is a pointer to an XFER_BUF structure, which contains:

- **id** The transfer type.
- **pBuf** A pointer to a buffer that contains the data.
- **len** The length of the data in buf.
Performing One-Shot Transfers

The buffer indicated by pBuf must be shared. The sender should allocate the buffer by calling OSSharedMemAlloc() or by calling OSHeapBlockAlloc() with a shared heap such as osProcessSharedHeapId.

Whenever possible, the client that knows how big the data is should allocate the shared buffer.

When the receiver has read the data from the shared buffer, it should deallocate the buffer (a shared buffer does not have to be allocated and deallocated by the same task).

Whenever possible, the client that requested the data should deallocate the shared buffer.

When the message sends back XFER_BUF, the len field contains the length of the data in pBuf.

ASCII Metrics Transfers

If the transfer type is XferASCII_metrics, the argument to msgXferGet is a pointer to an XFER_ASCII_METRICS structure, which contains:

- **id** The transfer type.
- **first** The text index of the first character in the range.
- **length** The length of the text range.
- **level** The parts of text being transferred. The level enables word processors to apply the correct styles to the data moved into a document. The possible values are:
  - 0 ignore
  - 1 characters
  - 2 words
  - 3 sentences
  - 4 paragraphs

This example shows a receiver sending msgXferGet to a sender.

```c
STATUS GetShortString(
    P_TAG pID,       // pointer to ID from XferMatch
    P_MY_DATA pMyData)   // pointer to instance data
{
    XFER_FIXED_BUF fb; // Fixed buffer
    STATUS status;

    fb.id = *pID;
    fb.data = pMyData;
    status = ObjectSendUpdate(msgXferGet, pMyData->sender, &fb);

    if (status >= stsOK)
    {
        strncpy(pMyData->pFixedData, fb.buf, fb.len);
        pMyData->fixedLen = fb.len;
        return stsOK;
    }
    ...
}
```
RePLYING TO ONE-SHOT TRANSFERS

When the sender receives msgXferGet, it should move the information into the buffer and return the message. This example shows how the sender responds when it receives msgXferGet:

```c
// Handle msgXferGet for fixed data
STATUS ReplyFixedData( P_XFER_FIXED_BUF pArgs )
{
    P_MY_DATA *myData;  // pointer for data
    STATUS rstatus;     // returned status

    rstatus = stsOK;

    // Find data to transfer
    FindData(myData);

    if (myData->len > 256)
    {
        pArgs->len = 256;
        rstatus = stsTrunc;  // data truncated
    }
    else
    {
        pArgs->len = myData->len;
    }

    strncpy(pArgs->buf, myData->data, pArgs->len);

    return rstatus;
}
```

Performing Stream Transfers

If the transfer type is a stream transfer, the receiver and sender must create their halves of a stream.

Creating the Receiver's Stream

The receiver always creates its half of the stream first, by calling the function XferStreamConnect(). The function's arguments are:

- The UID of the sender.
- The transfer type.
- An optional pointer to client data for the sender. This client data might indicate more specifically what portions of the data the receiver is interested in. When the stream sends msgXferStreamWrite, it passes this pointer to the producer.
- A pointer to the location that receives the UID of the receiver’s stream object.

The function does the following:

1. Initializes and creates a stream object (by sending msgNewDefaults and msgNew to clsXfer).
2. Tells the sender to create its own stream object by using ObjectPostAsync() to send msgXferStreamConnect to the sender.
Performing Stream Transfers

3. Initializes the stream by sending msgXferStreamInit to its stream object. If the function completes correctly, it returns stsOK.

Creating the Sender's Stream

When the sender receives msgXferStreamConnect, it should create its half of the stream by calling the function XferStreamAccept. The function arguments are:

- The receiver's stream object (pArgs->stream in msgXferStreamConnect).
- The size of the stream transfer buffer (can be up to 64K bytes).
- An optional producer UID. If this argument is null, the stream uses blocking protocol; if this argument indicates an object, the stream uses producer protocol, with the indicated object as the producer.
- A pointer to the location that receives the UID of the sender's stream object.

The function:

1. Initializes the stream object (by sending msgNewDefaults to clsXfer).
2. Establishes that the stream object has a producer and identifies the receiver.
3. Creates the stream object (by sending msgNew to clsXfer).
4. Copies the UID of the stream object to the location specified by the caller.

If the function completes correctly, it returns stsOK.

Freeing the Stream

When the transfer is complete, the sender or the producer frees the stream by sending msgFree to its stream object. When clsXfer receives msgFree, it sends msgXferStreamFreed to the producer.

When the receiver gets msgXferStreamFreed, it frees its half of the stream.

Accessing the Stream's Auxiliary Data

The stream's data includes a pointer value for auxiliary data. The receiver and the sender can write and read this value to exchange information about the transfer.

To store information in the stream's auxiliary data, send msgXferStreamSetAuxData to the stream object. The message takes a pointer to any type of data. When the message completes successfully, it returns stsOK.

To read the stream's auxiliary data, send msgXferStreamAuxData to the stream object. The message takes a pointer to the location that receives the pointer stored in the stream.

When the message completes successfully, it returns stsOK and the location contains the pointer stored in the stream.
Connecting a Stream to a Producer

When the receiver in a stream transfer creates its half of the stream, it sends msgXferStreamConnect to the sender. The message takes an XFER_CONNECT structure that contains:

- id  The transfer type.
- stream  The UID of the receiver's stream object.
- clientData  A pointer to optional, client-specified data.

When the sender receives msgXferStreamConnect, it should create its stream object by calling the function XferStreamAccept().

Initializing a Stream

When the sender returns msgXferStreamConnect, the receiver should initialize the stream by calling msgXferStreamInit. The message takes no arguments.
There are two help facilities built into the PenPoint™ operating system:

- Quick Help, which provides object-specific reminders when the user:
  - Makes the question mark 0 gesture over an object.
  - Taps on the Help icon, then taps on an object.
- The Help notebook, which provides users with full on-line help documentation.

Application designers can add both types of help to their application.

Topics covered in this chapter:

- Concepts of Help and Quick Help.
- How to add help to the Help notebook.
- How to define Quick Help resources.
- How to use the Quick Help messages.

**Help Concepts**

There are two help components in PenPoint: Quick Help and the Help notebook. Quick Help provides a reminder mechanism for windows which support Quick Help. The user invokes Quick Help either by making the question mark ? gesture over a window, or by tapping the Help icon, and then tapping on a window. Both methods display a brief help message in the Quick Help window.

The Help notebook is the location in which the full on-line help documentation for the system and applications exists. It is a true notebook and has a table-of-contents, tabs, sections, etc. The user invokes the Help notebook by tapping the Help icon, then tapping the Help notebook button in the Quick Help window.

**The Help Notebook**

The Help notebook contains tutorial Help information. It is a true notebook that can contain any application—there is no requirement on the type of application that can be placed in this notebook. An application can even put a instance of itself in the Help notebook.

**Help Directories**

To add a help application to the Help notebook, create a HELP directory in your application directory (\PENPOINT\APP\MYAPP\HELP) and place directories containing help applications in this directory.
When the user installs your application, the installer copies the help applications into the Help notebook, just as stationery documents in a STATNRY directory are copied into the Stationery notebook.

If you don't want to create a help application, you can use clsTexteditApp to display your help text. To use the text editor:

1. Create an ASCII text file called HELP.TXT or a Rich Text Format (RTF) file called HELP.RTF that contains your help text.
2. Store the text file in a subdirectory of your document's HELP directory. The name of the subdirectory is the title of the document in the Help notebook.

You can provide more than one help document (or help application), but each requires its own subdirectory under the HELP directory.

HELP subdirectories that are application documents must be labeled with the class of the application. If a directory is not labeled with an application class when the installer copies it into the Help notebook, the installer labels it with clsTexteditApp (the MiniText application class). When the user turns to your help text, the help notebook activates the text as a text editor document.

The following directory tree shows the help for an application named MyApp. Under the directory named \MyApp is a HELP directory. The HELP directory contains two subdirectories for two separate help text files, Summary Help and Detailed Help.

Creating Help Text

If you use clsTexteditApp to display your help, your help text can be either an ASCII text file called HELP.TXT or an RTF file called HELP.RTF.

You can create an ASCII text file in any text editor or word processor. If you use a word processor, make sure that you save the file as a plain text document—not the word processor's format.

RTF is a document description language that allows you to transfer formatted documents among different word processors. You can create RTF files by editing your help text in a word processor that can save its documents as RTF files.

An additional benefit of using RTF for your help text is that you can include gesture characters in your documents, using the PenPoint Gesture font (GS80). For information about how to add the PenPoint Gesture font characters to your help text, see “Advanced Topics,” later in this chapter.
Quick Help Concepts

The system support for Quick Help consists of a well-known object, `theQuickHelp`, which is the window which displays the current help information. The Quick Help window is a floating window with two buttons, Done and Help notebook.

The default support in the system for Quick Help is based in the gesture window class, `clsGWin`. A Quick Help ID can be stored with a gesture window as a property of the object. When the question mark gesture gets to `clsGWin`, it will send this Quick Help ID to the Quick Help window. The Quick Help window will then attempt to find the resource with the Quick Help ID. If it finds the resource, it reads in the text and displays it.

The resource described above currently contains the strings for the title and summary (there is also a gesture string which is unused). It is the responsibility of the developer to provide the Quick Help resources and store them in the application or system directories.

To use the Quick Help facility, developers create Quick Help resources and associate them with PenPoint objects. When the user makes a help gesture over the object, `clsGWin` intercepts the gesture, and sends `msgQuickHelpShow` to put the Quick Help window on-screen and present the text to the user. When the user taps on another object while the Quick Help window is on-screen, the Quick Help window simulates the gesture.

For objects that do not descend from `clsGWin`, the Quick Help API enables clients to direct the Quick Help manager to display text from a help resource.

Adding Quick Help to an object that inherits from `clsGWin` requires two steps:

- You define a Quick Help resource in a resource file, either by saving the resource to disk from an application or by using the resource compiler to compile a resource definition file (see Part 11: Resources for more information).
- You store the ID portion of the Quick Help resource ID in the metrics of the `clsGWin` object.

When the user makes a Quick Help gesture on the object, `clsGWin` receives the gesture (if it isn’t intercepted) and sends messages to the Quick Help manager to display the Quick Help resource.

If the object does not inherit from `clsGWin`, adding Quick Help is almost as easy. After defining the Quick Help resource, you ensure that the object stores the Quick Help resource ID. When needed, the object sends messages to the Quick Help manager, directing it to display the Quick Help resource.

Quick Help Resources

The Quick Help resource for an object consists of three strings that contain:

- A title for the Quick Help window.
• A gesture string (currently unused; should be empty).
• The Quick Help summary.

The strings can either contain plain text or RTF strings. RTF is useful if you need to use multiple fonts or add other stylistic changes to your text. For example, an RTF string can include characters from the PenPoint Gesture Font. For information on adding PenPoint Gesture Font characters to your help strings, see the “Advanced Topics,” later in this chapter.

You define a Quick Help resource as a string-array resource that contains all of the Quick Help strings for the class.

The Quick Help resources for an application must be stored in the APPRES file for that application. The Quick Help resources for system-wide objects should be stored in the system resource file.

Quick Help and clsGWin

When you create an instance of an object that inherits from clsGWin, you and store a Quick Help resource ID in the gwin.helpID field for the clsGWin object.

If the user makes the Quick Help gesture on an object that inherits from clsGWin and no subclass intercepts the gesture, clsGWin receives the gesture and:

1. Uses the application UID to locate the application’s resource list.
2. Uses the Quick Help ID in gwin.helpID to identify the resource.
3. Sends clsQuickHelp messages to the Quick Help manager (theQuickHelp).

Quick Help without clsGWin

If you use a class that does not inherit from clsGWin, but you want to be able to display Quick Help for instances of that class, you can send Quick Help messages to theQuickHelp (the same way that clsGWin invokes Quick Help).

You can also use this technique if you want to intercept the Quick Help gestures and display the Quick Help information yourself.

The messages defined by clsQuickHelp allow clients to:

• Display Quick Help.
• Open and close the Quick Help window.
• Determine which Quick Help string to display.

“Advanced Topics,” later in this chapter, describes the Quick Help messages and how to use them.

theQuickHelp Object

There is only one Quick Help object in the system: theQuickHelp. theQuickHelp is defined when the PenPoint operating system is booted. You do not need to send msgNew to any class to create theQuickHelp (in fact, there is no well known UID for the Quick Help class).
Defining Quick Help Resources

You define the Quick Help resources in C language files that are compiled by the Resource Compiler. Before you read about defining Quick Help resources, you should be familiar with the material in Part 11: Resources.

Each resource is defined by a string resource that defines the title and summary strings. All of the Quick Help string resources are combined into a single string array resource.

Defining the Quick Help String Array

To define a Quick Help string array resource you define a static string array and a resource definition that points to the string array. Each string in the string array combines the title and summary strings for a single Quick Help resource. Each of the Quick Help strings in the array has the following format:

```c
static CHAR label-of-string[] = {
  // title
  "title text"|
  // summary
  "summary text"
  "more summary text"
};
```

The title and summary strings are combined into a single string, with a sequence of two vertical bar characters (II) separating the two parts of the string. Note that ANSI C concatenates consecutive strings, so the above template actually defines a single string.

All of the Quick Help strings for a class are combined into a single string array, plus a null string to indicate the end of the array. This string array is the data for the Quick Help resource for the class.

The resource definition has this format:

```c
static RC_INPUT res-label = {
  MakeListResId(class-UID, resGrpQhelp, list-number), // list-number is typically 0
  label-of-string-array, // Name of the string array
  0, // dataLength is not needed for strings
  resTaggedStringArrayResAgent // Use string array resource agent
};
```
The following example shows a typical Quick Help resource using string resources.

Example 84-1

Defining a Quick Help Resource

This example shows how to define the Quick Help resource. It comes from the Quick Help for the view of the Tic-Tac-Toe sample program, in PENPOINT\SDK\SAMPLE\TTT. The tag for the resource is defined in TTTQHELP.RC as:

```plaintext
#ifndef RESCMPLR_INCLUDED
#include <rescmplr.h>
#endif
#ifndef QHELP_INCLUDED
#include <qhelp.h>
#endif
#ifndef TTTVIEW_INCLUDED
#include "tttview.h"
#endif

// Quick Help string for ttt's option card.
//
static CHAR tttOptionString[] = {
    // Title for the quick help window
    "TTT Card!"
    // Quick help text
    "Use this option card to change the thickness of the lines" "on the Tic-Tac-Toe board."
};

// Quick Help string for the line thickness control in ttt's option card.
//
static CHAR tttLineThicknessString[] = {
    // Title for the quick help window
    "Line Thickness!
    // Quick help text
    "Change the line thickness by writing in a number from 1-9."
};

// Quick Help string for the view.
//
static CHAR tttViewString[] = {
    // Title for the quick help window
    "Tic-Tac-Toe!"
    // Quick help text
    "The Tic-Tac-Toe window lets you to make X's and O's in a Tic-Tac-Toe" "grid. You can write X's and O's and make move, copy" "and pigtail delete gestures.\n\n" "It does not recognize a completed game, either tied or won.\n\n" "To clear the game and start again, tap Select All in the Edit menu," "then tap Delete."
};

// Define the quick help resource for the view.
static P_RC_TAGGED_STRING tttViewQHelpStrings[] = {
    tagCardLineThickness, tttOptionString,
    tagTttQHelpForLineCtrl, tttLineThicknessString,
    tagTttQHelpForView, tttViewString,
    pNull
};
```

continued
static RC_INPUT tttViewHelp = {
    MakeListResId(clsTttView, resGrpQhelp, 0),
    tttViewQHelpStrings, // Name of the string array
    0,
    resTaggedStringArrayResAgent // Use string array resource agent
};

The glue that ties everything together -- resInput.

resInput is an exported variable that the resource compiler expects.
Each element is a pointer to a structure describing the next resource.
The list is terminated with a null pointer.

P_RC_INPUT resInput [] = {
    &tttViewHelp, // this is the one defined in this example
    // any other resource pointers would go here
    pNull
};
Figure 84-1 illustrates the Quick Help that results from the resource shown here:

**Storing the Resource ID in a Gesture Window**

To associate a Quick Help resource with a window class that inherits from clsGWin, store the resource ID in the gWin.helpId field of the window. For example, while handling msgNewDefaults, clsTTTView sets its gWin.helpID to the resource ID of one of the strings defined in Example 84-1 (this code fragment is from \PENPOINT\SDK\SAMPLE\TTT\TTTVIEW.C):

```c
pArgs->gWin.helpId = tagTttQHelpForView;
```
Advanced Topics

Most application designers shouldn't need to communicate with the QuickHelp. However, if you create a class that does not inherit from clsGWin or want to intercept the Quick Help messages and handle them on your own, you can send messages to the QuickHelp.

Quick Help Message Summary

The Quick Help messages and structures are defined in QHELP.H. Table 84-1 lists the messages defined by Quick Help.

<table>
<thead>
<tr>
<th>Messages</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgQuickHelpHelpShow</td>
<td>P_XY32</td>
<td>Sent to a window to request it to display quick help. The window typically responds by posting msgQuickHelpShow.</td>
</tr>
<tr>
<td>msgQuickHelpShow</td>
<td>P_QUICK_DATA</td>
<td>Sent to the QuickHelp to display the Quick Help associated with a resource ID.</td>
</tr>
<tr>
<td>msgQuickHelpOpen</td>
<td>nothing</td>
<td>Forces the Quick Help window to appear.</td>
</tr>
<tr>
<td>msgQuickHelpHelpDone</td>
<td>OBJECT</td>
<td>Sent to a window when quick its quick help is no longer displayed.</td>
</tr>
<tr>
<td>msgQuickHelpOpened</td>
<td>nothing</td>
<td>Indicates that the quick help window has been opened.</td>
</tr>
<tr>
<td>msgQuickHelpClosed</td>
<td>nothing</td>
<td>Indicates that the quick help window has been closed.</td>
</tr>
<tr>
<td>msgQuickHelpInvokedNB</td>
<td>nothing</td>
<td>Indicates that the notebook associated with quick help should be open.</td>
</tr>
</tbody>
</table>

Using Quick Help Messages

This section describes how to use the Quick Help messages. You should rarely need to send any of these messages since default Quick Help handling is implemented by clsGWin, and most application windows inherit from clsGWin.

Displaying Quick Help Text

To tell the Quick Help window to open and display help text, send msgQuickHelpShow to the QuickHelp. Normally, the QuickHelp will send msgQuickHelpHelpShow to your window, and your window will respond by using ObjectPost() to send msgQuickHelpShow to the QuickHelp.

msgQuickHelpShow takes as its argument a pointer to a QUICK_DATA structure that contains:

- helpId  The Quick Help resource identifier.
- appUID  The UID of the application that owns the Quick Help resource.

The Quick Help window remains open until the user closes it.
Opening the Quick Help Window

You can force the Quick Help window to appear on screen. To open the window, send msgQuickHelpOpen to the QuickHelp. The message takes no arguments. The Quick Help window displays the text that was last displayed by msgQuickHelpShow.

Using the PenPoint Gesture Font

The PenPoint Gesture font allows you to incorporate glyphs that resemble gestures in your Help and Quick Help documents. The PenPoint Gesture font file is in \PENPOINT\FONT\GS080.PCK. You must use Rich Text Format (RTF) documents to include the gesture font with ordinary text in your document.

Table 84-2 shows the characters in the Gesture font as they appear on the PenPoint computer screen, their ASCII values, and the meanings of the characters. Not all gestures are currently used; not all have glyphs in the gesture font. The ASCII values are assigned tags in the file \PENPOINT\SDK\INC\XGESTURE.H.

<table>
<thead>
<tr>
<th>Gesture Tag</th>
<th>Symbol</th>
<th>ASCII Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>xgsLeftParens</td>
<td>[</td>
<td>40</td>
</tr>
<tr>
<td>xgsRightParens</td>
<td>]</td>
<td>41</td>
</tr>
<tr>
<td>xgsPlus</td>
<td>+</td>
<td>43</td>
</tr>
<tr>
<td>xgs1Tap</td>
<td>¥</td>
<td>46</td>
</tr>
<tr>
<td>xgsQuestion</td>
<td>?</td>
<td>63</td>
</tr>
<tr>
<td>xgsAGesture</td>
<td>A</td>
<td>65</td>
</tr>
<tr>
<td>xgsBGesture</td>
<td>B</td>
<td>66</td>
</tr>
<tr>
<td>xgsCGesture</td>
<td>C</td>
<td>67</td>
</tr>
<tr>
<td>xgsDGesture</td>
<td>D</td>
<td>68</td>
</tr>
<tr>
<td>xgsEGesture</td>
<td>E</td>
<td>69</td>
</tr>
<tr>
<td>xgsFGesture</td>
<td>F</td>
<td>70</td>
</tr>
<tr>
<td>xgsGGesture</td>
<td>G</td>
<td>71</td>
</tr>
<tr>
<td>xgsHGesture</td>
<td>H</td>
<td>72</td>
</tr>
<tr>
<td>xgsIGesture</td>
<td>I</td>
<td>73</td>
</tr>
<tr>
<td>xgsJGesture</td>
<td>J</td>
<td>74</td>
</tr>
<tr>
<td>xgsKGesture</td>
<td>K</td>
<td>75</td>
</tr>
<tr>
<td>xgsDownRight,</td>
<td>L</td>
<td>76</td>
</tr>
<tr>
<td>xgsLGesture</td>
<td>M</td>
<td>77</td>
</tr>
<tr>
<td>xgsNGesture</td>
<td>N</td>
<td>78</td>
</tr>
<tr>
<td>xgsCircle,</td>
<td>O</td>
<td>79</td>
</tr>
<tr>
<td>xgsOGesture</td>
<td>P</td>
<td>80</td>
</tr>
<tr>
<td>xgsQGesture</td>
<td>Q</td>
<td>81</td>
</tr>
</tbody>
</table>

continued
### Table 84.2 (continued)

<table>
<thead>
<tr>
<th>Gesture Tag</th>
<th>Symbol</th>
<th>ASCII Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>xgsRGesture</td>
<td>R</td>
<td>82</td>
</tr>
<tr>
<td>xgsSGesture</td>
<td>S</td>
<td>83</td>
</tr>
<tr>
<td>xgsTGesture</td>
<td>T</td>
<td>84</td>
</tr>
<tr>
<td>xgsUGesture</td>
<td>U</td>
<td>85</td>
</tr>
<tr>
<td>xgsWGesture</td>
<td>W</td>
<td>87</td>
</tr>
<tr>
<td>xgsCross</td>
<td>X</td>
<td>88</td>
</tr>
<tr>
<td>xgsXGesture</td>
<td>X</td>
<td>88</td>
</tr>
<tr>
<td>xgsYGesture</td>
<td>Y</td>
<td>89</td>
</tr>
<tr>
<td>xgsZGesture</td>
<td>Z</td>
<td>90</td>
</tr>
<tr>
<td>xgs2Tap</td>
<td>¥</td>
<td>128</td>
</tr>
<tr>
<td>xgs3Tap</td>
<td>¥</td>
<td>129</td>
</tr>
<tr>
<td>xgs4Tap</td>
<td>¥</td>
<td>130</td>
</tr>
<tr>
<td>xgsCheckTap</td>
<td>✓</td>
<td>136</td>
</tr>
<tr>
<td>xgsTapHold</td>
<td>⬇</td>
<td>137</td>
</tr>
<tr>
<td>xgsPressHold</td>
<td>⬇</td>
<td>138</td>
</tr>
<tr>
<td>xgsScratchOut</td>
<td>⫸</td>
<td>140</td>
</tr>
<tr>
<td>xgsPigtailVert</td>
<td>⬇</td>
<td>141</td>
</tr>
<tr>
<td>xgsCircleTap</td>
<td>∞</td>
<td>142</td>
</tr>
<tr>
<td>xgsUpCaret</td>
<td>⬆</td>
<td>143</td>
</tr>
<tr>
<td>xgsCircleLine</td>
<td>⬆</td>
<td>146</td>
</tr>
<tr>
<td>xgsCircleFlickUp</td>
<td>⬆</td>
<td>147</td>
</tr>
<tr>
<td>xgsCircleFlickDown</td>
<td>⬇</td>
<td>148</td>
</tr>
<tr>
<td>xgsUpCaretDot</td>
<td>⬆</td>
<td>149</td>
</tr>
<tr>
<td>xgsDbICircle</td>
<td>∞</td>
<td>152</td>
</tr>
<tr>
<td>xgsUpArrow</td>
<td>⬆</td>
<td>153</td>
</tr>
<tr>
<td>xgsUp2Arrow</td>
<td>⬆</td>
<td>154</td>
</tr>
<tr>
<td>xgsDownArrow</td>
<td>⬇</td>
<td>155</td>
</tr>
<tr>
<td>xgsDown2Arrow</td>
<td>⬇</td>
<td>156</td>
</tr>
<tr>
<td>xgsLeftArrow</td>
<td>⬇</td>
<td>157</td>
</tr>
<tr>
<td>xgsLeft2Arrow</td>
<td>⬇</td>
<td>158</td>
</tr>
<tr>
<td>xgsRightArrow</td>
<td>⬆</td>
<td>159</td>
</tr>
<tr>
<td>xgsRight2Arrow</td>
<td>⬆</td>
<td>160</td>
</tr>
<tr>
<td>xgsDbICaret</td>
<td>⬆</td>
<td>161</td>
</tr>
<tr>
<td>xgsRightUp</td>
<td>⬆</td>
<td>165</td>
</tr>
<tr>
<td>xgsRightUpFlick</td>
<td>⬆</td>
<td>166</td>
</tr>
<tr>
<td>xgsRightDown</td>
<td>⬇</td>
<td>167</td>
</tr>
<tr>
<td>xgsDownRightFlick</td>
<td>⬇</td>
<td>168</td>
</tr>
<tr>
<td>xgsDownLeft</td>
<td>⬇</td>
<td>169</td>
</tr>
<tr>
<td>xgsDownLeftFlick</td>
<td>⬇</td>
<td>170</td>
</tr>
</tbody>
</table>

continued
To add the gesture characters to help text, edit the help text file using a text editor that supports multiple fonts and can save a document in RTF format. Insert the value for the gesture font character and change the font of the character to symbol-h.

When MiniText reads an RTF document, it interprets characters styled with the symbol-h font as using the PenPoint Gesture font.

Adding Gestures to Help Text
Adding Gestures to Quick Help Strings

Adding gestures to Quick Help strings is somewhat more complicated, because you have to create your Quick Help strings in RTF to take advantage of the PenPoint Gesture font.

If you examine a few RTF strings, you will be able to understand most of the language. A complete description of RTF is available in the Microsoft Word for Windows Technical Reference, which is available from Microsoft Press.

The escape character for RTF strings is a backslash. When defining RTF strings in C, you must remember to double the backslash.

The first part of an RTF description of a document assigns fonts to font numbers, describes the style sheet (if there is one), and describes the document layout. PenPoint provides a shorthand method for describing this information with the \qh control word.

The change to the PenPoint Gesture font begins is specified with the string \f63; the return to the normal font is specified with the string \0. RTF files cannot contain characters other than the 127 ASCII characters, so characters beyond decimal 127 must be specified with the RTF hexadecimal character representation (\'hex-digits).
Chapter 85 / The Busy Manager

Occasionally your application might need to perform some time-consuming work—time-consuming enough for the user to notice that the machine is not responding. Such work might include performing compute-intensive work or copying large files across a network. To assure the user that the computer is still working, you can put up a busy clock, a small animated image of a clock face, on the screen.

The PenPoint™ Application Framework ensures that all applications automatically bring up the busy clock if they take more than about half a second to respond to an input event, then automatically take down the busy clock when they are no longer busy. Furthermore, you can control the busy clock programmatically by sending messages to theBusyManager. Messages for theBusyManager are defined in BUSY.H.

### Using theBusyManager

theBusyManager responds to the message, msgBusyDisplay. When your application starts some work that will not change the display for a number of seconds, send msgBusyDisplay to theBusyManager with busyOn (a U32) as the argument. When your application is no longer busy, send msgBusyDisplay to theBusyManager with busyOff (a U32) as the argument.

This example shows a use of theBusyManager.

```c
#include <busy.h>

STATUS s;

s = ObjectCall(msgBusyDisplay, theBusyManager, busyOn);

  // (Perform some time-consuming task)

  // Done, take down busy display
  s = ObjectCall(msgBusyDisplay, theBusyManager, busyOff);
```

### Placing the Busy Display

If you want to locate the busy clock at a particular point on-screen, send msgBusySetXY with a pointer to an XY32 as its argument. The next time theBusyManager receives msgBusyDisplay with an argument of busyOn, the busy clock will appear at the root window coordinates specified by the XY32.

msgBusySetXY sets the location of the busy clock only for the duration of one send of msgBusyDisplay. When the busy clock is turned off, the coordinates for the next display of the busy clock are set to minS32, minS32.
The Busy Clock Delay and Reference Count

Normally, the BusyManager allows a short delay from the time it receives `msgBusyDisplay` with an argument of `busyOn` and the time it displays the busy clock. This usually prevents situations in which the busy clock appears on screen for so brief a period that it simply flickers. However, if you know that your application will be busy for an extended period and would like the busy clock to come up immediately, you can use an OR operation to combine the flag `busyNoDelay` with `busyOn`. If you pass the result as the argument to `msgBusyDisplay`, the BusyManager will put the busy clock up with no delay.

The BusyManager maintains a reference count which records how many objects have turned on the busy clock and how many have turned it off. Using the reference count, the BusyManager can handle `msgBusyDisplay` more efficiently. For example, suppose two clients send `msgBusyDisplay` with an argument of `busyOn`. When the first client sends `msgBusyDisplay` with an argument of `busyOff`, the BusyManager does not need to take down the busy clock (which is still needed for the second client), and instead simply reduces the reference count and returns `stsOK`. The BusyManager doesn't execute the code to take down the busy clock until the reference count comes back down to zero.

It is possible to override the reference count mechanism, although this is not recommended. If you use an OR operation to combine the flag `busyNoRefCount` with `busyOn`, `msgBusyDisplay` will do nothing if the reference count is greater than zero. If you combine `busyNoRefCount` with `busyOff`, `msgBusyDisplay` will set the reference count to zero and bring down the busy clock.
Chapter 86 / Search and Replace

The search and replace API provides a protocol used by clients that need to search and replace text strings in embedded objects. Additionally, the search and replace library provides functions that search for and replace text in specified objects.

Topics covered in this chapter:

- Writing an application that searches and writing a class that can be searched.
- The search and replace protocol.
- Responding to traversal messages.
- The search and replace library.
- Advanced topics (for those not using the search and replace driver provided by the PenPoint™ operating system).

Concepts

Before reading this chapter, you should be familiar with embedded documents and using marks, which are both described in Part 2: PenPoint Application Framework.

A search and replace operation is a specialized form of traversal. The search and replace operation scans a document looking for a particular pattern of text. Traversal is necessary, because the document might contain embedded documents; those documents might contain embedded documents, and so on. When initiating the search, the user can specify whether to search in embedded documents or to only search the document that contains the selection.

At any one time, there are two interesting participants in a traversal: the driver and the slave. If you are writing a class that will contain data that can be searched, instances of the class will be the slave.

A search and replace operation is managed by a search and replace driver. The search and replace driver sends traversal messages to the objects being searched.

Most applications do not need to create a search and replace driver themselves. If an application allows clsApp to handle msgAppSearch, clsApp presents the user with a search dialog and creates the search and replace driver. (You can read about how to handle Standard Application Menu messages in Part 2: PenPoint Application Framework.)

If your class will be a slave (that is, instances of your class contain data that can be searched), you must be prepared to handle both mark messages and the search and replace messages. Most of this chapter discusses how to handle those messages.
Writing a Class That Can Be Searched

To start a search or search and replace operation, an application should allow the class manager to pass msgAppSearch to clsApp. clsApp displays the search dialog and searches the appropriate portion of the document.

If your application needs to start a search by itself, you can send msgAppSearch to self. The message takes no arguments.

If you want instances of your class to be searched, you must respond to:
- The clsMark messages sent to you by the search and replace driver.
- The search and replace messages.

Search and Replace Protocol

The following sections describe the protocol exchanged between theSearchManager and the object being searched. The search (and replace) operation follows these steps:

1. clsMark asks the object to create a token for a mark.
2. theSearchManager asks the object to set the initial search position.
3. theSearchManager asks the object to position to the next group of characters.
4. theSearchManager asks the object to pass it characters delimited by the token.
5. theSearchManager searches the characters for its search string. It repeats steps 3 through 5 until it finds the text (or reaches the end of data).
6. If theSearchManager finds a match, it asks the object to reposition its token to the matched string.
7. theSearchManager asks the object to select take the selection and show the matched string to the user.
8. If replacing, theSearchManager asks the object to replace the characters.
9. If replacing all, repeat steps 3 through 9 until the object reaches the end of its data.

Creating a Mark

When the user starts a search and replace operation, theSearchManager creates an instance of clsMark, which in turn requests the object being searched to create a mark by sending it msgMarkCreateToken.

The object that receives this messages should create a token for the mark, as described in Part 2: PenPoint Application Framework.

Setting the Initial Search Position

When the searched object has created the token, theSearchManager requests the object to position its token to the beginning (or end) of the data that it will search by sending one of msgMarkPositionAtEdge, msgMarkPositionAtSelection, or msgMarkPositionAtGesture.
Getting the Next Group

To search for text, the SearchManager requests the object to position its token to the next group of characters that might contain match the search. The arguments for msgSRNextChars identify the token to be moved.

The method that handles msgSRNextChars, and all methods that handle search and replace messages should call the function MarkHandlerForClass(). The only parameter for the function is the well-known UID of the class to which the method belongs.

Usually a group ends at a non-text element (such as an embedded document). However, the group can end anywhere that is convenient for your data.

The method must also set the blockStart and blockEnd BOOLEAN values. These indicate whether the group of characters is the beginning or end of a block of characters. Text within groups is not matched across block boundaries, but is matched across other groups.

There are three status values your method can return:

- stsOK Next group found and token repositioned.
- stsEndOfData There is no more data to be searched. (When your method finds the last group, it still returns stsOK; only when another msgSRNextChars arrives should it return stsEndOfData.)
- stsMarkEnterChild The next item is an embedded document.

Passing the Found Characters

If your method for msgSRNextChars returned stsOK, the SearchManager asks the object to pass back the characters matched by the token by sending msgSRGetChars.

The message passes a pointer to an SR_GET_CHARS structure, which contains:

- first An offset to the first character in the token to copy.
- len The number of characters to copy.
- bufLen The size of the buffer.
- pBuf A pointer to the buffer to which your method copies the characters.

Your method must copy a null terminate string into pBuf.

Searching the Text

The SearchManager does the actual searching by comparing its search string to the characters passed in by msgSRGetChars.

If the SearchManager finds a match, it asks the object to position its token to the matched characters by sending it msgSRPositionChars. The message passes a pointer to an SR_POSITION_CHARS structure, which identifies the new position for the token.
Highlighting the Text

The search manager then asks the object to display the found text by:

- Asking the object to take the selection by sending it `msgMarkSelectTarget`.
- Asking the object to display the matched characters to the user by sending it `msgMarkShowTarget`.

Replacing Characters

If the user is using find and replace and chooses to replace the matched text, `theSearchManager` asks the object to replace some or all of the characters matched by the token with new text by sending it `msgSRReplace`. The message passes a pointer to an `SR_REPLACE_CHARS` structure, which contains:

- `first`  The offset to starting character within the token to replace.
- `len`  The number of characters to replace.
- `bufLen`  The number of characters in the replacement string.
- `pBuf`  A pointer to the replacement string.

The `first` field can be negative, indicating that the text to be replaced starts before the token.

After replacing the text, your method must update the token to reflect any change in size of the replaced text.

Classes That Respond to Search Messages

If your class inherits from a class that responds to the search and replace messages, you may not have to do anything.

Currently `clsText` is the only class that responds to the search and replace messages.

The Search and Replace Messages

Table 86-1 lists the messages defined by `clsSR` in the file `SR.H`.

<table>
<thead>
<tr>
<th>Message</th>
<th>Tokens</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgSRNextChars</td>
<td>P_SR_NEXT_CHARS</td>
<td>Asks the client to move the token to the next group of characters.</td>
</tr>
<tr>
<td>msgSRGetChars</td>
<td>P_SR_GET_CHARS</td>
<td>The component passes back the characters from the location identified by the token.</td>
</tr>
<tr>
<td>msgSRReplaceChars</td>
<td>P_SR_REPLACE_CHARS</td>
<td>Ask the component to replace some of the characters at the location identified by the token.</td>
</tr>
<tr>
<td>msgSRPositionChars</td>
<td>P_SR_POSITION_CHARS</td>
<td>Asks the component to reposition the token to some of the characters in the current group.</td>
</tr>
</tbody>
</table>
Chapter 87 / Undo

The undo manager provides the mechanism that allows applications to respond to the standard application menu undo command.

Topics covered in this chapter:
- The concepts behind the undo manager.
- Undo manager messages.
- Using the undo manager messages.

Concepts

The undo manager provides a centralized facility within each application for managing undo information and handling undo commands local to a document. Applications use the undo manager to store records of user actions and can request the undo manager to undo those user actions. Currently there is no support for undoing actions between documents (such as move or copy between documents); each document sees its part of such operations as a separate undo item.

When designing the user interface for an application or component, you need to identify particular user actions that the user might want to undo. Each user action that is undoable is called a transaction. One or more objects contribute one or more undo items to a single transaction.

A transaction should appear to the user a single action, such as deleting a spread-sheet cell, inserting text in a document, or changing all the text in the selection to italics.

When the user issues an undo command, the undo manager undoes the most recent transaction by sending a message to the objects mentioned in the transaction. Those objects must know how to undo their parts of the transaction, but the undo manager keeps track of what to undo. The undo manager allows applications to support a transaction history, so that subsequent undo commands remove transactions from the history in the reverse order in which they were added.

Because PenPoint™ applications are built by integrating a number of components, the PenPoint undo is slightly different than undo under other operating environments. In traditional operating environments, the program that performs the undo keeps track of all of its own actions. However, under PenPoint, actions may be performed by the application or any of its component objects.

Each component that performs a function that a user might want to undo must cooperate in the undo strategy. If a component doesn’t cooperate in the undo strategy, changes made by that component are not undoable; if the component

The undo manager is not a general database transaction undo facility.
contributes to an overall action that is undoable, the undo might not work correctly.

**The General Strategy**

When the user turns to a document, the PenPoint Application Framework creates an instance of the undo manager for that document (named `theUndoManager`).

When the user performs some action that is undoable, the application tells `theUndoManager` that a transaction is beginning. Applications and components divide the steps that it takes to complete the transaction into a series of items. An undo item describes an operation by a class and how to undo that operation. The application and its components add items to the transaction data until the transaction ends. The application tells `theUndoManager` when the transaction ends.

When the user taps on *Undo*, `theUndoManager` gets the data for the most recent transaction. `theUndoManager` removes the last item in the transaction, examines it, and sends a message to the object that created the item. The object’s class (or one of its ancestors) uses the item data to undo that part of the operation and then frees the item data. `theUndoManager` continues to remove and examine the items in the transaction (in the reverse order in which they were received), until the transaction is completely undone.

When the user turns away from the document, the PenPoint Application Framework destroys `theUndoManager` and the undo history as part of deactivating the document.

At any time there is at most one undo transaction open. The data associated with each transaction includes:

- A unique identification of type `UNDO_ID`.
- A nesting count that tracks the number of “begin undo” transaction messages that have not been matched by a corresponding “end undo” transaction message.
- A heap with global scope, from which clients can allocate space to hold undo item information.
- A list of undo items contributed by applications or components.
- A variety of transaction state information.

The transaction information is stored in the `UNDO_METRICS` structure; applications can get this information by sending `msgUndoGetMetrics` to `theUndoManager`. 
The undo manager stores transaction data in the undo history. An application can change the size of its undo history, thereby allowing larger transaction histories.

The data for a transaction consists of one or more items. An item describes a change made to data by an application or component. For example, if a user replaced a value in a cell, the transaction might consist of a single item, the original value for the cell. On the other hand, a search and replace operation could potentially create items for each replacement.

An item contains:

- The object and class that performed the change.
- A 32-bit value that the class uses to store its own information about the change.
- A variety of attributes stored in a 16-bit flag variable.

In simple cases, the 32-bit value can contain the actual data that is changed; in more complex cases, it points to a buffer that describes the change. The object that creates an item and data is also responsible for restoring that item from the data. Thus, the organization of the data for each item is up to you.

Your application allocates the buffer used in the item. When you undo an item, you are also responsible for deallocating the buffer.

However, there are two other reasons why you might need to deallocate the buffer; in these cases either you or the undo manager can do the deallocation.

- When the transaction history is full, the undo manager removes the oldest transaction from the history. As part of removing a transaction, the undo manager must coordinate deallocation of the item data buffers.
- When an application or component aborts a transaction, the undo manager removes the items from the aborted transaction and must coordinate deallocating the data buffers for those items.

When you add an item to the transaction history, you can tell the undo manager how the data buffer was allocated. You can also tell the undo manager to deallocate the item’s buffer automatically when either of the above events occur (or you can deallocate the buffer yourself).

If you don’t tell the undo manager how to deallocate the buffer, the undo manager sends you a message telling you to deallocate the buffer.
The Undo Messages

The messages used by the UndoManager are described in the file UNDO.H. Table 87-1 lists the undo messages.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgUndoAbort</td>
<td>pNull</td>
<td>Aborts the current undo transaction.</td>
</tr>
<tr>
<td>msgUndoAddItem</td>
<td>P_UNDO_ITEM</td>
<td>Adds a new item to the current undo transaction if and only if it is still open.</td>
</tr>
<tr>
<td>msgUndoBegin</td>
<td>RES_ID</td>
<td>Creates a new undo transaction if there is no current transaction, or increments the nesting count if there is a current transaction.</td>
</tr>
<tr>
<td>msgUndoCurrent</td>
<td>pNull</td>
<td>Undoes the most recent undo transaction.</td>
</tr>
<tr>
<td>msgUndoEnd</td>
<td>pNull</td>
<td>Decrement the nesting count of (and thus may end) the current transaction.</td>
</tr>
<tr>
<td>msgUndoGetMetrics</td>
<td>P_UNDO_METRICS</td>
<td>Passes back the metrics associated with an undo transaction.</td>
</tr>
<tr>
<td>msgUndoLimit</td>
<td>U32</td>
<td>Sets the maximum number of remembered undo transactions.</td>
</tr>
</tbody>
</table>

Using the Undo Messages

The following sections describe how to use the clsUndo messages to save transactions and then how to restore them when the user taps on Undo.

Beginning a Transaction

When the user initiates an action that your application or component might need to undo, start a new transaction by sending msgUndoBegin to the UndoManager. The message takes no arguments.

If an application or component attempts to start a transaction while a transaction is already in progress, the UndoManager increments a counter that counts the number of begins; the UndoManager does little else. This is necessary because a component is not expected to know what happened in its owning application before it receives a message (nor should it know what application or component might own it). If your component performs some task that can be undone, it might send msgUndoBegin just to be on the safe side. If the owning application also sent msgUndoBegin, it won't matter.

By nesting transactions, the undo manager allows separately implemented components to behave consistently when used together in a transaction. Each component can add items to the current transaction, but the transaction shows no indication of the nesting.
Using the Undo Messages

The most important thing is that for each msgUndoBegin, there must be the same number of msgUndoEnd messages at the end of the transaction (msgUndoEnd is described below). Keep this in mind before you use the ObjCallRet(), ObjCallJmp(), StsRet(), or StsJmp() macros.

To prevent run-away applications from nesting transaction-begins too deeply (and to detect errors when a msgUndoBegin does not have a matching msgUndoEnd), theUndoManager returns stsFailed if you exceed the nesting limit, which is approximately 1000.

When msgUndoBegin completes successfully, it returns an UNDO_ID. This value identifies the transaction; you can use it to get metrics of a specific transaction.

The memory allocated for transactions is finite. When you start a new transaction, the undo manager might need to make space in the transaction history, which it does by freeing the earliest transaction. Thus, you shouldn't be surprised that before msgUndoBegin returns you receive a msgUndoFreeItemData, asking you to free buffers used by items in the freed transaction. You should free the buffers, return msgUndoFreeItemData and, eventually, msgUndoBegin returns.

Adding Items to a Transaction

As your application or component performs the steps within the transaction, it needs to note what change took place in each of the steps. For each change, send msgUndoAddItem to theUndoManager. The message takes a pointer to an UNDO_ITEM structure that describes a change to data. The structure contains:

- **object**  The UID of the object that is performing the transaction.
- **subclass**  The UID of the class that is making the change.
- **flags**  Flags that specify how the undo manager should free the item. The flags are described below.
- **pData**  A 32-bit value that either is the change data or points to a buffer that contains the change data.
- ** dataSize**  A value that contains the size of the data in pData. If this field is non-zero, theUndoManager assumes that pData points to a data buffer and copies that data into the transaction's heap.

**Item Flags**

There are two sets of flags in the flags field.

The first set contains four flags that are not interpreted by theUndoManager. You can use these flags to represent whatever you want in the undo item. The value ufClient is a mask that you can use to access the client-specific portion of flags.

The other set of flags describe how the pData buffer for an item was allocated. If you specify one of these flags, the undo manager frees the pData buffer automatically when it removes an old transaction from the history or when it aborts a transaction. The flags can be one of the following:
ufDataInUndoHeap pData points to a stand-alone node from the current undo transaction’s heap; the undo manager can free the buffer by freeing the entire heap.

ufDataIsHeapNode pData points to a stand-alone node from a global heap; the undo manager can free the buffer with OSHepBlockFree(pData).

ufDataIsObject pData contains the UID of an object; the undo manager can free the object by an ObjectSend of msgDestroy.

ufDataIsSegment pData points to a stand-alone segment; the undo manager can free the buffer with OSMemFree().

ufDataIsSimple pData is a simple value (for example a U32); the undo manager can free the value by just forgetting it.

You can use the value ufDataType as a mask to access the data type flags in the flags field. The data type flags are 12-bits long. If you store the ufDataType portion, use a U16 value.

If you do not specify one of these flags when you add an item, the undo manager sends you msgUndoFreeItem when it needs to remove an item from a transaction.

The transaction must be open when you send msgUndoAddItem. If there is no open transaction when you send msgUndoAddItem, the message returns stsFailed.

If there is an open transaction, but there isn’t enough memory to add the item to the transaction, msgUndoAddItem returns stsOSOutOfMem.

If msgUndoAddItem returns stsUndoAbortingTransaction, the transaction is aborting. You must free any storage allocated for the item that you attempted to add (unless the storage is in the transaction’s heap).

**Ending a Transaction**

When you conclude the transaction (before returning control to the user), you tell the undo manager to close the current transaction by sending msgUndoEnd to theUndoManager. The message takes no arguments.

For every msgUndoBegin there must be a corresponding msgUndoEnd. msgUndoEnd decrements the nesting count; if the message lowers the nesting count to zero, the undo manager closes the transaction. You cannot add items to a closed transaction.

**Aborting a Transaction**

When your application or component is unable to complete its work because of an error, you will probably want to abort the current transaction.

If you need to abort a transaction, send msgUndoAbort to theUndoManager. The message takes a pNull value.

When theUndoManager receives msgUndoAbort, it marks the current transaction as aborting. If you send msgUndoAddItem to an aborting transaction, it will return
stsUndoAbortingTransaction. You must deallocate the item’s data buffer, unless the buffer was allocated from the transaction's heap.

In an aborting transaction, the state field in the UNDO_METRICS structure has the flag undoStateAborting set.

Aborting a transaction does not close that transaction; you are still responsible for closing the transaction. The undoStateAborting flag remains set until the final msgUndoEnd closes the transaction.

Getting Transaction Metrics

You can get the transaction metrics for any of the transactions in the transaction history by sending msgUndoGetMetrics to theUndoManager. The message takes a pointer to an UNDO_METRICS structure. To get the metrics of the current or latest transaction, set the id field in the UNDO_METRICS structure to pNull. To get the metrics for another transaction, set the id field to the UNDO_ID value for that transaction.

When the message completes successfully, it returns stsOK and sends back the UNDO_METRICS structure with:

- id   An UNDO_ID value that contains the transaction ID.
- heapId An OS_HEAP_ID value that indicates the heap that you can use to store item data.
- state A U16 that indicates the transaction’s current state. The state field does not contain a value that indicates the state, rather it contains a set of flags that indicate attributes of the current transaction. The flags are:
  - undoStateBegun The transaction is open.
  - undoStateUndoing The transaction is being undone.
  - undoStateAborting The transaction is aborting. If the state field contains the value undoStateNil, all these flags are clear.
- transactionCount A count of the number of transactions in the undo history.
- itemCount A count of the number of items in the transaction.
- limit The maximum number of transactions allowed in the transaction history. The default for the limit value is 2; you can modify this value with msgUndoLimit.
- resID A resource ID identifying the string to use for the Undo menu item.
  This resource ID should specify a resGrpTK string resource list.
- info A U32 fields reserved for future system use.

Changing the Size of the Transaction History

To change the size of the transaction history, send msgUndoLimit to theUndoManager. The message takes a U32 value that contains the new maximum number of transactions that the undo manager can store. If the value is 0, you effectively disable the undo capabilities of your application or component.
Undoing a Transaction

When the user taps on the undo button in the Edit menu the menu sends `msgAppUndo` to self (the document that is open). When your application receives `msgAppUndo`, the PenPoint Application Framework sends `msgUndoCurrent` to the `UndoManager`; the message argument is always `pNull`. (For more information on Standard Application Menu messages, see Part 2: PenPoint Application Framework.

When the undo manager receives `msgUndoCurrent`, it locates the latest transaction (whether it is still open or not) and removes the last item from that transaction. Because each `UNDO_ITEM` structure contains an object UID and a class UID, the undo manager sends the item to the object UID with `msgUndoltem` (you must be prepared to receive and handle `msgUndoltem`, described below). The undo manager continues to remove items from the transaction and send them to their corresponding owners until the transaction is empty.

When the transaction is empty, `msgUndoCurrent` returns. If there is a previous transaction in the transaction history that transaction becomes the next transaction to be undone (the next time the undo manager receives `msgUndoCurrent`.

Handling `msgUndoltem`

When you receive `msgUndoltem`, compare the class in the `UNDO_ITEM` structure to your class. If the class doesn’t match, send the message to your ancestor. If the class does match, use the data indicated by `pData` to undo the action.

Again, the way that you store the data in an item and undo that action is totally up to you.

Handling `msgUndoFreItem`

If you did not specify one of the data type flags in `UNDO_ITEM`, the undo manager does not know how to free `pData`, so it sends `msgUndoFreItem` to you. The message sends you a pointer to an `UNDO_ITEM` structure.

As with `msgUndoltem`, you should compare the class field in the structure with your class, if they don’t match, send the message to your ancestor. If they do match, free the data. It is up to you to know how the data was allocated and how to free it.
Chapter 88 / Byte Buffer Objects

The byte buffer object class (clsByteBuf) allows clients to create a simple data object that contains an array of bytes. clsByteBuf allocates space for the byte array, handles filing messages, and deallocates the space when the client destroys the byte buffer object.

Concepts

Frequently applications maintain some data in an array of bytes. If this data is stateful, you need to file the data when you receive filing messages. One way to do this is to keep the data in one or more data objects and file the objects using msgResPutObject.

To save you from writing your own class to maintain and file byte arrays, PenPoint provides clsByteBuf, which you can use to store byte arrays. If you need to save a string, use the string object class defined in Chapter 89, String Objects.

If you have an array of bytes that you need to save, you create a clsByteBuf object and store the bytes in the object. You can store any type of bytes in the clsByteBuf object. clsByteBuf only sees the data as an array of bytes; the organization of the data is up to your application. clsByteBuf handles the allocation, deallocation, and filing messages for you. All you have to do is ensure that your byte buffer object receives filing messages at the proper time.

clsByteBuf allocates its memory from the system heap. When you ask a byte buffer object for its data, it sends back a pointer to the system heap, so you don’t have to allocate space to accommodate the data.

Because the storage for byte buffer data is in a system heap. The location in the heap changes when the data changes, so you cannot maintain or file other pointers to that buffer. When the byte buffer object is saved and restored, there is no guarantee that the object will be restored at the same address. If you need to maintain a location within a byte buffer object, use an index rather than a pointer.

clsByteBuf has no concurrency support. If two clients access the same byte buffer object at the same time, they manipulate the same data.
**Using the Byte Buffer Messages**

The clsByteBuf messages are listed in Table 88-1. The messages are defined in the file BYTEBUF.H.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_BYTEBUF_NEW</td>
<td>Creates a new buffer object.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_BYTEBUF_NEW</td>
<td>Initializes the BYTEBUF_NEW structure to default values.</td>
</tr>
<tr>
<td>msgByteBufGetBuf</td>
<td>P_BYTEBUF_DATA</td>
<td>Passes back a pointer to the object's buffer.</td>
</tr>
<tr>
<td>msgByteBufSetBuf</td>
<td>P_BYTEBUF_DATA</td>
<td>Copies the specified buffer data into the object's buffer.</td>
</tr>
<tr>
<td>msgByteBufChanged</td>
<td>OBJECT</td>
<td>Sent to observers when the object data changes.</td>
</tr>
</tbody>
</table>

**Creating a Byte Buffer Object**

To create a byte buffer object, send msgNewDefaults and msgNew to clsByteBuf. The messages take a pointer to a BYTEBUF_NEW structure that contains an OBJECT_NEW_ONLY structure and a BYTEBUF_NEW_ONLY structure. The BYTEBUF_NEW_ONLY structure contains:

- allowObservers A BOOLEAN value that specifies whether the object send messages to observers.
- data A BYTEBUF_DATA structure that contains:
  - pBuf A byte pointer to the byte array to be saved.
  - bufLen A U16 value that specifies the length of the byte array in pBuf.

When the message completes successfully, it returns stsOK.

**Getting the Byte Buffer Data**

To get the address of the byte buffer data, send msgByteBufGetBuf to the byte buffer object. The message takes a pointer to an empty BYTEBUF_DATA structure.

When the message completes successfully, it returns stsOK and passes back the BYTEBUF_DATA structure containing:

- pBuf A byte pointer to the byte array.
- bufLen A U16 value that indicates the length of the byte array in pBuf.

**Resetting a Byte Buffer Object**

When you create a byte buffer object, you set its initial value, but if you want to change the contents of a byte buffer object, you send msgByteBufSetBuf to the byte buffer object. The message takes a pointer to a BYTEBUF_DATA structure that contains:

- pBuf A byte pointer to the byte array.
- bufLen A U16 value that indicates the length of the byte array in pBuf.
If `bufLen` is different from the size of the byte buffer that was stored there before, `clsByteBuf` automatically changes the size of the allocated storage.

When the message completes successfully, it returns `stsOK`.

**Notification of Observers**

Clients can observe a byte buffer object (provided that the creator of the object specified `allowObservers` when it sent `msgNew`). When a client sends `msgByteBufSetBuf` to a byte buffer object, `clsByteBuf` sends `msgByteBufChanged` to all observers of that byte buffer object. The message passes an `OBJECT` value that identifies the byte buffer object that changed.
Chapter 89 / String Objects

The string object class (clsString) allows clients to create a simple data object that contains a null-terminated ASCII string. clsString allocates space for the string, handles filing messages, and deallocates the space when the client destroys the string object.

clsString inherits from clsByteBuf. The difference is that you must supply the length of a byte buffer as well as a pointer to it, whereas string objects are null-terminated, so you need only supply a pointer to the string.

Concepts

Most applications maintain some data in the form of null-terminated ASCII strings. If this data is stateful, you need to file the data when you receive filing messages. One way to do this is to keep the data in one or more data objects and file the objects using msgResPutObject.

To save you from writing your own class to maintain and file string data, PenPoint provides clsString, which you can use to store strings. If you need to save a byte array, use the byte buffer object class described in Chapter 88, Byte Buffer Objects.

If you have a string that you need to save, you create a clsString object and store the string in the object (you must create a clsString object for each string). clsString handles the allocation, deallocation, and filing messages for you. All you have to do is ensure that your string object receives filing messages at the proper time.

clsString allocates its memory from the system heap. When you ask a string object for its data, it sends back a pointer to the system heap, so you don't have to allocate space to accommodate the string.

Because the string object is in system heap, you cannot maintain (and file) other pointers to that string. When the string object is saved and restored, there is no guarantee that the object will be restored at the same address. If you need to maintain a location within a string object, use an index rather than a pointer.

clsString has no concurrency support. If two clients access the same string object at the same time, they manipulate the same data.
Using the String Object Messages

The clsString messages are listed in Table 89-1. The messages are defined in the file STROBJ.H.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_STROBJ_NEWONLY</td>
<td>Creates a new string object.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_STROBJ_NEW</td>
<td>Initializes the STROBJ_NEW structure to default values.</td>
</tr>
<tr>
<td>msgStrObjGetStr</td>
<td>PP_CHAR</td>
<td>Passes back the object's string.</td>
</tr>
<tr>
<td>msgStrObjSetStr</td>
<td>P_CHAR</td>
<td>Copies the specified string data into the object's string buffer.</td>
</tr>
<tr>
<td>msgStrObjChanged</td>
<td>OBJECT</td>
<td>Sent to observers when the string object data changes.</td>
</tr>
</tbody>
</table>

Creating a String Object

To create a string object, send msgNewDefaults and msgNew to clsString. The messages take a pointer to a STROBJ_NEW structure that contains an OBJECT_NEW_ONLY structure and a STROBJ_NEW_ONLY structure. The STROBJ_NEW_ONLY structure contains a pointer to the string to be saved (pString).

Getting the String Object

To get a pointer to a string in a string object, send msgStrObjGetStr to the string object. The message takes a pointer to a string pointer (PP_STRING).

When the message completes, it returns stsOK and passes back the pointer to system heap in the specified location.

Resetting a String Object

When you create a string object, you set its initial value, but if you want to change the contents of a string object, you send msgStrObjSetStr to the string object. The message takes a pointer to the string that you want stored in the string object.

If the string is a different size from the string that was stored there before, clsString automatically changes the size of the allocated storage.

When the message completes successfully, it returns stsOK.

Notification of Observers

Clients can observe a string object. When a client sends msgStrObjSetStr to a string object, clsString sends msgStrObjChanged to all observers of that string object. The message passes an OBJECT value that identifies the string object that changed.
Chapter 90 / Table Class

The class clsTable provides a general-purpose table mechanism with random and sequential access. You can use clsTable as a superclass for specialized table classes.

You create, destroy, modify, and access tables using a row and column metaphor. Tables are PenPoint objects; the data for tables is stored in table files. clsTable also provides a semaphore mechanism that you can use to control concurrent table access.

Tables are observable objects. Any object can add itself to a table's observer list, so that the object will receive notification of changes in the table. Changes can include adding, removing, or changing entries, or destruction of the table.

clsTable inherits from clsObject.

A Distributed DLL

The table class component is implemented in a DLL, \PENPOINT\SDK\DLL\TS.DLL. This DLL is not automatically loaded at boot time—it is not mentioned in \PENPOINT\SDK\INC\BOOT.DLC. You need to mention it in your application's .DLC file and include it in your application's installation disk. This ensures that all applications that use the table class will load the DLL if necessary and share the same copy of the code.

Structures and #defines used by clsTable are defined in \PENPOINT\SDK\INC\TS.H. The library support routines for clsTable are in the table library, \PENPOINT\SDK\LIB\TS.LIB.

The send list application uses clsTable to maintain the user's address list.

Table Concepts

A table is a two-dimensional array consisting of a fixed number of columns and a variable number of rows. Each column contains a single type of data, such as U32, variable-length string, fixed size byte field, date and time, and so on.

When you create a table, you define the number of columns in the table and the data type of each column. Once you create a table, you cannot change its organization.

Describing a Table

You describe a table by creating a structure that contains the total number of columns in the table and a pointer to an array of column descriptors. For each column in the table, you must create a column descriptor. The order of the
column descriptors indicates the order of the columns in the table; you use an
index into the column array to indicate specific columns.

Each of the column descriptors contains:

- The name of the column.
- The type of data in the column (such as U32, string, date and time, or UID).
- The width of the column.
- The number of times the columns is to be repeated.
- The byte offset to the column.
- Whether the column is a sort field or not.

Table Data Files

The first time you create a table object (by sending msgNew to clsTable), clsTable
creates a table file in which it stores the data for that table. The msgNew
arguments include the file name and file options. If you destroy the table object
and send msgNew again, clsTable accesses the table data stored in the table file.

The table object is transitory; the table file can last forever.

You also use arguments to msgNew to indicate whether the table should be
destroyed when it has no observers or no clients, and whether the table file should
be deleted when the table object is destroyed.

clsTable opens the file with exclusive access rights; this means that only one table
object can be associated with a given table file at one time. However, multiple
clients can access the table by using the same table object.

Beginning Table Access

After creating a table object, you register as a user of the table by sending
msgTBLBeginAccess to the table object. Each client that accesses the table object
must send a separate msgTBLBeginAccess (Even if you created the table object,
you must still register as a table user.) When you no longer need to access a table,
send msgTBLEndAccess to the table object.

The table object maintains a count of the number of accessing clients. When a
client begins access, the table increments the count; when a client ends access, the
table decrements the count. You cannot destroy a table object until the number of
accessing clients is zero. (One of the table destruction options uses this count to
destroy the table automatically when the number of accessors goes to zero.)

You can access a table without using msgTBLBeginAccess and
msgTBLEndAccess, but it is not recommended. If you use a table object for
which you haven’t registered, the table object might vanish without warning (if
another accessor destroys the object or table file). The table object has no
indication that you are an accessor of the table.
Positioning in Tables

There are two ways to locate records in a table:

- Searching for a specific value in a column, starting at the beginning of the table.
- Searching for a specific value in a column, starting at the current position.

The order in which the records are searched depends on the column that you search. Unlike other data bases, the table class has no concept of a last record.

When you request a new position from clsTable, it sends back a TBL_ROW_POS value that indicates the row position. This value is not an index or offset within the table. You cannot advance to the next row in a table by simply incrementing the row position value. When you request clsTable to advance to the next record, you must specify the current TBL_ROW_POS value.

If you know that you will need to examine records in the order in which they were added to the table, one of the columns in your table should be a sorted column containing a sequence number. When you add a record to the table, assign a new sequence number to the record.

Each table object has the concept of a current state. These states are "at the beginning," "at the end," and "somewhere in between." tsBegin indicates that the table's current row is the first row in that column; tsEnd indicates that the table's current row is the last row in that column; tsPosition indicates that the current row is not at the beginning or the end of the column. A client of an empty table is always positioned at tsEnd.

You use msgTBLGetState to get the current state and row position of a table.

Observing Tables

Table objects are observable. This means that an object can add itself to a table's observer list. When anything changes in the table or when other objects add or remove themselves from the table's observer list, the table object sends a message to objects in the observer list. A table client does not have to add itself to the observer list. In fact, adding observers has a detrimental impact on table performance. You should add an observer only when you really need it.

Shared Tables

Most clients use a table as an unshared, private database. The client creates the table object, accesses the table, and frees the object all on its own.

Clients can also share a table with other clients. Sharing a table raises several issues about simultaneous access to the table, such as:

Ownership An object exists only as long as its creator exists.
Access All clients that access a shared table must know the UID of the table object.
Concurrency Problems arise when two clients try to write to the table at the same time, or when one client writes a record which another client is reading.

Remember that the table object is transitory, but that the underlying table object file can exist practically forever. When clsTable creates a table object, it opens the table object file with exclusive access. This prevents any other client from creating a table object for the same table object file.

Ownership
The process that created a table object must remain active as long as any other process is using the table; the table object will be freed when the creating process terminates.

One way to handle ownership is through a requester-server model. In the requester-server model one client (the server) creates a table and serves requests from other clients (the requesters) that want to access the table. The server must be in existence before the first requester requests access. The server should exist as long as requesters needs to access the table.

Access to the Table Object
If you use the requester-server model, only the server has to know the UID of the table object. However, if you don't use the requester-server model, the other clients must be able to find the UID of the table object.

Making the UID well-known global is one way to accomplish this. Explicitly passing the UID to other processes is another approach (this works well in a server situation).

Another approach is to have each process that wants to use a table attempt to create it. If the creation is successful, the process should put the table object UID in a publicized global location. If the creation fails, another client has already created the table; client should look for the existing table object UID in the global location.

Concurrency
A table object has the following access concurrency characteristics:

- The last row in the table is always the last row that was added to the table. Because the table rows are enumerated in random order, there is no notion of an "end" to the table. However, while a client refers to a sorted column, the rows are sorted in the order determined by the data in the column.

- When a client writes data to the table file, it is permanently changed. Any other clients that hold data retrieved from the changed row and column now have invalid data.
clsTable provides a semaphore that you can use to synchronize access to a table among multiple clients. You can use this semaphore (by sending msgTBLSemaRequest to the table object) when you want to treat multiple updates as a single, atomic update.

If msgTBLSemaRequest is successful, you can access the table; all other clients that request the semaphore while you hold it will be suspended. If another client holds the semaphore, msgTBLSemaRequest will suspend your process until the other client releases the semaphore.

Using semaphores extensively can lead to a deadlock situation (that is, a circular list of clients waiting for each other’s resources). You can avoid deadlocks by releasing the semaphore (with msgTBLSemaClear) on one table before requesting the semaphore on another.

**Using Tables in a Database**

There is no explicit database class in PenPoint. However, you can use a single table as a flat-file database. A table is simply a flat database. Each row is a record, each column is a field.

With a bit more effort you can link a series of tables together (a table of tables) to create a relational database.

**Using Table Messages**

This section describes the tasks associated with table objects.

Table 90-1 lists the clsTable messages.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_TBL_NEW</td>
<td>Creates a new table object.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_TBL_NEW</td>
<td>Initializes the TBL_NEW structure to default values.</td>
</tr>
<tr>
<td>msgTBLAddRow</td>
<td>P_TBL_ROW_POS</td>
<td>Adds a row/record with no data to the table server object.</td>
</tr>
<tr>
<td>msgTBLDeleteRow</td>
<td>P_TBL_ROW_POS</td>
<td>Deletes the specified row.</td>
</tr>
<tr>
<td>msgTBLColGetData</td>
<td>P_TBL_COL_GET_SET_DATA</td>
<td>Passes back the data for the specified row and column.</td>
</tr>
<tr>
<td>msgTBLColSetData</td>
<td>P_TBL_COL_GET_SET_DATA</td>
<td>Sets the data for the specified row and column.</td>
</tr>
<tr>
<td>msgTBLRowGetData</td>
<td>P_TBL_GET_SET_ROW</td>
<td>Gets the contents of an entire row.</td>
</tr>
<tr>
<td>msgTBLRowSetData</td>
<td>P_TBL_GET_SET_ROW</td>
<td>Sets the contents of an entire row.</td>
</tr>
<tr>
<td>msgTBLGetInfo</td>
<td>P_TBL_HEADER</td>
<td>Gets the table header information.</td>
</tr>
<tr>
<td>msgTBLGetColCount</td>
<td>P_TBL_COL_COUNT</td>
<td>Gets the number of columns in the table.</td>
</tr>
</tbody>
</table>

---

continued
### Message | Takes | Description
--- | --- | ---
msgTBLGetColDesc | P_TBL_GET_COL_DESC | Passes back the column description for the specified column.
msgTBLGetRowCount | P_TBL_ROW_COUNT | Gets the current number of rows in the table.
msgTBLGetRowLength | P_TBL_ROW_LENGTH | Gets the length (in bytes) of the specified row.
msgTBLGetState | P_TBL_GET_STATE | Gets the current state of a specified row.
msgTBLBeginAccess | P_TBL_BEGIN_ACCESS | Initiates table access by a client on this table.
msgTBLEndAccess | P_TBL_END_ACCESS | Ends client access to the table.
msgTBLSemaClear | nothing | Releases the table's semaphore.
msgTBLSemaRequest | nothing | Requests access to the table's semaphore.
msgTBLFindFirst | P_TBL_FIND_ROW | Finds the first record that meets the search specification.
msgTBLFindNext | P_TBL_FIND_ROW | Find the next record following the specified TBL_ROW_POS that meets the search specification.
msgTBLFindColNum | P_TBL_COL_NUM_FIND | Passes back the column number for the specified column name.
msgTBLCompact | nothing | Compacts the table without closing it.
msgTBLRowNumToRowPos | P_TBL_CONVERT_ROW_NUM | Converts a TBL_ROW_NUM to its corresponding TBL_ROW_POS for the specified column.

### Observer Notification Messages

- **msgTBLRowAdded**<br>  Sent to observers indicating that a row has been added.<br>
- **msgTBLRowDeleted**<br>  Sent to observers indicating that a row has been deleted.<br>
- **msgTBLRowChanged**<br>  Sent to observers indicating that row data has been changed.

---

### Defining a Table

Before you create a table with `msgNew`, you need to declare the organization of the table with a `TBL_CREATE` structure. The `TBL_CREATE` structure consists of a value that indicates the number of columns in the table (`colCount`) and the address of an array of one or more `TBL_COL_DESC` structures (`colDescAry`).

Each `TBL_COL_DESC` structure describes one column and contains:

- **name** A string containing the name of the column. This name is used by applications when displaying a table. Internally, you usually identify columns by an index to their position. You can get the index from the name with `msgTBLFindColNum` or you can get the name using an index with `msgTBLColGetDesc`.  

---

**Table 90-1 (continued)**
Defining a Table

A TBL_TYPES value that specifies the type of data in the column (such as U32, string, date and time, or UID). The values defined by TBL_TYPES are described in Table 90-2, below.

The width of the column in bytes. This argument only has meaning for columns of type tsChar and tsCaseChar.

The number of times to repeat the column. This allows you to create an array in a column. You cannot address repeated columns individually; rather, you access the whole column and locate the appropriate bytes.

The byte offset to the column. This allows you to locate data in a row by its offset, rather than by column index. clsTable allows you to request the entire row, or just the individual item. It is often much faster to request the entire row and use byte offsets to find individual pieces of data. This will not work if any of the columns in the table are sorted.

A BOOLEAN value that indicates whether the contents of the column should be sorted or not. When a column is sorted, data access on a specific column occurs in that column’s sort order, regardless of the position of the row in the table.

Currently sorting is only ascending (in ASCII lexicographic order) and does not support alternate sorting keys. There is no limit to the number of columns in a table that can have the sorted flag set.

The data types defined by TBL_TYPES are listed in Table 90-2. If the column contains fixed length data types, the actual values are stored in the row data. However, if the column contains variable-length data types (such as tsString or tsByteArray), the actual values are stored in buffers; the value stored in the row data is a pointer to the buffer. These pointers to variable-length data buffers are internal and cannot be used by clients. For this reason, tables with variable-length columns cannot use the Get-Row and Set-Row operations.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsChar</td>
<td>Fixed-length array of case-sensitive characters.</td>
</tr>
<tr>
<td>tsCaseChar</td>
<td>Fixed-length array of case-insensitive characters.</td>
</tr>
<tr>
<td>tsU16</td>
<td>Unsigned 16-bit integer.</td>
</tr>
<tr>
<td>tsU32</td>
<td>Unsigned 32-bit integer.</td>
</tr>
<tr>
<td>tsFP</td>
<td>GOMath floating point (GO_DP) value.</td>
</tr>
<tr>
<td>tsDate</td>
<td>Date field in system timestamp format.</td>
</tr>
<tr>
<td>tsString</td>
<td>Variable-length, case-sensitive ASCII string (null terminated).</td>
</tr>
<tr>
<td>tsCaseString</td>
<td>Variable-length, case-insensitive ASCII string (null terminated).</td>
</tr>
<tr>
<td>tsByteArray</td>
<td>Variable-length byte array, contained in unsigned characters. Use TS_STRING structure.</td>
</tr>
<tr>
<td>tsUUID</td>
<td>64-bit UUID structure.</td>
</tr>
</tbody>
</table>

Table 90-2

Table Column Data Types
Creating a Table Object

To create a table object send msgNewDefaults and msgNew to clsTable. Both messages take a pointer to a TBL_NEW structure, which contains:

- **name**  The name of the table. This name allows multiple clients to share a common name for the table and its data. (Note that this is not the file name; you specify the file name in the locator, below.)
- **locator**  A file locator for the table file, which includes a directory handle and a path. For more information on locators, see Part 7: File System.
- **exist**  A TBL_EXIST value that specifies what to do if the table file does or doesn’t exist. The constants defined by TBL_EXIST are similar in name and function to the file system FS_EXIST constants.
- **create**  A TBL_CREATE structure that describes the columns in the table.
- **freeBehavior**  A TBL_FREE_BEHAVE value that describes how to dispose of the table file when the table object is destroyed. This structure also identifies conditions under which clsTable should automatically destroy the table. These options are described below.
- **createSemaphore**  A BOOLEAN value that specifies whether to create a semaphore for the table.

Use the values defined by TBL_FREE_BEHAVE to specify what to do with the table file when the table is freed (and when to automatically free the table). The values are:

- **tsFreeDeleteFile**  Delete the file when the table is destroyed.
- **tsFreeWhenNoClients**  Free the table when the number of accessors goes to zero.
- **tsFreeNoObservers**  Free the table when the number of observers goes to zero.

The default (tsFreeDefault) is not to delete the file when the table is destroyed, do not free the table when accessors or observers goes to zero.

Observing Tables

There are two ways to add yourself as the observer of a table. You can either include your UID in the TBL_BEGIN_ACCESS structure when you send msgTBLBeginAccess to table object (see below) or you can send msgAddObserver to an existing table object (before you send msgTBLBeginAccess).

If you use msgAddObserver to add yourself as an observer, you should use msgRemoveObserver to remove yourself from the observer list. If you send msgTBLEndAccess and include your UID in the message arguments, the message will send msgRemoveObserver to the table object automatically.
This example shows how to create a table with two columns. One column holds a 40-byte string (a name), the other column uses a date to store a birthday:

```c
STATUS s;
TBL_NEW tn;
TBL_COL_DESC colDesc[2]; // two column table
s = ObjectCall(msgNewDefaults, clsTable, &tn);

tn.table.exist = tsExistOpen | tsNoExistCreate;
tn.table.freeBehavior = tsFreeNoDeleteFile;
tn.table.createSemaphore = false;

tn.table.locator.uid = theWorkingDir;
tn.table.locator.pPath = "\\MyDir\\Table Data";
strcpy(colDesc[0].name, "Name");
colDesc[0].type = tsChar;
colDesc[0].length = 40;

strcpy(colDesc[1].name, "Birthday");
colDesc[1].type = tsDate;

strcpy(tn.table.name, "Names and Birthdays");

tn.table.create.colCount = 2; // Number of columns

tn.table.create.colDescAry = colDesc; // First column descriptor

s = ObjectCall(msgNew, clsTable, &tn);
```

### Start Access

To initiate access to a table, send `msgTBLBeginAccess` to the table you want to access. `msgTBLBeginAccess` takes a pointer to a `TBL_BEGIN_ACCESS` structure, which contains your object’s UID (`sender`). The `sender` value is required if you want to add yourself (or another object) to the table’s observer list. If you don’t want to add the object to the table’s observer list, use `objNull`. When `msgTBLBeginAccess` completes successfully, it sends a `TBL_ROW_LENGTH` value containing the table’s width (`rowLength`).

When you begin access to a table, the current row position is at the end of the table. When you access a table, it is your responsibility to keep track of your position within the table. There is no message to indicate where you are.

```c
STATUS s;
TBL_NEW tn;
TBL_BEGIN_ACCESS tba;
TBL_ROW_LENGTH width;
...
s = ObjectCall(msgNew, clsTable, (P_ARGS) (&tn));
sharedTable = tn.object.uid;
// Fill in TBL_BEGIN_ACCESS structure
tba.sender = objNull;
s = ObjectCall(msgTBLBeginAccess, sharedTable, &tba);
width = tba.rowLength;
```
Using Semaphores

To ensure that your task is the only one accessing a table, you can request a table semaphore. Each table has one semaphore, which is managed by that table object. You specify whether the semaphore should be available when you create the table.

The semaphore does not actually control access to the table. Applications that use the table must agree beforehand (at programming time) to use the semaphore to signal among themselves who has access to the table.

You acquire the semaphore for a table by sending msgTBLSemaRequest to the table. The message doesn't require any arguments. If the semaphore is available, the call completes immediately. If the semaphore is in use, your task will be suspended until the semaphore becomes available.

When you complete the operations that require exclusive access to the table, you release the semaphore with msgTBLSemaClear.

It is a good idea to acquire the semaphore just before the operations and release it as soon as possible. This reduces the chance that other processes that require access to the table will be suspended.

Example 90-3
Using Table Semaphores

This example shows an object that requests a table's semaphore, performs some action, then releases the semaphore.

```c
TBL_BEGIN_ACCESS tba;
STATUS s;
...

// Fill in TBL_BEGIN_ACCESS structure
// tba = Nil(OBJECT);
if (ObjCallFailed(msgTBLBeginAccess, sharedTable, &tba)) { // Handle error, if any (there should be no error if // the table created a semaphore)
...

// Perform protected operation
...

s = ObjectCall(msgTBLSemaClear, sharedTable, void);
```

Adding Rows to a Table

Use msgTBLAddRow to add a row to a table. In the table objects there is no concept of a last row; sequential ordering of rows is not guaranteed. However, if you define a column with the sorted attribute and use that column to access rows, access will occur in an ordered manner.

msgTBLAddRow has one argument, a pointer to a row position value of type TBL_ROW_POS. The message uses this location to return the position of the newly added row.
This code fragment shows an application adding a row to a table:

```c
TBL_ROW_POS curPos;
...
s = ObjectCall(msgTBLAddRow, myTable, &curPos);
```

**Setting Data**

You can set data in a single column of a row or you can set data in an entire row. To set data in a column, send `msgTBLColSetData` to the table object; to set data in a row, send `msgTBLRowSetData`. Tables with variable-length columns cannot use `msgTBLRowSetData`, because the data for variable-length columns is stored using an internal, private format.

`msgTBLColSetData` takes a pointer to a `TBL_COL_GET_SET_DATA` structure, which contains:

- `tblRowPos` The row position in the table that will receive the data.
- `colNumber` The column number in the table that will receive the data.
- `tblColData` A pointer to the data buffer.

To set data in a variable-width column, the `tblColData` field should contain the address of a `TBL_STRING` structure that specifies:

- `strLen` A `UI6` that specifies the length of the data.
- `strMax` A `UI6` that specifies the size of the buffer. Usually `strLen` should be the same as `strMax`.
- `pStr` A pointer to the buffer that receives the data.

`msgTBLRowSetData` takes a pointer to a `TBL_GET_SET_ROW` structure that contains:

- `tblRowPos` A `TBL_ROW_POS` value that will receive the data.
- `pRowData` A pointer to the data buffer that contains an image of the entire row. `msgTBLRowSetData` returns `stsTBLContainsIndexedCols` if any of the columns in the table are variable-length.

**Getting Data**

Getting data is similar to setting data. You can get data for a single column within a row (by sending `msgTBLColGetData` to the table object), or you can get data for an entire row (by sending `msgTBLRowGetData` to the table object).

Again the structures are similar. `msgTBLColGetData` takes a pointer to a `TBL_COL_GET_SET_DATA` structure that contains:

- `tblRowPos` A `TBL_ROW_POS` value that specifies the row position.
- `colNumber` A `TBL_COL_INX_TYPE` value that specifies the column number.
- `tblColData` A pointer to the buffer that will receive the data.

If your client allocates the data buffer on an as-needed basis, you can use `msgTBLGetColDesc` to find out the current width and the data type of
fixed-width columns. Note that tables do not allocate the data buffer; it is your client's responsibility.

To get data for variable-width columns, store the address of a TBL_STRING structure in the tblColData field. The TBL_STRING structure specifies:

- `strLen` A U16 to receive the length of the returned data.
- `strMax` A U16 that specifies the size of the buffer.
- `pStr` A pointer to the buffer that receives the data.

If the size of the data is larger than the buffer (`strMax`), the data is truncated, `strLen` contains the size of the data returned, and the message returns the status `stsTBLStrBuffTooSmall`.

`msgTBLRowGetData` takes a pointer to a TBL_GET_SET_ROW structure that contains:

- `tblRowPos` A TBL_ROW_POS value that specifies the row that we want.
- `pRowData` A pointer to a data buffer that will receive an image of the entire row.

If you allocate the data buffer dynamically, you can use `msgTBLGetRowLength` to find out the length of a row.

### Deleting a Row

You delete a row from a table by sending `msgTBLDeleteRow` to the table object. The row is not actually deleted until the client sends `msgTBLCompact` to the table object or until the table object is freed and the file is closed. If you want to prevent compaction when the file is closed, specify `tsFreeNoCompact` in the TBL_FREE_BEHAVE argument to `msgNew`.

After you delete a row, you can no longer access it, even though it hasn't actually been deleted from the file.

`msgTBLDeleteRow` takes one argument, a pointer to a TBL_ROW_POS value that identifies the row that you want to delete.

### Searching a Table

To search a table for a particular item, use the messages `msgTBLFindFirst` and `msgTBLFindNext`.

Use `msgTBLFindFirst` to search for the first occurrence of a particular item in a table. Use `msgTBLFindNext` to search for the next occurrence of an item when searching from a specified position.

Both messages take a pointer to a TBL_FIND_ROW structure that contains:

- `rowPos` A TBL_ROW_POS value that specifies the current table position.

  When the message completes, `rowPos` will contain the row position of the located row.
rowNum A TBL_ROW_NUM value that specifies the index position in the specified column.

srchSpec A TBL_SEARCH_SPEC structure that contains the search specification. In this structure you specify:

- colOperand The column number.
- relOp A TBL_BOOL_OP value that specifies the operator used to match the search string against the column string in each row. The operators are described below in Table 90-3.
- pConstOperand A pointer to a buffer containing the search item.
- pRowBuffer A pointer to a ROW_BUFFER that specifies the client's buffer space (this might be pNull).
- sortCol A TBL_COL_INX_TYPE value that specifies the column to sort the search by, if any. If sortCol is null, there is no sort.

The operands in the table search specification read the way they would if they were written in an equation, that is: column operand, operator, search constant. Thus, a less-than operator means "search until the column operand is less than the search constant."

The BOOLEAN operators are defined by TBL_BOOL_OP. Table 90-3 lists the BOOLEAN operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsEqual</td>
<td>Satisfied only if both items the same object.</td>
</tr>
<tr>
<td>tsLess</td>
<td>Satisfied if the column operand is less than the search constant.</td>
</tr>
<tr>
<td>tsGreater</td>
<td>Satisfied if the column operand is greater than the search constant.</td>
</tr>
<tr>
<td>tsGreaterEqual</td>
<td>Satisfied if the column operand is greater than or equal to the search constant.</td>
</tr>
<tr>
<td>tsLessEqual</td>
<td>Satisfied if the column operand is less than or equal to the search constant.</td>
</tr>
<tr>
<td>tsNotEqual</td>
<td>Satisfied if the column operand is not equal to the search constant.</td>
</tr>
<tr>
<td>tsSubstring</td>
<td>Satisfied if the column operand is a substring of the search constant. This operator is currently limited to case-dependent searches, even when searching a tsCaseChar or tsCaseString column.</td>
</tr>
<tr>
<td>tsStartsWith</td>
<td>Satisfied if the column starts with the specified string.</td>
</tr>
<tr>
<td>tsAlwaysTrue</td>
<td>Matches everything. Use tsAlwaysTrue to match the first row that the search encounters. For msgTBLFindFirst, this is the first row in the table; for msgTBLFindNext, this is the next row in the table (unless you are at the end of the table).</td>
</tr>
</tbody>
</table>

If the message finds a match, it returns stsOK and passes back the TBL_FIND_ROW structure with:

- rowPos A TBL_ROW_POS value that indicates the row where the match was found.
- rowNum A TBL_ROW_NUM value that indicates the indexed row number for sorted columns. If the column was not a sorted column, this value always contains 0.
If either `msgTBLFindFirst` or `msgTBLFindNext` does not find a match, or if the row position is at the end of the table, the message returns `stsTBLEndOfTable`.

### Getting Information About a Table

`clsTable` provides a number of ways to get information about a table. Table 90-4 lists the messages and the information they return. The sections following the table describe the messages in detail.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>msgTBLFindColNum</code></td>
<td><code>P_TBL_COL_NUM_FIND</code></td>
<td>Passes back the column number for the specified column name.</td>
</tr>
<tr>
<td><code>msgTBLRowNumToRowPos</code></td>
<td><code>P_TBL_CONVERT_ROW_NUM</code></td>
<td>Converts a <code>TBL_ROW_NUM</code> to its corresponding <code>TBL_ROW_POS</code> for the specified column.</td>
</tr>
<tr>
<td><code>msgTBLGetInfo</code></td>
<td><code>P_TBL_HEADER</code></td>
<td>Gets the table header information.</td>
</tr>
<tr>
<td><code>msgTBLGetColCount</code></td>
<td><code>P_TBL_COL_COUNT</code></td>
<td>Gets the number of columns in the table.</td>
</tr>
<tr>
<td><code>msgTBLGetColDesc</code></td>
<td><code>P_TBL_GET_COL_DESC</code></td>
<td>Passes back the column description for the specified column.</td>
</tr>
<tr>
<td><code>msgTBLGetRowCount</code></td>
<td><code>P_TBL_ROW_COUNT</code></td>
<td>Gets the current number of rows in the table.</td>
</tr>
<tr>
<td><code>msgTBLGetRowLength</code></td>
<td><code>P_TBL_ROW_LENGTH</code></td>
<td>Gets the length (in bytes) of the specified row.</td>
</tr>
<tr>
<td><code>msgTBLGetState</code></td>
<td><code>P_TBL_GET_STATE</code></td>
<td>Gets the current state.</td>
</tr>
</tbody>
</table>

The following sections describe these messages.

#### Finding a Column Number

If you have a column name string and need to find out the number of the column, send `msgTBLFindColNum` to the table. The message takes a pointer to a `TBL_COL_NUM_FIND` structure, in which you specify a pointer to the column name string (`name`).

When the message completes successfully, it returns `stsOK` and passes back the `TBL_COL_NUM_FIND` structure with a `TBL_COL_INX_TYPE` value that contains column number (`number`).

#### Converting a Row Number to a Row Position

To convert a row number to a row position for a specific column, send `msgTBLRowNumToRowPos` to the table object. The message takes a pointer to a `TBL_CONVERT_ROW_NUM` structure that contains:

- `rowNum`  A `TBL_ROW_NUM` value that specifies the row number to convert.
- `colNum`  A `TBL_COL_INX_TYPE` value that specifies the sorted column to use in the conversion.

If the message completes successfully, it returns `stsOK` and passes back a `TBL_ROW_POS` value in `rowPos` that specifies the position of the row.

If the column is not sorted, the message returns `stsTBLColNotIndexed`.
Getting the Number of Columns in a Table

Send msgTBLGetColCount to a table to get the number of columns in the table. The only argument for this message is a pointer to the TBL_COL_COUNT value that will receive the column count. When the message completes successfully, it returns stdout and passes back the column count.

Getting the Description of a Column

Send msgTBLGetColDesc to a table to get the description of a column in the table. The message takes a pointer to a TBL_GET_COL_DESC structure that contains a TBL_COL_INX_TYPE value that specifies the index of the column that you want (colIm).

When the message completes successfully, it returns stdout and passes back a TBL_COL_DESC structure that will contain the column descriptor (colDesc) information.

Getting the Entire Table Description

Send msgTBLGetInfo to a table to get the entire description of that table. The only argument for this message is a pointer to a TBL_HEADER structure that will receive the table description.

Getting the Number of Rows

Send msgTBLGetRowCount to a table to get the number of rows in that table. The only argument for this message is a pointer to a TBL_ROW_COUNT value that will receive the number of rows.

Getting the Length of a Row

Send msgTBLGetRowLength to a table to get the number of bytes in a table row. The only argument for this message is a pointer to a TBL_ROW_LENGTH value that will receive the number of bytes in a row.

The length passed back by this message does not include the length of variable-length data. To get the width of a variable length column for a particular row, send msgTBLColGetData to the table object, specifying the row, column, a pointer to a TBL_STRING structure, and specify strMax as 0. When the message returns strLen contains the length of the data.

Getting a Table's State

Occasionally you might need to find out a table's state. To get the state, send msgTBLGetState to a table object. The message takes a pointer to a TBL_GET_STATE structure, that contains:

- tblState A TBL_STATE enum value that will receive an indicator of the position within the table.
tblRowPos  A TBL_ROW_POS value that specifies the row for which you are requesting the table state.

Ending Access

When you have no longer need to access a table, send msgTBLEndAccess to the table. The message takes a pointer to a TBL_END_ACCESS structure that contains a single element, the UID of the sender (sender). This is usually self. If you specify sender, the message will remove the sender from the table's observer list.

The message also decrements the table's reference count.

Freeing a Table

When a table is no longer useful, destroy it by sending it msgDestroy. The options specified when the table was created determine if the file should be preserved and whether the rows should be compacted.

If a table's owner terminates without explicitly destroying the table, PenPoint sends msgFree to the table object.
Chapter 91 / The NotePaper Component

The NotePaper component, consisting primarily of the clsNotePaper view and the clsNPData data object, provides a capable building block for applications that manage ink as a data type. For example, the NotePaper component provides much of the function of the the MiniNote note-taking application that comes bundled with the PenPoint™ operating system.

clsNotePaper is a subclass of clsView. Like all subclasses of clsView, clsNotePaper is designed to interact with and display a data object—in this case an instance of clsNPData. A clsNPData data object manages a collection of data items whose classes inherit from the abstract class clsNPltem.

This chapter presents the API for clsNotePaper, clsNPData, and clsNPltem. To get a better understanding of the way these classes interact, you should study the NotePaperApp sample application, a simple note-taking application. The source code for this application is in the SDK sample application directory \PENPOINT\SDK\SAMPLE\NPAPP.

The clsNotePaper View

clsNotePaper is a subclass of clsView designed to observe a data object of clsNPData. clsNotePaper supports embedding, undo, move and copy, import, export, and option sheets. It also supports marks, which in turn provide support for search and replace, spell checking, and reference buttons. With all of these features, clsNotePaper is a very capable class.

clsNotePaper displays and alters the contents of a NotePaper data object, an instance of clsNPData. The data object maintains a database of items. The view sends messages to the data object to alter or query the database, and the data object notifies the view when it needs to update its presentation of the data items.

In displaying the data items, clsNotePaper maintains a coordinate system whose origin is the upper left corner of the view. This has the advantage that, as the NotePaper window changes in width and height, its contents remain relative to the upper left corner of the page (the expected behavior for notes). One thing to be aware of, though, is that an upper left origin means that all y (vertical) coordinates are negative.
NotePaper Metrics

clsNotePaper maintains a set of metrics including a paper style, a pen style, font and line spacing for displaying text, and a set of flags that determines various behaviors of the instance. All of this information is represented in a NOTE_PAPER_METRICS data structure, which includes the following fields (the data type follows the field name in parentheses):

paperStyle (NP_PAPER_STYLE)  The style of the “paper” the clsNotePaper instance displays. paperStyle can have any one of the following enumerated values:

  npPaperRuled  Horizontal rules but no vertical rules.
  npPaperRuledLeftMargin  Horizontal rules with a single vertical rule down the left side of the page.
  npPaperRuledCenterMargin  Horizontal rules with a single vertical rule down the center of the page.
  npPaperRuledLegalMargin  Horizontal rules with two vertical rules dividing the page into thirds.
  npPaperBlank  No horizontal or vertical rules.
  npPaperLeftMargin  No horizontal rules and a single vertical rule running down the left side of the page.
  npPaperCenterMargin  No horizontal rules and a single vertical rule running down the center of the page.
  npPaperGrid  Horizontal rules with vertical rules across the page with the same spacing as the horizontal rules.

penStyle (U8)  The pen color and weight. To generate valid values, use the NPPenStyle() macro.

paperFont (SYSDC_FONT_SPEC)  The font for displaying text items.

lineSpacing (COORD16)  The font size and distance between horizontal rules, measured in twips (a twip is \( \frac{1}{20} \) of a point, or \( \frac{1}{1440} \) of an inch.

style (NOTE_PAPER_STYLE)  A set of bit fields that determine various behaviors of the view. The bit field names and their meanings are:

  bEditMode  Gesture mode if true, ink mode if false.
  bAutoGrow  If true, automatically grow in height as user enters data.
  bWidthOpts  Include page width in option sheet if true.
  bHideTopRule  If true, don’t display the top-most horizontal rule when paperStyle is set to one of the npPaperRuled... styles.
# NotePaper Messages

Table 91-1 summarizes the messages clsNotePaper defines. See NOTEPAKR.H and the NPAPP sample application for more information about their use.

## Table 91-1  
<table>
<thead>
<tr>
<th>clsNotePaper Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
</tr>
<tr>
<td>msgNewDefaults</td>
</tr>
<tr>
<td>msgNotePaperGetMetrics</td>
</tr>
<tr>
<td>msgNotePaperSetEditMode</td>
</tr>
<tr>
<td>msgNotePaperSetPaperAndPen</td>
</tr>
<tr>
<td>msgNotePaperSetPenStyle</td>
</tr>
<tr>
<td>msgNotePaperGetPenStyle</td>
</tr>
<tr>
<td>msgNotePaperAddMenus</td>
</tr>
<tr>
<td>msgNotePaperAddModeCtrl</td>
</tr>
<tr>
<td>msgNotePaperClear</td>
</tr>
<tr>
<td>msgNotePaperClearSel</td>
</tr>
<tr>
<td>msgNotePaperInsertLine</td>
</tr>
<tr>
<td>msgNotePaperSetStyle</td>
</tr>
<tr>
<td>msgNotePaperGetStyle</td>
</tr>
<tr>
<td>msgNotePaperGetSelType</td>
</tr>
<tr>
<td>msgNotePaperTranslate</td>
</tr>
<tr>
<td>msgNotePaperUntranslate</td>
</tr>
<tr>
<td>msgNotePaperEdit</td>
</tr>
<tr>
<td>msgNotePaperGetDclnfo</td>
</tr>
<tr>
<td>msgNotePaperTidy</td>
</tr>
<tr>
<td>msgNotePaperCenter</td>
</tr>
<tr>
<td>msgNotePaperAlign</td>
</tr>
<tr>
<td>msgNotePaperMerge</td>
</tr>
<tr>
<td>msgNotePaperSplit</td>
</tr>
<tr>
<td>msgNotePaperSelectRect</td>
</tr>
<tr>
<td>msgNotePaperSelectLine</td>
</tr>
</tbody>
</table>
### Table 91-2 (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNotePaperDeselectLine</td>
<td>P_RECT32</td>
<td>Deselects items whose baselines intersect rect in the receiver’s data.</td>
</tr>
<tr>
<td>msgNotePaperDeleteLine</td>
<td>P_RECT32</td>
<td>Deletes items whose baselines intersect rect in the view’s data.</td>
</tr>
<tr>
<td>msgNotePaperScribble</td>
<td>OBJECT</td>
<td>Handles scribble (including creating and insert object into view’s data).</td>
</tr>
<tr>
<td>msgGWInGesture</td>
<td>P_GWIN_GESTURE</td>
<td>Self-sent to process the gesture.</td>
</tr>
<tr>
<td>msgAppSelectAll</td>
<td>P_NULL</td>
<td>Selects all items in the view.</td>
</tr>
<tr>
<td>msgSelDelete</td>
<td>P_NULL</td>
<td>Deletes selected items in the view.</td>
</tr>
<tr>
<td>msgOptionAddCards</td>
<td>P_OPTION_TAG</td>
<td>Creates and adds the Pen and Paper option sheets.</td>
</tr>
<tr>
<td>msgImportQuery</td>
<td>P_IMPORT_QUERY</td>
<td>Indicates whether or not passed-in file can be imported.</td>
</tr>
<tr>
<td>msgImport</td>
<td>P_IMPORT_DOC</td>
<td>Imports the passed in file.</td>
</tr>
<tr>
<td>msgExportGetFormats</td>
<td>PEXPORT_LIST</td>
<td>Passes back list of formats that can be exported.</td>
</tr>
<tr>
<td>msgExport</td>
<td>PEXPORT_DOC</td>
<td>Writes an ASCII version of receiver’s data to the passed in file.</td>
</tr>
<tr>
<td>msgNotePaperUpdateSel</td>
<td>NULL</td>
<td>Takes or releases the selection as is appropriate.</td>
</tr>
<tr>
<td>msgNotePaperSplitAsAtoms</td>
<td>NULL</td>
<td>Splits the selected item to its constituent pieces.</td>
</tr>
<tr>
<td>msgNotePaperSplitAsWords</td>
<td>NULL</td>
<td>Splits the selected scribbles and text into words.</td>
</tr>
<tr>
<td>msgNotePaperGrowHeightTo</td>
<td>COORD32</td>
<td>Grows the height of the view to at least COORD32.</td>
</tr>
<tr>
<td>msgNotePaperGrowHeightBy</td>
<td>COORD32</td>
<td>Grows the height of the view by COORD32.</td>
</tr>
</tbody>
</table>

## NotePaper Data

Like all views, clsNotePaper is designed to display a rendition of the data in an observed data object. The data object it uses is an instance of clsNPData, which maintains a database of “items,” accepts requests to change the items, and notifies observers of the changes.

clsNPData maintains items of any type, so long as they support the protocol defined by the abstract class clsNPItem (discussed later). All clsNPData has to do is maintain a set of objects whose classes inherit from clsNPItem. Each of the different subclasses of clsNPItem (PenPoint includes scribble and text item classes) responds differently to a single protocol of messages which clsNPData uses.

Table 91-2 summarizes the messages clsNPData defines.
### clsNPData Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_NP_DATA_NEW</td>
<td>Initializes pArgs.</td>
</tr>
<tr>
<td>msgNPDataInsertItem</td>
<td>OBJECT</td>
<td>Adds item to the data base.</td>
</tr>
<tr>
<td>msgNPDataInsertItemFromView</td>
<td>P_NP_DATA_ADDED_VNP_ITEM_VIEW</td>
<td>Adds item to the data base.</td>
</tr>
<tr>
<td>msgNPDataDeleteItem</td>
<td>OBJECT</td>
<td>Deletes an item from the data base.</td>
</tr>
<tr>
<td>msgNPDataMoveItem</td>
<td>P_NP_DATA_XY</td>
<td>Moves an item within the data base.</td>
</tr>
<tr>
<td>msgNPDataMoveItems</td>
<td>P_MOVE_ITEMS</td>
<td>Moves all items below pArgs-&gt;y by pArgs-&gt;yDelta.</td>
</tr>
</tbody>
</table>

#### Database Manipulation Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNPDataEnumOverlappedItems</td>
<td>P_ENUM_RECT_ITEMS</td>
<td>Enumerates each item that overlaps the given rectangle.</td>
</tr>
<tr>
<td>msgNPDataEnumBaselineItems</td>
<td>P_ENUM_RECT_ITEMS</td>
<td>Enumerates each item whose baseline overlaps the given rectangle.</td>
</tr>
<tr>
<td>msgNPDataEnumSelectedItems</td>
<td>P_ENUM_ITEMS</td>
<td>Enumerates each item that is selected (in paint order).</td>
</tr>
<tr>
<td>msgNPDataEnumSelectedItemsReverse</td>
<td>P_ENUM_ITEMS</td>
<td>Enumerates each item that is selected (in reverse paint order).</td>
</tr>
<tr>
<td>msgNPDataEnumAllItems</td>
<td>P_ENUM_ITEMS</td>
<td>Enumerates each item (in paint order).</td>
</tr>
<tr>
<td>msgNPDataEnumAllItemsReverse</td>
<td>P_ENUM_ITEMS</td>
<td>Enumerates each item (in reverse paint order).</td>
</tr>
<tr>
<td>msgNPDataSendEnumSelectedItems</td>
<td>P_SEND_ENUM_ITEMS</td>
<td>Enumerates each selected item (in paint order).</td>
</tr>
<tr>
<td>msgNPDataGetCurrentItem</td>
<td>P_OBJECT</td>
<td>Passes back the current item in the receiver.</td>
</tr>
<tr>
<td>msgNPDataGetNextItem</td>
<td>P_OBJECT</td>
<td>Increments the current item to the next item and sets *pArgs to it.</td>
</tr>
</tbody>
</table>

#### Enumeration Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNPDataItemCount</td>
<td>P_U32</td>
<td>Passes back the count of items in the receiver.</td>
</tr>
<tr>
<td>msgNPDataSelectedCount</td>
<td>P_U32</td>
<td>Passes back the count of selected items in receiver.</td>
</tr>
<tr>
<td>msgNPDataGetBaseline</td>
<td>P_XY32</td>
<td>Sets the receiver's baseline (used for alignment).</td>
</tr>
<tr>
<td>msgNPDataGetBaseline</td>
<td>P_XY32</td>
<td>Gets the receiver's baseline (used for alignment).</td>
</tr>
<tr>
<td>msgNPDataGetLineSpacing</td>
<td>P_XY32</td>
<td>Sets receiver's line spacing (used as the font size).</td>
</tr>
<tr>
<td>msgNPDataGetLineSpacing</td>
<td>P_XY32</td>
<td>Gets receiver's line spacing (used as the font size).</td>
</tr>
<tr>
<td>msgNPDataGetBounds</td>
<td>P_RECT32</td>
<td>Passes back the bounding rectangle for all items in receiver.</td>
</tr>
<tr>
<td>msgNPDataGetSelBounds</td>
<td>P_RECT32</td>
<td>Passes back the bounding rectangle for all selected items in receiver.</td>
</tr>
<tr>
<td>msgNPDataGetFontSpec</td>
<td>P_SYSDC_FONT_SPEC</td>
<td>Passes back the receiver's font specification.</td>
</tr>
<tr>
<td>msgNPDataSetFontSpec</td>
<td>P_SYSDC_FONT_SPEC</td>
<td>Sets the receiver's font specification.</td>
</tr>
<tr>
<td>msgNPDataGetCachedDCs</td>
<td>P_NP_DATA_DC</td>
<td>Passes back DC's with normal and bold fonts at the given line spacing.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 91-2**

### Class Messages

**msgNPDataEnumOverlappedItems**

- **P_ENUM_RECT_ITEMS**: This template describes the the callback function used in item enumeration.

**msgNPDataEnumBaselineItems**

- **P_ENUM_RECT_ITEMS**: Enumerates each item that overlaps the given rectangle.

**msgNPDataEnumSelectedItems**

- **P_ENUM_ITEMS**: Enumerates each item whose baseline overlaps the given rectangle.

**msgNPDataEnumSelectedItemsReverse**

- **P_ENUM_ITEMS**: Enumerates each item that is selected (in paint order).

**msgNPDataEnumAllItems**

- **P_ENUM_ITEMS**: Enumerates each item (in paint order).

**msgNPDataEnumAllItemsReverse**

- **P_ENUM_ITEMS**: Enumerates each item (in reverse paint order).

**msgNPDataSendEnumSelectedItems**

- **P_SEND_ENUM_ITEMS**: Enumerates each selected item (in paint order).

**msgNPDataGetCurrentItem**

- **P_OBJECT**: Passes back the current item in the receiver.

**msgNPDataGetNextItem**

- **P_OBJECT**: Increments the current item to the next item and sets *pArgs to it.
NotePaper Data Items

The NotePaper data item class, clsNPltem, defines a protocol of messages that defines the interactions possible between an instance of clsNPData and the items it maintains. clsNPltem is an abstract class; it handles only the generic behavior of the messages it defines. Instances of clsNPltem are not generally useful. Instead, the items a clsNPData object maintains are instances of subclasses of clsNPltem, not of clsNPltem itself. PenPoint includes two such subclasses: clsNPscirbbleItem, an ink scribble, and clsNPTextItem, a text item.

Table 91-3 summarizes the messages that clsNPltem and its subclasses handle:
<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNPItemAlignToBaseline</td>
<td>P_XY32</td>
<td>Moves item so that it aligns to passed-in line spacing.</td>
</tr>
<tr>
<td>msgNPItemPaint</td>
<td>P_NP_ITEM_DC</td>
<td>Paints item using the passed in drawing contexts.</td>
</tr>
<tr>
<td>msgNPItemSetPenStyle</td>
<td>U32</td>
<td>Sets the item's pen style. (Pen styles are defined in NOTEPAPR.H.)</td>
</tr>
<tr>
<td>msgNPItemSetOrigin</td>
<td>P_XY32</td>
<td>Set the receiver's origin.</td>
</tr>
<tr>
<td>msgNPItemScratchOut</td>
<td>P_RECT32</td>
<td>Handles the scratch-out gesture on an item.</td>
</tr>
<tr>
<td>msgNPItemSplit</td>
<td>P_XY32</td>
<td>Handles the split gesture on an item.</td>
</tr>
<tr>
<td>msgNPItemSplitAsWords</td>
<td>NULL</td>
<td>Splits an item into its constituent items.</td>
</tr>
<tr>
<td>msgNPItemJoin</td>
<td>OBJECT</td>
<td>Splits receiver into words. Deletes receiver, inserts new items.</td>
</tr>
<tr>
<td>msgNPItemJoin</td>
<td>OBJECT</td>
<td>Joins receiver and OBJECT and deletes OBJECT.</td>
</tr>
<tr>
<td>msgNPItemJoin</td>
<td>OBJECT</td>
<td>Joins OBJECT and receiver and deletes them. Inserts new object.</td>
</tr>
<tr>
<td>msgNPItemGetScribble</td>
<td>P_OBJECT</td>
<td>Item converts itself to a text item, passes back text item.</td>
</tr>
<tr>
<td>msgNPItemGetString</td>
<td>P_STRING</td>
<td>Pass back the item's scribble.</td>
</tr>
<tr>
<td>msgNPItemSetString</td>
<td>P_STRING</td>
<td>Passes back the text string for the item.</td>
</tr>
<tr>
<td>msgNPItemSetText</td>
<td>P_OBJECT</td>
<td>Sets the text string for the item.</td>
</tr>
<tr>
<td>msgNPItemJoin</td>
<td>OBJECT</td>
<td>Item converts itself to a text item, passes back text item.</td>
</tr>
<tr>
<td>msgNPItemGetScribble</td>
<td>P_ARGS</td>
<td>Item converts itself to a scribble item.</td>
</tr>
<tr>
<td>msgNPItemGetScribble</td>
<td>P_ARG32</td>
<td>Returns stsOK if receiver's path overlaps pArgs.</td>
</tr>
<tr>
<td>msgNPItemCalcBaseline</td>
<td>P_XY32</td>
<td>Calculates and sets the receiver's baseline.</td>
</tr>
<tr>
<td>msgNPItemCalcBounds</td>
<td>OBJECT</td>
<td>The receiver calculates and sets its new bounds.</td>
</tr>
<tr>
<td>msgNPItemGetWordSpacing</td>
<td>P_U16</td>
<td>The receiver passes back the size of its &quot;space&quot; character.</td>
</tr>
<tr>
<td>msgNPItemCanBeTranslated</td>
<td>pNull</td>
<td>The receiver returns stsOK if it can be translated.</td>
</tr>
<tr>
<td>msgNPItemCanBeUntranslated</td>
<td>pNull</td>
<td>The receiver returns stsOK if it can be untranslated.</td>
</tr>
<tr>
<td>msgNPItemHasString</td>
<td>pNull</td>
<td>The receiver returns stsOK if the item has a string.</td>
</tr>
<tr>
<td>msgNPItemCanJoin</td>
<td>OBJECT</td>
<td>Subclasses should return stsOK of they can join with OBJECT.</td>
</tr>
<tr>
<td>msgNPItemSetDataObject</td>
<td>OBJECT</td>
<td>Sets the receiver's data (e.g. instance of clsNPData).</td>
</tr>
<tr>
<td>msgNPItemSetAdjunct</td>
<td>OBJECT</td>
<td>Sets the receiver's adjunct item.</td>
</tr>
<tr>
<td>msgNPItemMark</td>
<td>P_U32</td>
<td>The receiver marks itself and provides its mark token.</td>
</tr>
<tr>
<td>msgNPItemUnmark</td>
<td>P_ARGS</td>
<td>The receiver unmarks itself and passes back the mark ID if applicable.</td>
</tr>
<tr>
<td>msgNPItemGetMarkId</td>
<td>P_U32</td>
<td>Passes back the mark ID of an item.</td>
</tr>
</tbody>
</table>
Part 10 / Connectivity
<table>
<thead>
<tr>
<th>Chapter 92 / Introduction</th>
<th>241</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout of This Part</td>
<td>92.1</td>
</tr>
<tr>
<td>Other Sources of Information</td>
<td>92.2</td>
</tr>
<tr>
<td>Chapter 93 / Concepts and Terminology</td>
<td>243</td>
</tr>
<tr>
<td>Principles of PenPoint Connectivity</td>
<td>93.1</td>
</tr>
<tr>
<td>PenPoint Connectivity Strategies</td>
<td>93.2</td>
</tr>
<tr>
<td>PenPoint Computer Connectivity</td>
<td>93.3</td>
</tr>
<tr>
<td>PenPoint Connectivity MIL Services</td>
<td>93.4</td>
</tr>
<tr>
<td>MIL Services and Other Services</td>
<td>93.4.1</td>
</tr>
<tr>
<td>Services and Interfaces</td>
<td>93.4.2</td>
</tr>
<tr>
<td>The Service Manager</td>
<td>93.5</td>
</tr>
<tr>
<td>PenPoint Facilities</td>
<td>93.6</td>
</tr>
<tr>
<td>Adding Network Protocols</td>
<td>93.7</td>
</tr>
<tr>
<td>Survey of the Remote Interfaces</td>
<td>93.8</td>
</tr>
<tr>
<td>Serial I/O</td>
<td>93.8.1</td>
</tr>
<tr>
<td>Parallel I/O</td>
<td>93.8.2</td>
</tr>
<tr>
<td>High-Speed Packet I/O</td>
<td>93.8.3</td>
</tr>
<tr>
<td>File System</td>
<td>93.8.4</td>
</tr>
<tr>
<td>File Import Export</td>
<td>93.8.5</td>
</tr>
<tr>
<td>Modem Interface</td>
<td>93.8.6</td>
</tr>
<tr>
<td>Networking Interface</td>
<td>93.8.7</td>
</tr>
<tr>
<td>Chapter 94 / Using Services</td>
<td>255</td>
</tr>
<tr>
<td>Concepts</td>
<td>94.1</td>
</tr>
<tr>
<td>Installing Services</td>
<td>94.1.1</td>
</tr>
<tr>
<td>Service Managers</td>
<td>94.1.2</td>
</tr>
<tr>
<td>Access Overview</td>
<td>94.1.3</td>
</tr>
<tr>
<td>Ownership</td>
<td>94.1.4</td>
</tr>
<tr>
<td>Targeting</td>
<td>94.1.5</td>
</tr>
<tr>
<td>Connections</td>
<td>94.1.6</td>
</tr>
<tr>
<td>Accessing Services</td>
<td>94.2</td>
</tr>
<tr>
<td>Predefined Service Managers</td>
<td>94.2.1</td>
</tr>
<tr>
<td>Binding to a Service</td>
<td>94.2.2</td>
</tr>
<tr>
<td>Opening a Service</td>
<td>94.2.3</td>
</tr>
<tr>
<td>The Service Manager Messages</td>
<td>94.3</td>
</tr>
<tr>
<td>Using the Service Manager</td>
<td>94.4</td>
</tr>
<tr>
<td>Accessing a Service</td>
<td>94.4.1</td>
</tr>
<tr>
<td>Finding a Service</td>
<td>94.4.2</td>
</tr>
<tr>
<td>Binding to a Service</td>
<td>94.4.3</td>
</tr>
<tr>
<td>Opening and Closing a Service</td>
<td>94.4.4</td>
</tr>
<tr>
<td>Unbinding from a Service</td>
<td>94.4.5</td>
</tr>
<tr>
<td>Finding a Handle</td>
<td>94.4.6</td>
</tr>
<tr>
<td>Receiving Connection State Notification</td>
<td>94.4.7</td>
</tr>
<tr>
<td>Setting a Service Owner</td>
<td>94.4.8</td>
</tr>
<tr>
<td>Chapter 95 / Serial I/O</td>
<td>265</td>
</tr>
<tr>
<td>Serial I/O Concepts</td>
<td>95.1</td>
</tr>
<tr>
<td>Interrupt Driven I/O</td>
<td>95.1.1</td>
</tr>
<tr>
<td>Buffered Data</td>
<td>95.1.2</td>
</tr>
<tr>
<td>Flow Control</td>
<td>95.1.3</td>
</tr>
<tr>
<td>Events</td>
<td>95.1.4</td>
</tr>
<tr>
<td>Concurrency Issues</td>
<td>95.1.5</td>
</tr>
<tr>
<td>Using Serial Messages</td>
<td>95.2</td>
</tr>
<tr>
<td>Requesting and Releasing a Serial Handle</td>
<td>95.2.1</td>
</tr>
<tr>
<td>Reinitializing the Serial Port</td>
<td>95.2.2</td>
</tr>
<tr>
<td>Serial Port Configuration</td>
<td>95.2.3</td>
</tr>
<tr>
<td>Reading and Writing with the Serial Port</td>
<td>95.2.4</td>
</tr>
<tr>
<td>Flow Control</td>
<td>95.2.5</td>
</tr>
<tr>
<td>Sending BREAK</td>
<td>95.2.6</td>
</tr>
<tr>
<td>Detecting Events</td>
<td>95.2.7</td>
</tr>
<tr>
<td>High-Speed Packet I/O Concepts</td>
<td>95.3</td>
</tr>
<tr>
<td>HSPKT on Serial Lines</td>
<td>95.3.1</td>
</tr>
<tr>
<td>Parallel Cable Connection Detection</td>
<td>95.3.2</td>
</tr>
<tr>
<td>Protocol Variations</td>
<td>95.3.3</td>
</tr>
<tr>
<td>Notes</td>
<td>95.3.4</td>
</tr>
<tr>
<td>Chapter 96 / Parallel I/O</td>
<td>275</td>
</tr>
<tr>
<td>Parallel Port Concepts</td>
<td>96.1</td>
</tr>
<tr>
<td>Parallel Port Interrupts</td>
<td>96.1.1</td>
</tr>
<tr>
<td>Parallel Port Messages</td>
<td>96.2</td>
</tr>
<tr>
<td>Using the Parallel Port</td>
<td>96.3</td>
</tr>
<tr>
<td>Requesting a Parallel Port Handle</td>
<td>96.3.1</td>
</tr>
<tr>
<td>Releasing the Parallel Port Object</td>
<td>96.3.2</td>
</tr>
<tr>
<td>Parallel Port Configuration</td>
<td>96.3.3</td>
</tr>
<tr>
<td>Initializing the Printer</td>
<td>96.3.4</td>
</tr>
<tr>
<td>Writing to the Parallel Port</td>
<td>96.3.5</td>
</tr>
<tr>
<td>Getting Status</td>
<td>96.3.6</td>
</tr>
<tr>
<td>Cancellng Printing</td>
<td>96.3.7</td>
</tr>
<tr>
<td>Chapter 97 / Data Modem Interface</td>
<td>279</td>
</tr>
<tr>
<td>Concepts</td>
<td>97.1</td>
</tr>
<tr>
<td>Getting a Serial Port Handle</td>
<td>97.1.1</td>
</tr>
<tr>
<td>Configuring the Serial Port</td>
<td>97.1.2</td>
</tr>
<tr>
<td>The clsModem API</td>
<td>97.2</td>
</tr>
<tr>
<td>The clsModem Messages</td>
<td>97.3</td>
</tr>
<tr>
<td>Creating a clsModem Object</td>
<td>97.3.1</td>
</tr>
<tr>
<td>Configuring the Modem</td>
<td>97.3.2</td>
</tr>
</tbody>
</table>


Establishing a Connection with a Data Modem 97.3.3 287
Dial String Modifiers 97.3.4 287
Waiting for a Connection with a Data Modem 97.3.5 289
Sending and Receiving Data 97.3.6 289
MNP Data Communication 97.3.7 289
Direct Communication with the Data Modem 97.4 290
The Data Modem AT Command Set 97.4.1 290

Chapter 100 / The Address Book 317

Concepts
Participants 100.1 317
The Address Book Protocols 100.1.1 318
Organization of Data 100.1.2 318
Groups 100.1.3 320
The GO Address Book Application 100.1.4 322
Loading the GO Address Book 100.2 323
Using the GO Address Book 100.2.1 323
The Address Book Protocols 100.2.2 323

Chapter 98 / The Transport API 295

Transport Concepts 98.1 295
Participants in Communication 98.1.1 295
Transport Service Types 98.1.2 296
Agreeing on Conventions 98.1.3 297
Asynchronous Communication 98.1.4 297
Using clsTransport 98.2 298
Accessing a Socket 98.2.1 299
Closing a Socket Handle 98.2.2 299
Sending Datagrams 98.2.3 299
Receiving Datagrams 98.2.4 300
Requesting a Transaction Service 98.2.5 300
Responding to a Transaction Service 98.2.6 301
Binding to a Local Transport Address 98.2.7 301
Using clsTransport for AppleTalk 98.3 301
Using the AppleTalk Protocol 98.3.1 301
AppleTalk Name and Zone Protocols 98.3.2 302

Chapter 99 / In Box and Out Box 305

Introduction to the In Box and Out Box 99.1 305
General Device Concepts 99.2 306
Service Sections 99.2.1 306
Services and Devices 99.2.2 306
Installing Devices and Services 99.2.3 307
Targeting Communications Devices 99.2.4 307
Enabling and Disabling Services 99.2.5 307
Out Box Concepts 99.3 308
Out Box Operation 99.3.1 308
Out Box Protocol Messages 99.3.2 308
Documents in the Out Box 99.3.3 309
Writing Your Own Out Box Service 99.3.4 310
In Box Concepts 99.4 312
Passive and Active In Box Services 99.4.1 312
In Box Documents 99.4.2 313
In Box and Out Box Service Messages 99.5 313

Chapter 101 / The Sendable Services 331

The Sendable Services Protocol 101.1 331
Creating Address Descriptors 101.1.1 331
Displaying a User Interface 101.1.2 332
The Sendable Services Messages 101.2 333
Getting Address Descriptors 101.2.1 333
Creating and Filling Address Windows 101.2.2 333
Summarizing Address Information 101.2.3 334

List of Figures
93-1 Applications and Ports 245
93-2 Applications, Drivers, and Devices 246
93-3 Layered Services 248
93-4 Devices and Interfaces 249
100-1 The GO Address Book 324
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>94-1</td>
<td>clsServiceMgr Messages</td>
<td>260</td>
</tr>
<tr>
<td>95-1</td>
<td>clsMILAsyncSIODevice Messages</td>
<td>267</td>
</tr>
<tr>
<td>95-2</td>
<td>Event Mask Indicators</td>
<td>272</td>
</tr>
<tr>
<td>96-1</td>
<td>Parallel Port Messages</td>
<td>276</td>
</tr>
<tr>
<td>97-1</td>
<td>clsModem Messages</td>
<td>281</td>
</tr>
<tr>
<td>97-2</td>
<td>Modem Reset Settings</td>
<td>283</td>
</tr>
<tr>
<td>97-3</td>
<td>Modem Connection Types</td>
<td>285</td>
</tr>
<tr>
<td>97-4</td>
<td>Summary of AT Command Set</td>
<td>291</td>
</tr>
<tr>
<td>98-1</td>
<td>clsTransport Messages</td>
<td>297</td>
</tr>
<tr>
<td>98-2</td>
<td>NBP and ZIP Messages</td>
<td>301</td>
</tr>
<tr>
<td>99-1</td>
<td>clsOBXService Messages</td>
<td>314</td>
</tr>
<tr>
<td>99-2</td>
<td>clsINBXService Messages</td>
<td>315</td>
</tr>
<tr>
<td>99-3</td>
<td>clsIOBXService Messages</td>
<td>316</td>
</tr>
<tr>
<td>100-1</td>
<td>Attribute Identifiers</td>
<td>321</td>
</tr>
<tr>
<td>100-2</td>
<td>Address Book Gestures</td>
<td>323</td>
</tr>
<tr>
<td>100-3</td>
<td>clsAddressBookApplication Messages</td>
<td>324</td>
</tr>
<tr>
<td>100-4</td>
<td>clsABMgr Messages</td>
<td>325</td>
</tr>
<tr>
<td>101-1</td>
<td>Sendable Services Messages</td>
<td>333</td>
</tr>
</tbody>
</table>
Chapter 92 / Introduction

The PenPoint™ operating system allows applications to communicate with devices and networks through device ports in the PenPoint computer.

For most operating systems, this level of connectivity is sufficient. However, other operating systems expect connections to be always present, uninterrupted, and require the user to reboot the machine between installing a new device driver and using it.

The PenPoint approach to connectivity enables users to:

- Store data so that it will be sent only when the machine connected to the correct network (deferred connectivity).
- Disconnect the machine from a network and reconnect without losing the current data transfer (automatic connection detection).
- Install and deinstall new device drivers (or other non-application software) without rebooting the machine (service architecture).

The PenPoint operating system also defines a protocol for applications or services that contain addressing information. These applications or services are called address books. By defining a common address book protocol, all service providers can access a single resource for all addresses, phone numbers, and other such information.

Layout of This Part

Part 10 is organized into nine chapters.

Chapter 92, this chapter, presents a brief overview of the PenPoint remote interface.

Chapter 93, Concepts, provides you with the fundamental concepts needed to understand the PenPoint remote interface.

Each of the following chapters describes one remote interface.

Chapter 94, Using Services, describes the service manager and how to access services through the service manager. Part 13: Writing PenPoint Services describes how to implement a service.

Chapter 95, Serial I/O, describes the serial I/O interface.

Chapter 96, Parallel I/O, describes the parallel I/O interface.

Chapter 97, Data Modem Interface, describes the interface to the data modem.

Chapter 98, The Transport API, describes PenPoint support for local area network communication using the AppleTalk protocol.
Chapter 99, In Box and Out Box, describes support for delayed data transfer through the In box and Out box.

Chapter 100, The Address Book, describes the address book protocol used to query and modify the system address book.

Chapter 101, The Sendable Services, describes the protocol used by the address book for communicating with services that provide Send capabilities, such as fax and e-mail.

**Other Sources of Information**

For useful insights on the topics covered in this part, refer to these books:

- For further reading on data communication, see *Computer Networks*, 2nd. Edition by Tannenbaum (Prentice Hall, 1989).

If you are developing MIL devices (device drivers that interface to the PenPoint machine interface layer), you may need to refer to the documentation in the PenPoint HDK (hardware development kit). For more information on the HDK and MIL Devices, please contact GO Developer Technical Support.
Chapter 93 / Concepts and Terminology

The PenPoint™ operating system has a flexible architecture that can accommodate many forms of networking and connectivity.

This chapter presents the "big-picture" concepts that tie together all of the PenPoint communications and networking software. This chapter covers these topics:

- The principles of PenPoint connectivity.
- The hardware ports.
- Software access to those ports.
- The service manager.
- A summary of the actual implementation of PenPoint connectivity.
- How to integrate other network protocols with PenPoint.
- A survey of the remote interface features.

Principles of PenPoint Connectivity

There are four recurrent principles of PenPoint connectivity:

1. Many different communication facilities can coexist in a running PenPoint system.
2. All device drivers can be installed and deinstalled dynamically, unlike some other operating systems, where device drivers must be configured as part of the cold-boot process. To add a new device driver to these systems, you must shut the system down, add the device driver, and reboot.
3. PenPoint expects that there might be a chain of device drivers that handle communication between an application and a device (rather than just one driver per device).
4. All devices can be dynamically connected and disconnected. The operating system and the device drivers are prepared to handle disconnection and reconnection events while accessing a device.
PenPoint Connectivity Strategies

PenPoint was designed to allow many connectivity options. The PenPoint connectivity strategies include:

- A volume connectivity strategy that enables users to install and configure volumes from many different file systems on the fly. Volumes include internal disks, hard or floppy disk volumes, and volumes on remote file systems. PenPoint detects when a volume is connected or disconnected and manages volume connection.

- An adaptable file system designed to use multiple existing file systems that allows PenPoint to offer data sharing capabilities with many file system architectures.

- A device connectivity strategy enabling users to install and configure devices, device drivers, and network protocol stacks on the fly. As with volumes, PenPoint provides automatic connection detection and management.

- Basic hardware support, including a parallel port, serial port, high-speed packet, and a SCSI interface. With these hardware interfaces, PenPoint machines can connect (through adapters) to many networks and peripherals.

- Multiple network protocol stacks that can coexist concurrently. A protocol stack is a layered set of protocols, each of which handles a specific communication task, such as establishing and maintaining connections, transporting data, or presenting the data to the application.

- A general-purpose document import and export architecture. The operating system and its user interface makes it easy to exchange information between PenPoint applications and other file formats.

- An In box and an Out box that allows deferred document I/O for printing, faxing, e-mail, and so on. If the user makes an output request while the machine is disconnected, the Out box queues the document. The document is output automatically when the machine is reconnected. While the machine is connected, input accumulates in the In box; the user can review the input documents after the machine is disconnected.

PenPoint Computer Connectivity

It is up to the manufacturers of PenPoint computers to determine what connectivity their machine will offer. While some machines will not have any ports, most machines will include at least one, if not several of these connectors:

- Serial connectors
- Parallel connectors
- SCSI connectors
- AppleTalk connectors
- Modem connectors
- Other communications connectors
If a machine doesn't implement a port, the user doesn't have to install the service for that port. Applications can query service managers for available services. If the service is not installed on a machine, the application that uses the service can handle the situation by notifying the user or suggesting an alternate service.

Additionally, hardware manufacturers can easily create and distribute services that support non-standard ports, or ports that are not supported by the PenPoint operating system.

### PenPoint Connectivity

The essential point of any connectivity architecture is to allow applications and other programs to communicate with hardware devices. The problem is in translating the applications needed to write and read bytes of data into device instructions to perform those tasks.

![Figure 93-1: Applications and Ports](image)

#### MIL Services

MIL services (PenPoint device drivers) provide an interface between programs and devices. The MIL service can configure and initialize the device, buffer data (if necessary), and inform the program when a device error has occurred. MIL services are usually written by GO and other hardware vendors.

Although different devices might perform equivalent functions, they take different instructions to perform those functions. For example, data can be written to a disk volumes or to the serial port, but the instructions to control those devices are very different. Thus, each device requires a separate MIL service.
Most MIL services use the stream interface, you can use the same stream messages to send and receive data from the MIL service. However, most MIL services also provide control functions in their API; these control functions vary from device to device. Your application should always be aware of the type of device with which it is communicating.

### MIL Services and Other Services

MIL services control physical ports, such as the serial port or the SCSI port. There is a fixed set of MIL services created at boot time for all of the available hardware ports. Additional MIL services can be loaded after boot time for things like networks or plug-in peripherals.

Each MIL service has a programmatic interface; MIL services that perform related functions have similar interfaces. These interfaces can themselves be controlled by another service, which has a more general interface or provides another layer of functionality. Networks use services to implement the various network protocols.

In the PenPoint operating system a service is an instance of a particular service class. A service class provides the interface to a particular type of device (for
example, there is a service for Hewlett-Packard's LaserJet printers and there is a service class for Microsoft MS-DOS disk drives).

When the user attaches a device to a PenPoint computer and identifies it through the device option sheet, PenPoint creates an instance of a service instance that corresponds to that device. An instance of the LaserJet class might handle the user's LaserJet; an instance of the MS-DOS disk drive class handles the floppy disk volume named DISK1. The service instance contains configuration information (for example one paper tray or two), has a user-visible name, and can be bound (directly or through another service) to a MIL service which represents an actual hardware port.

**Binding**

A service is joined to a port or another service in a process called **binding**. Binding can be static or dynamic. Static binding is performed at compile time and is permanent (such as binding a MIL service to a port). Dynamic binding is performed when a service is installed, which allows the service to bind to a service chosen by the user.

More than one service can be bound to a given service. For example, the serial port could have two different services, such as a printer service and the modem service.

Some services, such as networks or the SCSI device, have no problem with this situation, because the peripherals connected to these devices are self-identifying. However, a device such as the serial port cannot be shared by peripherals; only one thing can be connected to it at any time. Devices of this nature can be accessed by only one service at a time.

To solve this problem, the user must identify the application or service that owns the service (the **owner**). A service keeps track of all the services that are bound to it, but only allows communication with its owner. Services that can be shared by peripherals can allow more than one owner.

**Connection Management**

MIL services can recognize when actual connections are made and broken. For intelligent peripherals, the services are able to detect connection by monitoring their hardware port. For dumb peripherals, those that aren't able to detect connections, the user must tell the service which device is connected. For example, the SCSI port can detect when SCSI devices come and go, but standard PC floppy drives can't. Therefore, the user must triple tap on the Connections notebook for it to recognize a new floppy.

When a service detects that a connection is made or broken, it sends object notification messages to the services that are bound to it. In turn, a service can relay the connection status message to any services that are bound to it. Any object can add itself to the notification list to receive these connection status messages. All MIL services support a common set of connection status messages.
Services also broadcast connection status. They observe the MIL service to which they are bound and essentially pass the connection messages on to their own observers.

Figure 93-3 illustrates the logical and MIL services.
Figure 93-4 shows the PenPoint networking and connectivity architecture. Each interface in the figure (denoted by dotted lines) is a PenPoint class. Applications can send messages to instances of these classes to control individual devices. Thus, an application might use the file system interface to communicate with a remote...
computer, but it can also use the SofTalk interface (available from Sitka Corporation) to communicate with the remote computer.

The interface that an application uses depends on the level of control that the application needs to exercise, offset by the amount of work the application wants to perform itself. Usually in bypassing one of the upper-level interfaces, an application takes on more work for itself.

The Service Manager

At the root of PenPoint connectivity architecture is the service manager class. A service is any general, non-application DLL that enables PenPoint clients to communicate with a device or to access a function, such as a database engine.

Related services are managed by an instance of the service manager class. A service manager instance performs these tasks:

- Locates and accesses the service.
- Manages the connections between a client and a service (services may themselves by clients of another service).
- Notifies its observers of additions and deletions.
- Monitors the state of each connection between a service and its target and passes change notifications to its observers.

PenPoint Facilities

Using the principles and architecture described above, PenPoint provides these facilities for networking and connectivity:

- A file system that works on top of MS-DOS disk organization.
- A Connections notebook, in which the user can view the available volumes and printer queues.
- Basic Appletalk services at the ATP (AppleTalk Transport Protocol) level and below. This does not include ASP (AppleTalk Session Protocol) and ADSP (AppleTalk Data Stream Protocol).
- 19.2K asynchronous serial support, which allows a PenPoint computer to communicate with a single IBM PC through the PC's standard serial port.
- Appletalk connections between a PenPoint computer with a modem and another computer with a modem.
- High-speed serial (115K) and parallel data connections.
- Printer services for 8- and 24-pin dot-matrix printers and the H-P LaserJet printer.
Adding Network Protocols

PenPoint can support multiple protocol stacks and multiple remote volumes.

To allow users to connect a PenPoint computer directly to a network, you have to obtain or write a network protocol stack, or supplement an existing one and write support for another remote file system.

The service manager allows users to add new network protocol stacks dynamically; they don’t have to cold-boot the system to add a new protocol. To connect a PenPoint computer to a new network device (provided the device uses SCSI or AppleTalk connectors), all the user has to do is:

• Install the protocol stack for the new network.
• Connect the PenPoint computer to the device.
• Tell the connection manager which protocol stack to use with the device.

Survey of the Remote Interfaces

The rest of this chapter provides an overview of the remote interface APIs provided by the PenPoint operating system. Most of these APIs are described in full in the remaining chapters.

Most of the device interfaces described here descend from clsStream. To read from and write to these devices, applications use the stream read and stream messages. clsStream is described in Part 9: Utility Classes.

Serial I/O

The serial I/O interface provides access to and control of a serial port. The serial interface is managed by theSerialDevices service manager. When an application gets a handle on a serial port from theSerialDevices, it can perform these tasks:

• Alter the serial port configuration, including baud rate, line control, and flow control.
• Transmit and receive data on the serial port.
• Detect events on the serial port.

For more information on the serial port, see Chapter 95, Serial I/O.

Parallel I/O

The parallel I/O interface provides access to and control of a parallel port. The parallel interface is managed by theParallelDevices service manager. When an application gets a handle on a parallel port from theParallelDevices, it can:

• Initialize a printer attached to the parallel port.
• Transmit and receive data on the parallel port.
• Change the auto-line-feed characteristics of a printer.
• Get and set the initialization and interrupt time-out intervals.
For more information on the parallel port, see Chapter 96, Parallel I/O.

**High-Speed Packet I/O**

The high-speed packet I/O interface (HSPKT) provides access to a protocol that performs high-speed data transfers using either a serial or parallel port. The high-speed packet interface is managed by the `HighSpeedPacketHandlers` service manager.

High-speed packet I/O implements a built-in RTS/CTS type protocol by sending a lead-in character and expecting a data acknowledge character in return before actually sending a packet of data.

High-speed packet I/O is also discussed in Chapter 96, Parallel I/O.

**File System**

The PenPoint file system provides access to files and directories on a RAM volume, local disk volumes, and remote volumes. The file system API provides all standard file system control functions. In addition, it provides these capabilities:

- Reliable handling of disconnect and reconnect events for volumes.
- Memory-mapped files that allows you to open files with direct byte addressing capability.
- A volume traversal mechanism that allows you to visit all file system nodes that match a particular criterion.

The PenPoint file system currently is layered on the MS-DOS volume structure. Implementations using other volume structures (such as Apple's HFS) are possible in the future.

For more on the PenPoint file system, see Part 7: File System.

**File Import Export**

The PenPoint operating system provides system-based support for importing and exporting documents in different file formats; each application developer determines which import and export formats to supply.

When the user moves or copies a file into a PenPoint computer, PenPoint examines the file to see if it is a PenPoint document. If not, PenPoint asks every application that is currently installed whether it can import the file. Those applications that can import the file are displayed in a dialog box. When the user chooses an application, the import mechanism tells the application to translate the file into a document of that application type.

To export a file, the export mechanism asks the application what file formats it can write and displays the list of formats in a dialog box. The user selects a file format and the export mechanism tells the application to write the file using that format.

An additional mechanism allows applications and file translators to communicate with each other, allowing third parties to provide additional file translators.
The text editor application, MiniText, that is bundled with PenPoint imports and exports using three file formats:

- PenPoint internal text format.
- ASCII text.

For more information on the import/export mechanism see Chapter 81, File Import and Export, in Part 9: Utility Classes.

**Modem Interface**

The modem interface provides access to modem control functions through a high-level API, which allows you to configure the modem, dial out, and auto-answer.

The modem interface handles all the low-level modem line functions for you.

To use the modem interface, you open the serial port, create a modem object using the serial port handle, and then send messages to the modem object.

For more information on the modem manager, see Chapter 97, Data Modem Interface.

**Networking Interface**

As described previously in this chapter, PenPoint includes support for networking protocols. PenPoint provides these protocols:

- A transport protocol, which establishes communication with another program that understands the same protocol and transports data between the two programs.
- Several link protocols, which send and receive the data using various physical media (parallel and serial lines).

The session layer protocols are available from other vendors, such as Sitka Corporation, who provides TOPS for PenPoint.

**Transport**

The transport protocol establishes communication with a partner and handles data exchange with the partner. Currently the only transport protocol supported by the transport API is the AppleTalk transport protocol (ATP).

For more information on the Transport Protocol, see Chapter 98, Transport API.

**Link**

The link protocol provides communication on physical network devices (LocalTalk). The interface to the link protocol is not described in this part. For further information, please contact GO Developer Technical Support.
Chapter 94 / Using Services

PenPoint™ service classes provide installable, configurable system extensions. To access a particular service instance, clients need to communicate with a service manager. The service manager controls access to service instances and keeps track of all clients that are interested in a particular service instance.

There are a number of service managers in PenPoint. Each service manager is responsible for a particular category of services, such as the modems, the printing devices, or the handwriting translators.

Some of these categories may actually overlap (such as the printing devices and the parallel ports). A particular service instance can be managed by several service managers.

This chapter describes some fundamental service concepts, but is primarily concerned with the API for the service managers.

Chapter 94 covers these topics:

- Service concepts, including user installation and configuration of services, service managers, binding and opening services, and chaining services.
- Messages for service managers.
- How to use the service manager messages.

This chapter does not describe how to write services; for that information, see Part 13: Writing PenPoint Services.

Concepts

A service is a general, non-application DLL that provides an extension to the system. Services can perform many types of work, including—but not limited to—database engines, e-mail backends, and device drivers.

Each service is a class that inherits from clsService. For some services the service writer provides a user interface so that the user can create instances of a service; for other services, their instances are created automatically when the service is installed.

The services architecture can be thought of as being quite similar to the PenPoint application framework. Services are similar to application classes; service instances are similar to documents.

However, the key concept to services is not with the services themselves, but the service manager architecture. Most clients of services deal far more with the service managers and service manager messages than with the actual services.
Installing Services

There are three ways to install services:

- The user can explicitly install a service through the services page of the Connections notebook. This action may be initiated by the service developer enabling service quick installation on a distribution disk.

- Applications or services can list a set of services in a SERVICE.INI file. If the service is not already installed in the system, PenPoint will install the service (if it is already installed, PenPoint will increase its reference count).

- A program can explicitly install a service by sending msgIMInstall to theInstalledServices (most applications will not do this).

You cannot list a service DLL file in any DLC file.

When an application requests a service in a SERVICE.INI file, the application is said to be dependent on the service. When someone attempts to deinstall the service, any application that is dependent on the service can veto the deinstallation.

The user can deinstall services by the Connections notebook interface or theInstalledServices. When the user deinstalls a service, the service manager destroys the service handles and removes the service’s code and all saved service instances from PenPoint.

Clients that are bound to a service receive msgIMDeinstalled when the service class is deinstalled. Clients can observe the service manager to find out when the service is reinstalled.

A service cannot be deinstalled if it is marked as being in use when any instance of the service is open. By default, services are marked in use when anyone has their instances open.

Service Managers

A service manager provides a system-wide way of managing categories of services.

A service manager is an instance of clsServiceMgr. There is no single, central service manager; rather, there are a number of service managers. For example, the print services are managed by the print service manager; the serial ports are managed by the serial port manager.

A service manager performs several functions:

- It maintains a list of service instances.
- It provides protocol that allow clients to access service instances.
- It controls access to service instances.
- It notifies all observers when service instances are added or removed from its list.

However, when the client that initially loaded the service is deinstalled, the dependent applications receive notification, but cannot veto the deinstallation.
A service can appear on more than one service manager list. For example, theSerialDevices lists the available serial ports, but thePrinterDevices lists all the devices that can support printers, including the serial ports. This frees clients from having to search many lists to find the correct service; they just look for a service listed on by the most likely service manager.

All services controlled by a particular service manager support the same minimal set of functions.

**Access Overview**

The service manager provides access protocols that allow clients to:

- Find a particular service.
- Express an interest in a particular service instance (called binding).
- Open and close service instances.
- Acquire and release ownership of service instances.

Rather than perform these actions explicitly, the service manager also provides access messages that perform all of these actions.

For example, when a client needs to access a specific serial port driver, the client performs the following tasks:

1. Sends a message to a service manager that manages the driver (such as theSerialDevices), giving it the name of the serial port and asking it to find the service for that device. theSerialDevices replies with a handle on the service.
2. The client sends a message to theSerialDevices asking to bind with that service.
3. When the client needs to send or receive serial data, it sends an open message to theSerialDevices.
4. When the client is done sending or receiving, it sends a close message to theSerialDevices and eventually unbinds from the service.

A client should open a service only when it is ready to actually use the service. Opening a service returns the service object. The client should close the service as soon as it is done with the service.

**Ownership**

The service manager also works to control access to service instances.

Some services (such as those that communicate with ports) can only be used by one client at a time. For example, if the fax service owns a serial port and a fax/data modem is connected to that port, you don't want an E-mail service to be able to access the port.

A service manager can maintain an owner of a service instance and, if suitably configured, will allow only the owner of a service instance to open that service.
Targeting

A service can have a target, which can be either a physical device or another service. By using targeting, several services can be chained together, much like a network protocol stacks.

Targeting is used to pass both data and control information up and down target chains. (Here data is the actual information that a client is trying to transmit through a device; control information is information about the other services in the chain—such as connection information.)

Connections

Services can maintain connection status. Usually connection status is important to services that control devices, but connection status is not limited to those services.

Changes in connection state are transmitted to observers of the service (in addition to observers of the service manager in which the service is listed). By default, a service’s connection state follows that of its target.

With targeting and connection information, PenPoint applications can easily inform their users that a required device is available (or unavailable).

Accessing Services

To access a service, a client must observe the service protocol of binding to the service when interested in it, opening the service when access is needed, closing the service when done, and releasing the service when no longer interested in it.

Clients use messages defined by both clsInstallMgr and clsServiceMgr to communicate with the service manager. For example, your application gets the name of a service given its handle by sending msgIMGetName to the service manager.

clsInstallMgr is described in Chapter 112, Installation Managers of Part 12: Installation API.

Predefined Service Managers

GO defines a number of service managers in UID.H.

theMILDevices
theParallelDevices
theAppleTalkDevices
theSerialDevices
thePrinterDevices
thePrinters
theSendableServices
theTransportHandlers
theLinkHandlers
theHWXEngines
theModems
theHighSpeedPacketHandlers
theFaxIOServices

The names of these service managers make them fairly self-explanatory.

Binding to a Service

A client binds to a service by sending `msgSMBind` to the service's manager. The service manager then adds the client to the service's notification list. That way the service can inform the client about its availability through notification messages.

These notification messages include messages defined by the service, plus the following service manager messages:

- `msgIMActiveChanged`
- `msgIMDeinstalled`
- `msgIMModifiedChanged`
- `msgIMInUseChanged`
- `msgIMCurrentChanged`
- `msgSMConnectChanged`

Opening a Service

Most services have service-specific arguments that the client must include in the arguments to `msgSMOpen`. The structure and organization of these arguments is described in the documentation for the specific service. Some services provide their own defaults for their open arguments. To initialize the service specific arguments to their default values and to allow the clsService to perform other work for the service, the client must always call `msgSMOpenDefaults` before calling `msgSMOpen`.

A client opens a service by sending `msgSMOpen` to the service's manager. If the service has no other openers, or if the service can be shared, the service manager passes back the UID of the service object. If the service manager refuses the open request, it returns `stsFailed`.

Once the client has the UID of the service object, it can send service-specific messages to the service.

The client should open a service just before it needs to use it, and should close the service as soon as its use is completed. Services are a resource; many of them only allow one client at a time. An open service cannot be deinstalled.
## The Service Manager Messages

Most `clsServiceMgr` messages are sent by clients attempting to access a service; they are sent to instances of the service manager.

Table 94-1 lists the `clsServiceMgr` messages.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>msgIMGetState</code></td>
<td><code>P_IM_GET_STATE</code></td>
<td>Gets the state of a service manager.</td>
</tr>
<tr>
<td><code>msgIMSetModified</code></td>
<td><code>P_IM_SET_MODIFIED</code></td>
<td>Changes an item’s modified setting.</td>
</tr>
<tr>
<td><code>msgSMAccess</code></td>
<td><code>P_SM_ACCESS</code></td>
<td>Accesses a service instance, given its name.</td>
</tr>
<tr>
<td><code>msgSMRelease</code></td>
<td><code>P_SM_RELEASE</code></td>
<td>Releases a service instance.</td>
</tr>
<tr>
<td><code>msgSMBind</code></td>
<td><code>P_SM_BIND</code></td>
<td>Binds to a service.</td>
</tr>
<tr>
<td><code>msgSMUnbind</code></td>
<td><code>P_SM_BIND</code></td>
<td>Unbinds from a service.</td>
</tr>
<tr>
<td><code>msgSMAccessDefaults</code></td>
<td><code>P_SM_ACCESS</code></td>
<td>Sets pArgs defaults for <code>msgSMAccess</code>.</td>
</tr>
<tr>
<td><code>msgSMOpenDefaults</code></td>
<td><code>P_SM_OPEN_CLOSE</code></td>
<td>Initializes <code>SMOpen</code> pArgs to default value.</td>
</tr>
<tr>
<td><code>msgSMOpen</code></td>
<td><code>P_SM_OPEN_CLOSE</code></td>
<td>Opens a service, given its handle.</td>
</tr>
<tr>
<td><code>msgSMClose</code></td>
<td><code>P_SM_OPEN_CLOSE</code></td>
<td>Closes an open service.</td>
</tr>
<tr>
<td><code>msgSMGetState</code></td>
<td><code>P_SM_GET_STATE</code></td>
<td>Gets the state of a service.</td>
</tr>
<tr>
<td><code>msgSMFindHandle</code></td>
<td><code>P_SM_FIND_HANDLE</code></td>
<td>Finds a handle, given a service instance UID.</td>
</tr>
<tr>
<td><code>msgSMGetOwner</code></td>
<td><code>P_SM_GET_OWNER</code></td>
<td>Gets the current owner of a service.</td>
</tr>
<tr>
<td><code>msgSMSetOwner</code></td>
<td><code>P_SM_SET_OWNER</code></td>
<td>Sets a new service owner.</td>
</tr>
<tr>
<td><code>msgSMSetOwnerNoVeto</code></td>
<td><code>P_SM_SET_OWNER</code></td>
<td>Sets a new service owner without giving owners veto power.</td>
</tr>
<tr>
<td><code>msgSMQueryLock</code></td>
<td><code>P_SM_QUERY_LOCK</code></td>
<td>Gets the UID of a service and locks out any opens.</td>
</tr>
<tr>
<td><code>msgSMQuery</code></td>
<td><code>P_SM_QUERY_LOCK</code></td>
<td>Gets the UID of a service.</td>
</tr>
<tr>
<td><code>msgSMQueryUnlock</code></td>
<td><code>P_SM_QUERY_UNLOCK</code></td>
<td>Unlocks a service that was locked via <code>msgSMQueryLock</code>.</td>
</tr>
<tr>
<td><code>msgSMSave</code></td>
<td><code>P_SM_SAVE</code></td>
<td>Saves a service instance to a specified external location.</td>
</tr>
<tr>
<td><code>msgSMGetClassMetrics</code></td>
<td><code>P_SM_GET_CLASS_METRICS</code></td>
<td>Gets the service’s class metrics.</td>
</tr>
<tr>
<td><code>msgSMRemoveReference</code></td>
<td><code>P_SM_REMOVE_REF</code></td>
<td>Removes a service from a service manager without destroying the service.</td>
</tr>
<tr>
<td><code>msgSMConnectedChanged</code></td>
<td><code>P_SM_CONNECTED_NOTIFY</code></td>
<td>A service’s connection state changed.</td>
</tr>
<tr>
<td><code>msgSMOwnerChanged</code></td>
<td><code>P_SM_OWNER_NOTIFY</code></td>
<td>A service’s owner has changed.</td>
</tr>
</tbody>
</table>

### Table 94-1

`clsServiceMgr` Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>msgIMGetState</code></td>
<td><code>P_IM_GET_STATE</code></td>
<td>Gets the state of a service manager.</td>
</tr>
<tr>
<td><code>msgIMSetModified</code></td>
<td><code>P_IM_SET_MODIFIED</code></td>
<td>Changes an item’s modified setting.</td>
</tr>
<tr>
<td><code>msgSMAccess</code></td>
<td><code>P_SM_ACCESS</code></td>
<td>Accesses a service instance, given its name.</td>
</tr>
<tr>
<td><code>msgSMRelease</code></td>
<td><code>P_SM_RELEASE</code></td>
<td>Releases a service instance.</td>
</tr>
<tr>
<td><code>msgSMBind</code></td>
<td><code>P_SM_BIND</code></td>
<td>Binds to a service.</td>
</tr>
<tr>
<td><code>msgSMUnbind</code></td>
<td><code>P_SM_BIND</code></td>
<td>Unbinds from a service.</td>
</tr>
<tr>
<td><code>msgSMAccessDefaults</code></td>
<td><code>P_SM_ACCESS</code></td>
<td>Sets pArgs defaults for <code>msgSMAccess</code>.</td>
</tr>
<tr>
<td><code>msgSMOpenDefaults</code></td>
<td><code>P_SM_OPEN_CLOSE</code></td>
<td>Initializes <code>SMOpen</code> pArgs to default value.</td>
</tr>
<tr>
<td><code>msgSMOpen</code></td>
<td><code>P_SM_OPEN_CLOSE</code></td>
<td>Opens a service, given its handle.</td>
</tr>
<tr>
<td><code>msgSMClose</code></td>
<td><code>P_SM_OPEN_CLOSE</code></td>
<td>Closes an open service.</td>
</tr>
<tr>
<td><code>msgSMGetState</code></td>
<td><code>P_SM_GET_STATE</code></td>
<td>Gets the state of a service.</td>
</tr>
<tr>
<td><code>msgSMFindHandle</code></td>
<td><code>P_SM_FIND_HANDLE</code></td>
<td>Finds a handle, given a service instance UID.</td>
</tr>
<tr>
<td><code>msgSMGetOwner</code></td>
<td><code>P_SM_GET_OWNER</code></td>
<td>Gets the current owner of a service.</td>
</tr>
<tr>
<td><code>msgSMSetOwner</code></td>
<td><code>P_SM_SET_OWNER</code></td>
<td>Sets a new service owner.</td>
</tr>
<tr>
<td><code>msgSMSetOwnerNoVeto</code></td>
<td><code>P_SM_SET_OWNER</code></td>
<td>Sets a new service owner without giving owners veto power.</td>
</tr>
<tr>
<td><code>msgSMQueryLock</code></td>
<td><code>P_SM_QUERY_LOCK</code></td>
<td>Gets the UID of a service and locks out any opens.</td>
</tr>
<tr>
<td><code>msgSMQuery</code></td>
<td><code>P_SM_QUERY_LOCK</code></td>
<td>Gets the UID of a service.</td>
</tr>
<tr>
<td><code>msgSMQueryUnlock</code></td>
<td><code>P_SM_QUERY_UNLOCK</code></td>
<td>Unlocks a service that was locked via <code>msgSMQueryLock</code>.</td>
</tr>
<tr>
<td><code>msgSMSave</code></td>
<td><code>P_SM_SAVE</code></td>
<td>Saves a service instance to a specified external location.</td>
</tr>
<tr>
<td><code>msgSMGetClassMetrics</code></td>
<td><code>P_SM_GET_CLASS_METRICS</code></td>
<td>Gets the service’s class metrics.</td>
</tr>
<tr>
<td><code>msgSMRemoveReference</code></td>
<td><code>P_SM_REMOVE_REF</code></td>
<td>Removes a service from a service manager without destroying the service.</td>
</tr>
</tbody>
</table>

### Notification Messages

- `msgSMConnectedChanged` (P_SM_CONNECTED_NOTIFY): A service’s connection state changed.
- `msgSMOwnerChanged` (P_SM_OWNER_NOTIFY): A service’s owner has changed.
Using the Service Manager

Whenever a client needs to access a service, it sends install manager and service manager messages to a service manager. The service managers are always present in a running system.

Accessing a Service

Your application can access the service in one of two ways:

- It can use the msgSMAccess to find, bind to, set owner, and then open a service.
- It can explicitly find, bind to, set owner, and open the service.

This section discusses msgSMAccess. The following sections discuss msgIMFind, msgSMBind, and msgSMOpen.

msgSMAccess provides a convenient way for clients to perform the most common sequence of messages used to access a service. msgSMAccess takes a pointer to an SM_ACCESS structure that contains:

- pServiceName: The name of the service that your application needs to access.
- caller: The UID of the object making the call (usually self).
- pArgs: A pointer to a set of arguments, if required by the service.

If the service requires arguments, your application must send msgSMAccessDefaults first.

If the message does not succeed, it can return these status values:

- stsNoMatch: The specified service instance wasn’t found.
- stsSvcLocked: An exclusive-access service is locked by another client.
- stsSvcNotOwner: Your application is not the current owner of the service.
- stsSvcAlreadyOpen: An exclusive-open service is open by another client.

If the message does succeed, it passes back:

- handle: The handle on the service object.
- service: The UID of the service instance.

Applications should never store an device’s object UID in their instance data. While a document is saved and the application is terminated, the user could deinstall the service, or reconfigure the service, rendering the UID invalid. Your application should find and bind to a service when it receives msgApplInit or msgAppRestore; it should unbind from the service when it receives msgFree.

Finding a Service

Before your application can bind to a service, it must get the name of the service and a handle on the service.

To get the name of a service, your application can send msgIMGetList to a service manager, which passes back a list of service instance UIDs. Your application can
then use msgIMGetName to find the name of each service; the application then
displays the names to the user and allows the user to choose one.

If your application knows the name of the service ahead of time, it can send
msgIMFind to the service manager for that type of service, specifying the name.
msgIMFind passes back the handle on the service.

This example gets a handle on parallel I/O port:

```
STATUS s;
IM_FIND imf;
SM_BIND sm;
SM_OPEN so;
OBJECT siOid;

imf.pName = PPortName; // PPortName is defined by the user.

ObjCallRet(msgIMFind, theParallelDevices, &imf, s);
```

### Binding to a Service

When your application binds to a service, the service manager adds your
application to the observer list for that service. Observers of a service receive
messages that relate to the status and availability of the service.

When your application has the handle on the service, it can send msgSMBind to
the handle. The message takes a pointer to an SM_BIND structure that contains:

- **handle** The handle on the service.
- **caller** The UID of the object sending the message. Usually caller is self.

When the message completes successfully, it returns *stsOK*. The message does not
send back anything. If the handle is not found, the message returns *stsNoMatch*.

This example continues from the example above. The caller binds to the option
slot serial I/O port:

```
sm.handle = imf.handle;
sm.caller = self;

ObjCallRet(msgSMBind, theParallelDevices, &sm, s);
```

### Opening and Closing a Service

When your application is ready to use the service, your application can open it by
sending msgSMOpenDefaults and then msgSMOpen to the service manager for
the service. *msgSMOpenDefaults* allows the service class to provide defaults for
its arguments, and also allows clsService to initialize other internal data structures,
such as open service objects.

Clients should only open the service when they are ready to use it and should
leave the service open for as little time as possible.

Both messages take a pointer to an SM_OPEN_CLOSE structure that contains:

- **handle** The handle on the service. This is the same handle that your
  application sent with msgIMFind.
- **caller** The UID of the object sending the message. Usually caller is self.
Using the Service Manager

pArgs  A pointer to service-specific arguments. Your application must check the description of the service to see if it has any service-specific arguments. Some services can initialize their pArgs structure to default values; for these services, call msgSMOpenDefaults before calling msgSMOpen.

If the message completes successfully, it returns stsOK and sends back the UID of the service in the service field of the SM_OPEN_CLOSE structure.

If the service manager refuses the request, it returns stsRequestDenied. A common reason for refusal is that only one client is allowed to open the service at a time and the service is busy. See the documentation for individual services for the exact reasons for returning stsRequestDenied.

This example continues from the previous two examples to show the client opening the service:

```c
so.handle = imf.handle;
so.caller = self;
ObjCallRet(msgSMOpen, theSerialParallel, &SO, s);
pportUid = so.service;
```

When your application has finished with a service, send msgSMClose to the service manager for that service. The message takes a pointer to an SM_OPEN_CLOSE structure that contains:

- handle  The handle of the service to close.
- caller  The UID of the object sending the message. Usually caller is self.
- pArgs  A pointer to service-specific arguments.
- service  The UID of the service to close.

Your application must specify both the handle on the service and the UID of the service.

Unbinding from a Service

To remove yourself from a service’s observer list, send msgSMUnbind to the service manager for that service. If your application has the service open when it send msgSMUnbind, the service manager also sends msgSMClose to close the service. The message takes a pointer to an SM_BIND structure, which contains:

- handle  The handle on the service.
- caller  The UID of the object sending the message. Usually caller is self.

When the message completes successfully, it returns stsOK.

Finding a Handle

If your application has the UID of a service object, but don’t have the handle, it can send msgSMFindHandle to the service manager for the service. The message takes a pointer to an SM_FIND_HANDLE structure that contains the service UID (service).
When the message completes successfully, it sends back the handle on the service in the handle field of the SM_FIND_HANDLE structure.

**Receiving Connection State Notification**

Most services support the notion of being connected. When the connection state of a service changes, clients that are bound to that service receive msgSMConnectedChanged, indicating that the connection state has changed.

The message pArgs point to an SM_CONNECTED_NOTIFY structure that contains:

- **manager** The UID of the manager that sent the notification.
- **handle** The handle of the service whose state changed.
- **connected** A BOOLEAN value that indicates the new connection state.

When connected is TRUE, the service is connected.

**Setting a Service Owner**

To set a new owner of a service, send msgSMSetOwner to a service manager.

The message takes a pointer to an SM_SET_OWNER structure that contains:

- **handle** The handle of the service that your application is changing.
- **owner** The UID of the new owner. To make yourself the new owner of a service, owner should be self.

When your application sends this message to a service manager, the service manager sends msgSvcOwnerReleaseRequested to the current owner and msgSvcOwnerAcquireRequested to the new owner. The current owner and new owner have the option to veto the change owner.

If the current and new owner do not veto, the service manager sends msgSvcOwnerChangeRequested to the owned service, so that the service has the opportunity to veto the change owner.

If the service doesn’t veto the change owner, the service manager sends msgSvcOwnerReleased to the current owner, msgSvcOwnerAcquired to the new owner, and msgSMOwnerChanged to all clients that are bound to the service.

If the current or new owner, or the owned service vetoes the change, the message returns any status value less than stsOK.

If your application sends msgSMSetOwnerNoVeto instead of msgSMSetOwner, the service manager does not send msgSvcOwnerChangeRequested to the current owner.
Chapter 95 / Serial I/O

This chapter presents clsMILAsyncSIODevice, the serial port device driver interface class. The class provides an object-oriented interface to the serial port and supports the functionality necessary for a wide variety of serial applications.

To access a serial port, you send open and bind messages to the serial device service manager (theSerialDevices), requesting the port by name. If that port is available, the service manager gives you a handle on the port.

clsMILAsyncSIODevice inherits from clsStream. Structures and defines used by clsMILAsyncSIODevice are in \GO\INC\SIO.H.

Serial I/O Concepts

The following sections discuss the concepts of serial I/O under the PenPoint™ operating system.

Interrupt Driven I/O

Serial communications can burden a system with a high volume of asynchronous real-time events. The serial hardware generates interrupts in response to these events. The serial driver takes full advantage of this interrupt capability to buffer the data before passing it to its client.

Buffered Data

When data is received, the hardware generates a receive character interrupt. The serial driver interrupt handler takes the received character and places it in the input buffer. Thus, the client code avoids the responsibility of responding in real time to each input character.

Output is similar. The client code does not wait around sending characters one by one. Instead, the client places outgoing data in an output buffer and passes the buffer to the serial driver. The serial driver moves the characters to the serial port as the port becomes ready for them. When a character is loaded, the hardware generates a send character interrupt.

The serial port client selects the size of the input and output buffers.

Flow Control

At high data rates, or when the system is busy with other tasks, it’s possible for receive data to arrive in the input buffer faster than the client task can remove it. To prevent buffer overflow the serial driver utilizes two flow control protocols: XON/XOFF flow control and hardware RTS/CTS flow control. Of course, both sides of the serial connection must be using the same protocol for proper operation.
XON/XOFF flow control is limited to working with ASCII data, but has the advantage of being independent of the serial hardware and cable. Two characters, XON (usually Ctrl-Q) and XOFF (usually Ctrl-S), have special meaning. When the input buffer is nearly full and in danger of overflowing the driver sends an XOFF character to the remote serial port. When the remote serial port receives the XOFF character it immediately suspends transmission. When there is enough room in the input buffer the driver sends an XON character to the remote serial port which can start transmitting again.

Hardware RTS/CTS flow control works with any data but requires a particular hardware setup. It uses the serial port hardware input clear-to-send (CTS) and output request-to-send (RTS) control lines; these lines must be connected to the remote serial port. The CTS input enables data transmission to the remote serial port. The RTS output disables transmission from the remote serial port.

**Events**

The serial driver includes an event mechanism that can notify you when an interesting serial event happens. The event mechanism allows you to respond quickly to serial events without polling loops, which drain batteries and waste machine cycles. You can select which event messages you want to receive by sending `msgSioEventSet` to the serial driver. When one of the events occurs, the serial driver uses `ObjectSend` to send `msgSioEventHappened` and an event indication to you.

The possible interesting serial events are:

- CTS input line has changed state.
- DSR input line has changed state.
- DCD input line has changed state.
- RI input line has changed state.
- Input buffer is no longer empty.
- Break character has been received.
- Output buffer has become empty.
- Receive error condition has occurred (parity, framing error, or overrun error).

**Concurrency Issues**

Only one client can access the serial port at a time. If you request a handle for a port that is owned by another client, `clsMILasyncSIODevice` returns an error status code `stsSioPortInUse`.

You can define a global handle for a serial port, however, the serial port has no built-in access control mechanism. If you share a port with other clients, you must create your own access control mechanism.
## Using Serial Messages

Table 95-1 lists the messages defined by clsMILAsyncSIODevice.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgStreamRead</td>
<td>P_STREAM_READ_WRITE</td>
<td>Reads data from stream.</td>
</tr>
<tr>
<td>msgStreamWrite</td>
<td>P_STREAM_READ_WRITE</td>
<td>Writes data to stream.</td>
</tr>
<tr>
<td>msgStreamReadTimeOut</td>
<td>P_STREAM_READ_WRITE_TIMEOUT</td>
<td>Reads data from stream with timeout.</td>
</tr>
<tr>
<td>msgStreamWriteTimeOut</td>
<td>P_STREAM_READ_WRITE_TIMEOUT</td>
<td>Writes to the stream with timeout.</td>
</tr>
<tr>
<td>msgSioInit</td>
<td>P_SIO_INIT</td>
<td>Initializes the serial device to its default state.</td>
</tr>
<tr>
<td>msgSioBaudSet</td>
<td>U32</td>
<td>Sets the serial port baud rate.</td>
</tr>
<tr>
<td>msgSioLineControlSet</td>
<td>P_SIO_LINE_CONTROL_SET</td>
<td>Sets serial port data bits per character, stop bits, and parity.</td>
</tr>
<tr>
<td>msgSioControlOutSet</td>
<td>P_SIO_CONTROL_OUT_SET</td>
<td>Controls serial port output lines dtr and rts.</td>
</tr>
<tr>
<td>msgSioControlInStatus</td>
<td>P_SIO_CONTROL_IN_STATUS</td>
<td>Reads the current state of the serial port in put control lines.</td>
</tr>
<tr>
<td>msgSioFlowControlCharSet</td>
<td>P_SIO_FLOW_CONTROL_CHAR_SET</td>
<td>Defines serial port XON/XOFF flow control characters.</td>
</tr>
<tr>
<td>msgSioBreakSend</td>
<td>P_SIO_BREAK_SEND</td>
<td>Sends a break for the specified duration.</td>
</tr>
<tr>
<td>msgSioBreakStatus</td>
<td>P_SIO_BREAK_STATUS</td>
<td>Sends back the number of breaks received so far.</td>
</tr>
<tr>
<td>msgSioInputBufferStatus</td>
<td>P_SIO_INPUT_BUFFER_STATUS</td>
<td>Provides input buffer status.</td>
</tr>
<tr>
<td>msgSioOutputBufferStatus</td>
<td>P_SIO_OUTPUT_BUFFER_STATUS</td>
<td>Provides output buffer status.</td>
</tr>
<tr>
<td>msgSioFlowControlSet</td>
<td>P_SIO_FLOW_CONTROL_SET</td>
<td>Selects flow control type.</td>
</tr>
<tr>
<td>msgSioEventStatus</td>
<td>P_SIO_EVENT_STATUS</td>
<td>Sends back current state of event word, and then clears the event word.</td>
</tr>
<tr>
<td>msgSioEventSet</td>
<td>P_SIO_EVENT_SET</td>
<td>Enables event notification.</td>
</tr>
<tr>
<td>msgSioEventGet</td>
<td>P_SIO_EVENT_SET</td>
<td>Gets the current sio event setting.</td>
</tr>
<tr>
<td>msgSioEventHappened</td>
<td>P_SIO_EVENT_HAPPENED</td>
<td>Notifies client of event occurance.</td>
</tr>
<tr>
<td>msgSioGetMetrics</td>
<td>P_SIO_METRICS</td>
<td>Sends back the sio metrics.</td>
</tr>
<tr>
<td>msgSioSetMetrics</td>
<td>P_SIO_METRICS</td>
<td>Sets the sio metrics.</td>
</tr>
<tr>
<td>msgSioReceiveErrorsStatus</td>
<td>P_SIO_RECEIVE_ERRORS_STATUS</td>
<td>Sends back the number of receive errors and the number of dropped bytes (due to buffer overflows).</td>
</tr>
<tr>
<td>msgSioInputBufferFlush</td>
<td>pNull</td>
<td>Flushes the contents of the input buffer.</td>
</tr>
<tr>
<td>msgSioOutputBufferFlush</td>
<td>pNull</td>
<td>Flushes the contents of the output buffer.</td>
</tr>
<tr>
<td>msgSioSetReplaceCharProc</td>
<td>P_SIO_REPLACE_CHAR</td>
<td>Replaces the built in receive character interrupt routine.</td>
</tr>
</tbody>
</table>
Requesting and Releasing a Serial Handle

Before you can communicate through the serial port, you must request a handle on the port from theSerialDevices service manager. To locate the port by name, send msgIMFind to theSerialDevices, then bind the port to your process by sending msgSMBind to theSerialDevices. Finally, you open the port by sending msgSMOpen to theSerialDevices.

This example shows how a client requests a handle on a serial port.

```
STATUS s;
IM_FIND imf;
SM_BIND sm;
SM_OPEN so;
OBJECT sioUid;
imf.pName = userName;  // The name was requested from the user earlier
ObjCallRet(msgIMFind, theSerialDevices, &imf, s);
sm.handle = imf.handle;
sm.caller = self;
ObjCallRet(msgSMBind, theSerialDevices, &sm, s);
so.handle = imf.handle;
so.caller = self;
ObjCallRet(msgSMOpen, theSerialDevices, &so, s);
sioUid = so.service;
```

The symbol sioUid now contains the UID of the serial port object. You can communicate with the serial port by sending messages to this object.

When you have finished with the serial port, you should free the port by sending msgSMClose to theSerialDevices. Do not send msgDestroy to the serial port object.

This example shows the correct way to free a port:

```
...  
sioUid = so.service;
...
ObjCallRet(msgSMClose, theSerialDevices, wknKey);
```

Reinitializing the Serial Port

To reinitialize a serial port to its default state and change the buffer sizes, send msgSioInit to the handle on the serial port. The message takes a pointer to an SIO_INIT structure that contains:

- `inputSize` A U16 value that specifies the size of the input buffer.
- `outputSize` A U16 value that specifies the size of the output buffer.

Serial Port Configuration

Before you communicate over the serial port, you must configure the port so that it can communicate with the device at the other end of the line. Both devices must be configured identically, or communication will not occur. The configurable items are:

- Baud rate
- Number of bits per byte
Parity
Stop bits
Flow control
Flow control characters.

When you first create a handle on a serial port, it has these defaults:
- 8 bits
- No parity
- One stop bit
- XON/XOFF flow control
- Ctrl-Q and Ctrl-S are the XON and XOFF characters
- DTR and RTS are on.

**Setting the Baud Rate**

You set the baud rate by sending `msgSioBaudSet` to the serial port handle. The message takes a single argument, a U32 value that specifies the data rate in bits per second. The maximum allowed setting is 115200; there is no default setting.

**Setting the Line Control**

You set the number of data bits, the stop bits, and the parity by sending `msgSioLineControlSet` to the serial port handle. `msgSioLineControlSet` takes a pointer to an `SIO_LINE_CONTROL_SET` structure that specifies:

- **dataBits** The number of bits in a byte. Three constants define the possible byte sizes: `sioSixBits`, `sioSevenBits`, and `sioEightBits`.
- **stopBits** The stop bits. Three constants define the possible stop bits: `sioOneStopBit`, `sioOneAndAHalfStopBits`, and `sioTwoStopBits`.
- **parity** The parity. Three constants define the possible parity settings: `sioNoParity`, `sioOddParity`, and `sioEvenParity`.

**Specifying the Flow Control**

You specify the flow control for a serial port by sending `msgSioFlowControlSet` to a serial port handle. The message takes a pointer to an `SIO_FLOW_CONTROL_SET` structure. The structure contains a single member, `flowControl`, which indicates whether the port will use XON/XOFF, CTS/RTS, or no flow control. The constants defined by `SIO_FLOW_TYPE` are:

- **sioNoFlowControl** No flow control.
- **sioXonXoffFlowControl** Use XON/XOFF flow control. Use `msgSioFlowControlCharSet` to change the control characters from their defaults).
- **sioHardwareFlowControl** Use the CTS and RTS lines for flow control.
Changing the Flow Control Characters

If, for some reason, you cannot use Ctrl-Q (ASCII 17) as the XON character or Ctrl-S (ASCII 19) as the XOFF character, you can change the default flow control character values by sending `msgSioFlowControlCharSet` to the serial port handle. The message takes a pointer to an `SIO_FLOW_CONTROL_CHAR_SET` structure that specifies:

- `xonChar` A U8 value for the XON character.
- `xoffChar` A U8 value for the XOFF character.

Controlling DTR and RTS For Output

If both the local and remote hardware support data-terminal-ready (DTR) and request-to-send (RTS) lines, you can set the state of the output DTR and RTS lines by sending `msgSioControlOutSet` to the serial port handle. The message takes a pointer to an `SIO_CONTROL_OUT_SET` structure, which contains:

- `dtr` A BOOLEAN value that specifies the state of the DTR line.
- `rts` A BOOLEAN value that specifies the state of the RTS line.

In both BOOLEAN values, `true` activates the line.

Requesting the Input Line States

If the local and remote hardware supports DTR and RTS lines, you can request the states of the input lines by sending `msgSioControlInStatus` to the serial port handle. The message takes a pointer to an `SIO_CONTROL_IN_STATUS` structure, which contains:

- `cts` A BOOLEAN value that receives the state of the CTS line.
- `dsr` A BOOLEAN value that receives the state of the DSR line.
- `rlsd` A BOOLEAN value that receives the state of the RLSD line.
- `ri` A BOOLEAN value that receives the state of the ring-indicator (RI) line.

In all BOOLEAN values, `true` means active.

Requesting All Serial Port Settings

You can request all the serial port settings by sending `msgSioGetMetrics` to the serial port handle. The message takes a pointer to an `SIO_METRICS` structure that contains:

- `baud` An `SIO_BAUD_SET` structure that receives the baud rate.
- `line` An `SIO_LINE_CONTROL_SET` structure that receives the line control.
- `controlOut` An `SIO_CONTROL_OUT_SET` structure that receives the serial port output line settings.
- `flowChar` An `SIO_FLOW_CONTROL_CHAR_SET` structure that receives the flow control characters.
- `flowType` An `SIO_FLOW_CONTROL_SET` structure that receives the flow control settings.
All of the structures here are described in the foregoing message descriptions.

You can use the same `SIO_METRICS` structure with `msgSioSetMetrics` to set all of the current serial port settings.

**Reading and Writing with the Serial Port**

To read from or write to the serial port, send `msgStreamRead` or `msgStreamWrite` to the serial port handle. Both messages take a pointer to a `STREAM_READ_WRITE` structure that specifies:

- `numBytes` The number of bytes to read or write.
- `pReadBuffer` A pointer to a buffer that receives the data, or containing data to be written. On `msgStreamRead`, the buffer must hold at least `numBytes` of data.

To read or write with a timeout value, send `msgStreamReadTimeout` or `msgStreamWriteTimeout` to the serial port handle. These messages require a `STREAM_READ_WRITE_TIMEOUT` structure that contains `numBytes` and `pReadBuffer` and a timeout value in milliseconds (`timeOut`).

For more information on reading or writing streams, see Chapter 79, Class Stream in *Part 9: Utility Classes*.

**Input and Output Buffer Status**

To find out the number of characters in the input or output buffer and the amount of room left in the buffer, send `msgSioInputBufferStatus` or `msgSioOutputBufferStatus` to the serial port handle. The `msgSioInputBufferStatus` message takes a pointer to an `SIO_INPUT_BUFFER_STATUS` structure that contains:

- `bufferChars` A location that receives the number of characters in the buffer.
- `bufferRoom` A location that receives amount of room left in the buffer.

The `msgSioOutputBufferStatus` message takes an `SIO_OUTPUT_BUFFER_STATUS` structure that contains:

- `bufferChars` A location that receives the number of characters in the buffer.
- `bufferRoom` A location that receives amount of room left in the buffer.
- `transmitterFrozen` A BOOLEAN value that indicates whether the transmitter is frozen. This can happen when the serial port receives XOFF or the RTS line is not active. For more information, see “Flow Control,” below.

**Flushing the Input and Output Buffers**

To flush (delete the contents of) the input or output buffers, send `msgSioInputBufferFlush` or `msgSioOutputBufferFlush` to the serial port handle. The messages do not take any arguments.
Flow Control

Now comes the moment when all the flow control information becomes useful. If you are performing your own buffer management and detect that your buffer is in danger of overflowing, you need to tell the device with which you are communicating to stop sending data; usually you do this with CTS/RTS.

Sending BREAK

A more drastic way to signal your partner to stop transmission is to send a BREAK signal (a series of zeros). BREAK sends a series of zeros to the stream. Of course, you can’t use a stream write message, because you don’t want the zeros buffered. Instead, you send msgSioBreakSend to the serial port handle. The message takes a pointer to an SIO_BREAK_SEND structure, which contains a single member, milliseconds. milliseconds specifies the length of time that zeros should be sent on the line, in milliseconds. A typical break duration is 200 to 400 milliseconds.

Detecting Events

The previous section discussed how to halt data transmission. This section describes how to detect those signals.

The best way to detect the halt signals is to make yourself an observer of the serial port manager. To do this, send msgSioEventSet to the serial port handle. The message takes a pointer to an SIO_EVENT_SET structure that contains:

- eventMask  An event mask, which describes the events for which you want to receive notification. Table 95-2 lists the event mask indicators.
- client The UID of an object to inform when the event happens. This is usually yourself.

<table>
<thead>
<tr>
<th>Indicator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>sioEventCTS</td>
<td>The CTS line changed state.</td>
</tr>
<tr>
<td>sioEventDSR</td>
<td>The DSR line changed state.</td>
</tr>
<tr>
<td>sioEventDCD</td>
<td>The DCD line changed state.</td>
</tr>
<tr>
<td>sioEventRI</td>
<td>The RI line changed state.</td>
</tr>
<tr>
<td>sioEventRxChar</td>
<td>Your receive buffer is no longer empty.</td>
</tr>
<tr>
<td>sioEventRxBreak</td>
<td>Received a break condition.</td>
</tr>
<tr>
<td>sioEventTxBufferEmpty</td>
<td>The sender's transmission buffer is empty.</td>
</tr>
<tr>
<td>sioEventRxError</td>
<td>A parity, framing, or overrun error occurred.</td>
</tr>
</tbody>
</table>

If you send msgSioEventSet to the serial port handle and one of the specified events occurs, the object named in the message receives msgSioEventHappened. The argument for this message is a pointer to an SIO_EVENT_HAPPENED structure, which contains:
eventMask  An SIO_EVENT_MASK structure that indicates the event or events that occurred. When the serial port manager sends msgSioEventHappened, it also clears its event mask.

self  The UID of the object that generated this message.

Note that some eventMask indicators might be set for events that you are not observing.

To get the current SIO_EVENT_SET structure, send msgSioEventGet to the handle on the serial port. The message takes a pointer to an SIO_EVENT_SET structure that will receive the event information.

**Polling for Events**

An alternative method for detecting events is to poll the event word, by sending msgSioEventStatus to the serial port handle. The message takes a pointer to an SIO_EVENT_STATUS structure that contains a location to receive the current state of the event mask (eventMask).

When the serial port manager receives this message, it returns the event mask to the requestor and clears the mask.

**Checking BREAK Status**

You can also poll the BREAK counter by sending msgSioBreakStatus to the serial port handle. The message takes a pointer to an SIO_BREAK_STATUS structure, which contains the location to receive the current break count (breaksReceived).

When the serial port manager receives the message, it sends back the number of breaks received since the last time the counter was cleared and then clears the counter.

**High-Speed Packet I/O Concepts**

The high-speed packet I/O interface (HSPKT) provides access to a protocol that performs high-speed data transfers using either a serial or parallel port. The high-speed packet interface is managed by the HighSpeedPacketHandlers service manager.

High-speed packet I/O implements a built-in RTS/CTS type protocol by sending a lead in character and expecting a data acknowledge character in return before actually sending a packet of data.

**HSPKT on Serial Lines**

When running on a parallel line, HSPKT uses a connection detection protocol. When running on a serial line, DSR high signals that it is connected, regardless of what the serial port is connected to.

Also when running on serial lines, the high-speed packet I/O can dynamically negotiate the baud rate.

In case of send errors, baud rates are renegotiated with remote station automatically.
**Parallel Cable Connection Detection**

The parallel connection protocol involves sending a parallel connect character and expecting to receive a connect acknowledge character back.

**Protocol Variations**

By setting leadInChar to 0, no lead in character is sent (and of course no dataAckChar is expected).

By setting dataAckChar to 0, no acknowledge character is sent upon receiving the lead in character, i.e. DVHSPKT goes on expecting data to arrive right after the lead in character.

By setting parConnectChar to 0, DVHSPKT reports always connected, leaving it up to users at a higher level to determine actual connection.

By setting parConnectAckChar to 0, no connect acknowledge character is sent upon receiving the connect character.

**Notes**

As data is transmitted/received in parallel mode, DVHSPKT remains synchronized with the other side during the entire data transfer. As a result, both transmissions and receptions are subject to failure, making the use of a lead in/ack protocol not always a necessity when communicating through the parallel port. The use of at least a lead in character may however improve performance as fake interrupts would be noticed early (by DVHSPKT itself) saving the upper layer code the trouble of validating the beginning of a packet.

Parallel transfer speeds depend on the speed of the machines transmitting/receiving data.

In asynchronous serial mode, DVHSPKT synchronizes itself with the other side only upon receiving the first byte (lead in) by sending a data ack character to inform the other side. The absence of a lead in/ack protocol might then cause overruns on slower machines.

See MIL specifications for an explanation of the protocols used by this device.
This chapter presents both clsParallelPort, the parallel port device driver interface class, and clsHighSpeedPacket, the high-speed packet I/O class (HSPKT).

**Parallel Port Concepts**

Usually application writers don’t need to communicate directly with the parallel port. Rather, applications (with the assistance of the PenPoint™ Application Framework) communicate with a printer driver, the printer driver then communicates with the parallel port. However, you can use the parallel port to communicate with devices other than printers.

To access a parallel port, a printer driver sends open and bind messages to the parallel device service manager (theParallelDevices), requesting the port by name. If that port is available, the service manager gives the printer driver a handle on the parallel port.

clsParallelPort inherits from clsMILService, which is a descendent of clsStream. Structures and #defines used by clsParallelPort are in \GO\INCPORT.H.

**Parallel Port Interrupts**

Because of a problem in the 8259 programmable interrupt controller (PIC) used by most PCs, some machines can generate sporadic interrupt 7s under certain, unpredictable conditions. The symptom is that you see several “Int w/o RB: 7” messages at boot time. We have not seen this behavior on tablet hardware.

The PenPoint operating system attempts to avoid hanging conditions by disabling interrupt 7 whenever too many bad interrupts occur, and the re-enabling interrupt 7 at a later time. This does not limit PenPoint’s functionality.

Theoretically, parallel port I/O could become impossible, if PenPoint were to constantly disable and enable interrupt 7. However, we have not seen this situation.
Parallel Port Messages

Table 96-1 lists the messages defined by clsParallelPort. These messages, their structures, and #defines are all defined in PPORT.H.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_PPORT_NEW</td>
<td>Creates a new pport object.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_PPORT_NEW</td>
<td>Initializes a structure to create a new pport object.</td>
</tr>
<tr>
<td>msgPPortStatus</td>
<td>P_PPORT_STATUS</td>
<td>Returns the current status of the printer.</td>
</tr>
<tr>
<td>msgPPortInitialize</td>
<td>P_NULL</td>
<td>Initializes the printer.</td>
</tr>
<tr>
<td>msgPPortAutoLineFeedOn</td>
<td>P_NULL</td>
<td>Inserts a line feed after each carriage return.</td>
</tr>
<tr>
<td>msgPPortAutoLineFeedOff</td>
<td>P_NULL</td>
<td>Disables inserting a line feed after each carriage return.</td>
</tr>
<tr>
<td>msgPPortGetTimeDelays</td>
<td>P_PPORT_TIME_DELAYS</td>
<td>Gets the initialization and interrupt time-out intervals.</td>
</tr>
<tr>
<td>msgPPortSetTimeDelays</td>
<td>P_PPORT_TIME_DELAYS</td>
<td>Sets the initialization and interrupt time-out intervals.</td>
</tr>
<tr>
<td>msgPPortCancelPrint</td>
<td>P_NULL</td>
<td>Cancels the printing of the buffer currently being printed.</td>
</tr>
</tbody>
</table>

Using the Parallel Port

Like other device drivers, the parallel driver is a service. To access a parallel port, you must use the services protocol to find a parallel port service, bind to the service, and then open it. The service manager for parallel ports, the ParallelDevices, controls access to the parallel ports.

Requesting a Parallel Port Handle

Before you can use a parallel port, you must request a handle on a port from the ParallelDevices service manager. To locate the port by name, send msgIMFind to the ParallelDevices. To bind the port to your process, send msgSMBind to the ParallelDevices. Finally, to open the port, send msgSMOpen to the ParallelDevices. This example demonstrates how to open a parallel port.

```c
STATUS s;
IM_FIND imf;
SM_BIND sm;
SM_OPEN so;
OBJECT pportUid;
imf.pName = userName; // The name was requested from the user earlier
ObjCallRet(msgIMFind, theParallelDevices, &imf, s);
sm.handle = imf.handle;
sm.caller = self;
ObjCallRet(msgSMBind, theParallelDevices, &sm, s);
so.handle = imf.handle;
so.caller = self;
ObjCallRet(msgSMOpen, theParallelDevices, &so, s);
pportUid = so.service;
```
The symbol pportUid now contains the UID of the parallel port object. You can communicate with the parallel port by sending messages to this object.

**Releasing the Parallel Port Object**

When you have finished with the parallel port, you should release the port by sending msgSMClose to theParallelDevices, as shown in the example below. Do not send msgDestroy to the parallel port object.

\[
\text{pportUid} = \text{so.service}; \\
\text{ObjCallRet}(	ext{msgSMClose}, \text{theParallelDevices}, \text{wknKey});
\]

**Parallel Port Configuration**

When you have the UID of the parallel port object, you should configure the parallel port for the printer that is attached to it. This involves setting the auto line feed state and setting the initialization and interrupt time-out intervals.

**Setting Auto Line Feed**

Some printers allow you to specify whether the printer should do a line feed after each carriage return. Mostly it is up to your application to determine whether it should send a line feed or enable the printer to insert line feeds automatically.

To turn on auto line feed, send msgPPortAutoLineFeedOn to the parallel port object. To turn off auto line feed, send msgPPortAutoLineFeedOff to the parallel port object. Both messages take a pointer to NULL.

**Get and Set Time Delays**

The parallel port driver allows you to get and set two time values:

- The duration of the initialization pulse.
- The interval at which characters can be sent to the printer.

To get or set these values, send msgPPortGetTimeDelays or msgPPortSetTimeDelays to the parallel port object. Both messages take a pointer to a PPORT_TIME_DELAYS structure, which contains:

- initDelay A U32 that specifies the duration of the initialization pulse (in microseconds).
- interruptTimeOut A U32 that specifies the maximum amount of time to wait for the printer to indicate it is ready to accept another character (in milliseconds).

**Initializing the Printer**

Before printing, you should initialize the printer attached to the parallel port by sending msgPPortInitialize to the parallel port object. The message takes no arguments.
msgPPortInitialize sends the initialization signal for the amount of time specified
by the initDelay argument to msgPPortSetTimeDelays.

**Writing to the Parallel Port**

Your application sends data through the parallel port by sending msgStreamWrite
to the parallel port object. This is the only clsStream message that clsParallelPort
handles.

**Getting Status**

To get the current printer status, send msgPPortStatus to the printer. The message
takes a pointer to a PPORT.STATUS structure that contains a single U16 value
(pportStatus), which contains one or more of the following values:

- **pportStsBusy**  The printer is busy.
- **pportStsAcknowledge**  The printer has accepted a character.
- **pportStsEndOfPaper**  The paper is out.
- **pportStsSelected**  The printer is on line.
- **pportStsIOError**  There was an I/O error on the printer.
- **pportStsInterruptHappened**  An interrupt occurred.

**Cancelling Printing**

To cancel printing, send msgPPortCancelPrint to the parallel port object. The
message takes a pointer to NULL.
Chapter 97 / Data Modem Interface

This section describes the modem interface implemented by clsModem. clsModem provides device-independent access to a data modem attached to a serial port and makes the data modem command sets transparent to clients.

Chapter 97 covers these topics:
- How to access the data modem.
- Sending modem commands and data.
- The data modem command set.

Concepts

The data modem plugs into an option slot on the PenPoint computer. Before you communicate with the data modem, you need to establish communication with the option slot serial port. You establish communication with the serial port by sending bind and open messages to theSerialDevices.

When you have access to a serial port, there are two ways to communicate with the data modem:
- Communicating through the modem interface.
- Sending commands and data directly through the serial port.

The modem interface is implemented by clsModem and provides a device-independent, object-oriented interface to the modem. If you use the modem interface, you let clsModem perform all of the management tasks associated with establishing data modem communications. clsModem can also auto-answer the modem.

Communicating directly with the modem through the serial port is not recommended, but certain applications may need to do so. If you choose to communicate with the modem through the serial port, your code will be device-dependent. You must not initiate the modem driver (clsModem) when you communicate with the modem directly. Additionally, you are responsible for separating modem responses from data received by the modem.

Getting a Serial Port Handle

The data modem connects to a serial port on a PenPoint computer. You access the serial port by sending bind and open messages to theSerialDevices service manager; the open message sends back a handle on the serial port, you send commands and data to the port by sending serial I/O messages to the handle.
To get a handle on the serial port, you must send `msgSMBind` and `msgSMOpen` to the `SerialDevices`, specifying the name of the serial port. For example:

```c
STATUS s;
IM_FIND imf;
SM_BIND sm;
SM_OPEN so;
OBJECT serialHandle;
imf.pName = "Option Slot";
ObjCallRet(msgIMFind, theSerialDevices, &imf, s);
sm.handle = imf.handle;
sm.caller = self;
ObjCallRet(msgSMBind, theSerialDevices, &sm, s);
so.handle = imf.handle;
so.caller = self;
ObjCallRet(msgSMOpen, theSerialDevices, &so, s);
serialHandle = so.service;
```

### Configuring the Serial Port

You can change the configuration of the serial port before or after you create the modem object. Typically you might want to set the baud rate, data bits, parity, stop bits to match the configuration of the remote modem.

When you first reinitialize the serial port handle, it has these defaults:

- 8 bits
- No parity
- One stop bit
- XON/XOFF flow control
- Ctrl-Q and Ctrl-S are the XON and XOFF characters
- DTR and RTS are on.

You can either use the get and set metrics messages for `clsSio`, or if you only need to adjust one or two characteristics, you can send messages to configure those characteristics independently.

In the following example, the application reconfigures the serial port to communicate at 2400 baud with 7 bit bytes by sending `msgSioBaudSet` and `msgSioLineControlSet` to the serial port handle opened in the previous example (serialHandle).

```c
STATUS s;
SIO_BAUD_SET sioBaud;
SIO_LINE_CONTROL_SET sioLineControl;
sioBaud.baudRate = 2400;
s = ObjectCall(msgSioBaudSet, serialHandle, &sioBaud);
sioLineControl.dataBits = sioSevenBits;
sioLineControl.StopBits = sioOneStopBit;
sioLineControl.parity = sioNoParity;
s = ObjectCall(msgSioLineControlSet, serialHandle, &sioLineControl);
```
The clsModem API

The modem interface provides an object-oriented interface for you to communicate with a modem. The modem interface also provides two other capabilities:

- Auto-answer and connection detection.
- Asynchronous event handling.

To use the modem interface, you must get a handle on the serial port to which the modem is attached. When you have the serial port handle, you send msgNew to clsModem, specifying the serial port handle. clsModem sends back a new handle, to which you send all modem, serial, and stream messages until you destroy the modem interface object.

Do not send modem commands directly to a serial port that is being used by a modem interface object; you must send messages to the modem object. clsModem expects the modem to be configured in a certain way. If you send an AT command that changed the modem's responses, clsModem will produce unpredictable results.

The clsModem Messages

The clsModem messages are defined in the file MODEM.H. Table 97-1 lists the clsModem messages.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_MODEM_NEW</td>
<td>Creates a new instance of a modem service.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_MODEM_NEW</td>
<td>Initializes the MODEM_NEW structure to default values.</td>
</tr>
<tr>
<td>msgModemReset</td>
<td>nothing</td>
<td>Resets the modem firmware, I/O port, and service state.</td>
</tr>
<tr>
<td>msgModemOnline</td>
<td>nothing</td>
<td>Forces the modem online into data mode.</td>
</tr>
<tr>
<td>msgModemSetDiaIType</td>
<td>MODEM_DIAL_MODE</td>
<td>Establishes the mode for dialing telephone numbers.</td>
</tr>
<tr>
<td>msgModemHangUp</td>
<td>nothing</td>
<td>Hang-ups and disconnects to terminate a connection.</td>
</tr>
<tr>
<td>msgModemOffHook</td>
<td>nothing</td>
<td>Picks up the phone line.</td>
</tr>
<tr>
<td>msgModemSetSpeakerControl</td>
<td>MODEM_SPEAKER_CONTROL</td>
<td>Enables, disables and controls modem speaker behavior.</td>
</tr>
<tr>
<td>msgModemSetAutoAnswer</td>
<td>P_MODEM_SET_AUTO_ANSWER</td>
<td>Disables or enables the modem auto-answer feature.</td>
</tr>
<tr>
<td>msgModemSendCommand</td>
<td>P_MODEM_SEND_COMMAND</td>
<td>Sends a specified command to the modem.</td>
</tr>
<tr>
<td>msgModemSetDuplex</td>
<td>MODEM_DUPLEX_MODE</td>
<td>Sets the duplex mode for inter-modem-communications while on-line.</td>
</tr>
</tbody>
</table>

continued
### Table 97-1 (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgModemDial</td>
<td>P_MODEM_DIAL</td>
<td>Performs dialing and attempts to establish a connection.</td>
</tr>
<tr>
<td>comsgModemSetResponseBehavior</td>
<td>P_MODEM_RESPONSE_BEHAVIOR</td>
<td>Set the modem response mode, and command-to-response time-out values.</td>
</tr>
<tr>
<td>msgModemGetResponseBehavior</td>
<td>P_MODEM_RESPONSE_BEHAVIOR</td>
<td>Passes back the current modem response mode and the current command-to-response time-out values.</td>
</tr>
<tr>
<td>msgModemGetConnectionInfo</td>
<td>P_MODEM_CONNECTION_INFO</td>
<td>Passes back information about the current connection.</td>
</tr>
<tr>
<td>msgModemSetAnswerMode</td>
<td>MODEM_ANSWER_MODE</td>
<td>Filters the type of calls to answer and connection reporting.</td>
</tr>
<tr>
<td>msgModemAnswer</td>
<td>nothing</td>
<td>Immediately answers a telephone call.</td>
</tr>
<tr>
<td>msgModemSetSignallingModes</td>
<td>P_MODEM_SIGNALLING_MODES</td>
<td>Restricts the modem to use specific signalling modes or standards.</td>
</tr>
<tr>
<td>msgModemSetToneDetection</td>
<td>MODEM_TONE_DETECTION</td>
<td>Enables or disables busy tone or dial tone detection.</td>
</tr>
<tr>
<td>msgModemSetSpeakerVolume</td>
<td>MODEM_SPEAKER_VOLUME</td>
<td>Sets the volume of the modem speaker.</td>
</tr>
<tr>
<td>msgModemSetCommandState</td>
<td>nothing</td>
<td>Sets the modem into command mode.</td>
</tr>
<tr>
<td>msgSvcGetMetrics</td>
<td>P_SVC_GET_SET_METRICS</td>
<td>Passes back the current modem metrics.</td>
</tr>
<tr>
<td>msgSvcSetMetrics</td>
<td>P_SVC_GET_SET_METRICS</td>
<td>Set current modem metrics, and re-initialize modem with specified metrics.</td>
</tr>
<tr>
<td>msgSvcCharacteristicsRequested</td>
<td>P_SVC_CHARACTERISTICS</td>
<td>Passes back the characteristics of the modem service.</td>
</tr>
<tr>
<td>msgModemSetMNPMode</td>
<td>MODEM_MNP_MODE</td>
<td>Sets the MNP mode of operation.</td>
</tr>
<tr>
<td>msgModemSetMNPCompression</td>
<td>MODEM_MNP_COMPRESSION</td>
<td>Sets MNP class 5 compression on or off.</td>
</tr>
<tr>
<td>msgModemSetMNPBreakType</td>
<td>MODEM_MNPBREAK_TYPE</td>
<td>Specifies how a break is handled in MNP mode.</td>
</tr>
<tr>
<td>msgModemSetMNPFlowControl</td>
<td>MODEM_MNP_FLOW_CONTROL</td>
<td>Specifies the flow control to use in MNP mode.</td>
</tr>
</tbody>
</table>

### Observer Messages

- **msgModemConnected**: The modem has connected with a remote node modem.
- **msgModemDisconnected**: The current connection has been terminated.
- **msgModemRingDetected**: A ring indication has been received from the modem.
- **msgModemErrorDetected**: An unexpected error indication has been received from the modem.
- **msgModemTransmissionError**: An error has been detected during data transmission.

---

**Creating a clsModem Object**

To create the clsModem object, you first get a handle on the serial port to which the modem is connected. You then send `msgNewDefaults` and `msgNew` to
The clsModem Messages

The messages take a pointer to a MODEM_NEW structure that contains an OBJECT_NEW_ONLY structure (object) and a MODEM_NEW_ONLY structure (modem). The MODEM_NEW_ONLY structure contains:

modem.client  The UID of the client that will use the modem object. This is usually self.

modem.sioPort  The UID of the serial port handle.

When the message completes, the object.uid field of the MODEM_NEW structure contains the UID of the new modem object.

Configuring the Modem

When you get the modem object, the first thing you should do is reset the modem to its default settings; that way you know the state of the modem before continuing. To reset the modem, send msgModemReset to the modem object. The message has no arguments. When the message completes successfully, it returns stsOK.

Table 97-2 lists the default settings established by msgModemReset. clsModem requires certain settings so that it can interpret responses from the modem and pass its interpretation on to the client. Because you are using clsModem, you shouldn't have any reason for setting these values. The required settings are:

- E0  Echo mode must be off.
- Q0  Return result codes.
- V1  result codes must be whole words.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>AT Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bell or CCITT protocol</td>
<td>B1</td>
<td>Use Bell protocol.</td>
</tr>
<tr>
<td>Duplex</td>
<td>F1</td>
<td>Use full duplex.</td>
</tr>
<tr>
<td>Speaker</td>
<td>M1</td>
<td>Speaker on until carrier.</td>
</tr>
<tr>
<td>Dialing mode</td>
<td>T</td>
<td>Use touch tone dialing.</td>
</tr>
<tr>
<td>Result code range</td>
<td>X4</td>
<td>Wait for dial tone, detect busy signal.</td>
</tr>
<tr>
<td>Long space disconnect</td>
<td>Y0</td>
<td>Disabled.</td>
</tr>
<tr>
<td>Carrier detect (CD)</td>
<td>&amp;C1</td>
<td>On in presence of carrier.</td>
</tr>
<tr>
<td>DTR off action</td>
<td>&amp;D2</td>
<td>Hang up, no auto-answer.</td>
</tr>
<tr>
<td>Pulse dial ratio</td>
<td>&amp;P0</td>
<td>Ratio is 39/61 (USA).</td>
</tr>
<tr>
<td>Guard tones</td>
<td>&amp;G0</td>
<td>No guard tones.</td>
</tr>
<tr>
<td>RDL test</td>
<td>&amp;T4</td>
<td>Grant RDL test request.</td>
</tr>
<tr>
<td>Auto-answer mode</td>
<td>S0=000</td>
<td>Auto-answer disabled.</td>
</tr>
<tr>
<td>Escape character</td>
<td>S2=043</td>
<td>Escape character is +.</td>
</tr>
<tr>
<td>Carriage return</td>
<td>S3=013</td>
<td>Carriage return is 13.</td>
</tr>
<tr>
<td>CD timeout</td>
<td>S7=30</td>
<td>Wait 30 seconds for CD.</td>
</tr>
<tr>
<td>Response to carrier</td>
<td>S9=6</td>
<td>Respond to CD after .6 seconds.</td>
</tr>
<tr>
<td>Lost carrier hang-up</td>
<td>S10=14</td>
<td>Wait 1.4 seconds after loss of carrier before hang-up.</td>
</tr>
<tr>
<td>Touch tone spacing</td>
<td>S11=95</td>
<td>Wait 70 ms between tones.</td>
</tr>
<tr>
<td>Escape code guard time</td>
<td>S12=50</td>
<td>Wait 1 second before sensing escape characters.</td>
</tr>
<tr>
<td>Test timer</td>
<td>S18=0</td>
<td>Test timer is 0.</td>
</tr>
<tr>
<td>DTR detect delay</td>
<td>S25=5</td>
<td>Wait .05 seconds for DTR.</td>
</tr>
</tbody>
</table>
clsModem provides messages that allow you to set the following modem characteristics:

- Dial type
- Autoanswer mode
- Carrier state
- Speaker mode
- Command/data mode
- Duplex mode
- MNP mode.

### Setting the Dial Type

The modem can dial in either pulse or touch-tone mode. To set the dialing type of the modem, send `msgModemDialTypeSet` to the modem object. The message takes a pointer to a `MODEM_DIAL_TYPE_SET` structure that contains a single element, a `MODEM_DIAL_TYPE` value (dType) that specifies `pulseDialing` or `touchtoneDialing`.

### Setting the Auto-Answer Mode

You can configure the modem so that it automatically answers the phone after a specified number of rings. To set the modem's auto-answer mode, send `msgModemAutoAnswerSet` to the modem object. The message takes a pointer to a `MODEM_AUTO_ANSWER_SET` structure that contains:

- **answerType** An `AUTO_ANSWER_TYPE` value that specifies the answer mode. Possible values are:
  - `autoAnswerOff` Do not answer the phone.
  - `dataModemAnswer` Answer the phone in data mode.
- **rings** The number of rings after which to answer the phone. This value can be between 0 and 255. If the value is 0, or is not specified, the modem answers after one ring; if the value is greater than 255, clsModem uses 255.

### Setting the Carrier State

When you have a connection with another modem, you can enable or disable the carrier state by sending `msgModemCarrierStateSet` to the modem object. The message takes a pointer to a `MODEM_CARRIER_STATE_SET` structure that contains a single element, a `MODEM_CARRIER_STATE` value (carrierState) that specifies either `enableCarrier` or `disableCarrier`.

### Controlling the Speaker

You can turn the modem speaker on and off. When the speaker is on, you can choose to turn it off when the carrier is detected or you can keep it on all the time. Most applications turn the speaker on and then turn it off when the carrier is detected. This
provides the user with an audible signal when the connection is made (a tone, followed by a higher tone, and then a burst of noise indicates that a carrier was detected).

To set the speaker, send `msgModemSpeakerControlSet` to the modem object. The message takes a pointer to a `MODEM_SPEAKER_STATE_SET` structure that contains a single element, a `SPEAKER_STATE` value that can specify:

- `speakerOff`  Turn the speaker off.
- `speakerOnConnectOff`  Turn the speaker on, but turn it off when the carrier is detected.
- `speakerOn`  Turn the speaker on.

### Setting Command and Data Modes

When the modem is in command mode, it can receive the AT commands. In data mode, the modem sends data to and receives data from the modem to which it is connected.

To put the modem in command mode, send `msgModemCommandModeSet` to the modem object. The message takes no arguments.

To put the modem in data mode, send `msgModemOnline` to the modem object. The message takes a pointer to a `MODEM_ONLINE` structure that contains a single object, a pointer to a `MODEM_CONN_TYPE` value. When the message completes successfully, it sends back the connection type in the `MODEM_CONN_TYPE` value.

Table 97-3 lists the modem connection types and their meanings.

If the modem connects successfully, the message returns `stsOK`. If the modem does not connect within a reasonable amount of time, the message returns `stsTimeOut`. Otherwise, the message returns `stsModemUnexpectedResponse`.

When the modem detects a carrier, it automatically goes to data mode.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>noConnection</td>
<td>No connection.</td>
</tr>
<tr>
<td>connect300</td>
<td>Data connection at 300 baud.</td>
</tr>
<tr>
<td>connect600</td>
<td>Data connection at 600 baud.</td>
</tr>
<tr>
<td>connect1200</td>
<td>Data connection at 1200 baud.</td>
</tr>
<tr>
<td>connect2400</td>
<td>Data connection at 2400 baud.</td>
</tr>
<tr>
<td>connect4800</td>
<td>Data connection at 4800 baud.</td>
</tr>
<tr>
<td>connect9600</td>
<td>Data connection at 9600 baud.</td>
</tr>
<tr>
<td>connect19200</td>
<td>Data connection at 19200 baud.</td>
</tr>
<tr>
<td>connectMNP1200</td>
<td>MNP connection at 1200 baud.</td>
</tr>
<tr>
<td>connectMNP2400</td>
<td>MNP connection at 2400 baud.</td>
</tr>
<tr>
<td>connectLAPM1200</td>
<td>Lap M connection at 1200 baud.</td>
</tr>
<tr>
<td>connectLAPM2400</td>
<td>Lap M connection at 2400 baud.</td>
</tr>
</tbody>
</table>
Setting Duplex Mode

The modem allows you to choose either duplex or half duplex mode. In duplex mode, the modem echoes characters as they are sent; in half duplex mode, the modem does not echo the characters.

To set the duplex mode, send `msgModemDuplexSet` to the modem object. The message takes a pointer to a `MODEM_DUPLEX_SET` value that contains a single element, a `DUPLEX_TYPE` value that can be either `halfDuplex` or `fullDuplex`.

Setting MNP Mode

The Microcom Network Protocol (MNP) provides data compression and enhanced error checking. If both modems in a connection implement MNP, the modems communicate using MNP.

The MNP mode specifies how the modem will act when it connects to another modem that does or does not support MNP. To set the MNP mode, send `msgModemMNPModeSet` to the modem object. The message takes a pointer to a `MODEM_MNP_MODE_SET` structure that contains a single element, a `MNP_MODE` value that specifies:

- `mnpAutoReliableMode` Attempt to connect using MNP at the baud rate specified in `msgModemDial` or `msgModemOnline`. If not successful, drop the baud rate back until you get an MNP connection. If you still don't get a connection, attempt a connection at the highest baud rate without MNP. If not successful at that baud rate, drop the baud rate back until you get a connection.

- `mnpReliableMode` Attempt to connect using MNP at the baud rate specified in `msgModemDial` or `msgModemOnline`. If not successful, drop the baud rate back until you get an MNP connection. If you still don't get a connection, give up.

- `mnpDirectMode` Communicate directly with no error checking or compression, that is, without MNP.

- `mnpLapMMode` Use Link Access Protocol for Modems (LAP-M) Mode. LAP-M is the CCITT V.42 error correcting code. LAP-M also includes support for MNP levels 2 through 4.

Further messages that affect the MNP protocol are described in the section titled “MNP Data Communication.”

Sending Your Own AT Commands

If you need to send specific commands to the modem through the modem interface, you can send `msgModemSendCommand` to the modem object. However, this message does not work with all modem commands. Do not use `msgModemSendCommand` to send dial commands or to set the echo mode (En), quiet mode (Qn), and result code style (Vn); `clsModem` will fail if these values are modified.
msgModemSendCommand takes a pointer to a MODEM_SEND_COMMAND structure that contains:

- **pCmd** A pointer to the command string.
- **responseTimeout** A timeout value for a response from the modem.

When the message completes, it sends back the response from the modem in the response field of the MODEM_SEND_COMMAND structure.

### Establishing a Connection with a Data Modem

To establish a connection, you can use clsModem commands to dial or connect directly with another modem.

To dial another modem, make sure the dial type matches your phone equipment (it is usually touch-tone) and send msgModemDial to the modem object. The message takes a pointer to a MODEM_DIAL structure that contains:

- **pPhoneNumber** A pointer to the phone number to dial in ASCII.
- **pCType** A pointer to the MODEM_CONN_TYPE value that will receive the connection type.

The phone number usually contains the number to dial. It can also contain a number of **dial string modifiers** defined by the AT command set. These dial string modifiers are described in the following section Dial String Modifiers.

When the other modem answers, clsModem waits for the connection to be established, then sends back the connection type in the value indicated by pCType.

When msgModemDial successfully completes, it establishes a connection and send back the connection type. If you go back to command mode (by sending msgModemCommandModeSet to the modem object) to change the modem configuration, you can return to data mode by sending msgModemOnline to the modem object.

When you have finished with a connection by either telephone or direct wiring, send msgModemHangup to the modem object. The message takes no arguments.

The message terminates the connection and, if you are connected by telephone, hangs up the phone.

### Dial String Modifiers

The phone number in pPhoneNumber usually contains the number to dial. It can also contain a number of **dial string modifiers** defined by the AT command set. These dial string modifiers can:

- Pause dialing (useful when waiting for an outside line or using alternative long distance companies).
- Wait for silence (waiting through a prerecorded message).
Switch between pulse and tone dialing (useful when dealing with heterogeneous phone systems).

Flash the switch hook.

The phone number can contain these characters:

- The numbers 0 through 9, the alphabetic characters A B C D, and the special characters * and #.
- The alphabetic characters: P T R W.
- The special characters: comma (,), semicolon (;), at sign (@), exclamation point (!), and slash (/).
- The string $S=n$.

The characters 0-9, A, B, C, D, *, and #, are the same as the keys on a touch-tone phone. The characters A B C D, and the symbols * and #, can be used only during tone dialing; they are typically used to access newer features of modern telephone systems.

The character P directs the modem to pulse dial the digits that follow it. The character T directs the modem to tone dial the digits that follow it. These modifiers are useful when you dial into systems that require a mix of tone and pulse signals.

The character R forces the modem to use answer mode frequencies after dialing the number. This allows you to dial up an originate-only modem. This character must only be at the end of the dialing string.

The character W causes the modem to wait a specified amount of time for a dial tone before proceeding. The default is 30 seconds.

The semicolon character (;) causes the modem to go back into the command state after dialing, which allows you to enter other commands while online.

The comma (,) causes the modem to pause. The default time for the pause is two seconds, and can be changed by modifying register S8.

The at sign (@) causes a "hookflash." This simulates hanging up for 0.5 second and then reconnecting. It is typically used for transferring calls.

A slash (/) causes the modem to wait for 0.125 second before proceeding with the rest of command line.

The $S=n$ dial string modifier causes the modem to dial one of the four phone numbers previously stored in the modem's non-volatile memory. You store numbers with the &Zn=number command.
Waiting for a Connection with a Data Modem

You can tell the modem object to automatically answer the phone, or you can answer the phone yourself (with clsModem messages).

To instruct the modem to answer the phone automatically, send msgModemAutoAnswerSet to the modem object, specifying dataModemAnswer. When another modem dials your number and the phone rings, clsModem picks the phone up, determines the type of connection, and sends you msgModemConnected, which indicates the type of connection and the baud rate.

To answer the phone yourself, send msgModemAutoAnswerSet to your modem object, specifying autoAnswerOff.

When the phone rings, the modem object sends you msgModemRingDetected. The message has no arguments. You answer the phone by sending msgModemOffHook to the modem object (the message takes no arguments).

You then send msgModemOnline to the modem object. The message takes a pointer to a MODEM_ONLINE structure that contains a pointer to a MODEM_CONN_TYPE. When the message returns, it sends back the connection type in the MODEM_CONN_TYPE value. The connection types are listed above in Table 97-3.

It is up to you to examine the connection type and determine if you want that connection or not.

Sending and Receiving Data

To send data to the other modem, send msgStreamWriteTimeout to the modem object. To receive data from a modem, send msgStreamReadTimeout to the modem object.

If the connection is lost at any time, you will receive msgModemDisconnected.

MNP Data Communication

As mentioned earlier, MNP performs additional error checking and data compression when two MNP-capable modems have a connection. To set MNP mode, send msgModemMNPModeSet to the modem object. This message is described above in “Setting MNP Mode.”

To turn MNP compression on or off, send msgModemMNPCompressionSet to the modem object. This message takes a pointer to a MODEM_MNP_COMPRESSION_SET structure that contains a single MNP_COMPRESSION_TYPE value (mnpCompression), which specifies mnpNoDataCompression or mnpDataCompression.

If you use MNP, you can specify when clsModem will send break messages to the other modem. Sometimes you want to send break immediately, at other times you want to wait until the send buffer is empty. To set the break type, send msgModemMNPBreakTypeSet to the modem object. The message takes a
pointer to a MODEM_MNP_BREAK_TYPE_SET structure that contains a single element, an MNP_BREAK_TYPE value that specifies:

- mnpSendNoBreak  Do not send breaks.
- mnpEmptyBuffersThenBreak  Send breaks when the buffer is empty.
- mnpImmediatelySendBreak  Send breaks immediately.
- mnpSendBreakInSequence  Send breaks in the sequence that they are received.

When using MNP, you can direct it to use different types of flow control, or not to use flow control at all. To set the MNP flow control, send msgModemMNPFlowControlSet to the modem object. The message takes a pointer to a MODEM_MNP_FLOW_CONTROL_SET structure that contains a single element, an MNP_FLOW_CONTROL value (mnpFlowControl), which specifies:

- mnpDisableFlowControl  Do not use flow control.
- mnpXonXoffFlowControl  Use XON/XOFF flow control.
- mnpCtsRtsFlowControl  Use the hardware CTS/RTS flow control.

Direct Communication with the Data Modem

If you do not want to use clsModem to interpret the modem's responses and handle asynchronous events, you can communicate directly with the modem.

After configuring the serial port to which the modem is attached, use clsStream messages to send information to and receive information from the modem.

If you write an application that communicates directly with the modem, your application will not be device-independent. Additionally, you cannot communicate directly with the modem and use the modem interface at the same time.

You send the following instructions to the modem:

- Commands to control and configure your modem.
- Data to be transmitted to another modem.

You receive the following information back from the modem:

- Responses from your modem (about your modem commands).
- Data transmitted by another modem.

The Data Modem AT Command Set

You use the AT command set to control the modem switch-hook operations, baud rate, and other standard functions.

This section contains a brief summary of the AT command set.

The AT commands have the following syntax:

```
ATcommand[ arg ][=n][ command... ]
```
The only commands that do not follow this syntax are the / and +++ commands. All commands can be in either upper or lowercase. After the initial AT you can append a number of AT commands on the same command line, however the total length of the line cannot exceed 40 characters.

If you omit a parameter, its value is assumed to be 0.

### Summary of AT Command Set

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+++</td>
<td>Escape from modem data mode.</td>
</tr>
<tr>
<td>A</td>
<td>Answers a call, regardless the of S0 setting.</td>
</tr>
<tr>
<td>A/</td>
<td>Re-executes the last command.</td>
</tr>
<tr>
<td>Bn</td>
<td>Selects Bell (B0) or CCITT (B1) protocols. Ignored at 2400 baud.</td>
</tr>
<tr>
<td>Cn</td>
<td>Turns the carrier signal on and off. C0 turns the carrier off; C1 turns the carrier on.</td>
</tr>
<tr>
<td>Dn..n</td>
<td>Dials a phone number using most recently used dialing method (pulse or tone).</td>
</tr>
<tr>
<td>DTn..n</td>
<td>Dials a phone number using pulse dialing.</td>
</tr>
<tr>
<td>DTn..n</td>
<td>Dials a phone number using tone dialing.</td>
</tr>
<tr>
<td>En</td>
<td>Sets echo mode. E0 turns echo mode off; E1 turns echo mode on.</td>
</tr>
<tr>
<td>Fn</td>
<td>Sets Halfduplex (F0) or Fullduplex (F1).</td>
</tr>
<tr>
<td>Hn</td>
<td>Operates the switch hook. H0 is on switch hook (hang up); H1 is off switch hook (pick up).</td>
</tr>
<tr>
<td>In</td>
<td>Asks the modem to identify itself. I0 returns the product identification code; I1 returns the firmware ROM checksum; I2 computes the firmware ROM checksum and returns either OK or ERROR.</td>
</tr>
<tr>
<td>Ln</td>
<td>Sets speaker volume. Values for n are between 0 and 3; L0 is lowest volume, L3 is highest volume.</td>
</tr>
<tr>
<td>Mn</td>
<td>Turns speaker on and off. M0 means speaker is always off; M1 means speaker is on until carrier is detected, M2 means speaker is always on, and M3 means the speaker goes on after the last digit is dialed and goes off after carrier is detected.</td>
</tr>
<tr>
<td>On</td>
<td>Returns modem to data state. O0 returns without breaking connection; O1 returns after an equalizer retrain sequence.</td>
</tr>
<tr>
<td>P</td>
<td>Sets dial mode to pulse.</td>
</tr>
<tr>
<td>Qn</td>
<td>Sets quiet mode. Q0 reports result codes; Q1 remains quiet.</td>
</tr>
<tr>
<td>Sr?</td>
<td>Returns the current value of register r. r is between 0 and 27.</td>
</tr>
<tr>
<td>Sr=n</td>
<td>Sets the value of register r. r is between 0 and 27; n is between 0 and 255.</td>
</tr>
<tr>
<td>T</td>
<td>Sets dial mode to tone.</td>
</tr>
<tr>
<td>Vn</td>
<td>Sets reports to either textual or numeric result codes. V0 returns numeric result codes; V1 returns textual result codes.</td>
</tr>
<tr>
<td>Xn</td>
<td>Sets extended result code reporting. See the data modem documentation for details.</td>
</tr>
<tr>
<td>Yn</td>
<td>Enables and disables long space disconnect mode. Y0 disables the mode; Y1 enables it.</td>
</tr>
<tr>
<td>Zn</td>
<td>Resets the modem to one of two stored profiles. Z0 resets the modem to stored profile 0; Z1 resets it to profile 1.</td>
</tr>
<tr>
<td>&amp;Cn</td>
<td>Sets how the modem controls the CD signal. &amp;C0 forces CD on; &amp;C1 sets CD on when there is a valid carrier signal.</td>
</tr>
</tbody>
</table>

*continued*
<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;Dn</td>
<td>Sets how the modem responds to a DTR OFF signal. &amp;D0 ignores DTR; &amp;D1 causes the modem to enter command state when DTR goes off; &amp;D2 causes the modem to hang up, disable auto-answer, and enter command state when DTR goes off; &amp;D3 causes the modem to reset and use the stored profile set with the &amp;Yn command.</td>
</tr>
<tr>
<td>&amp;F</td>
<td>Replaces the active profile with the factory standard profile stored in ROM.</td>
</tr>
<tr>
<td>\Vn</td>
<td>Enables or disables MNP and V.42 result codes. When Q0 is set, \V0 disables MNP and V.42 result codes and \V1 enables MNP and V.42 result codes.</td>
</tr>
<tr>
<td>\Yn</td>
<td>Switch from MNP direct mode to MNP reliable mode.</td>
</tr>
<tr>
<td>&amp;Gn</td>
<td>Selects which guard tones to generate (not used in the United States). &amp;G0 sets no guard tone; &amp;G1 sets 550Hz guard tone; &amp;G2 sets 1800Hz guard tone.</td>
</tr>
<tr>
<td>&amp;Pn</td>
<td>Sets the pulse dialing switch hook interval. &amp;P0 sets a 39%:61% make:break ratio (United States); &amp;P1 sets a 33%:67% make:break ratio (United Kingdom and Hong Kong).</td>
</tr>
<tr>
<td>&amp;Rn</td>
<td>Sets CTS state. Both &amp;R0 and &amp;R1 set CTS on.</td>
</tr>
<tr>
<td>&amp;Sn</td>
<td>Sets DSR state. Both &amp;S0 and &amp;S1 set DSR on.</td>
</tr>
<tr>
<td>&amp;Tn</td>
<td>Controls the test mode. See the Data Modem documentation for details.</td>
</tr>
<tr>
<td>&amp;Vn</td>
<td>Displays the currently active configuration profile, both stored configuration profiles, and any stored telephone numbers. This command cannot be accompanied by any other commands.</td>
</tr>
<tr>
<td>&amp;Wn</td>
<td>Stores the currently active configuration profile in one of two non-volatile memory locations (&amp;W0 or &amp;W1).</td>
</tr>
<tr>
<td>&amp;Yn</td>
<td>Sets the default configuration profile. &amp;Y0 sets the default to the profile stored by &amp;W0; &amp;Y1 sets the default to the &amp;W1 profile.</td>
</tr>
<tr>
<td>&amp;Zn=x</td>
<td>Stores the phone number, x, in the non-volatile storage location n. n is between 0 and 3.</td>
</tr>
<tr>
<td>%Cn</td>
<td>Enables of disables data compression in MNP mode. %C0 disables compression; %C1 enables MNP data compression.</td>
</tr>
<tr>
<td>\Kn</td>
<td>Sends a specific break. \K0 does not send break; \K1 empties data buffers and sends break; \K2 sends a break immediately; \K3 sends a break in sequence with data.</td>
</tr>
<tr>
<td>\Nn</td>
<td>Sets the MNP operating mode for the modem. \N0 and \N1 set direct mode; \N2 sets reliable mode; \N3 sets auto-reliable mode; \N4 sets V.42 (LAP-M) mode.</td>
</tr>
<tr>
<td>\O</td>
<td>Force modem to initiate an MNP reliable link.</td>
</tr>
<tr>
<td>\Qn</td>
<td>Sets the flow control. \Q0 disables MNP flow control; \Q1 enables XON/XOFF flow control; \Q2 enables RTS/CTS flow control.</td>
</tr>
<tr>
<td>\Un</td>
<td>Accepts an MNP reliable link.</td>
</tr>
</tbody>
</table>
In the following example, the application sends the commands to set the result codes to numeric and then dials a phone number, using tone dialing:

```c
STREAM_READ_WRITE srw;
U8 setnum[] = "ATV0\n";
U8 dialnum[] = "ATDT5551234\n";

    // Send V0 command to set status reports to numeric
    srw.numBytes = strlen(setnum);
    srw.pReadBuffer = setnum;
    s = ObjectCall(msgStreamWrite, pInst->ModemHandle, &srw);

    // Check returned status report for OK (0)
    srw.numBytes = 256;
    s = ObjectCall(msgStreamRead, pInst->ModemHandle, &srw);
    if (srw.pReadBuffer != '0') // If not OK, handle error
      goto Error1;

    // Send dial instructions to modem
    srw.numBytes = strlen(dialnum);
    srw.pReadBuffer = dialnum;
    s = ObjectCall(msgStreamWrite, pInst->ModemHandle, &srw);
```

Chapter 98 / The Transport API

This section addresses how to communicate through a local area network (LAN) with programs running on other computers.

The Transport API provides a number of features for communicating through a local area network. If you are writing an application that communicates through the network with another program, GO recommends that you read this section for background information, but encourages you to consider using the TOPS SoftTalk API, provided by Sitka Corporation. TOPS SoftTalk provides session-layer protocols, which are much easier to use.

Chapter 98 covers these topics:
- Transport concepts, including transport service types and conventions.
- Using clsTransport, including how to access a socket, sending and receiving datagrams, requesting and responding to transaction services, and binding to local transport addresses.
- Using clsTransport for AppleTalk, including AppleTalk protocol and AppleTalk name and zone protocols.

Transport Concepts

The aim of data communications is to move information from one device, through a network, to another device.

The PenPoint™ operating system transport API provides end-to-end services between the PenPoint computer and other computers. You can use Transport API services to send information to and receive information from a program running on another computer. The same services that are available to you are available to the other program. This set of common services and their encoding is called a protocol.

There are many different protocols for transporting information. The PenPoint Transport API provides a general set of capabilities, which can be used by a number of different transport protocols. Currently PenPoint supports only the AppleTalk transport protocol (ATP).

Participants in Communication

When you communicate, you exchange data with a remote server. A remote server is a program or device on another computer that can communicate over the network.

To access the network, you must access a socket, through which you send and receive transport messages. A socket is a communication endpoint. The remote
server also accesses the network through its own socket. A **connection** is an association between two sockets.

Each socket is assigned a transport address. When other programs (usually on other computers) want to establish communication with your program, they search for your socket's **transport address**. A transport address consists of identifiers for these network components:

- Network
- Node (or computer)
- Socket.

The socket identifier is also known as a **protocol port** or **port**. There are two types of ports: **well-known ports** that have pre-assigned, specific uses, and **dynamic ports** that are assigned when they are requested.

### Transport Service Types

There are two types of transport services: connection-oriented and connectionless communication.

In **connection-oriented communication**, you establish a connection with another socket, exchange data, and then break the connection. Connection-oriented communication is useful when the connection will last a long time. Currently the PenPoint Transport API does not support connection-oriented communication.

In **connectionless communication**, data is transferred from one socket to another without explicitly establishing a connection. The data transfer is accomplished at the transport layer—the layer that knows the location of each socket. This method of communication is also called **datagram delivery**. In each datagram you specify the transport address of the other socket.

There is no guarantee that the datagram will be delivered to the receiving socket; nor is there a guarantee that several datagrams will be delivered in the order that they were sent.

However, the advantage of datagram delivery is that on a fast, highly reliable medium, such as a LAN, datagrams are an efficient way of communicating information.

There are two types of datagram delivery. **At-least-once delivery** guarantees that the datagrams are delivered to their destination at least once. **Exactly-once delivery** guarantees that the datagrams are delivered to their destination exactly once.

Datagram delivery also provides transaction services. The **transaction services** send a datagram to another socket and expect the receiver to reply to it with another datagram.
Agreeing on Conventions

Communication is based on agreement. When you design and write data communication applications, you must either create conventions that both you and your communications partner agree on or use established conventions (especially if your partner was written separately and has already specified how communication is to take place).

When creating conventions, you must make these specifications:

- Communication protocol and the layer you will use.
- Type of communication (connection-oriented or connectionless).
- Service type (transaction or datagram).
- Order in which you will send or receive data, including establishing who sends first.

A good source of information on conventions for communicating information is *Computer Networks* by Tannenbaum (see Chapter 92 for publishing details).

Asynchronous Communication

When you send a clsTransport I/O message, you must wait until the message returns (the I/O operation succeeds or fails) — it’s communication is totally synchronous. If you need to communicate asynchronously, you must create another task with OSSubtaskCreate. The subtask sends the I/O message for you and returns the received data and some completion status when it is done.

You can read more about creating tasks and OSSubtaskCreate() in *Part 8: System Services*.

Using clsTransport

clsTransport provides a set of messages that allow you to communicate with remote processes. clsTransport can support a variety of transport protocols. This means that you can potentially connect the PenPoint computer to a variety of different networks. Currently AppleTalk (ATP/DDP) is the only protocol supported by GO.

The clsTransport messages are defined in the file TPH. Table 98-1 lists the clsTransport messages.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgTPBind</td>
<td>P_TP_BIND</td>
<td>Binds a socket to an address.</td>
</tr>
<tr>
<td>msgTPRecvFrom</td>
<td>P_TP_RECVFROM</td>
<td>Receives a datagram.</td>
</tr>
<tr>
<td>msgTPSendTo</td>
<td>P_TP_SENDTO</td>
<td>Sends a datagram.</td>
</tr>
<tr>
<td>msgTPSendRecvTo</td>
<td>P_TP_SENDRECVTO</td>
<td>Sends a request, wait for a response.</td>
</tr>
</tbody>
</table>
Accessing a Socket

Each socket is a service instance maintained by the `theTransportHandlers` service manager. When the user configures ATP, the Connections notebook creates a service instance for a socket.

To establish transport-level communication with another socket, you must locate, bind to, and open a local socket by:

- Sending `msgIMFind` to `theTransportHandlers` to locate a socket service handle.
- Sending `msgSMBind` to the socket service handle.
- Sending `msgSMOpenDefaults` to the socket service handle.
- Sending `msgSMOpen` to the socket service handle, which passes back the UID for the socket.

Both `msgSMOpenDefaults` and `msgSMOpen` take a standard `SM_OPEN_CLOSE` structure. `pArgs` in `SM_OPEN_CLOSE` contains a pointer to a `TP_NEW` structure that contains an `OSO_NEW` structure (oso) and a `TP_NEW_ONLY` structure (tp). The `OSO_NEW` (open service object) structure is used internally by `clsTransport`. For more information on open service objects, see Part 13: Writing PenPoint Services. The `TP_NEW_ONLY` structure contains:

- `pArgs.tp.service` A `TP_SERVICE` value that specifies the service type. The possible service types are:
  - `tpReliableService`
  - `tpDatagramService`
  - `tpTransactionService`

When the message completes successfully, `clsTransport` passes back the UID of the service in the `service` field of the `SM_OPEN_CLOSE` structure as in the following example:

For example:

```pascal
STATUS PASCAL OpenTransportHandle(
    P_TP_NEW       pTPnew,
    P_OBJECT       pTPhandle,
    OBJECT         self)
{
    SM_OPEN_CLOSE   smOpen;
    STATUS          s;
    smOpen.caller = self;
    smOpen.handle = targetServiceHandle;
    smOpen.pArgs = ( P_ARGS ) pTPnew;
    s = ObjectCall( msgSMOpenDefaults, theTransportHandlers, &smOpen );
    s = ObjectCall( msgSMOpen, theTransportHandlers, &smOpen );
    *pTPhandle = smOpen.service;
    return( stsOK );
}
```
**Closing a Socket Handle**

To close a socket handle, send `msgSMClose` to `theTransportHanders`. You must specify both the handle of the ATP service and the UID of the service that you received from `msgSMOpen`, as shown in the following example:

```pascal
STATUS PASCAL CloseTransportHandle(
    OBJECT tpHandle,
    OBJECT self
)
{
    SM_OPEN_CLOSE smClose;
    STATUS s;

    smClose.caller = self;
    smClose.handle = targetServiceHandle;
    smClose.service = tpHandle;
    smClose.pArgs = pNull;

    Debugf("Closing socket handle %lx. pArgs=%lx, %lx", tpHandle, &smClose, smClose.pArgs);

    s = ObjectCall( msgSMClose, theTransportHandlers, &smClose );
    if ( s != stsOK ) {
        //Debugf( "msgSMClose failed %lx.", s );
        return ( s );
    }

    //Debugf( "Closed socket handle." );

    return ( stsOK );
}
```

**Sending Datagrams**

To send a datagram, send `msgTPSendTo` to the socket handle. The message takes a `TP_SENDTO` structure that specifies:

- **pBuffer** A pointer to the buffer of data. A datagram can contain up to 586 bytes of data.
- **count** The length of the data.
- **pOptions** A pointer to a transport options block. This block varies, depending on the transport protocol. The AppleTalk options are described in “Using clsTransport for AppleTalk.”
- **pAddress** A pointer to a buffer that contains the transport address of the destination socket.

When the message completes successfully, it returns `stsOK`. If you attempt to send more than 586 bytes you will receive the status message `stsTPlength` from `ObjectCall`.

---

98.2.2

---

98.2.3
Receiving Datagrams

When you are ready to receive a datagram, send msgTPRecvFrom to the socket handle. The message takes a TP_RECVFROM structure that contains:

- **pBuffer** A pointer to the buffer that will receive the data.
- **length** The size of the buffer.
- **count** A location that receives the actual number of bytes written to pBuffer.
- **pAddress** A pointer to the buffer that receives the transport address of the sending socket.
- **pOptions** A pointer to a transport options block. This block varies depending on the transport protocol. The AppleTalk options are described below “Using clsTransport for AppleTalk.”

When the message completes successfully, it returns stsOK and sends back the TP_RECVFROM structure in which:

- The buffer indicated by pBuffer contains the data received.
- count contains the length of data in pBuffer.
- The buffer indicated by pAddress contains the transport address of the socket that sent the data.

Requesting a Transaction Service

Frequently you will send a datagram to a socket and expect the socket to send you a datagram in return. You could use both msgTPSendTo and msgTPRecvFrom messages, or you can use msgTPSendRecvTo. To send msgTPSendRecvTo, you must specify tpTransactionService in the service field of TP_NEW when you create your socket handle.

msgTPSendRecvTo takes a TP_SENDRECVTO structure that contains:

- **pSendBuffer** A pointer to the buffer of data to send.
- **sendCount** The length of the data be sent.
- **pRecvBuffer** A pointer to the buffer that receives data.
- **recvLength** The size of the receive buffer.
- **recvCount** A location that receives the actual number of bytes written to pRecvBuffer.
- **pAddress** A pointer to a buffer that contains the transport address of the destination socket.
- **pOptions** A pointer to a transport options block. This block varies, depending on the transport protocol. The AppleTalk options are described below “Using clsTransport for AppleTalk.”

When the message completes successfully, it returns stsOK and sends back the TP_SENDRECVTO structure in which:

- The buffer indicated by pRecvBuffer contains the data received.
- recvCount contains the length of data in pBuffer.
Responding to a Transaction Service

When you use transaction services, both your application and the server must agree on certain conventions. Among these conventions are the types of communication (connection-oriented or connectionless) and the service type (transaction or datagram). If you have agreed to transaction services and you receive an unsolicited message from the remote server, then it is implied that you must respond to the message.

Binding to a Local Transport Address

Before a server can locate your socket, you must bind your socket to a local transport address.

To bind your socket, send msgTPBind to the socket handle. The message requires a TP_BIND structure that contains a pointer to the buffer that receives the address (pAddress).

Using clsTransport for AppleTalk

This section describes how to use clsTransport to communicate on an AppleTalk network. If you need information on AppleTalk concepts and protocols, see Inside AppleTalk by Gursharan S. Sidhu (see Chapter 92 for publishing details).

Table 98-2 lists the messages that are defined in the file ATALK.H.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgATPRespPktSize</td>
<td>P_ATP_RESPPKTSIZE</td>
<td>Sets the size of the response packets.</td>
</tr>
<tr>
<td>msgNBPRegister</td>
<td>P_NBP_REGISTER</td>
<td>Registers a name with NBP.</td>
</tr>
<tr>
<td>msgNBPRemove</td>
<td>P_NBP_REMOVE</td>
<td>Removes a name from NBP.</td>
</tr>
<tr>
<td>msgNBPLookup</td>
<td>P_NBP_LOOKUP</td>
<td>Looks up a name in NBP.</td>
</tr>
<tr>
<td>msgNBPConfirm</td>
<td>P_NBP_CONFIRM</td>
<td>Confirms address and name.</td>
</tr>
<tr>
<td>msgZIPGetZoneList</td>
<td>P_ZIP_GETZONES</td>
<td>Gets list of zones.</td>
</tr>
<tr>
<td>msgZIPGetMyZone</td>
<td>P_ZIP_GETZONES</td>
<td>Gets my zone name.</td>
</tr>
</tbody>
</table>

Using the AppleTalk Protocol

AppleTalk Protocol Options

One of the arguments to the transport API messages msgTPSendTo, msgTPRecvFrom, and msgTPSendRecvTo is a pointer to a block of protocol options. When the Transport API builds its transport packets, it uses these options.

Protocol options for AppleTalk are contained in the ATP_OPTIONS structure (defined in ATALK.H). The AppleTalk options contained in the structure are:

- **ddpType** A specifier for the type of ddp traffic.
- **flags** A set of flags that specify ATP options. The flags are defined in ATP_FLAGS and specify:
ATP_XO_Flag  Whether datagrams should be sent exactly once or at least once.

ATP_Checksum_Flag  Whether datagrams should be sent with a checksum. Checksums are used only for messages that will be sent through a bridge or a router (internet messages).

ATP_ALONoResponse_Flag  Tells ATP not to expect a response to the datagram.

transactionID  A transaction ID. When you reply to a datagram, you use the transaction ID to specify which datagram you are replying to.

interval  The interval between send retries.

retries  The number of retries to attempt. If the transport API exceeds the number of retries, it returns stsTPfailed in the message's completion code argument.

minRespPackets  The minimum number of response packets that the transport API must use to send a response. This number can be in the range 0 through 8. Usually minRespPackets indicates the number of sets of userBytes in this array.

userBytes  An array of up to eight sets of user bytes. Each set of user bytes is four bytes long. userBytes are used for communicating additional, encoded information.

Changing the Size of ATP Packets

For each AppleTalk request, a server can return up to eight AppleTalk Protocol (ATP) packets.

When you use AppleTalk for transaction services, the AppleTalk protocol specifies that the Transport API can create only one ATP packet for each request. Usually the ATP packets contain 578 bytes of data (586-byte datagram, minus an 8-byte header). Responses can contain up to eight ATP packets, or 4624 (578 * 8) bytes of data.

Some protocols, such as PAP (the printer access protocol), require a smaller response packet size.

You can change the size of ATP packets, by sending msgATPRespPktSize to the socket. The message takes a pointer to an ATP_RESPPKTSIZE structure that contains a single element, a U16 value that specifies the new size for ATP packets (size). The maximum size for an ATP packet is 578 bytes.

AppleTalk Name and Zone Protocols

AppleTalk defines messages that you can use to manipulate the names of the sockets on the network. The name binding protocol (NBP) messages allow you to search for, add, and delete names from the NBP tables.

Each station maintains its own NBP table. When you add a name, NBP adds it to your table. When you search for a name, NBP searches all NBP tables in the network.
NBP messages can accept wildcard characters, but they are valid only within a user-specified zone. A network is divided into two or more zones when it is joined by a bridge. A bridge is a computer or other smart device that can link two networks and route traffic from one zone to another.

The zone information protocol (ZIP) messages allow you to get a list of zone names, or get the name of your zone.

The following sections describe how to use NBP and ZIP messages. If you need to cancel any of these NBP requests, send msgNBPCancel to the socket.

Registering a Name

When you create a server on the network, you will want others to know where to find it. When your server starts, create a socket and send msgNBPRegister to it. The message requires an NBP_REGISTER structure that contains a pointer to the buffer that contains the name (pName).

The names have the format:

\[ \text{name:object@zone} \]

where:

- **name** Is the name of the actual server.
- **object** Is the object type (such as TOPSServer, LaserWriter, and so on).
- **zone** Is the name of the zone.

If you register a name that is already bound to a local transport address, the Transport API stores both the name and address in the NBP table. If the name is not bound to a local transport address, the Transport API creates a transport address and binds it to the name, then stores them in the NBP table.

Removing a Name

When your server is removed, you must remove its name from NBP by sending msgNBPRemove to the socket. The message requires an NBP_REMOVE structure that contains a pointer to the buffer that contains the name to delete (pName).

Looking for a Name

To look for a server name, send msgNBPLookup to the socket. The message requires a NBP_LOOKUP structure that contains:

\[
\begin{align*}
\text{pName} & \quad \text{A pointer to a buffer containing the name of the server you want to find. The name can contain wildcard characters, described at the end of this list.} \\
\text{pBuffer} & \quad \text{A pointer to the buffer that receives the names that match pName.} \\
\text{length} & \quad \text{The number of bytes in pBuffer.} \\
\text{numMatches} & \quad \text{A location that receives the number of matches.}
\end{align*}
\]

The server name can contain the equal sign (=) as a wildcard in any field. An asterisk (*) in the zone field means the current zone. For example, \[ =:Printer@Marketing \] would
find all printers in the zone Marketing; MyServe:TOP2Server@ would find all TOPS servers with the name MyServe in all zones.

If more names match pName than will fit in the buffer, the message stores as many as it can in the buffer and returns stsTPNoRoom.

You cannot search for a server registered on a PenPoint computer from that same PenPoint computer.

Confirming an Address

Because the user might disconnect the PenPoint computer from the network and reconnect it at a later time, your application might want to confirm that a server is still associated with the same transport address. If your application hasn't communicated with a server for a long time, it might also be a good idea to confirm the address.

To confirm the network address associated with a name send msgNBPConfirm to the socket handle. The message requires an NBP_CONFIRM structure that specifies pointers to two buffers containing:

- pName The name
- pAddress The address.

Listing the Zone Names

To get a list of the zone names, send msgZIPGetZoneList to the socket. The message takes a pointer to a ZIP_GETZONES structure that contains:

- pBuffer A pointer to the buffer that will receive the list of zone names. You must allocate the buffer.
- length A U16 value that specifies the length of the buffer. If length is less than the space required for the list of zone names, the message passes back only those zone names that will fit in the available space (it does not return partial zone names). If length is less than the first zone name, it will not pass back any zone names.

When the message completes successfully, it returns stsOK and stores the zone names in pBuffer. Each zone name is stored as a Pascal string (that is, a length byte followed by that number of characters).

The message also passes back:

- length The total size of all zone names passed back in pBuffer.
- numZones A U16 value that contains the number of zone names passed back in pBuffer.

Getting Your Zone Name

To get the name of your own zone, send msgZIPGetMyZone to your socket. The message takes a pointer to a ZIP_GETZONES structure, as described in the preceding section.
Chapter 99 / In Box and Out Box

This chapter describes the In box and Out box services, how to insert or retrieve documents from the queues, and how you subclass the service classes for the In box and Out box (to create queues for new types of output devices).

Chapter 99 covers these topics:
- General device concepts.
- Out box service concepts.
- In box service concepts.
- In box and Out box service messages.

Introduction to the In Box and Out Box

One of the unique features of the PenPoint™ operating system is its capability for deferred input and output. Deferred input and output means that the user doesn’t have to connect the PenPoint computer to an input or output device before beginning an input or output operation. The actual input or output process is deferred until the input or output device becomes ready.

For example, if a printer is not connected when the user prints a document, PenPoint places the document into an output queue associated with the printer. When a printer is connected, PenPoint prints documents that have been queued for that printer.

The PenPoint user interface presents the deferred input and output queues as sections within the In box and Out box notebooks. The user can open the notebook and look at a particular section to see documents in the queue.

In PenPoint, deferred input operations are handled by a special class of services known as In box services; deferred output operations are handled by Out box services. Input services are subclasses of clsINBXService; output services are subclasses of clsOBXService. Both clsINBXService and clsOBXService inherit from clsIOBXService. clsIOBXService creates the service sections and provides the queuing mechanism for the In box and Out box.

All three service classes, clsINBXService, clsOBXService, and clsIOBXService define many messages that must be implemented by subclasses.

If you are writing a service for deferred input or output, you must subclass one of these three services and implement methods to handle their input or output messages.
Before reading this section, you should be familiar with PenPoint services, which are described in Chapter 94. If you are going to write an In box or Out box service, you must also read Part 13: Writing PenPoint Services.

**General Device Concepts**

The PenPoint operating system expects that the PenPoint computer will not always be attached to most input and output devices. Therefore it must:

- Defer output until the output services are available.
- Gather input from input services to a central location when the services become available.

**Service Sections**

When the user creates an instance of an In box or Out box service class, the service class creates a corresponding section in the In box or Out box.

The service sections are different from normal sections because:

- Service sections must restrict what the user can do to documents in the service section. Usually the user cannot move or copy documents between service sections (because each service section can support a different type of document), nor can the user move documents directly into a service section.
- Each service section is bound to an instance of a specific service. For example, if the user has installed two printers named First Floor and Second Floor, there are two corresponding Out box sections, also named First Floor and Second Floor. The First Floor service section is bound to an instance of the printer service named First Floor; the Second Floor service section is bound to an instance of the printer service named Second Floor.
- Each Out box service section is related to the queue for the service to which it is bound. The user can change ordering of documents in a queue by moving the documents within the service section.

**Services and Devices**

Chapter 93, Concepts and Terminology, described how services, such as logical and physical device drivers enabled applications to communicate directly with devices. Chapter 94, Using Services, described how applications acquire access to particular services through the service managers.

Usually a PenPoint computer is not attached to any peripheral devices, such as printers, LAN servers, and the like. For this reason, PenPoint must be able to defer input and output operations until the user connects the appropriate peripheral device.

Each service section in the In box and Out box is associated with an instance of a service class. Each service instance is associated with a specific device—possible through a series of logical device drivers.
For example, the Out box service section for “Marketing Printer” might be associated with an instance of the HP LaserJet printer service. That instance of the LaserJet printer service owns and is connected to the parallel port service instance that controls a parallel port in the PenPoint computer.

As another example, the In box service section for fax might be associated with an instance of a fax receiver. The fax receiver owns and is connected to the serial port service instance that controls a serial port in the PenPoint computer.

**Installing Devices and Services**

Installing In box and Out box services is no different from installing any other service. Once the service is installed, the service must provide its own user interface by which the user creates new instances of the service. (Although users create new instances of printer services through the Connections notebook, there is no plan to add an extensible interface to the Connections notebook.)

When creating the service instance, the user:

- Names the specific device (such as “Accounting Printer: 4th Floor”).
- Specifies the type of device (such as HP LaserJet II).
- Specifies the communication port used by the service (such as the parallel port).

**Targeting Communications Devices**

PenPoint communicates with its peripheral devices through its communication devices (such as serial ports, parallel ports, data or fax modems, and LAN servers). The software that controls each of these communication devices is implemented as a PenPoint service.

As mentioned before, deferred input and output operations are handled by In box and Out box services. An In box or Out box service targets a communication service so that it will be notified whenever the physical communication device becomes connected or disconnected. See Chapter 94 for more information on targeting services.

**Enabling and Disabling Services**

An Out box service must be enabled before its output process can begin. This enabled state is represented by a checkbox in the Enabled column of the Out box notebook. Typically, a communications device permits only exclusive access. If multiple Out box services are connected to the same output device, only one can be enabled at a time. Enabling an Out box service causes it to become the owner of its target service. The service remains enabled until either:

- The user disables it (by unchecking the Enabled box).
- The service willingly releases ownership of the communications device so that another service can become the new owner.
Service ownership is discussed in more detail in Chapter 94.

Enabling or disabling an In box or Out box service also provides a convenient mechanism for managing communications devices that can't automatically determine when they become connected or disconnected. Because these devices cannot inform the Out box service when they are connected or disconnected, their status will always remain connected, regardless of the connection status of the physical device. Such services can be explicitly disabled to prevent documents from being sent to a device that is not ready for output.

**Out Box Concepts**

Each instance of an Out box service has a corresponding section in the system Out box notebook. The name of the service and the name of the section are the same. For example, the user can create two instances of a printer Out box service class. The instances are named “Engineering Printer” and “Marketing Printer.” Each instance of the printer Out box service class has its own output queue. These output queues appear as sections in the Out box named “Engineering Printer” and “Marketing Printer.”

Documents are placed in the Out box service sections by an output service manager. There are two output service managers, thePrintManager and theSendManager. thePrintManager is responsible for placing documents in the appropriate printer service sections. theSendManager supports the other service sections, including fax and e-mail. If you write a new output service, you can integrate it into theSendManager.

**Out Box Operation**

When the user gives a Print or Send command, PenPoint invokes the appropriate output manager (thePrintManager or theSendManager). These output managers are provided by GO, but have no API.

The output manager responds by performing these tasks:

- Prompts the user for the destination device.
- Prompts the user for the options that are related to that output manager.
- Adds the document to the queue for the device.

For thePrintManager, the output service is a printer service that is associated with a destination printer.

For theSendManager, the output service provides e-mail, fax, deferred file transfers, and so on. theSendManager is an open-ended mechanism, which allows it to handle any services that are installed and active.

**Out Box Protocol Messages**

The primary function of an Out box service is to manage the output queue for each service instance. This function is implemented by a standard Out box protocol consisting of eight interrelated messages.
The client of an Out box service sends msgOBXSvcMoveInDoc or msgOBXSvcCopyInDoc to the Out box service instance, telling the Out box to add an existing PenPoint document to its output queue.

Once a document is added to the Out box, msgOBXSvcPollDocuments informs an Out box service that it should check to see if conditions are right to start an output process. Other events may also cause the Out box service or the client to send msgOBXSvcPollDocument to the Out box service. For example, an Out box service will selfsend this message when the service has just been enabled.

If the service is enabled and the output device is connected, the service sends msgOBXSvcNextDocument to self to locate the next document ready for output.

If a document exists in the output queue but is not ready for output, the service selfsends msgOBXSvcScheduleDocument to reschedule output at a later time.

If a document is ready for output, the service will lock the document with msgOBXSvcLockDocument, and kick off the output process with msgOBXSvcOutputStart.

At the end of the output process, the document being sent will send msgOBXDocOutputDone to the Out box service.

Finally, if the output finished normally, the service selfsends msgOBXSvcPollDocuments again to see if anything else is ready for output.

If the output didn't finish normally, the service selfsends msgOBXSvcUnlockDocument to restore the document to its preoutput state.

Documents in the Out Box

The primary focus of an Out box service is to manage its output queue. An output queue is essentially a collection of documents located in an Out box section. The primary focus of a document in the Out box is to manage a single output job.

An Out box document can be any PenPoint document, that is, an instance of an application inheriting from clsApp. The document can be created, opened, and closed just like a regular page in the notebook. There are two ways to implement Out box documents:

- The application that created the document knows how to respond to Out box messages. For example, an electronic mail application might also respond to Out box messages so that it can send its own documents to an electronic mail service. The document in the Out box would contain destination information for the document.
The Out box document is a wrapper, which contains (embeds) the document being output. The wrapper document responds to the Out box messages and other PenPoint Application Framework messages. In response to these messages, the wrapper document controls how the embedded document is output. Printing in PenPoint is implemented in this way.

An Out box document is also responsible for interacting with the Out box service and controlling the output process, such as sending out an electronic mail message through a communication device. Thus, in addition to responding to clsApp messages, an Out box document also understands the following clsOBXService messages:

- msgOBXDocOutputStartOK
- msgOBXDocOutputStart
- msgOBXDocOutputCancel
- msgOBXDocOutputDone
- msgOBXDocStatusChanged

**Writing Your Own Out Box Service**

clsOBXService is an abstract class. If you are writing an Out box service, you must create a subclass of clsOBXService. (clsOBXService manages the output queue only, it does not actually cause the output to happen.) Typically, your Out box service inherits its output queue management behavior from clsOBXService. You must add any servicespecific behaviors for the communication protocol or devices you need to handle.

By default, clsOBXService provides a simple first-in-first-out queue. If your Out box service requires sophisticated scheduling algorithms, you must replace the default behaviors with your own. In this case, your service might need to handle these messages:

- msgOBXSvcMoveInDoc
- msgOBXSvcCopyInDoc
- msgOBXSvcNextDocument
- msgOBXSvcLockDocument
- msgOBXSvcUnlockDocument
- msgOBXSvcScheduleDocument
- msgOBXSvcOutputStart

Another example would be msgOBXSvcLockDocument and msgOBXSvcUnlockDocument. Their default behavior is to mark the document so that gestures over the document icon will not be processed while output is in progress (in fact they cause an error note to appear). A msgOBXSvcUnlockDocument typically indicates that the output has been aborted for some reason. You may wish to add to the default behavior, such as notifying your observers that some error has just occurred.
Working with Existing Out Box Services

All output operations should be performed through an Out box service in order to take advantage of PenPoint's deferred output feature. An application or a service can bypass the Out box protocol only if the output device is always present or is rarely detached from the PenPoint computer.

The key to working with an existing Out box service is to conceptually break up the output process into two distinct phases:

- The first phase is either adding an existing PenPoint document to the output queue, or creating a special document of some sort in temporary storage and then move it into the output queue.
- The second phase is the actual output process, during which a devicespecific data stream is sent out through some communication device.

clsOBXService provides a framework for managing the transition from one phase to another.

The separation of these two phases of output operation has an additional benefit. In many cases, an application developer can avoid writing a new Out box service in order to handle application-specific output functions. It is often sufficient to handle only one of the two phases of the output operation. There are several options:

- One inexpensive solution is to have the application export the data into a format that is easier to output under an existing Out box service. For example, a database document can generate a report as an ASCII file or a word processor document and move it into a printer, fax or e-mail Out box section. Similarly, a spreadsheet document can export its pie chart into a popular drawing program document and move it to the Out box for output.
- Another approach is to allow the database or spreadsheet document itself to be moved or copied into the output queue. When the document receives msgOBXSvcOutputStart, it knows that the output device is ready. It then proceeds to perform the output operation the old-fashioned way. Such applications already have sophisticated output capabilities, and we only need to ensure not to start the output process until the device is ready. The obvious disadvantage of this approach is that it requires additional memory if we have to make a copy of the document in order to put it into the Out box.
- A third approach represents a compromise between the two. During the first phase of the output operation, a "surrogate" document, rather than the real one, is copied into the output queue. This surrogate document not only understands the Out box output protocol, but also knows how to communicate with the original document. It is effectively a "pointer" back to the original document. When the output process begins, the surrogate document communicates with the original one to cause the device-specific data stream to be sent to the correct output port.
Services that Handle Input or Output

clsOBXService deals only with output operations; clsINBXService deals only with input operations. If a service wants to handle both input and output (for example, an electronic mail service), it can use clsIOBXService, which is another abstract class (and the ancestor of both clsINBXService and clsOBXService). clsIOBXService associates the service with both an input queue and an output queue. The service, the In box section, and the Out box section all have the same name.

In Box Concepts

Although the In box contains service sections that are associated with queues and services, the similarity between the In box and Out box stops there. The Out box must schedule documents for output when the communication devices become available; the output service drives the output.

The In box on the other hand must handle incoming data and convert it into a PenPoint document, whenever possible.

Some services convert incoming data streams into PenPoint documents. Other services do not know what type of data they are receiving and must create a data file.

Passive and Active In Box Services

There are two type of In box services: passive and active. A passive In box service waits for an input event to happen. An active In box service initiates the communication process. For example, a fax input service may wish to periodically poll a store-and-forward facility (to receive a fax image).

Typically, when a user enables a passive In box service, the service becomes the owner of its communication device. While the In box service owns the I/O device, no other services can transmit or receive data through the same device. A simple fax In box service, for example, becomes the owner of the fax modem (and the serial port) and sets it up to start receiving fax images whenever a phone call comes in.

Some In box services may want to actively solicit input from a remote agent. For example, a service that queries a remote database will have to establish the communication link between the PenPoint computer and the remote database server. For these active In box services, clsINBXService provides default behaviors to manage three things:

- The state of the device (connected or disconnected).
- The protocol to initiate input operation (whether the service is enabled or disabled).
- Automatic polling behavior similar to that of an Out box service.

Thus, the user can defer the input operation until it becomes possible to establish a communication link with a remote agent. Note, however, that to enable such behavior for an In box service, the polling flag (iobxsvc.in.autoPoll) must be true when the service is created.
After a remote agent initiates the input operation, the In box service detects the input event and then receives the incoming data stream.

**In Box Documents**

Normally, an input event results in a PenPoint document being created in an In box section. For example, a fax In box section can create a document containing the fax images received by the fax modem. Such documents are normal PenPoint documents. Their contents have nothing to do with the input device or where the document came from.

Sometimes an In box document contains not only data, but also some control information about the input operation to be performed. For example, the user may construct a specific query statement for an online database and put it into the appropriate In box section before the PenPoint machine is connected to the remote database. When the input service becomes ready, the query statement is sent to the remote database, and the result is put into either another document or the same document containing the query statements. This type of In box document is very similar to the Out box document that controls the actual output operation.

Note that the deferred I/O protocol implemented by clsINBXService assumes that an input operation is controlled by an In box document. This assumption may be too cumbersome and confusing for many services. If this is the case, an In box service can simply store the input control information (such as a database query statement) with the service itself. When the service receives msgINBXSvcPollDocuments, it simply handles the input operation directly and bypasses the rest of the protocol.

**In Box and Out Box Service Messages**

The following three tables list the messages defined by clsOBXService, clsINBXService, and clsIOBXService. For more detailed description of these messages, see Part 10: Connectivity in the PenPoint API Reference, or read the header files (OBXSVC.H, INBXSVC.H, and IOBXSVC.H).
### clsOBXSERVICE Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgOBXSvcMoveInDoc</td>
<td>P_OBXSVCS_MOVE_COPY_DOC</td>
<td>Move a document into the Out box section.</td>
</tr>
<tr>
<td>msgOBXSvcCopyInDoc</td>
<td>P_OBXSVCS_MOVE_COPY_DOC</td>
<td>Copy a document into the Out box section.</td>
</tr>
<tr>
<td>msgOBXSvcPollDocuments</td>
<td>nothing</td>
<td>Poll all documents in an output queue and output those that are ready.</td>
</tr>
<tr>
<td>msgOBXSvcNextDocument</td>
<td>P_OBXSVCS_DOCUMENT</td>
<td>Pass back the next document ready for output.</td>
</tr>
<tr>
<td>msgOBXSvcScheduleDocument</td>
<td>P_OBXSVCS_DOCUMENT</td>
<td>Schedule a document that is not ready for output.</td>
</tr>
<tr>
<td>msgOBXSvcLockDocument</td>
<td>P_OBXSVCS_DOCUMENT</td>
<td>Lock the document in preparation for output.</td>
</tr>
<tr>
<td>msgOBXSvcUnlockDocument</td>
<td>P_OBXSVCS_DOCUMENT</td>
<td>Unlock a document that was previously locked.</td>
</tr>
<tr>
<td>msgOBXSvcOutputStart</td>
<td>P_OBXSVCS_DOCUMENT</td>
<td>Start the output process for a document in the output queue.</td>
</tr>
<tr>
<td>msgOBXDocOutputDone</td>
<td>P_OBX_DOC_OUTPUT_DONE</td>
<td>Tell the Out box service that output is finished.</td>
</tr>
<tr>
<td>msgOBXSvcSwitchIcon</td>
<td>nothing</td>
<td>Toggle the Out box icon (to empty or filled) if necessary.</td>
</tr>
<tr>
<td>msgOBXSvcGetTempDir</td>
<td>P_OBJECT</td>
<td>Pass back a handle for a temporary directory.</td>
</tr>
<tr>
<td>msgOBXSvcOutputCancel</td>
<td>nothing</td>
<td>Cancel the output process.</td>
</tr>
<tr>
<td>msgOBXSvcOutputCleanUp</td>
<td>P_OBX_DOC_OUTPUT_DONE</td>
<td>Clean up after the current output is done.</td>
</tr>
<tr>
<td>msgOBXSvcStateChanged</td>
<td>OBJECT</td>
<td>Tell observers that the service state just changed.</td>
</tr>
<tr>
<td>msgOBXSvcQueryState</td>
<td>P_OBXSVCS_QUERY_STATE</td>
<td>Pass back the state of the service.</td>
</tr>
<tr>
<td>msgOBXSvcGetEnabled</td>
<td>P_BOOLEAN</td>
<td>Get the enabled state of the service.</td>
</tr>
<tr>
<td>msgOBXSvcSetEnabled</td>
<td>BOOLEAN</td>
<td>Set the enabled state of the service.</td>
</tr>
<tr>
<td>msgOBXDocGetService</td>
<td>P_OBX_DOC_GET_SERVICE</td>
<td>Get the service name.</td>
</tr>
<tr>
<td>msgOBXDocInOutbox</td>
<td>P_OBX_DOC_IN_OUTBOX</td>
<td>Check if a document is in a section in the Out box.</td>
</tr>
<tr>
<td>msgOBXDocOutputStartOK</td>
<td>nothing</td>
<td>Ask the Out box document if it is OK to start the output process.</td>
</tr>
<tr>
<td>msgOBXDocOutputStart</td>
<td>nothing</td>
<td>Tell an Out box document to start the output process.</td>
</tr>
<tr>
<td>msgOBXDocOutputCancel</td>
<td>nothing</td>
<td>Tell an Out box document to cancel the output process.</td>
</tr>
<tr>
<td>msgOBXDocStatusChanged</td>
<td>P_OBX_DOC_STATUS_CHANGED</td>
<td>Tell the Out box service that the document status is changed.</td>
</tr>
</tbody>
</table>
## In Box and Out Box Service Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgINBXSvcSwitchIcon</td>
<td>nothing</td>
<td>Toggle the In box icon (to empty or filled) if necessary.</td>
</tr>
<tr>
<td>msgINBXDocGetService</td>
<td>P_INBX_DOC_GET_SERVICE</td>
<td>Get the service name.</td>
</tr>
<tr>
<td>msgINBXDocInInbox</td>
<td>P_INBX_DOC_IN_INBOX</td>
<td>Check if a document is in a section in the In box.</td>
</tr>
<tr>
<td>msgINBXSvcMoveInDoc</td>
<td>P_INBXSvc_MOVE_COPY_DOC</td>
<td>Move a document into the In box section.</td>
</tr>
<tr>
<td>msgINBXSvcCopyInDoc</td>
<td>P_INBXSvc_MOVE_COPY_DOC</td>
<td>Copy a document into the In box section.</td>
</tr>
<tr>
<td>msgINBXSvcGetTempDir</td>
<td>OBJECT</td>
<td>Pass back a handle for a temporary directory.</td>
</tr>
<tr>
<td>msgINBXSvcPollDocuments</td>
<td>nothing</td>
<td>Poll all documents in an input queue and input those that are ready.</td>
</tr>
<tr>
<td>msgINBXSvcNextDocument</td>
<td>P_INBXSvc_DOCUMENT</td>
<td>Pass back the next document ready for input.</td>
</tr>
<tr>
<td>msgINBXSvcLockDocument</td>
<td>P_INBXSvc_DOCUMENT</td>
<td>Lock the document in preparation for input.</td>
</tr>
<tr>
<td>msgINBXSvcUnlockDocument</td>
<td>P_INBXSvc_DOCUMENT</td>
<td>Unlock a document that was previously locked.</td>
</tr>
<tr>
<td>msgINBXSvcScheduleDocument</td>
<td>P_INBXSvc_DOCUMENT</td>
<td>Schedule a document that is not ready for input.</td>
</tr>
<tr>
<td>msgINBXSvcInputStart</td>
<td>P_INBXSvc_DOCUMENT</td>
<td>Start the input process for a document in the input queue.</td>
</tr>
<tr>
<td>msgINBXSvcInputCancel</td>
<td>nothing</td>
<td>Cancel the input process.</td>
</tr>
<tr>
<td>msgINBXSvcInputCleanUp</td>
<td>P_INBX_DOC_INPUT_DONE</td>
<td>Clean up after the current input is done.</td>
</tr>
<tr>
<td>msgINBXSvcStateChanged</td>
<td>OBJECT</td>
<td>Tell observers that the service state just changed.</td>
</tr>
<tr>
<td>msgINBXSvcQueryState</td>
<td>P_INBXSvc_QUERY_STATE</td>
<td>Pass back the state of the service.</td>
</tr>
<tr>
<td>msgINBXSvcGetEnabled</td>
<td>P_BOOLEAN</td>
<td>Get the enabled state of the service.</td>
</tr>
<tr>
<td>msgINBXSvcSetEnabled</td>
<td>BOOLEAN</td>
<td>Set the enabled state of the service.</td>
</tr>
<tr>
<td>msgINBXDocInputStartOK</td>
<td>nothing</td>
<td>Ask the In box document if it is OK to start the input process.</td>
</tr>
<tr>
<td>msgINBXDocInputStart</td>
<td>nothing</td>
<td>Tell an In box document to start the input process.</td>
</tr>
<tr>
<td>msgINBXDocInputDone</td>
<td>P_INBX_DOC_INPUT_DONE</td>
<td>Tell the In box service that input is finished.</td>
</tr>
<tr>
<td>msgINBXDocInputCancel</td>
<td>nothing</td>
<td>Tell an In box document to cancel the input process.</td>
</tr>
<tr>
<td>msgINBXDocStatusChanged</td>
<td>P_INBX_DOC_STATUS_CHANGED</td>
<td>Tell the In box service that the document status is changed.</td>
</tr>
<tr>
<td>Message</td>
<td>Takes</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>--------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>msgIOBXSvcSwitchIcon</td>
<td>nothing</td>
<td>Toggle the In box or Out box icon (to empty or filled) if necessary.</td>
</tr>
<tr>
<td>msgIOBXDocGetService</td>
<td>P_IOBX_DOC_GET_SERVICE</td>
<td>Get the service name.</td>
</tr>
<tr>
<td>msgIOBXDocInIOBox</td>
<td>P_IOBX_DOC_IN_IOBOX</td>
<td>Check if a document is in a section in the In box or Out box notebook.</td>
</tr>
<tr>
<td>msgIOBXSvcMoveInDoc</td>
<td>P_IOBXSVC_MOVE_COPY_DOC</td>
<td>Move a document into the Out box section.</td>
</tr>
<tr>
<td>msgIOBXSvcCopyInDoc</td>
<td>P_IOBXSVC_MOVE_COPY_DOC</td>
<td>Copy a document into the In box or Out box section.</td>
</tr>
<tr>
<td>msgIOBXSvcGetTempDir</td>
<td>P_OBJECT</td>
<td>Pass back a handle for a temporary directory.</td>
</tr>
<tr>
<td>msgIOBXSvcPollDocuments</td>
<td>nothing</td>
<td>Poll all documents waiting for input/output.</td>
</tr>
<tr>
<td>msgIOBXSvcNextDocument</td>
<td>P_IOBXSVC_DOCUMENT</td>
<td>Pass back the next document ready for input/output.</td>
</tr>
<tr>
<td>msgIOBXSvcLockDocument</td>
<td>P_IOBXSVC_DOCUMENT</td>
<td>Lock the document in preparation for input/output.</td>
</tr>
<tr>
<td>msgIOBXSvcUnlockDocument</td>
<td>P_IOBXSVC_DOCUMENT</td>
<td>Unlock a document that was previously locked.</td>
</tr>
<tr>
<td>msgIOBXSvcScheduleDocument</td>
<td>P_IOBXSVC_DOCUMENT</td>
<td>Schedule a document that is not ready for input/output.</td>
</tr>
<tr>
<td>msgIOBXSvcIOStart</td>
<td>P_IOBXSVC_DOCUMENT</td>
<td>Start the input/output process for a document in the input/output queue.</td>
</tr>
<tr>
<td>msgIOBXSvcIOCancel</td>
<td>nothing</td>
<td>Cancel the input/output process.</td>
</tr>
<tr>
<td>msgIOBXSvcIOCLeanUp</td>
<td>P_IOBX_DOC_OUTPUT_DONE</td>
<td>Clean up after the current input/output is done.</td>
</tr>
<tr>
<td>msgIOBXSvcStateChanged</td>
<td>OBJECT</td>
<td>Tell observers that the service state just changed.</td>
</tr>
<tr>
<td>msgIOBXSvcQueryState</td>
<td>P_IOBXSVC_QUERY_STATE</td>
<td>Pass back the state of the service.</td>
</tr>
<tr>
<td>msgIOBXSvcGetEnabled</td>
<td>P_BOOLEAN</td>
<td>Get the enabled state of the service.</td>
</tr>
<tr>
<td>msgIOBXSvcSetEnabled</td>
<td>BOOLEAN</td>
<td>Set the enabled state of the service.</td>
</tr>
<tr>
<td>msgIOBXDocIOStartOK</td>
<td>nothing</td>
<td>Ask the In box or Out box document if it is OK to start the input/output process.</td>
</tr>
<tr>
<td>msgIOBXDocIOCStart</td>
<td>nothing</td>
<td>Tell an In box or Out box document to start the input/output process.</td>
</tr>
<tr>
<td>msgIOBXDocIODone</td>
<td>P_IOBX_DOC_OUTPUT_DONE</td>
<td>Tell the In box or Out box service that input/output is finished.</td>
</tr>
<tr>
<td>msgIOBXDocIOCCancel</td>
<td>nothing</td>
<td>Tell an In box or Out box document to cancel the input/output process.</td>
</tr>
<tr>
<td>msgIOBXDocStatusChanged</td>
<td>P_IOBX_DOC_STATUS_CHANGED</td>
<td>Tell the In box or Out box service that the document status is changed.</td>
</tr>
</tbody>
</table>
Chapter 100 / The Address Book

The PenPoint™ operating system address book protocol allows you to write an application or service that responds to requests from services or other applications for address information. The addresses can be, but are not limited to, street addresses, voice phone numbers, phone numbers for data communication devices (such as fax machines, e-mail routers, and so on).

An address book application should be a subclass of clsAddressBookApplication. Although it doesn’t have to, a subclass of clsAddressBookApplication inherits some additional address book behavior.

This chapter covers these topics:

• Concepts of the address book, including the address book and address book manager classes, organization of data, and groups.

• How to install and use the GO address book application.

• Using the address book messages to add, delete, set, and get address book entries and their service data.

The PenPoint SDK provides a simple address book (described later in this chapter) with a user interface that allows you to test your services and applications that access an address book.

The information in this chapter is not exhaustive. Please refer to the header files ADDRBOOK.H and ABMGR.H for more information.

Concepts

When you try to communicate with someone, you have to consider which communication method you want to use before you can make contact. If you phone someone, you have to find their phone number. If you send a fax to someone, you have to find their fax number. If you want to send electronic mail to someone, you need to find the phone number of the mail service — then you need to find the e-mail address for that person.

Services on PenPoint computers are faced with much the same problem. Any one person might have several different addresses for several different communications methods.

In traditional operating systems, it is up to each application to provide an address book through which the application can find a person’s address. This means that the user has to enter names and phone numbers in a separate address book for each application.

To avoid this duplication and scattering of information, PenPoint defines a protocol for address books. An address book is an application or a service that
responds to the address book protocol. All clients that need to access the address book use the same protocol. The protocol specifies particular messages that get and set information in an address book; the protocol also specifies the types of information to be stored in address books and the structures that store this information.

All clients that want to access an address book do it by sending messages to the system object, theAddressBookMgr, which then forwards the message to the system address book. While there can be many address books running in PenPoint, the system address book is identified as the address book that is currently receiving messages through theAddressBookMgr. All address books must register with theAddressBookMgr, but only one address book can be the system address book.

The address book protocols allow an application to activate an address book, that is, to make it the system address book. The user chooses which address book to make the system address book through a user interface provided by an address book, or by any application that chooses to do so. theAddressBookMgr provides a way for applications to provide this user interface through option sheets. All the application needs to do is forward msgOptionAddCards to theAddressBookMgr after the application has handled the message.

The user interface for a particular service or application can access an address book (through theAddressBookMgr) and allow the user to modify information in the system address book.

**Participants**

There are three principal participants in any address book operation:

- The client is any service or application that requests an address book operation.
- theAddressBookMgr is a well-known object, through which all address book requests are channeled.
- An address book is an application or a service that is registered with theAddressBookMgr and is made current by theAddressBookMgr. The address book is responsible for storing and retrieving address information.

An address book announces its availability by registering with theAddressBookMgr. Any number of address books can register with theAddressBookMgr.

**The Address Book Protocols**

There are three protocols used to communicate with the address book:

- The address book protocol Enables clients to communicate with address books. This protocol is defined in the file ADDRBOOK.H.
- The address book manager protocol Enables address books to register themselves with theAddressBookMgr and to get information about other
registered address books. This protocol is defined in ABMGR.H. The class for theAddressBookMgr is private.

**The sendable services protocol** Enables address books to request addressing information from a service that is accessed through theSendableServices service manager. This protocol is defined by clsSendableService in the file SENDSERV.H. The whole topic of sendable services is discussed in Chapter 101.

---

**The Address Book Protocol**

The address book protocol, defined in ADDRBOOK.H, allows clients to communicate with the system address book. The protocol allows the client to:

- Get information.
- Set information.
- Delete information.
- Get address book metrics.
- Search for information.

Remember that the address book is not a file; it is an application or service that responds to the clsAddressBook messages. Thus, it is better to think of an address book as a server; the client sends requests to the server and it responds with addresses.

Your client sends all address book requests to theAddressBookMgr. Before requesting information from theAddressBookMgr, your client must declare that it wants to use the system address book by sending msgABMgrOpen to theAddressBookMgr. If the system address book is an application, theAddressBookMgr activates the application; if the system address book is a service, theAddressBookMgr binds to the service.

When your client needs information from the address book, it sends a message to theAddressBookMgr. theAddressBookMgr forwards the message to the system address book (with ObjectSend()). Because theAddressBookMgr uses ObjectSend() (rather than ObjectCall()), your client must use shared memory to allocate buffers that it will use to send or retrieve information from the address book.

All address book requests that deal with a specific entry in the address book require a key to the address book. The key is *not* an index to the address book entries, rather it contains information that is used by the address book to locate an entry.

To get a key, send msgAddrBookSearch to theAddressBookMgr. If you are simply interested in retrieving the the information contained at that location, your search arguments can specify what information you need; msgAddrBookSearch returns the information when it finds the entry. (You can also cache the key passed back by msgAddrBookAdd.)
The Address Book Manager Protocol

The address book manager (theAddressBookMgr) keeps track of the system address book. When a client needs information from the address book, it uses ObjectCall() to pass a message to theAddressBookMgr. theAddressBookMgr then forwards the message to the system address book using ObjectSend().

Although any application or client can be an address book, an address book must register itself with theAddressBookMgr. The protocol for registering address books is defined in ABMGR.H. The protocol allows address books to:

- Register an address book.
- Unregister an address book.
- List the registered address books.
- Activate an address book (make it the system address book).
- Deactivate an address book.

If an address book is an application that inherits from clsAddrBookApplication, an instance of application is automatically registered with theAddressBookMgr when it is created; the application is automatically unregistered when it is destroyed. clsAddrBookApplication provides no other special behavior.

Organization of Data

The address book is organized by entries. An individual entry contains:

- Attributes, which contain individual information for the entry such as names, phone numbers, a street address, and so on.
- Service addresses, which contain service-related information such as e-mail addresses and so on.

For each entry in the address book there can be many addresses.

Entry Attributes

Within each entry, there are one or more attributes, which are used to store the information commonly thought of as “address book information.” An attribute is described by the ADDR_BOOK_ATTR structure. Each attribute contains:

- An attribute identifier, which indicates what the data is used for (for example given name, surname, street, company, and so on). Table 100-1 lists the minimum set of attribute identifiers that all address books must implement.
- An attribute type, which indicates the type of data contained in this attribute. The possible types are:
  
  - abNumber  A 32-bit number.
  - abString  A null-terminated string.
  - abPhoneNumber  A phone number, as defined by clsDialEnv.
  - abOther  An encoded byte array.
The actual value.

• A label that can be used when displaying the attribute value.

The address book protocol defines a minimum set of attribute identifiers and attribute types that all address books must implement. If you know of an identifier or type that is commonly used, but not represented in this set of attributes, please contact GO Developer Technical Support.

Table 100-1

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddrBookGroupNameId</td>
<td>abString</td>
<td>Name of a group.</td>
</tr>
<tr>
<td>AddrBookGivenNameId</td>
<td>abString</td>
<td>A person's given name.</td>
</tr>
<tr>
<td>AddrBookSurNameId</td>
<td>abString</td>
<td>A person’s surname (family name).</td>
</tr>
<tr>
<td>AddrBookHomePhoneId</td>
<td>abPhoneNumber</td>
<td>Home phone number.</td>
</tr>
<tr>
<td>AddrBookBussPhoneId</td>
<td>abPhoneNumber</td>
<td>Work phone number.</td>
</tr>
<tr>
<td>AddrBookBussPhone2Id</td>
<td>abPhoneNumber</td>
<td>Second work phone number.</td>
</tr>
<tr>
<td>AddrBookCountryId</td>
<td>abString</td>
<td>Country in postal address.</td>
</tr>
<tr>
<td>AddrBookStateId</td>
<td>abString</td>
<td>State or prefecture.</td>
</tr>
<tr>
<td>AddrBookZipId</td>
<td>abString</td>
<td>Zip code or post code.</td>
</tr>
<tr>
<td>AddrBookCityId</td>
<td>abString</td>
<td>City.</td>
</tr>
<tr>
<td>AddrBookDistrictId</td>
<td>abString</td>
<td>District within a city.</td>
</tr>
<tr>
<td>AddrBookStreetId</td>
<td>abString</td>
<td>Street, building, apartment, and so on.</td>
</tr>
<tr>
<td>AddrBookCompanyId</td>
<td>abString</td>
<td>Company name.</td>
</tr>
<tr>
<td>AddrBookTitleId</td>
<td>abString</td>
<td>A person’s title.</td>
</tr>
<tr>
<td>AddrBookPositionId</td>
<td>abString</td>
<td>A person’s position.</td>
</tr>
<tr>
<td>AddrBookNickNameId</td>
<td>abString</td>
<td>A person’s nickname.</td>
</tr>
<tr>
<td>AddrBookFaxId</td>
<td>abPhoneNumber</td>
<td>A fax phone number.</td>
</tr>
<tr>
<td>AddrBookSvcNameId</td>
<td>abString</td>
<td>The name of a service.</td>
</tr>
<tr>
<td>AddrBookSvcNoteId</td>
<td>abString</td>
<td>User define nickname for service.</td>
</tr>
</tbody>
</table>

The identifier AddrBookStreetId can contain number, street, building, apartment number, suite, floor, or any other addressing information. If the actual address contains several lines when written, you can use the character \012 (LF in ASCII) to separate the lines. For example, an address might contain:

    919 East Hillsdale Blvd,
    Suite 400

You could store this information as:

    "919 East Hillsdale Blvd,\012Suite 400"

But it is up to the address book how to display such information.

When requesting information from an address book, one of the fields in the ADDR_BOOK_ENTRY structure specifies the number of entry attributes for which you expect information (numAttrs). To specify all attributes, set numAttrs to
AddrBookAll. The address book will allocate the necessary storage for the information. It is up to the client to free these buffers.

**Service Addresses**

When first created, an address book contains the minimum set of information as described by the attribute identifiers (listed above). However, it has no address descriptors for service information. To gather the address descriptors, the address book queries the sendable services for the type of addressing information they require. The address book can then add the descriptors to its attributes. The sendable service protocol is described further in Chapter 101.

Within each entry there can be one or more service addresses. Each service address contains:

- A key for a service instance.
- An address descriptor that contains a series of attributes that pertain to the specific service. For instance, an e-mail address would contain both a string attribute for the e-mail address and a phone number attribute for the dial-up access line to the e-mail service.

Typically the service attributes use AddrBookSvcNameld to identify the service and AddrBookSvcNoteId for a user-specified nickname.

When requesting information from an address book, one of the fields in the ADDR_BOOK_ENTRY structure specifies the number of services for which you expect information (numServices). You can use this field to specify different combinations of services and their attributes.

- If you want all service attributes for all services, set numServices to AddrBookAll.
- If you want selected attributes for all services, set numServices to AddrBookAllSvcSelectAttrs. You must provide an additional attributes structure that specifies these attributes.
- If you want all attributes for selected services, set numServices to AddrBookSelectSvcAllAttrs. You must provide an addition services structure that specifies these services.
- If you want selected attributes for selected services, set numServices to AddrBookSelectSvcSelectAttrs. You must provide both additional attribute and service structures that specify these attributes and services.

The address book will usually allocate the necessary storage for the information. It is up to the client to free these buffers.

**Groups**

The address book protocol can enable address books to collect entries into groups, which are accessed through a single entry. A group consists of a group identifier, which contains the name for the group, and an array of keys to the members of the group.
The GO Address Book Application

The PenPoint SDK includes an address book (in \PENPOINT\SDK\APP\AB.EXE) that both serves as an address book and provides a user interface to its information. Although it is possible for an address book to both hold information and provide a user interface, usually an address book will simply contain the address information; other services and applications will access the address book and provide a user interface.

The GO Address Book allows you to test your services and applications that access an address book. You must have a separate license from GO Corporation to distribute the GO Address Book application with your software.

Loading the GO Address Book

You can load the GO Address Book by adding this line to your APP.INI file in \PENPOINT\BOOT:

```
\boot\penpoint\sdk\app\Address Book
```

To create an address book, make a caret in the Notebook’s table of contents and tap on Address Book in the stationery menu. Tap on the address book icon to turn to the Address Book.

Using the GO Address Book

The screen in Figure 100-1 shows the GO Address Book.

The Address Book is initially empty. You add new entries by tapping on the Add Entry... menu item and filling out information. Initially you add name and postal address information. To add information related to sendable services, select an entry, then tap on Add Service... in the Edit menu.

You can use many of the core gestures in the GO Address Book. This table lists the gestures that are specific to the GO Address Book. For quick help on the available gestures, make a Quick Help gesture on the Address Book.

<table>
<thead>
<tr>
<th>Gesture</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code> </code> Tap</td>
<td>Selects the field.</td>
</tr>
<tr>
<td><code> </code> Double tap</td>
<td>Selects the current line.</td>
</tr>
<tr>
<td><code> </code> Triple tap</td>
<td>Selects the entire entry.</td>
</tr>
<tr>
<td><code> </code> Circle</td>
<td>Displays an edit pad for a selection.</td>
</tr>
<tr>
<td><code> </code> Caret</td>
<td>Displays a new entry page; adds new lines to postal address information.</td>
</tr>
<tr>
<td><code> </code> Caret tap</td>
<td>Displays a pop-up list to add a new service to the selected entry.</td>
</tr>
<tr>
<td><code> </code> Delete</td>
<td>Deletes the selected information.</td>
</tr>
<tr>
<td><code> </code> Tap press</td>
<td>Selects the entire entry and starts a copy operation.</td>
</tr>
<tr>
<td><code> </code> Press</td>
<td>Selects the entire entry and starts a move operation.</td>
</tr>
</tbody>
</table>
The Address Book Messages

Table 100-3 summarizes the clsAddressBookApplication messages. Following it, Table 100-4 summarizes the clsABMgr messages. Please see the header files ADDRBOOK.H and ABMGR.H for more detail on these messages.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAddrBookGet</td>
<td>P_ADDR_BOOK_ENTRY</td>
<td>Fills in the specified entry field data, given an address book key for the entry.</td>
</tr>
<tr>
<td>msgAddrBookSet</td>
<td>P_ADDR_BOOK_ENTRY</td>
<td>Sets the specified entry and service data.</td>
</tr>
<tr>
<td>msgAddrBookAdd</td>
<td>P_ADDR_BOOK_ENTRY</td>
<td>Adds the specified entry and service data.</td>
</tr>
<tr>
<td>msgAddrBookDelete</td>
<td>P_ADDR_BOOK_ENTRY</td>
<td>Deletes the specified entry or service data.</td>
</tr>
</tbody>
</table>

Table 100-3

clsAddressBookApplication Messages
CHAPTER 100 / THE ADDRESS BOOK

Using an Address Book

Table 100-3 (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAddrBookSearch</td>
<td>P_ADDR_BOOK_SEARCH</td>
<td>Searches for the entry that matches the search spec.</td>
</tr>
<tr>
<td>msgAddrBookCount</td>
<td>P_ADDR_BOOK_COUNT</td>
<td>Finds the number of entries that match the search spec.</td>
</tr>
<tr>
<td>msgAddrBookGetMetrics</td>
<td>P_ADDR_BOOK_METRICS</td>
<td>Passes back the metrics for the address book.</td>
</tr>
<tr>
<td>msgAddrBookGetServiceDesc</td>
<td>P_ADDR_BOOK_SERVICES</td>
<td>Gets the service address description from the address book.</td>
</tr>
<tr>
<td>msgAddrBookAddAttr</td>
<td>P_ADDR_BOOK_ATTR</td>
<td>Adds a new attribute to the active address book.</td>
</tr>
</tbody>
</table>

Notification Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAddrBookEntryChanged</td>
<td>P_ADDR_BOOK_ENTRY_CHANGE</td>
<td>Sent to observers when an entry has been changed, added or deleted.</td>
</tr>
</tbody>
</table>

Table 100-4
clsABMgr Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgABMgrRegister</td>
<td>P_AB_MGR_ID</td>
<td>Registers an application or a service as an address book instance.</td>
</tr>
<tr>
<td>msgABMgrUnregister</td>
<td>P_AB_MGR_ID</td>
<td>Unregisters an application or a service as an address book instance.</td>
</tr>
<tr>
<td>msgABMgrOpen</td>
<td>nothing</td>
<td>Used by address book clients to begin access to address books.</td>
</tr>
<tr>
<td>msgABMgrClose</td>
<td>nothing</td>
<td>Used by address book clients to end access to address books.</td>
</tr>
<tr>
<td>msgABMgrList</td>
<td>P_LIST</td>
<td>Creates a list of currently registered address book in pArgs.</td>
</tr>
<tr>
<td>msgABMgrActivate</td>
<td>P_AB_MGR_ID</td>
<td>Make a registered address book the system address book.</td>
</tr>
<tr>
<td>msgABMgrDeactivate</td>
<td>P_AB_MGR_ID</td>
<td>Deactivates the current system address book.</td>
</tr>
<tr>
<td>msgABMgrIsActive</td>
<td>P_AB_MGR_ID</td>
<td>Passes back the UID of the system address book.</td>
</tr>
<tr>
<td>msgABMgrChanged</td>
<td>P_AB_MGR_LIST</td>
<td>Sent to observers of the AddressBookMgr when the system address book changes.</td>
</tr>
</tbody>
</table>

Using an Address Book

The rest of this chapter covers two topics:

- Developing an address book client.
- Developing an address book.

This section discusses the messages that an address book client sends. The section “Writing an Address Book” discusses the messages that an address book sends and receives.
Opening the Address Book

Before you can access the system address book, you must send msgABMgrOpen to theAddressBookMgr.

When you are done with the address book, you must send msgABMgrClose to theAddressBookMgr.

Searching the Address Book

The root behind most address books operations is in finding a specific entry. You use msgAddrBookSearch to find an entry and to return information from the found entry.

msgAddrBookSearch enables you to make fairly sophisticated searches. You can specify:

- A search specification that can contain one or more comparisons.
- A sort order for the entries matched by the search specification.
- A number of matching entries to skip.
- A direction for the search.

When you search the address book, you must always specify a key (key in ADDR_BOOK_SEARCH). When you are performing a series of searches, you start the next search with the key passed back by the previous search. For the first search, you should specify a null key; the direction field specifies whether the search begins at the first or last entry in the sort order.

You can specify a sort order by specifying an attribute identifier in the sort field. Attribute identifiers are described above.

To specify a search direction, set the dir field to abEnumNext to search forward and abEnumPrevious to search backwards. When key is null, abEnumNext begins the search at the first element, abEnumPrevious begins the search at the last element.

To specify the nth entry (from the current location) that matches the search specification, specify a non-zero value for the nth argument. If nth is 1, the search gets the next entry that matches the search.

The Search Query

A search query can contain a number of different comparisons. The query argument points to an ADDR_BOOK_QUERY structure that lists the number of comparisons (numAttrs) and contains a pointer to the first in a list of ADDR_BOOK_QUERY_ATTR structures (attrs), each of which specifies a separate comparison.

The ADDR_BOOK_QUERY_ATTR structure specifies:

- id The attribute identifier of the attribute being compared.
- length The length of an encoded byte array (used only if attribute type is abOther).
valueOp  The comparison operator; the comparison operators are:
  abEqual   Tests if the attribute is equal to value.
  abNotEqual Tests if the attribute is not equal to value.
  abGreater Tests if the attribute is greater than value.
  abLess   Tests if the attribute is less than value.
  abGreaterEqual Tests if the attribute is greater than or equal to value.
  abLessEqual Tests if the attribute is less than or equal to value.
  abMatchBeginning Tests if the string specified in value is a substring of
          the attribute, starting at the beginning.
  abMatchEnd Tests if the string specified in value is a substring of the at-
          tribute, starting at the end.
  abMatchPartial Tests if the string specified in value is a substring of the
                   attribute (not anchored to the beginning or end).
value  The value against which the attribute of the current entry is
       compared.
attrOp  The operator that specifies the relationship between this attribute
        structure and the following attribute structure (if there is one). The last
        attribute in the list does not need to specify an attrOp. The possible
        values are:
  abAnd   Both the current attr structure and the next attr structure must
           be true for the comparison to succeed.
  abOr    Either the current attr structure or the next attr structure can be
           true for the comparison to succeed.

For example, if a client wants to specify a query that says match an entry whose
last name is Smith and whose zip code is greater than or equal to 95000, the query
would contain two attr structures:

```
pArgs->query id length value valueOp attrOp
  fsp
  attr[0] AddrBookSurNameId n/a "Smith" abEqual abAnd
  attr[1] AddrBookZipId n/a 95000 abGreaterEqual n/a
```

The Search Result

When msgAddrBookSearch successfully matches an entry, it passes back the
information you specified in an ADDR_BOOK_ENTRY structure (result). The
ADDR_BOOK_ENTRY structure contains one or more ADDR_BOOK_ATTR
structures (described above) that specifies the actual attributes that you need.

Because theAddressBookMgr uses ObjectSend() to relay messages to address
books, the ADDR_BOOK_ENTRY structure must be allocated from shared memory.

Getting More Information

To get more information from an entry, send msgAddrBookGet to
theAddressBookMgr, specifying the key returned by msgAddrBookSearch.
You specify the information you want from the address book entry in the same way that you specify the search result in msgAddrBookSearch.

For example, you might use msgAddrBookSearch to find an entry that meets particular criteria. The message passes back the fundamental information that you need. You then compare that information against some other set of information and, if you have a match, you can then send msgAddrBookGet, using the same key to retrieve further information about the entry.

### Changing Information

To change information in an address book entry, send msgAddrBookSet to theAddressBookMgr. The message takes a pointer to an ADDR_BOOK_ENTRY that contains the changed information for the entry.

Because theAddressBookMgr uses ObjectSend() to relay messages to address books, the ADDR_BOOK_ENTRY structure must be allocated from shared memory.

### Adding a New Entry

To add a new entry to an address book, send msgAddrBookAdd to theAddressBookMgr. The message takes a pointer to an ADDR_BOOK_ENTRY that contains the new information for the entry.

As above, the ADDR_BOOK_ENTRY structure must be allocated from shared memory.

### Deleting an Entry

To delete an entire entry from an address book (or to delete services from an entry), send msgAddrBookDelete to theAddressBookMgr. The message takes a pointer to an ADDR_BOOK_ENTRY that identifies the contains the changed information for the entry. The key field identifies the entry to delete.

To delete an entire entry with services, the numServices field must be 0.

When deleting services, the numServices field specifies the number of services to delete. The services field points to the first in an array of ADDR_BOOK_SERVICE structures that contain the service IDs of the services to delete.

### Writing an Address Book

When you write an address book, it must be prepared to receive all address book messages defined in ADDRBOOK.H.

In addition, your address book needs to send a number of address book manager messages, which are defined in ABMGR.H. Particularly, your address book must be able to:

- Register itself as an address book instance.
- Unregister itself when it is deleted.
- Give up the role as system address book, when appropriate.
The fundamental structure used in the address book manager messages is the AB_MGR_ID. This structure contains:

- **name**: A string that contains the name of the address book.
- **type**: An AB_MGR_ID_TYPE that indicates that the address book is an application (abMgrApplication) or a service or data object (abMgrObject).
- **value**: A union that contains a UID or a UUID, depending on whether the address book is a service or an application.
  - **uid**: Is an OBJECT that contains the UID of the service.
  - **uuid**: Is a UUID that contains the UUID of the application's working directory.

### Registering an Address Book

When an instance of your address book is created, it should register itself by sending msgABMgrRegister to theAddressBookMgr. The message takes a pointer to an AB_MGR_ID structure that identifies the address book.

Only when an address book is registered with theAddressBookMgr can it later be selected as the system address book.

If your address book inherits from clsAddrBookApplication, it is registered automatically when it is created.

### Unregistering an Address Book

When your address book is terminated (if an application, when it is deleted; if a service, when deinstalled), it should send msgABMgrUnregister to theAddressBookMgr. The message takes a pointer to an AB_MGR_ID structure that identifies the address book.

If your address book inherits from clsAddrBookApplication, it is unregistered automatically when it is deleted.

### Becoming the System Address Book

When your address book needs to become the system address book, it sends msgABMgrActivate to theAddressBookMgr. The message takes a pointer to an AB_MGR_ID structure that identifies the address book.

In the current implementation of the address book manager, only one address book can be the system address book at one time.

If there is currently a system address book, theAddressBookMgr deactivates that address book first.

There are two important status values that you should note when sending msgABMgrActivate:

- **stsABMgrAddrBookOpen**: The current system address book is currently open, and therefore cannot be deactivated.
stsABMgrAddrBookNotRegistered  The specified address book is not a registered address book.

Deactivating the System Address Book

To deactivate the system address book, send msgABMgrDeactivate to theAddressBookMgr. The message takes a pointer to an AB_MGR_ID structure.

Observing theAddressBookMgr

When an address book is activated or deactivated as the system address book, theAddressBookMgr sends msgABMgrChanged to its observers (which includes all registered address books). The message passes a pointer to an AB_MGR_NOTIFY structure, which contains:

- **type**  An AB_MGR_CHANGE_TYPE value that indicates the type of change. Possible values are:
  - abMgrRegister  An address book was registered.
  - abMgrUnregister  An address book was unregistered.
  - abMgrActivated  An address book was activated.
  - abMgrDeactivated  An address book was deactivated.
  - abMgrOpened  An address book was opened.
  - abMgrClosed  An address book was closed.

- **addressBook**  An AB_MGR_ID structure that indicates the address book that changed.

Handling Option Sheet Protocol

To provide users with a common way of selecting the system address book, an address book or other applications that want to provide this facility must be prepared to handle msgOptionSheetAddCards. When it receives this message, the address book can add cards to the option sheet, if it needs. The address book must then forward the message to theAddressBookMgr.

When theAddressBookMgr receives the message, it creates a card that allows the user to select the current address book. Subsequent option card related messages are sent directly to theAddressBookMgr.
Chapter 101 / The Sendable Services

A sendable service is a service that provides some form of deferred data transmission, such as e-mail or fax. Before sending data, the user must be able to address the fax or e-mail to a particular recipient. The sendable service protocol allows address books to query sendable services for the type of addressing information they require; the protocol also allows address books to request a sendable service to present a user interface, through which the user can update address information in the current address book.

Sendable services inherit from the abstract superclass clsSendableService, which defines the sendable service protocol. All sendable services are managed by the SendableServices service manager.

The Sendable Services Protocol

The PenPoint™ operating system has two general categories of data transfer mechanisms:

- Printing, which is assumed to be accessible from any PenPoint computer, and which has a simple addressing mechanism: the name of the printer.
- Other forms of data transfer, which the user could install separately, and which has any number of transmission mechanisms and addressing complexities.

This division is reflected in the Document menu in the PenPoint Application Framework's standard application menus: it contains a Print button and a Send button. The Print button is specific to a single action (printing a document). However, the Send button provides access to all other deferred data transfer mechanisms. In other words, it provides access to the sendable services.

The Address Book uses the sendable services protocol at two different times:

- Gathering address descriptors from sendable services.
- Requesting the sendable service to display a user interface through which the user can add, modify, or delete address data for that service.

Creating Address Descriptors

As described in Chapter 100, all address books contain a fixed set of attributes for usual addressing information, such as names, postal addresses, voice-phone numbers, and so on. In addition to this fixed set, the address book can maintain addressing information for services. Some services (such as fax) just need a phone number; others (such as e-mail) need a phone number for the e-mail provider and one or more strings for the actual mail address.
However, until the address book and all the sendable services are installed, the address book does not know the addressing attributes that are required by each service.

The address book requests an attribute descriptor from a sendable service by sending msgSendServGetAddrDesc to the sendable service. In response to the message, the sendable service creates an address descriptor and passes it back to the address book. The address book then adds the descriptor to its attributes.

There are two times when an address book should send msgSendServGetAddrDesc:

- When a new instance of the address book is created, the address book should enumerate all the services that belong to theSendableServices service manager. It should then send msgSendServGetAddrDesc to each of the sendable services.

- When the user installs a new sendable service, theSendableServices notifies the address book that a new service is available (by sending msgIMInstalled). The address book should then send msgSendServGetAddrDesc to the new service. An address book should always observe theSendableServices.

Displaying a User Interface

When you, or anyone else, writes an application that presents the user interface to an address book, the application must allow additions or changes to all the required attributes (name, postal address, and so on). It is easy to know what is expected, by simply examining the list of attributes and ensuring that you have implemented them all.

Unfortunately, you can't plan an address book user interface for modifying service information, because the amount and types of information maintained for service addresses is entirely up to each service. The solution is to provide a message whereby the address book requests the service to present a user interface for its own addressing information.

When an address book sends msgSendServCreateAddrWin to a sendable service, the sendable service should create a window in which the user can add or modify address information for the service.

When the address book sends msgSendServFillAddrWin to a sendable service, the sendable service should fill or clear its window depending on the address information passed in by the address book.

The address book can send msgSendServGetAddrSummary to a sendable service to request a string that contains an address summary.
The Sendable Services Messages

The sendable services messages are defined in SENDSERV.H. These messages are listed in Table 101-1.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgSendServGetAddrDesc</td>
<td>P_ADDR_BOOK_SVC_DESC</td>
<td>Responsibility of a sendable service to return its service attribute-value pairs that describe its service address.</td>
</tr>
<tr>
<td>msgSendServCreateAddrWin</td>
<td>P_SEND_SERV_ADDR_WIN</td>
<td>Converts address data into a window displaying the data.</td>
</tr>
<tr>
<td>msgSendServFillAddrWin</td>
<td>P_SEND_SERV_ADDR_WIN</td>
<td>Refreshes pArgs-&gt;win with information in pArgs-&gt;attrs.</td>
</tr>
<tr>
<td>msgSendServGetAddrSummary</td>
<td>P_SEND_SERV_ADDR_WIN</td>
<td>Given pArgs-&gt;attrs, set pArgs-&gt;addrSummary to be a displayable string that sums up the address.</td>
</tr>
</tbody>
</table>

Getting Address Descriptors

When an address book needs to get the address descriptor from a sendable service, it sends `msgSendServGetAddrDesc` to the service. The message takes a pointer to a `P_ADDR_BOOK_SVC_DESC` structure, which contains an uninitialized pointer for an array of address attributes (`attrs`).

The sendable service allocates storage for the attribute-value pairs and stores the address in `attrs`. The sendable service also sets the `numAttrs` field to the number of attributes used by the service.

Creating and Filling Address Windows

When an address book needs to present a user interface that gathers information about a sendable service, the address book sends `msgSendServCreateAddrWin` to request the sendable service to create an empty window for this information.

The address book then sends `msgSendServFillAddrWin` to the sendable service to request it to fill in or update the window using the data passed in by the message.

Both messages take a pointer to a `SEND_SERV_ADDR_WIN` structure, which contains a pointer to an array of address attributes (`addrAttr`) and the number of attributes in the array (`numAttrs`).

When the sendable service receives `msgSendServCreateAddrWin`, it should create a window suitable for containing its addressing information. The type of window it creates and whether the window is inserted in the window hierarchy or not is dependent on the implementation of the address book. For example, the GO address book actually uses the option sheet protocol; when it requests the sendable service to create an address window, the address book adds the new window to the list of option cards.
The sendable service should not fill information in the window when it creates it. Instead it should wait for the address book to send `msgSendServFillAddrWin`. When it receives `msgSendServFillAddrWin`, the sendable service should display its addressing information in the window with proper formatting and labels. If `numAttrs` is 0, the sendable service should display empty fields, presumably for the user to fill in information.

### Summarizing Address Information

The user interface for an address book might simply display a summary of an address (or a series of addresses) to the user, rather than displaying all the addressing information. For instance an address book might want to present the e-mail names for recipients, but doesn’t need to present things like the phone number used to access the e-mail system.

An address book can send `msgSendServGetAddrSummary` to a sendable service to request summary of addressing information. The message takes a pointer to a `SEND_SERV_ADDR_WIN` structure, which lists all the service attributes.

It is up to the sendable service to determine which attributes it can use and to create a summary string, which it passes back in the `addrSummary` field of the `SEND_SERV_ADDR_WIN` structure.
Part 11 / Resources
Chapter 102 / Introduction

Resources provide a mechanism for storing objects and data in a file for later retrieval. You can use resource files to store and retrieve application objects in response to msgSave and msgRestore, to manage document configuration information such as user preferences, and to store application-specific data such as user interface text strings.

Overview

A resource is an object or collection of data stored in a file. A resource file can contain several resources, each with a resource ID that is unique for the resource file. Resource files respond to messages for locating, reading, writing, updating, and deleting individual resources.

You can search for resources in an ordered list of resource files called a resource list. The PenPoint™ Application Framework provides every document with a default resource list that allows the document to override application resources, an application to override user preference resources, and user preference resources to override system defaults.

The PenPoint operating system has built-in support for object resources—filed representations of objects—as well as three kinds of data resource:

- **Byte array** resources are filed byte streams. These are the simplest kind of data resource, and your application must interpret the meaning of the bytes.
- **String** resources are filed, null-terminated text strings.
- **String array** resources are filed, indexed lists of text strings.

You can write resource agents to support additional semantics for the bytes of a resource.

To make it easier for you to create data resources, the PenPoint SDK includes a header file that lets you define the data resources in a C language source file and compile the source into a resource file with a resource compiler utility included with the PenPoint SDK.

Developer’s Quick Start

As an application writer, you use resources to save and restore application instance data and to create objects and data from resources stored in a resource file.

If you need to access a replaceable resource, you need to know the resource ID of the resource and the resource file in which it is stored. It is the responsibility of resource creators to publish their resource IDs in a header file.
Once you have the file handle and the resource ID, you can send `msgResReadData` (for data resources) or `msgResReadObject` (for object resources) to read the resource.

In the following example, we want to create a menu from a resource. We know that the resource file `MyResources` contains the object, and that it's well-known resource ID is defined by the symbol `tagMyMenu`.

The first step is to generate a handle on the resource file as shown in the example below:

```c
STATUS status;
RES_FILE_NEW resNew;
FILE_HANDLE resFile;
... status = ObjectCall(msgNewDefaults, clsFileHandle, &resNew);
resNew.fs.locator.pPath = "MyResources";
status = ObjectCall(msgNew, clsFileHandle, &resNew);
resFile = fsNew.object.uid;
```

Once you have created the resource file handle (and thereby opened the resource file), you can use `msgResReadObject` to read the specific object from the resource file as shown in the example below:

```c
RES_READ_OBJECT rro;
STATUS status;
OBJECT myMenu;
... rro.mode = resReadObjectMany; // let other procs have their own copy
rro.resId = tagMyMenu; // resource ID of target
ObjectCallRet(msgNewDefaults, clsObject, &rro.objectNew, status)
status = ObjectCallWarn(msgResReadObject, resFile, &rro);
myMenu = rro.objectNew.uid; // the UID of the retrieved object
```

---

**Layout of This Part**

Chapter 102, Introduction (this chapter) briefly introduces PenPoint resources.

Chapter 103, Concepts and Terminology, gives a more detailed overview of the concepts and terminology of PenPoint resources.

Chapter 104, Using clsResFile, describes the API for clsResFile, the resource file class.

Chapter 105, Defining Resources with the C Language, describes the resource language used to create generic resources.

Chapter 106, Compiling Resources, describes the DOS utility that converts the resource language files into resource files.

Chapter 107, System Preferences, describes the system preferences stored in the system resource file. This chapter also describes notification messages sent to observers when system preferences change.
Other Sources of Information

Part 7: File System describes the organization of the PenPoint file system and the messages used to access files. Resource files inherit characteristics (such as file handles) from ordinary files, so you should be familiar with ordinary files before programming with resource files.

Part 2: PenPoint Application Framework describes the message protocol involved in saving and restoring application instance data. Handling the PenPoint Application Framework save and restore messages is the most common reason for using object resources.
Chapter 103 / Concepts and Terminology

Resource files are used throughout the PenPoint™ operating system to store and recreate objects and data. Generally, you use resource files in the following ways:

- To file your application’s objects in response to msgSave, or to unfile them in response to msgRestore.
- To read and modify configuration information, such as user preferences, from a list of resource files.
- To store application-specific objects or data.

Object Resources

An object resource contains information required for creating or restoring a PenPoint object. Object resources are used to file object instance data as well as store replaceable object resources.

Clients save and restore object instance data in response to the PenPoint Application Framework messages msgSave and msgRestore. The PenPoint Application Framework maintains one document resource file for each document. When a document receives msgSave, it saves its state and sends msgResPutObject once for each of its child windows. By default, the child windows handle this by saving their state and sending msgResPutObject for each of their children, and so on, until the document has saved all of its state. When the document receives msgRestore, it simply reads the window objects from the document resource file rather than create the entire window tree from scratch.

Replaceable object resources are used to store information to create a specific object, such as a button, a menu, or a dialog box. You can improve code efficiency by storing a complex object to a resource file before run time (at install time, for example). At run time, rather than take the time to build the complex object while the user waits, you can simply read the object from the resource file.

If your application creates a number of objects that are the same, you can reduce the size of your executable file by storing one copy of the object in a resource file. At run time, you read the resource several times to create the multiple objects. You can also use this technique for objects that are not identical but are largely similar, using the resource to create most of the object and writing only the code necessary to modify each copy after reading it.
Once and Many Modes for Object Resources

When you read or write an object resource, you must specify either once mode, in which only one copy of the object exists in memory and only one copy of the resource exists in the file, or many mode. In many mode, any reader of the file can restore the object resource, creating its own copy of the object resource in memory (replacing the prior copy, if any); and any writer can replace the filed resource with an object resource of its creation. The only time you normally use once mode is when handling msgSave and msgRestore. For most other uses of resource files, you use many mode.

Data Resources

A data resource is an array of bytes stored in a resource file. You can write a resource agent to interpret the byte array in a meaningful way. For example, the PenPoint operating system provides resource agents for text string data resources and for string list data resources.

Later versions of PenPoint will provide specific support for multiple-language applications.

Data resources are useful for storing static data that the user might update or replace. For example, you can make it easier to port your application to a second language by using a resource file to store the all of the text strings your application displays. When you want to support a second language from your application, you can change the language of your application without writing new code, by simply providing the user with a resource file containing the text strings in the new language.

Resource Files

Resource files begin with a resource file header that contains general information about the file. Each resource in the file has its own resource header. The resource header contains the resource class, resource ID, and a resource type.

When you have created a resource file, you can write resources to it programmatically by sending messages to the file. There are separate sets of filing messages for the two types of resources. You can read and write object resources only with the object resource filing messages, and data resources only with the data resource filing messages. A single resource file can contain both object and data resources.

You can create data resources (but not object resources) by editing a resource definition file, which you then compile into a resource file with the C compiler.

Identifying Resources

Each resource is tagged with a unique resource ID—a 32-bit TAG that applications use to locate the resource. Each resource in a file has a resource ID that is unique within the file. As with most TAGs, the low 21 bits of a resource ID is the administered portion and scope of the ID. The next two bits identify the resource type of resource—well-known, dynamic, or well-known list. The remaining nine
bits include eight bits to identify the unique tag number (tagNum) of the resource. The high bit has a special meaning, explained below.

**Resource Types**

There are three types of resource ID: well-known resource IDs, dynamic resource IDs, and well-known list resource IDs. The flags of the resource ID TAG indicate which type of resource the ID identifies.

- A well-known resource ID (flags == 0) identifies a resource that any client can use.
- A dynamic resource ID (flags == 1) generally identifies an object that is nested within another object. For example, when you store an option sheet as an object resource, it can store its child windows in the same file using dynamic resource IDs.
- A well-known list resource ID (flags == 2) identifies a list resource such as an array of strings. “List Resources” later in this chapter describes list resources in more detail.

The high bit of the resource ID, when set, indicates that the object being identified is a well-known object, and that the remainder of the resource ID is the well-known UID of the object. You should take care not to modify the high bit yourself, as its special meaning is maintained by the application.

The file RESFILE.H defines macros and functions that you can use to create resource IDs of various types, and to extract individual values (such as flags) from a resource ID.

**Well-Known Resource IDs**

Well-known resource IDs are defined in the header file of the class that defines the resource, so any client of the class can access the identified resource. You use the same administered portion used in defining the class, and assign a unique tagNum value for each well-known resource ID that uses the same administered portion.

Use the MakeWknResId() macro to create a well-known resource ID when you compile your application. The macro has the following syntax:

```
MakeWknResId(wkn, tagNum)
```

For example:

```
RES_ID myButtonID; // create the resource ID
myButtonID = MakeWknResId(clsButton,1);
```

**Dynamic Resource IDs**

You normally send msgResPutObject in response to the PenPoint's Application Framework msgSave. msgResPutObject generates dynamic resource IDs for storing application objects that are not well-known objects. Dynamic resource IDs
are 29-bit numbers composed of the 21 bits normally used for the administered portion and the 8 bits normally used for the tagNum value.

If you want to store dynamic items without using msgResPutObject, you can create your own dynamic resource ID by sending msgResNextDynResId to a resource file handle. The message takes a pointer to a RES_ID value that will receive the dynamic resource ID.

If the message is successful, it returns stsOK and stores the newly generated resource ID in the specified RES_ID. In the example below:

```c
RES_ID myDataResource;
OBJECT resFile;
STATUS s;

// create the dynamic resource ID
ObjCallJmp(msgResNextDynResId, resFile, &myDataResource, s, error);
```

Be aware that dynamic resource ID values are not recycled. Once a file uses up its 29 bits worth of dynamic resource IDs, msgResNextDynResId returns stsFailed.

### Well-Known List Resource IDs

A well-known list resource ID identifies a list of indexed resources such as an array of text strings. For list resource IDs, the eight bits normally used for the tagNum value is used to indicate a six-bit list group and a two-bit list number within that group. The list group identifies the type of list, while the list number identifies one of up to four lists of that type in the resource file.

- List groups numbered from 0 to 1F hex are available for your use.
- List group 20 hex is reserved for arrays of strings for toolkit tables.
- List group 21 hex is reserved for arrays of strings for standard message text.
- List group 22 hex is reserved for arrays of strings for Quick Help text.
- List groups numbered from 23 to 3F hexadecimal are reserved for system use.

The data of a list resource includes a pseudo-resource ID index for each of the elements of the list. These index IDs look like resource IDs, except that the 8 bits normally used for the tagNum value instead indicate the item’s index in its list resource, while the flags indicate which list. This allows each list to include up to 256 entries. Since each list group can have 4 lists per group, this yields up 1,024 items per list group for each unique administered portion. To access a list resource item, you must have the resource ID as well as the index within the list. You can generate the resource ID and index from the group number and list resource ID.

### Using Resource IDs

RESFILE.H defines several macros that let you determine the type of resource ID returned by a message. The macros are:

- **WknItemResId(resId)** True if the resource ID is a well-known resource ID.
- **WknListResId(resId)** True if the resource ID is a well-known list resource ID.
**Resource Agents**

Resource agents enhance the default reading and writing behavior of data resources, interpreting the resource data in special ways. The simplest resource agent is the default resource agent, which treats data resources as a stream of bytes. However, data isn't always just a byte stream; often the sequence of bytes has meaning. The data might be a null-terminated string, a series of unsigned 32-bit values, or a series of 8-byte floating point values. Agents work for you by interpreting the formats of the data they read from the data resource file, and converting data to a stream of bytes to write to the data file.

The PenPoint operating system comes with three resource agents:

- **resDefaultResAgent** is the default resource agent. This agent treats data resources as a stream of bytes.
- **resStringResAgent** handles data resources that are NULL-terminated strings.
- **resStringArrayResAgent** handles data resources that are arrays of NULL-terminated strings. The array of strings must be terminated by a `pNull` string pointer.

Chapter 104, Using `clsResFile`, explains how to write your own agents to handle other types of data resources.

**Resource Lists**

A resource list is an instance of `clsResList`, which inherits from `clsList`. The entries in a resource file are resource file handles, other resource file lists, or null entries. When you send a read or find message to a resource file list object, it sends the message to each of its entries in turn (skipping null entries) until the message returns `stsOK` (`msgResEnum`, which enumerates the items in each file, is a special case).

Each application class maintains a default resource list object in its class metrics. The default application installation process creates this list with four initial elements:

- The document resource file, `DOC.RES`.
- The application resource file, `APPRES`.
- The system preferences resource file.
- The PenPoint system resource file.
The system resource file has the well-known file handle theSystemResFile; the system preferences resource file has the well-known file handle theSystemPreferences. Even though these resource file handles are well-known, they are on the default resource list for easier resource searching.

The application resource file is stored in the directory for the corresponding application (under \PENPOINT\SYS\APP). You should add dynamic resources to the application resource file when you install the application (that is, when running instance 0 of the application). For more information about adding dynamic resources during installation, see Part 12: Installation API.

You can create a resource list in addition to the default application resource list by sending msgNewDefaults and msgNew to clsResList. Instances of clsResList pass all other messages to clsList. For more information on clsList messages see the description of clsList in Part 9: Utility Classes.
Chapter 104 / Using clsResFile

The resource file class, clsResFile, inherits from clsFileHandle. This chapter presents the data structures used by clsResFile messages and discusses the messages as well. For more information on using clsResFile, see the RESFILE.H header file shipped with the PenPoint™ SDK.

**clsResFile Messages**

Table 104-1 summarizes the messages that clsResFile defines.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_RES_FILE_NEW</td>
<td>Creates a resource file object.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_RES_FILE_NEW</td>
<td>Initializes the RES_FILE_NEW structure to default values.</td>
</tr>
<tr>
<td>msgResReadData</td>
<td>P_RES_READ_DATA</td>
<td>Reads resource data from a resource file or resource list.</td>
</tr>
<tr>
<td>msgResWriteData</td>
<td>P_RES_WRITE_DATA</td>
<td>Writes resource data to a file.</td>
</tr>
<tr>
<td>msgResUpdateData</td>
<td>P_RES_WRITE_DATA</td>
<td>Updates existing data resource data.</td>
</tr>
<tr>
<td>msgResReadObject</td>
<td>P_RES_READ_OBJECT</td>
<td>Reads a resource object from a resource file or resource list.</td>
</tr>
<tr>
<td>msgResWriteObject</td>
<td>P_RES_WRITE_OBJECT</td>
<td>Writes a resource object to a file.</td>
</tr>
<tr>
<td>msgResGetObject</td>
<td>P_OBJECT</td>
<td>Reads the filed object resource from the current file position.</td>
</tr>
<tr>
<td>msgResPutObject</td>
<td>OBJECT</td>
<td>Writes the object as a filed object resource to the current file position. For use only during msgSave.</td>
</tr>
<tr>
<td>msgResReadObjectWithFlags</td>
<td>P_RES_READ_OBJECT</td>
<td>Reads a resource object, passing the supplied flags.</td>
</tr>
<tr>
<td>msgResWriteObjectWithFlags</td>
<td>P_RES_WRITE_OBJECT</td>
<td>Writes a resource object, passing the supplied flags.</td>
</tr>
<tr>
<td>msgResDeleteResource</td>
<td>RES_ID</td>
<td>Marks as deleted the resource identified by RES_ID.</td>
</tr>
<tr>
<td>msgResGetInfo</td>
<td>P_RES_INFO</td>
<td>Gets information on a data or object resource in a resource file or a resource list.</td>
</tr>
<tr>
<td>msgResEnumResources</td>
<td>RES_ENUM</td>
<td>Enumerates resources in a resource file or resource list.</td>
</tr>
<tr>
<td>msgResNextDynResId</td>
<td>P_RES_ID</td>
<td>Gets the next available dynamic resource ID from the file.</td>
</tr>
</tbody>
</table>

continued
Creating a Resource File Handle

To create a resource file handle, send `msgNewDefaults` and `msgNew` to `clsResFile`. When `msgNew` completes, the `object.uid` field of the `RES_NEW` structure contains the UID of the resource file handle.

`msgNew` and `msgNewDefaults` both take a pointer to a `RES_FILE_NEW` structure as their argument. The `RES_FILE_NEW` structure includes all the fields that `clsFileHandle`'s `FS_NEW` structure includes. This is how `clsResFile` inherits from `clsFileHandle`. In addition to the `FS_NEW` fields, `RES_FILE_NEW` includes the following fields (the data type of each field is shown in parentheses following its name):

- **mode** (*RES_NEW_MODE*) A set of flags that controls various attributes of the resource file handle. The flags can include any combination of the following:
  - `resSharedResFile` (default: false) More than one client can use this file handle concurrently. This does not mean that more than one client can open a file handle on the file, just that multiple clients can safely share the one file handle.
  - `resCompactOnClose` (default: true) Send `msgCompact` to file whenever the resource file is closed.
  - `resCompactAuto` (default: false) Send `msgCompact` to file when ratio of deleted to non-deleted resources in the file reaches `compactRatio`.
  - `resVerifyVersions` (default: true) Check that current system version is at least the minimum supported version for each resource.

- **compactMinimum** (*U16*) The minimum number of resources required in the file before automatic compaction occurs.

- **compactRatio** (*U16*) A percentage representing the minimum ratio of deleted resources marked to undeleted resources required before automatic compaction occurs.

When you delete a resource from a file, the resource is marked as deleted but is not deleted until the file is compacted. If `resCompactOnClose` is set, the file is compacted every time the controlling document closes. If `resCompactAuto` is set, the file is compacted automatically if there are more than `compactMinimum` resources in the file and a deletion causes the ratio of deleted resources to

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>msgResCompact</code></td>
<td>void</td>
<td>Compacts the resource file, actually deleting the resources marked as deleted.</td>
</tr>
<tr>
<td><code>msgResFlush</code></td>
<td>void</td>
<td>Flushes the index of resources read from the file.</td>
</tr>
<tr>
<td><code>msgResAgent</code></td>
<td>P_RES_AGENT</td>
<td>Message sent by resource file to resource agent when forwarding messages.</td>
</tr>
</tbody>
</table>
undeleted resources to exceed compactRatio. For example, if compactRatio is 30, the file is compacted when the ratio of deleted resources to undeleted resources reaches 30%.

**Locating a Resource**

To search for and get information about a resource, send msgResGetInfo to a resource file handle or resource file list. msgResGetInfo takes a pointer to a RES_INFO structure as its argument. You pass in resId, and msgResGetInfo passes back information about the identified resource. RES_INFO includes the following fields (the data type of each field is shown in parentheses following its name):

- **resId** (RES_ID) The ID of the resource to find.
- **file** (RES_FILE) The file where the identified resource resides. This is mainly useful when you send msgResGetInfo to a resource list, so that you know which of the files in the list contains the resource.
- **agent** (UID) The agent that saved the resource.
- **objClass** (UID) If an object resource, the class of the object.
- **offset** (U32) The offset in bytes from the start of the file to the first byte of the resource. Use only with caution.
- **size** (U32) The size of the resource in bytes. Use only with caution.
- **minSysVersion** (U16) The minimum system version with which the resource is compatible.

**Reading a Data Resource**

To search for and read a data resource, send msgResReadData to either a resource file handle or a resource file list. msgResReadData takes a pointer to a RES_READ_DATA structure as its argument. RES_READ_DATA includes the following fields (the data type of each field is shown in parentheses following its name):

- **resId** (RES_ID) The resource ID of the resource to read.
- **pData** (P_UNKNOWN) The retrieved resource data.
- **heap** (OS_HEAP_ID) A heap from which to allocate memory for storing the retrieved resource data. Set to Nil(OS_HEAP_ID) if pData points to an allocated buffer.
- **length** (U32) The length of the resource data.
- **pAgentData** (P_UNKNOWN) Agent-specific data, such as the index into a list resource.

**Writing and Updating Data Resources**

To write a data resource to a resource file, send msgResWriteData to a resource file handle. To update a data resource, send msgResUpdateData to a resource file handle. Updating a data resource is like writing a data resource, except that the
update operation reads the resource file to determine the correct agent to use. You cannot send write or update messages to a resource file list, only to a resource file.

`msgResWriteData` and `msgResUpdateData` both take a pointer to a `RES_WRITE_DATA` structure as their argument. `RES_WRITE_DATA` includes the following fields (the data type of each field is shown in parentheses following its name):

- `resId` (RES_ID) The resource ID with which to write the resource.
- `pData` (p_UNKNOWN) The data to be written.
- `length` (U32) The length of the data (agent might compute this value).
- `agent` (UID) The agent to use in writing the data. Not used for `msgResUpdateData`.
- `pAgentData` (p_UNKNOWN) Agent-specific data, such as the index into a list resource.

### Reading an Object Resource

There are two messages that you can use to read object resources: `msgResReadObject` and `msgResReadObjectWithFlags`. Both messages search for and read an object resource from either a resource file handle or a resource file list. The difference is that `msgResReadObjectWithFlags` takes additional flags arguments, which it passes in `msgRestore` to the new object.

`msgResReadObject` and `msgResReadObjectWithFlags` both take a pointer to a `RES_READ_OBJECT` structure as their argument. `RES_READ_OBJECT` includes the following fields (the data type of each field is shown in parentheses following its name):

- `mode` (RES_READ_OBJ_MODE) Whether to read the object in once mode (don't read the object from file if already read) or many mode (allow each read to create its own copy of the object). Possible values are:
  - `resReadObjectOnce` Read once mode.
  - `resReadObjectMany` Read many mode.
- `resId` (RES_ID) The resource ID of the object resource to be read.
- `objectNew` (OBJECT_NEW) Before reading the object resource, this field must be initialized with a `msgNewDefaults` to clsObject. After reading the object resource, the object UID will be `objectNew.uid`.
- `sysFlags` (RES_SAVE_RESTORE_FLAGS) System-defined flags for `msgResReadObjectWithFlags`.
- `appFlags` (UI6) Application-defined flags for `msgResReadObjectWithFlags`.

When the message returns successfully, the `RES_READ_OBJECT` structure contains the UID of the newly created object in `objectNew.uid`. If you specified `resReadObjectOnce` and the object existed already, `objectNew.uid` contains the UID of the object created when the resource was first read, but the message does not result in a second copy of the object.
When you use `msgResReadObjectWithFlags`, you must be very careful in specifying the `sysFlags` and `appFlags` values. The values for `sysFlags` are specific to PenPoint, and are currently used only on copy operations. You can define specific `appFlags` for a class, but you must be careful that you only send the flags defined for a specific class when restoring objects of that class.

**Writing an Object Resource**

There are two messages that you can use to write object resources: `msgResWriteObject` and `msgResWriteObjectWithFlags`. Both messages write an object resource to a resource file; you cannot send these messages to a resource file list. The difference between the messages is that `msgResWriteObjectWithFlags` takes additional flags arguments, which it passes in `msgSave` to the object being written.

`msgResWriteObject` and `msgResWriteObjectWithFlags` both take a pointer to a `RES_WRITE_OBJECT` structure as their argument. `RES_WRITE_OBJECT` includes the following fields (the data type of each field is shown in parentheses following its name):

- `mode` (`RES_WRITE_OBJ_MODE`) Whether to write the object in once mode or many mode. Possible values are:
  - `resWriteObjectOnce` Write once mode. Don’t write the object resource to the file if it has already been written.
  - `resWriteObjectMany` Write many mode. Write the object resource to the file, creating it if it isn’t already in the file, or overwriting the old version if it is already in the file.
- `resId` (`RES_ID`) The resource ID with which to write the object resource.
- `object` (`UID`) The UID of the object to be written.
- `sysFlags` (`RES_SAVE_RESTORE_FLAGS`) System-defined flags for `msgResWriteObjectWithFlags`.
- `appFlags` (`U16`) Application-defined flags for `msgResWriteObjectWithFlags`.

When you use `msgResWriteObjectWithFlags`, you must be very careful in specifying the `sysFlags` and `appFlags` values. The values for `sysFlags` are specific to PenPoint, and are currently only used on copy operations. You can define specific `appFlags` for a class, but you must be careful that you send only the flags defined for a specific class when restoring objects of that class (the flags are passed along with recursive writes, which may end up at objects of a different class than the top-level object; use `msgResWriteObjectWithFlags` with care.

**Enumerating Resources**

To create an array of all resource IDs in a resource file or resource file list that match a particular selection criteria, send `msgResEnumResources` to a resource file handle or a resource file list.
msgResEnumResources takes a pointer to a RES_ENUM structure as its argument. RES_ENUM includes the following fields (the data type of each field is shown in parentheses following its name):

- **match** (UID)  The search key, for example an agent or class.
- **mode** (RES_ENUM_MODE)  How to filter the search. This can be one of the following values:
  - resEnumAll (the default)  Find next resource in file.
  - resEnumByResIdClass  Find next resource in file whose resource ID administrated part matches match.
  - resEnumByObjectClass  Find next resource in file whose object class matches match.
  - resEnumByObjectUID  Find next resource in file whose object UID matches match.
  - resEnumByAgent  Find next resource in file whose agent matches match.
- **pResId** (p_RES_ID)  A pointer to the array of enumerated resource IDs.
- **pResFile** (p_RES_FILE)  A pointer to the array of file handles corresponding to each entry of pResId.
- **max** (UI6)  Maximum number of entries allocated for pResId and pResFile arrays.
- **count** (UI6)  The number of entries in the arrays. When passed in with a message, count specifies the number of entries requested. When passed back out, count specifies the number of entries actually retrieved.

When the message completes, pResId[] contains the resource IDs that match the criterion, and pResFile[] contains the resource file handles for the corresponding entries in pResId[]. If the final count is larger than max, the resource manager allocates heap to accommodate the additional resource IDs and resource file handles. You can tell that heap was allocated when the arrays indicated by pResId and pResFile are different from their original values when the message returns. If the resource manager allocated heap, you must free the heap when you are done.

If you are concerned about memory requirements, you can request the message to return only a few resource IDs at a time. To do so, you keep the pResId and pResFile arrays small and specify a small count value on the first call to msgResEnumResources. On subsequent calls, you specify small count values, and you must OR in resEnumNext in the mode field and repeat the data returned in cache.

**Deleting a Resource**

To mark a resource as deleted, send msgResDeleteResource to a resource file handle. msgResDeleteResource takes as its argument a pointer to the RES_ID that identifies the resource to be marked.

msgResDeleteResource marks a resource as deleted. Resources marked as deleted are not actually removed from the resource file until the resource file compacts in response to msgResCompact. You can send msgResCompact yourself to force a
resource file to compact, or it may compact automatically depending on the mode settings at msgNew time (see “Creating a Resource File Handle” earlier in this chapter).

**Compacting and Flushing Resource Files**

To explicitly compact a resource file, send msgResCompact to the resource file handle. This deletes all resources in the resource file that are marked as deleted. msgResCompact doesn’t take any arguments.

Flushing the resource file has the effect of clearing the internal table maintained by the resource manager for the file. To do this, send msgResFlush to the resource file handle. The message doesn’t require any arguments. This internal table keeps track of the resource IDs, the UIDs associated with object resources, and maintains the flags that indicate when an object resource has been written.

In addition to clearing the internal table associated with a resource file, msgResFlush flushes any buffered file output to disk.

Essentially, a flush restores the resource file to the state it was in when you opened it. This is particularly important if you are reading an object resource with resReadObjectOnce mode, because the resource manager uses the table to determine whether it has read the object before.

If you want to flush buffered output without flushing the resource file table, you can send the clsFileHandle message msgFSFlush.

**Resource Agents**

As mentioned in Chapter 103, data resources are usually managed by agents that interpret the bytes of a data resource in a special way. PenPoint comes with three data resource agents: one that interprets the bytes simply as bytes, a second that interprets them as NULL-terminated strings, and a third that interprets them as arrays of NULL-terminated strings. This section explains how the resource manager interacts with resource agents and how to write your own agents to handle special types of data resource.

**Reading and Writing Data Resources**

When you send msgResWriteData to a resource file handle, you specify the agent that will write the data. The resource manager finds a suitable location to store the resource and writes the header information and the agent's UID in the resource file at that location. The resource manager then sends msgResAgent to the agent, and the agent handles the rest of the write.

When you send msgResReadData or msgResUpdateData to a resource file handle or resource file list, the message determines the correct agent to use by looking at what msgResWriteData stored in the file. The resource manager then sends msgResAgent to the agent, and the agent handles the rest of the read or update.
Writing Your Own Agents

A resource agent is a well-known object that responds to `msgResAgent`. The easiest way to create your own agent is to create a single instance of the agent class and make the UID of that instance well-known.

The agent needs to handle just one message: `msgResAgent`. As discussed above, the resource manager will send `msgResAgent` to the agent during any `msgResReadData`, `msgResWriteData`, or `msgResUpdateData` that specifies the agent. `msgResAgent` takes a pointer to a RES_AGENT structure as its argument. RES_AGENT includes the following fields (the data type of each field is shown in parentheses following its name):

- file (RES_FILE) The file containing the resource.
- length (U32) Length of the resource entry, in bytes.
- msg (MESSAGE) The original message (`msgResReadData`, `msgResWriteData`, or `msgResUpdateData`).
- pArgs (p_UNKNOWN) The argument passed with the original message.
- sysVersion (U16) The minimum system version compatible with the resource, to record during write operations.

When you handle `msgResAgent`, you must examine `msg` to determine what action you will take. You must use `dsStream` messages to read and write data. Do not use the `stdio` library functions. `dsStream` messages update the file pointer maintained by `dsResFile`. If the file pointer is not positioned where the resource manager expects, unpredictable results occur, potentially including loss of data. On the same note, do not send seek or rewind messages to the file handle, as these also alter the file pointer.

The arguments to `msgResReadData` and `msgResUpdateData` include a pointer to other data (`pAgentData`). You can use this information to locate specific information within a resource. For example, if your agent stores a number of variable length values, you can allow clients to specify an index to one of the values in `pAgentData`. When you receive `msgResAgent`, you can use the `pAgentData` to locate the specific value that the client wants.
Chapter 105 / Defining Resources with the C Language

You can define data resources in the C language. This section describes the specific format of static data declarations that you use to define resources. At the end of this chapter is a short example of a resource source file.

Resource definition files contain C code that define the data resource. To compile the code into a resource file, you follow these steps:

1. Include a special header file (RESCMPLR.H) into your code.
2. Define each data resource as an RC_INPUT data structure.
3. Define an array of pointers (resInput) to the data resources.
4. Use the RC command, described in Chapter 106 to compile the C code into a resource file.

Resource Source File Organization

Every resource definition file has pretty much the same structure, varying mainly in the number of resources and the data they contain. Example 105-1 shows the source for a resource file that contains a single NULL-terminated string resource.

Example 105-1

A Tiny Resource Definition File

This example shows the source for a resource definition file that defines a single resource, a NULL-terminated string reading “Hello, World.” More complex resource definitions would simply include more RC_INPUT definitions and a larger resInput[] array to hold them. RESCMPLR.H provides examples for data types other than strings.

```c
#ifndef RESCMPLR_INCLUDED
#define RESCMPLR_INCLUDED
#include <rescmplr.h>
#endif
#include mydefine.h // defines myResId to MakeWknResId(clsExample, 1)

static RC_INPUT myResource = {
  myResId, // Resource ID for the resource
  "Hello, World.", // Pointer to the resource data
  0, // Data length; 0 means let compiler figure it out
  resStringAgent, // Resource agent for the resource
}

P_RC_INPUT resInput[] = {
  &myResource,
  pNull
};
```

The first non-comment lines must contain an #include <rescmplr.h> statement. You should also include any other header files that your resource definitions require, such as the header file that defines the resource IDs.
Follow the \#define statements with the resource definitions. Each resource
definition is a static struct for which you specify the elements. The elements of the
struct depend on the resource data type. The structs and the resource data types
are described in the next section.

Finally, you define an array named resInput that contains pointers to each of the
static struct resource definitions. resInput is an exported variable that is expected
by the resource compiler. The array ends with a null entry.

### Resource Definitions

The definition for each resource in the source file has the following common
structure:

```c
static RC_INPUT resource-label = {
    resId; // the resource ID (RES_ID)
    pData; // points to the resource data (P_UNKNOWN)
    dataLen; // length of the resource data, in bytes (U16)
    agent; // the resource agent, usually resDefaultResAgent (OBJECT)
    minSysVersion; // min sys version for resource (UI6)
    objectData; // false means the resource is a data resource (BOOLEAN)
    pAgentWriteProc; // pNull, unless supplying routine (P_AGENT_WRITE)
    pAgentWriteData; // usually pNull (P_UNKNOWN)
};
```

All of the fields after agent default to zero, which is normally a correct value, so
you shouldn't have to set them explicitly.

### Example

Example 105-2, taken from TTTQHELP.RC in the \PENPOINT\SDK\SAMPLE\TTT
directory of the PenPoint SDK distribution, shows the definition of four Quick
Help resources. Quick Help resources determine the text that appears in the
Quick Help window when you tap the pen on a window. Quick Help is described
in more detail in Part 9: Utility Classes.

Note that in ANSI C, two consecutive quoted strings are treated as a single quoted
string. This lets you break a long string into several parts, and this technique is
used in the following example.

```c
#define RESCMPLR_INCLUDED
#include <rescmplr.h>
#endif

#define QHELP_INCLUDED
#include <qhelp.h>
#endif

#define TTVIEW_INCLUDED
#include "tttview.h"
#endif
```

---

Example 105-2

Defining Quick Help Resources

continued
Example 105-2 (continued)

// Quick Help string for ttt's option card.
//
static CHAR tttOptionString[] = {
    // Title for the quick help window
    "TTT Card\n"
    // Quick help text
    "Use this option card to change the thickness of the lines "
    "on the Tic-Tac-Toe board."
};

// Quick Help string for the line thickness control in ttt's option card.
//
static CHAR tttLineThicknessString[] = {
    // Title for the quick help window
    "Line Thickness\n"
    // Quick help text
    "Change the line thickness by writing in a number from 1-9."
};

// Quick Help string for the view.
//
static CHAR tttViewString[] = {
    // Title for the quick help window
    "Tic-Tac-Toe\n"
    // Quick help text
    "The Tic-Tac-Toe window lets you to make X's and O's in a Tic-Tac-Toe "
    "grid. You can write X's and O's and make move, copy "
    "and pigtail delete gestures.\n\n"
    "It does not recognize a completed game, either tied or won.\n"
    "To clear the game and start again, tap Select All in the Edit menu, "
    "then tap Delete."
};

// Define the quick help resource for the view.
static _P_RC_TAGGED_STRING tttViewQHelpStrings[] = {
    tagCardLineThickness, tttOptionString,
    tagTttQHelpForLineCtrl, tttLineThicknessString,
    tagTttQHelpForView, tttViewString,
    pNull
};

static _RC_INPUT tttViewHelp = {
    MakeListResId(clsTttView, resGrpQhelp, 0),
    tttViewQHelpStrings, // Name of the string array
    0,
    resTaggedStringArrayResAgent // Use string array resource agent
};

// The glue that ties everything together -- resInput.

// resInput is an exported variable that the resource compiler expects.
// Each element is a pointer to a structure describing the next resource.
// The list is terminated with a null pointer.
_P_RC_INPUT resInput [] = {
    &tttViewHelp, // this is the one defined in this example
    // any other resource pointers would go here
    pNull
};
Chapter 106 / Compiling Resources

When you have created the resource source file, you must compile it with the resource compiler to create the resource file. The resource compiler performs four tasks:

1. Creates a C file that #includes your resource definition file.
2. Uses the C compiler to compile your resource source file.
3. Links the resulting object file with RESCMPLR.LIB.
4. Runs the resulting executable file to produce a resource file.
5. Deletes its intermediate files.

Running the Resource Compiler

The resource compiler is the MS-DOS executable file RC.EXE. RC has the following syntax:


The resInputFile specifies the input source file. The input file can be either a resource source file (with the extension .C or .RC) or a previously compiled object file (with the extension .OBJ). If you omit the extension from the file name, RC looks for files with .C and .RC extensions.

The optional resOutputFile specifies the name of the final resource file. If you omit resOutputFile, RC creates a file with the same name as the resInputFile, but with a .RES extension.

The optional /V flag directs RC to give verbose status messages. The verbose messages:

- Tell you the commands that RC executes.
- Report all of its steps.
- Give you information about the resources that it compiled.

The optional /D flag directs RC to report when its spawned processes complete or fail. The /D flag also turns on verbose status messages and prevents RC from deleting intermediate files.

RC creates a resource file (either using the name of the source file or using the output name specified). If the resource file already exists, RC appends the new resources to the resource file. If the resources already exist in the file, RC marks those resources as deleted and appends the new resources. To remove deleted resources from a resource file, use the RESAPPND utility described below.

To review the contents of a resource file, use the RESDUMP utility, also described below.
RC cleans up after itself by deleting any intermediate files that it created (.OBJ and .EXE). However, if the input file is an object file, RC does not delete that .OBJ file.

The **RESAPPND Utility**

The RESAPPND utility allows you to append all the resources from one resource file to another resource file. This allows you to create and compile a number of small resources and append those resources to one large resource file.

The syntax for RESAPPND is:

\[
\text{RESAPPND sourceResFile destResFile}
\]

The `sourceResFile` is the resource file that contains the resources that will be appended. The `destResFile` is the file to which the source resources will be appended.

While appending resources, RESAPPND ignores any resources that are marked as deleted. Thus, you can use RESAPPND to remove resources marked as deleted from a resource file. To do this, use RESAPPND to append the resources in a file to a new, temporary file. Then rename the temporary file to the name of the original resource file.

For example:

```
C:\> REM myres.res contains deleted resources
C:\> resappnd myres.res temp.res
C:\> copy temp.res myres.res
   1 File(s) Copied
C:\> del temp.res
```

The **RESDUMP Utility**

The RESDUMP utility allows you to display the contents of a resource file. The information displayed by RESDUMP includes:

- The file header, which describes the class and resource file version information.
- The resource ID for each resource.
- The length of the data in each resource.
- The data in each resource.

The syntax for RESDUMP is:

\[
\text{RESDUMP resourceFile}
\]

`resourceFile` is the name of the resource file to be dumped.

If any of the resources are marked as deleted, RESDUMP identifies them as marked as deleted.
Chapter 107 / System Preferences

The system preferences are stored as resources in the preferences resource file. This chapter describes system preference resources and the notification messages that observers receive when the system preferences change.

Topics covered in this chapter include:

- System preference concepts, including their storage location and typical uses.
- The system preferences and what they do.
- Observing system preferences.

Concepts

On a new PenPoint installation, the preferences resource file is \PENPOINT\SYS\PREF\GENERIC, but the user can change which preferences resource file is current. No matter what the current preferences resource file, the PenPoint™ operating system maintains the SystemPreferences as a handle on the current preferences resource file.

System preferences describe system settings, such as the current system font, screen orientation, left- or right-handed operation, and so on. Usually a user modifies the system preferences from the Settings notebook. However, if an application needs to examine or, more rarely, modify system preferences, it can access the system preferences in the preferences resource file.

The system preference resources are identified by well-known resource IDs. The preferences and their resource IDs are listed in the next section. The class used in the administered portion of the system preference resource IDs is clsPreferences. If you add resources for other preferences, the class should be the class that created the preference.

Any request to read or write a preference forces a read or write to a preference resource file. This minimizes the amount of space required to store preferences.

Clients can get and set the preferences resources by sending msgResReadData, msgResWriteData, and msgResUpdateData to the well-known object clsPreferences (which is an instance of clsPreferences). When users want to change the system preferences, they open the Preferences section of the Settings notebook, which then communicates with clsPreferences to alter the system preferences.

The preferences resource file is stored in a well-known directory, managed by clsPreferences. If necessary, a client can supply an entirely new system preferences file, containing a different set of preferences.
When PenPoint is cold- or warm-booted, the SystemPreferences contains the set of preferences associated with the current preference set managed by the InstalledPreferences. If there is no current set of preferences at boot time, the SystemPreferences copies a generic set of preferences from \PENPOINT\SYS\PREF\GENERIC and makes that the system preferences file.

### The System Preferences

The following sections describe the standard system preferences, defined in PREFS.H. In each section, the preference is described along with its resource ID and the symbols that identify possible states for that preference.

Table 107-1 summarizes the standard preferences and their resource IDs. These preferences are defined in the header file PREFS.H.

<table>
<thead>
<tr>
<th>Preference</th>
<th>Resource ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>System font</td>
<td>prSystemFont</td>
</tr>
<tr>
<td>User font</td>
<td>prUserFont</td>
</tr>
<tr>
<td>Screen orientation</td>
<td>prOrientation</td>
</tr>
<tr>
<td>Hand preference</td>
<td>prHandPreference</td>
</tr>
<tr>
<td>Writing style</td>
<td>prWritingStyle</td>
</tr>
<tr>
<td>Handwriting timeout</td>
<td>prHWXTimeout</td>
</tr>
<tr>
<td>Press-hold timeout</td>
<td>prPenHoldTimeout</td>
</tr>
<tr>
<td>Gesture timeout</td>
<td>prGestureTimeout</td>
</tr>
<tr>
<td>Power management</td>
<td>prPowerManagement</td>
</tr>
<tr>
<td>Auto suspend timeout</td>
<td>tagPrAutoSuspend</td>
</tr>
<tr>
<td>Auto shutdown timeout</td>
<td>tagPrAutoShutdown</td>
</tr>
<tr>
<td>Floating allowed</td>
<td>prDocFloating</td>
</tr>
<tr>
<td>Zooming allowed</td>
<td>prDocZooming</td>
</tr>
<tr>
<td>Bell</td>
<td>prBell</td>
</tr>
<tr>
<td>Scroll margins</td>
<td>prScrollMargins</td>
</tr>
<tr>
<td>Input pad style</td>
<td>prInputPadStyle</td>
</tr>
<tr>
<td>Character box width</td>
<td>prCharBoxWidth</td>
</tr>
<tr>
<td>Character box height</td>
<td>prCharBoxHeight</td>
</tr>
<tr>
<td>Line height</td>
<td>prLineHeight</td>
</tr>
<tr>
<td>Pen cursor</td>
<td>prPenCursor</td>
</tr>
<tr>
<td>Time and date</td>
<td>prTime</td>
</tr>
<tr>
<td>Date format</td>
<td>prDateFormat</td>
</tr>
<tr>
<td>Time format</td>
<td>prTimeFormat</td>
</tr>
<tr>
<td>Display seconds</td>
<td>prTimeSeconds</td>
</tr>
<tr>
<td>Primary input device</td>
<td>prPrimaryInput</td>
</tr>
<tr>
<td>Unrecognized character</td>
<td>prUnrecCharacter</td>
</tr>
</tbody>
</table>
System and User Fonts

`prSystemFont` is the resource ID for the system font. `prUserFont` is the resource ID for the field or user font. Both of these resources affect the returned value from `PrefsSysFontInfo()`. Changing either of these resources with `msgResWriteData` will cause the entire system to layout after notification of observers, which degrades system performance. As a result, `clsPreferences` will compare this resource to its previous value to prevent layout and observer notification if the write did not change the value.

Both of these resources contain a `PREF_SYSTEM_FONT_INFO` structure that includes the following fields:

- **scale** A `U8` representing the font size in points.
- **sysFontId** A `U16` that identifies the font family (Courier, for example) of the system font.
- **userFontId** A `U16` that identifies the font family used to render translated user input.

See Part 3: Windows and Graphics of volume I for more information about font rendering under the ImagePoint™ imaging model.

Screen Orientation

`prOrientation` is the resource ID for the screen orientation preference. Changing this resource with `msgResWriteData` will cause the system to layout after notification of observers, expensive in terms of system performance. As a result, `clsPreferences` will compare this resource to the previous value to prevent layout and observer notification if the write did not change the value.

The screen orientation preference is a `P_U8` that can have one of the following values:

- `prPortrait` The long edge of the screen is vertical.
- `prLandscape` The long edge of the screen is horizontal.
- `prPortraitReversed` Similar to `prPortrait`, but rotated 180 degrees.
- `prLandscapeReversed` Similar to `prLandscape`, but rotated 180 degrees.

Hand Preference

`prHandPreference` is the resource ID for the preference that indicates whether the user is left-handed or right-handed. This affects aspects of the screen layout such as whether scroll margins appear along the right or left sides of windows.

Changing this resource with `msgResWriteData` will cause the system to layout after notification of observers, degrades system performance. As a result, `clsPreferences` will compare this resource to the previous value to prevent layout and observer notification if the write did not change the value. Reads and writes of this ID use a `P_U8` which can have one of two values:

- `prLeftHanded` User is left-handed.
- `prRightHanded` User is right handed.
Writing Style

prWritingStyle is the resource ID for the handwriting preference style. This preference indicates whether the user prefers to write in all capital letters or to use mixed upper and lower case. Reads and writes of this ID use a P_U8, whose possible values are:

prMixedCase  Mixed case writer.
prCapsOnly   All caps writer.

Handwriting Timeout

prHWXTimeout is the resource ID indicating the handwriting timeout, and is measured in 0.01 second increments. Reads and writes of this ID use a P_U16 indicating the number of 0.01-second increments from the time the pen stops moving to the time the handwriting translator begins to translate the strokes.

Press-Hold Timeout

prPenHoldTimeout is the resource ID for the press-hold timeout, and is measured in 0.01 second increments. Reads and writes of this ID use a P_U16 indicating the number of 0.01-second increments from the time the pen touches the screen to the time the gesture translator recognizes the gesture as a press-hold.

Gesture Timeout

prGestureTimeout is the resource ID for the gesture timeout, and is measured in 0.01 second increments. Reads and writes of this ID use a P_U16 indicating the number of 0.01-second increments from the time the pen stops moving to the time the gesture translator captures and begins to translate the strokes.

Power Management

prPowerManagement is the resource ID that indicates whether the system should attempt to limit the computer's power consumption by turning off inactive devices. prPowerManagement is a P_U8 that can have one of two values:

prPowerManagementOff  No power management attempted.
prPowerManagementOn   Power management attempted.

Auto Suspend

tagPrAutoSuspend is the resource ID for automatic suspend timeout (the amount of idle time allowed before the machine goes into a power-saving suspended mode). Reads and writes of this ID use a P_U16, whose units are minutes. If tagPrAutoSuspend is 0, the machine will not automatically suspend itself. Machines that do not support automatic suspend do not use the auto shutdown preference value. Instead, they use the auto suspend preference value as an auto shutdown timeout.
The System Preferences

**Auto Shutdown**

Tag `PrAutoShutdown` is the resource ID for automatic shutdown timeout (the amount of idle time before the system automatically shuts itself down). Reads and writes of this ID use a P_U16, whose units are hundredths of hours. If the value is 0, the machine will not shut itself down automatically.

Machines that do not support auto suspend use the automatic suspend timeout preference as the automatic shutdown timeout.

**Floating Allowed**

`prDocFloating` is the resource ID that indicates whether documents in notebooks can be floated (displayed in their own window, independent of the notebook frame). Reads and writes of this ID use a P_U8 which can have one of two values:

- `prDocFloatingOff` Document floating not allowed.
- `prDocFloatingOn` Document floating allowed.

**Zooming Allowed**

`prDocZooming` is the resource ID that indicates whether floating documents can be zoomed (expanded to fill most of the display). Reads and writes of this ID use a P_U8 which can have one of two values:

- `prDocZoomingOff` Document zooming not allowed.
- `prDocZoomingOn` Document zooming allowed.

**Bell**

`prBell` is the resource ID for the preference that indicates whether to sound the warning bell to gain the user’s attention. It reads and writes a P_U8, whose possible values are:

- `prBellOn` Sound the bell to indicate a warning condition.
- `prBellOff` Don’t sound the bell under any circumstances.

**Scroll Margin Style**

`prScrollMargins` is the resource ID that determines whether scrolling windows use “full” or “light” scroll bars.

Changing this resource with `msgResWriteData` will cause the system to layout after notification of observers, which degrades system performance. As a result, `clsPreferences` will compare this resource to the previous value to prevent layout and observer notification if the write did not change the value.

Reads and writes of this ID use a P_U8 which can have one of two values:

- `prScrollMarginsFull` Use full scroll margins.
- `prScrollMarginsLight` Use light scroll margins.
Input Pad Style

prInputPadStyle is the resource ID indicating the preferred style of handwriting pads. Reads and writes of this ID use a P_U8, which can have one of three values:

- prInputPadStyleBoxed  Input pads are boxed.
- prInputPadStyleRuled  Input pads are ruled.
- prInputPadStyleRuledAndBoxed  Input pads start ruled, then go to boxed for editing.

This preference affects only input pads created after the preference is changed.

Character Box Width

prCharBoxWidth is the resource ID indicating the width of character boxes for boxed writing fields. Reads and writes of this ID use a P_U8, which indicates the width of the box in points (a point is 1/72 inch). This preference affects only character boxes created after the preference is changed.

Character Box Height

prCharBoxHeight is the resource ID indicating the height of character boxes for boxed writing fields. Reads and writes of this ID use a P_U8, which indicates the height of the box in points (a point is 1/72 inch). This preference affects only character boxes created after the preference is changed.

Line Height

prLineHeight is the resource ID for the height of lines in ruled edit pads. Reads and writes of this ID use a P_U16, indicating 0.01 inch increments. Changing this preference only affects subsequently created ruled pads.

Pen Cursor

prPenCursor is the resource ID that determines whether the system displays a pen cursor (mainly useful for public demonstrations where an image of the screen in projected). Reads and writes of this ID use a P_U8 which can have one of two values:

- prPenCursorOff  Do not display pen cursor.
- prPenCursorOn  Display pen cursor.

Time and Date

prTime is the resource ID for the system time (including the date). Reads and writes of this ID use a P_PREF_TIME_INFO containing the current time information. The PREF_TIME_INFO structure includes the following fields:

- dateTime  An OS_DATE_TIME containing the date and time (see OS.H for a more detailed description of the OS_DATE_TIME structure).
- mode  A PREF_TIME_MODE determining which parts of dateTime to write.
**Date Format**

`prDateFormat` is the resource ID for the date format. This preference will affect the format of the string returned from `PrefsDateToString()`. Reads and writes use a `P_U8`, whose possible values are:

- `prDateMDYFull` Example: January 15, 1990
- `prDateMDYAbbre` Example: Jan. 15, 1990
- `prDateMDYSlash` Example: 1/15/90
- `prDateMDYHyphe` Example: 11590
- `prDateMDYDot` Example: 1.15.90
- `prDateDMYFull` Example: 15 January 1990
- `prDateDMYAbbre` Example: 15 Jan. 1990
- `prDateDMYSlash` Example: 15/1/90
- `prDateDMYHyphe` Example: 15190
- `prDateDMYDot` Example: 15.1.90.

**Time Format**

`prTimeFormat` is the resource ID for the preferred time format (military, 24-hour time or civilian, 12-hour time). This preference will affect the returned string from `PrefsTimeToString()`. Reads and writes of this ID use a `P_U8`, which can have one of two values:

- `prTime12Hour` Display time in 12-hour format, including an A.M. or P.M. indicator.
- `prTime24Hour` Display time in 24-hour format.

**Display Seconds**

`prTimeSeconds` is the resource ID indicating whether to show seconds in clock displays. This preference will affect the returned string from `PrefsTimeToString()`. Reads and writes of this ID use a `P_U8`, which can have one of two meanings:

- `prTimeSecondsDisplay` Show seconds in time displays.
- `prTimeSecondsOff` Don't show seconds in time displays.

**Primary Input Device**

`prPrimaryInput` is the resource ID defining the primary input device (pen or keyboard). Reads and writes of this ID use a `P_U8` which can have one of two values:

- `prPrimaryInputPen` The pen is the primary input device.
- `prPrimaryInputKbd` The keyboard is the primary input device.
Unrecognized Character

prUnrecCharacter is the resource ID used for the unrecognized character glyph (the glyph shown in place of a handwritten character that the handwriting translation algorithm did not recognize). Reads and writes of this ID use a P_U8 which can have one of two values:

- prUnrecCharacterQuestion: Use a circled question mark to represent unrecognized characters.
- prUnrecCharacterUnder: Use an underscore character to represent unrecognized characters.

Preference Change Notification

You can make yourself an observer of the system preferences by sending msgAddObserver to the SystemPreferences. All applications observe the system preferences by default.

When the system preferences change, all observers of the system preferences receive msgPrefsPreferenceChanged. When you receive msgPrefsPreferenceChanged, the message sends you a pointer to a PREF_CHANGED structure. The structure contains:

- manager: The UID of the object that sent the notification. This is usually the SystemPreferences.
- prefID: The resource ID of the preference that changed.

If you need to know the details of the change, you can send msgResReadData to the SystemPreferences to get the new value for prefID.
Part 12 /
Installation API
### Chapter 108 / Introduction

- Overview 108.1 373
- Organization of This Part 108.2 373
- Other Sources of Information 108.3 374

### Chapter 109 / Installation Concepts

- Installation Managers 109.1 375
- Installation Process 109.2 376
- Installing Applications 109.3 377
- Installing Fonts and Handwriting Prototypes 109.4 378
- Service Installation 109.5 378
- The Installation Classes 109.6 379
  - The Installation Managers 109.6.1 379
  - The Application Monitor 109.6.2 380
  - Auxiliary Notebooks 109.6.3 380

### Chapter 110 / PenPoint File Organization

- Reasons for a Required Organization 110.1 381
- PenPoint Directory Concepts 110.2 382
  - Other Directories 110.2.1 382
- The Organization 110.3 382
  - General Structure 110.3.1 384
  - The PenPoint Directory 110.3.2 384
  - System Distribution 110.3.3 385
  - Installable Entities 110.3.4 386
  - Installable Applications 110.3.5 386
  - Installable Services 110.3.6 387
  - The Run-Time System 110.3.7 387
  - SDK Distribution 110.3.8 389
- Your Own Internal Development 110.4 389
- Organization of Distribution Volumes 110.5 390
  - PENPOINT.DIR Files 110.5.1 390
  - A Quick Look at STAMP 110.5.2 390
  - Application Directories 110.5.3 391
  - Service Directories 110.5.4 395
  - The Quick Installer 110.5.5 397
  - Multiple Applications and Multiple Volumes 110.5.6 398
  - Upgrading 110.5.7 398

### Chapter 111 / Dynamic Link Libraries

- References to DLL Files 111.1 399
- DLL File Issues 111.2 400
- Identifying DLLs 111.3 400
- DLC Files 111.4 401
- Sharing DLL Files 111.5 401
- Versions 111.6 402
- DLL Processes 111.7 402
- Operating System DLL Files and Versions 111.8 403
- DLL Files and MAKE Files 111.9 403

### Chapter 112 / Installation Managers

- Installer Concepts 112.1 405
  - Installers 112.1.1 406
  - Installed Item Database 112.1.2 406
  - Controlling Items 112.1.3 406
- Observing Installation Managers 112.2 407
  - Handling msgIMNameChanged 112.2.1 408
  - Handling msgIMCurrentChanged 112.2.2 408
  - Handling msgIMInUseChanged 112.2.3 408
  - Handling msgIMModifiedChanged 112.2.4 408
  - Handling msgIMInstalled 112.2.5 409
  - Handling msgIMDeinstalled 112.2.6 409
- Using clsInstallMgr Messages 112.3 409
  - Managing Installable-Item Managers 112.3.1 410
  - Managing Installable Items 112.3.2 411
  - Altering Installable Item Attributes 112.3.3 412
  - Getting Information About an Installable Item 112.3.4 413
  - Advanced clsInstallMgr Topics 112.4 414
    - Using the Semaphore 112.4.1 414
- Code Installation Manager 112.5 414
  - Installing an Application or Service 112.5.1 415
  - Application Installation Manager 112.5.2 415
  - Service Installation Manager 112.5.3 416
- Font Installation Manager 112.6 416
  - Font Identification 112.6.1 416
  - clsFontInstallMgr Messages 112.6.2 417
    - Getting and Setting a Font's ID 112.6.3 418
    - Finding a Font Handle 112.6.4 418
    - Getting a Font Name 112.6.5 418
    - Getting a List of Installed Fonts 112.6.6 418
Chapter 113 / The Auxiliary Notebook Manager

Auxiliary Notebook Concepts
The Auxiliary Notebooks
Auxiliary Notebooks and the File System
Back Up Considerations

Auxiliary Notebook Manager Messages

Generalized Auxiliary Notebook Manager Messages
Opening an Auxiliary Notebook
Getting the Path to an Auxiliary Notebook

Specialized Auxiliary Notebook Manager Messages
Creating Auxiliary Notebook Sections
Creating Auxiliary Notebook Documents
Moving and Copying Documents to an Auxiliary Notebook
Deleting an Auxiliary Notebook Section or Document

Modifying the Stationery Menu
Adding a Document to the Stationery Menu
Removing a Document to the Stationery Menu

Chapter 114 / The System Class

Concepts
Booting Sequence
File System Paths

The System Messages
Boot Progress Messages
System Directory Messages

List of Figures
110-1 PenPoint Volume Structure
110-2 Creating a Quick Install Disk

List of Tables
112-1 clsInstallMgr Notification Messages
112-2 clsInstallMgr Messages
112-3 clsCodeInstallMgr Messages
112-4 clsAppInstallMgr Messages
112-5 clsServiceInstallMgr Messages
112-6 clsFontInstallMgr Messages
113-1 Auxiliary Notebooks
113-2 Auxiliary Notebook Manager Messages
114-1 Boot Sequence Symbols
114-2 File System Path Constants
114-3 clsSystem Messages
Chapter 108 / Introduction

This part describes the APIs used to install software in the PenPoint™ operating system. In describing the installation API, we also describe the organization of the files under PenPoint, the relationship of executable and DLL files, and the initialization files.

Overview

Most of the work that an application developer has to do is to make sure that the application directory is structured correctly and that the .DLC file is correct. If these things are attended to, the application installer takes care of the rest of the work.

The people who are interested in anything in this part beyond PenPoint directory organization and DLL file information are OEMs who want to turn off configuration, stationery, and installation; and people who create device drivers and other installable entities, in addition to fonts, handwriting prototypes, and applications. These people will have to understand how the auxiliary notebook manager works, in addition to the install manager.

Organization of This Part

This chapter introduces the topic of installation and describes the organization of the part.

Chapter 109, Installation Concepts, introduces the concepts needed to understand the rest of the part, including a brief overview of installation from the user's perspective and an internal description of installation.

Chapter 110, PenPoint File Organization, describes the directory structure used in PenPoint volumes and describes how to create application or service distribution volumes.

Chapter 111, Dynamic Link Libraries, describes the dynamic link library (DLL) files used by PenPoint and how application installation handles issues such as shared DLL files and version control.

Chapter 112, Installation Managers, describes the installation manager (clsInstallMgr), the application installation manager (clsAppInstallMgr), the font installation manager (clsFontInstallMgr), and other installation managers.

Chapter 113, The Auxiliary Notebook Manager, describes clsAuxNotebookMgr and its descendents.

Chapter 114, The System Class, describes clsSystem, which provides information about booting and other system-wide information.
Other Sources of Information

For a description of the application monitor class (clsAppMon), see Part 2: PenPoint Application Framework.
Chapter 109 / Installation Concepts

This chapter describes concepts related to installation. After describing the application installation process from the user's perspective, we describe how application installation works internally. This discussion leads to a description of the classes used in installation.

The installation process is essentially similar for applications, fonts, handwriting prototypes, device drivers and other objects.

When you distribute an application to users, you ship it on **application distribution diskettes**. The file structure on these diskettes is similar to the PenPoint™ volume structure in the boot volume and in your application development system. (The file structure is explained in Chapter 110, PenPoint File Organization.)

### Installation Managers

An **installable item** is a collection of data that represents a specific object or group of objects used by PenPoint applications. Typical examples of installable items are applications, services, fonts, handwriting prototypes, device drivers, and dictionaries. As a rule, an installable item isn't vital to the operation the PenPoint operating system, but adds capabilities to it. Installable items are grouped into sets of similar installable items.

**Installation Managers** provide applications with the facilities needed to manage installable items. All installation managers are subclasses of `clsInstallMgr`.

An **installable manager** is an instance of `clsInstallMgr` that manages a set of similar items. Usually an installable manager is a well-known global object (called `theInstalledThing`), so that all applications can easily access the set of installable items.

Each installable item managed by an install manager has these traits:

- A name.
- An attribute that indicates whether it has been modified.
- An attribute that indicates whether it is the current item.

The name of an installable item can be between 1 and 31 characters. Any character is valid in the component name except backslash (`\`) and null (ASCII 0).

These are the same naming conventions that apply to node names in the PenPoint file system.

The installable manager can identify one item as the current item. By identifying a current item, the installable manager can quickly return a specific item to clients.
that request it. Additionally, the clients do not have to keep track of which is the current item.

In managing a set of installable items, instances of clsInstallMgr can:

- List all items in a set of items.
- Install a item.
- Deinstall a item.
- Get and set the current item.

The installable manager uses the PenPoint file system to implement much of its behavior. An instance of clsInstallMgr indicates a item directory in the RAM volume where a set of items is stored. clsInstallMgr creates a file handle or directory handle for each of the items in the item directory. If the item is a file, the file system uses a file node; if the item is a directory, the file system uses a directory node. The file or directory has the same name as the item it contains.

When a client requests a list of items, the item manager returns a list of all handles in the item directory.

The attributes (in use, modified, current, deinstalled) are saved as file system attributes. To modify these attributes, you should use the clsInstallMgr messages, rather than the file system attribute messages. The main reason for this is that it allows the item to send notifications to observers when it changes.

If you subclass clsInstallMgr, you can modify the contents of these attributes, or you can add other attributes.

clsInstallMgr monitors the item directory and sends notification whenever a change occurs in the item directory. Notification messages indicate when an installable item is added, installed, deinstalled, updated, modified, or becomes current. Most notification messages end in “ed” (such as msgIMAdded, msgIMCurrentChanged, and so on).

**Installation Process**

If you have used PenPoint 1.0 for any length of time, the mechanics of installation will not be new to you. This is just a brief review. The most common installation you perform is application installation. Remember, however, that installation applies to any installable item.

Installation can be initiated in one of three ways:

- If the distribution diskette was marked for quick install, PenPoint invokes the appropriate quick-installer automatically when the user inserts the diskette.
- The user opens the Connections notebook to the disks page, selects the view for a particular type of installable item, and then taps on the Install... menu item.
- The user opens the settings notebook to one of the installed software pages and then taps on the Install... menu item.
The user selects an item and taps on the installed checkbox for that item.

If this is the first time the user has installed the item, the item being installed can present an option sheet that allows the user to specify the configuration for the item. The user can choose whether to load stationery and any option modules, such as help or accessories. An example of an accessory might be a report generator. When the user has chosen the configuration, the installer installs the item and all other relevant pieces.

If this isn’t the first time the user has installed the item, the installer locates the configuration for the item and installs the item and other relevant pieces.

**Installing Applications**

In brief, the file structure on the distribution volume and the PenPoint volume divides the installable item into pieces that are installed and managed by various system facilities. This is particularly true in the case of applications.

In addition to the application executable file, an application can include help templates, stationery documents, tools documents, and other miscellaneous files. The application installer must install all these items.

The stationery documents are created by either you or the user and already contain some information (such as a letterhead, margins, predrawn figures and so on). The application installer loads the stationery documents into the Stationery notebook. The accessories documents are similar to stationery documents, except the installer loads the documents into Accessories.

**How Application Installation Works**

When the user selects an application and taps on the install checkbox, the application installer starts the installation process. The application installation manager is a system application. There is only one instance of the application installation manager (clsAppInstallIMgr) in the system; it has the well-known identifier theInstalledApps.

The application installation manager starts installation by:

1. Creating an application directory for the selected application.
2. Copying the application resource file from the distribution diskette to the new application directory.
3. Creating an attribute in the directory that specifies the application class.
4. Invoking OSPProgramInstall() to start instance 0 of the application and the application monitor.

OSPProgramInstall() copies the executable and DLL files into the loader database and starts the application. Chapter 111, DLL Files, describes how the installer locates DLL files.

Instance 0 of an application contains an application manager instance. The first thing that instance 0 of the application does is to call AppMonitorMain(), which fires up the
application monitor, an instance of clsAppMonitor. AppMonitorMain() is a part of PenPoint.

The application monitor drives the rest of the installation process. The application monitor:

1  Receives msgAppInit sent by AppMonitorMain().
2  If this is the first time the application has been installed, the application monitor puts up the property sheet so the user can specify the configuration. The results from the property sheet are stored in the application's resource file. PenPoint defines a default property sheet; the application can override the defaults.
3  If it isn't the first time, the application monitor searches for the application's resource file in the application distribution disk.
4  The application monitor uses the resource file to drive the rest of the installation.
5  Using the resource file, the application monitor loads all other portions of the application, such as help and any miscellaneous resources.

When the application monitor finishes its work and returns from msgAppInit, the installation process is complete; OSPProgramInstall() completes and the application is installed and ready to create instances.

The application monitor does not go away, however. The application monitor maintains the identity of its application class. When requested, the application monitor can perform actions such as deinstalling the application.

Installing Fonts and Handwriting Prototypes

Installing fonts and handwriting prototypes is similar to installing applications. The main difference is, of course, that fonts and handwriting prototypes have no instance 0, or related processes.

Both can be installed from either the disk view of the Connections notebook or the installed software section of the Settings notebook.

Font installation is controlled by theInstalledFonts, a unique instance of clsFontInstallMgr. Handwriting prototype installation is controlled by theInstalledHWX, a unique instance of clsHWXInstallMgr.

Service Installation

Service installation is similar to application installation, however, the services are DLLs—there are no EXEs to load.

The service installer starts installation by:

1  Creating a service directory for the selected service.
2  Creating an attribute in the directory that specifies the service class.
3  Invoking OSPProgramInstall() to start instance 0 of the service class.
OSProgramInstall() copies the service DLL files into the loader database and
starts the service class by invoking DLLMain(). DLLMain() is the service's
external entry point.

The first thing that a service's DLLMain() should do is invoke InitService(), a
function defined by clsService, that initializes the service class and searches the
service INST directory for service nodes.

A service node is a service state file or a directory that contains a service state file
and other information required by the service. If the autoCreate flag is specified,
InitService() creates instances for each of the service nodes.

### The Installation Classes

The PenPoint operating system defines a number of installation classes that assist
users in installing applications, fonts, and so on. You can often use these classes as
they are; you only need to subclass when you are defining a new installable entity
or removing functionality from users. The classes are:

- clsInstallMgr - The installation manager, subclasses of which control the
  overall aspects of installing applications, fonts, services, and so on.
- clsAppInstallMgr - An instance of handles application installation.
- clsFontInstallMgr - An instance of handles font installation.
- clsCodeInstallMgr - The code installation manager, which controls the
  particular details of installing applications and services.
- clsAppMonitor - The application monitor, instances of which run in the
  process 0 of each application. The application monitor handles installing
  and deinstalling applications.
- clsAuxNotebookMgr - The auxiliary notebook manager, which creates and
  maintains the sections in the auxiliary notebooks and moves, copies, and
  deletes documents within those notebooks.
- clsIniFileHandler - The initialization file handler, which reads and processes
  APP.INI and SERVICE.INI files. These files contain the paths to
  applications and services that are required to run another application
  or service.

The classes described here are described in much greater detail in chapters 112 and
113.

### The Installation Managers

The installation managers are all descendents of clsInstallMgr. Each installation
manager has only one instance, which is created at boot time, and is called
theInstalledXxxx (such as theInstalledApps and theInstalledFonts). clsInstallMgr
defines the common tasks that the installation managers perform.

The application installation manager is an instance of clsAppInstallMgr. It is
responsible for starting application installation. The application installation
manager creates the application directory and directs PenPoint to create instance 0 of the application.

The font installation manager is an instance of clsFontInstallMgr. It is responsible for locating and installing fonts.

**The Application Monitor**

The application monitor drives application installation. It locates or prompts the user for the configuration of the application and copies the necessary files into the boot volume of a running PenPoint computer.

Again, most application developers will have to subclass the application monitor for the particular requirements of your application. If you conform to the file organization specified in Chapter 110, the application monitor can locate and copy all necessary files.

**Auxiliary Notebooks**

All auxiliary notebooks are managed by theAuxNotebookMgr, which is defined by clsAuxNotebookMgr. This class defines the types of auxiliary notebooks that are available to the system. It performs the tasks required to create sections and documents in the notebooks and to move, copy, and delete documents within the notebooks.

The Configuration notebook, Stationery notebook, and the Help notebook are all auxiliary notebooks and, hence, are managed by theAuxNotebookMgr. Their APIs are through clsAuxNotebookMgr.

If you create another type of installable entity, you must use theAuxNotebookMgr to create a page for your entity in the configuration notebook.
Chapter 110 / PenPoint File Organization

This document describes the organization of files and directories on all PenPoint™ operating system volumes and how to organize application or service distribution volumes. These volumes include software distribution diskettes (for both applications and system software), hard disks, solid-state disks, and the PenPoint RAM file system.

If you intend to use the PenPoint-provided installer, your application distribution diskettes must conform to this organization.

Reasons for a Required Organization

The required organization is necessary to:

• Make a clean separation between the operating system and installable entities such as applications.
• Make a separation between the operating system files required at boot time and those used at run time.
• Provide an orderly development environment for application developers.
• Set up an environment where advanced filing facilities such as multiple notebooks can exist.
• Break out PenPoint’s private, internal development facilities from what we deliver to the outside world.

When volumes are organized in a well-known structure, the PenPoint Application Framework, the installer, and many other tools and utilities can easily locate specific files and directories.

PenPoint volumes are used for these six purposes; the organization of files and directories allows PenPoint to accommodate all six uses:

• Volumes used to distribute PenPoint operating system software to end users.
• A volume that contains a running PenPoint system.
• Volumes used to distribute other installable resources such as fonts and dictionaries.
• Volumes used to distribute installable application software to end users.
• Volumes used to distribute PenPoint development software to applications developers (SDK).
• Volumes that contains GO’s internal development environment.
PenPoint Directory Concepts

There are two basic areas on a PenPoint disk volume; the area administered by PenPoint (the \PENPOINT directory) and the area not administered by PenPoint. The \PENPOINT directory has a strict set of structural rules. GO discourages end-users from directly manipulating this area; it is meant for you—PenPoint application developers—and GO engineers.

The \PENPOINT directory includes subdirectories that can contain one or more of:

- The boot files for the PenPoint operating system.
- Files in use by an active PenPoint system.
- Installable items (applications, services, fonts, and so on).
- The PenPoint SDK software.

It is important to note that not all PenPoint volumes will have all the files and directories listed in this chapter. The following sections describes the union of all possible files and directories that can exist on a PenPoint volume.

For example, an application distribution diskette will contain a \PENPOINT\APP directory that contains an application directory. A PenPoint system distribution diskette will contain a \PENPOINT\BOOT directory for files loaded at boot time, and a \PENPOINT\SYS directory for the active system.

On the other hand, a PenPoint SDK distribution diskette probably won’t have a \PENPOINT\APP, \PENPOINT\BOOT, or \PENPOINT\SYS directory, but will have a \PENPOINT\SDK directory.

Other Directories

Any other directories on a PenPoint disk, outside of the \PENPOINT directory, have no structural rules. You can use these other directories to maintain your application development environment or store whatever data you choose. End-users can also use other directories to keep data. No PenPoint system software depends on structure in any other directory outside of the \PENPOINT directory.

Internally, GO uses another directory on PenPoint volumes, which allows us to maintain development efforts outside of the \PENPOINT directory. We will describe the organization of this \GO directory in later sections as a suggestion as to how you might organize a similar development environment.

The Organization

Figure 110-1 shows the overall organization of the PenPoint directory. Subsequent figures depict details of this organization. In all figures uppercase letters are used for directory names (such as \PENPOINT), lowercase letters are used for file names (such as penpoint.exe), and italic letters are used as place holders for file or directory names that are application or user dependent (such as A Volume Name).
Volume Name

BOOT (These files are loaded at boot time)
  - ppboot.exe (PC only)
  - boot.dlc
  - environ.ini
  - sysapp.ini
  - app.ini
  - win.ini (PC only)
  - service.ini
  - syscopy.ini
  - DLL
    - system dll files
    - dll (PC only)
    - debug system dll files

APP
  - NBAPP
  - NBTOC
  - A System Application
    - A System Application
  - DB (PC only)
    - DB.INI

DOC
  - HELP
    - NOTEBOOK

\ginstall

\PREFS (Installable preferences)
\FONT (Installable fonts)
\HWXPROM (Installable handwriting samples)
\APP (Installable applications)

\An Application
  - app.res
  - dll files
  - dlc file
  - exe files

\STATRY
  \Stationery Docs

\ACCESSRY
  \Tools Docs

\HELP
  \Help Docs

\MISC

\SERVICE (Installable services)
  \A Service
    - dll files
    - dlc file
    - service.res

\INST
  \Service State Node

\SYS (PenPoint run-time files)
  \penpoint.res
  \penpoint.idx

\HWXPROM

\FONT

\PREFS

\APP
  \An App’s Runtime
    - app.res
      \MISC

\SERVICE
  \An Installed Service
    \INST
      \Service State Node

\DOC
  \A Bookshelf
    \A Notebook
      \A Section
      \A Section
      \A Document

\SDK (SDK distribution)
  \INC
  \LIB
  \DLL
  \SAMPLE
  \UTIL
    \MAKE
    \TAG
    \DOS

\Your Company (Your own internal development)

34 \yourLIB
  \yourINC
  \A Project
  \A Project
General Structure

PenPoint volumes have this general structure:

```
\A Volume Name
    \PENPOINT...
    \Your Company...
```

As described above, the \PENPOINT directory contains files and directories for the PenPoint run-time system and the files and directories used to distribute software. It can also contain the files and directories of a running PenPoint system. Both GO and other application developers use the \PENPOINT directory to ship software to end users. Directories under \PENPOINT contain resource files, executable and DLL files, fonts, preferences, and handwriting prototypes.

The \Your Company directory contains the files and directories you use for your own internal development. It can include directories for each project. This directory is included in this chapter to point out a possible way to organize your own PenPoint development. The actual placement and organization of your own development directories is almost entirely up to you. However, we recommend that you do not store sources for your own development within the \PENPOINT directory (but you certainly will store your resulting applications in the \PENPOINT directory).

The PenPoint Directory

The \PENPOINT directory contains the files and directories used for software distribution and run-time system files. The \PENPOINT directory has the following structure. The directories with trailing ellipses (...) are described in detail in later sections.

```
\A Volume Name
    \PENPOINT
        \BOOT...
        \qinstall
        \PREFS
        \FONT
        \HWXPROT
        \APP...
        \SERVICE...
        \SYS...
        \SDK...
```

The BOOT directory contains the system files used when cold-loading the PenPoint computer.

The following four directories: FONT, HWXPROT, APP, and SERVICE all contain installable items. These directories are used in distribution volumes.

The FONT directory contains installable fonts.

The HWXPROT directory contains installable handwriting prototypes.

The APP directory contains a series of installable applications. Each application is stored in a directory that contains the files needed to install an application.

The SERVICE directory contains the installable services. Each service is stored in a directory that contains the files needed to install the service.
CHAPTER 110 / PENPOINT FILE ORGANIZATION

The SYS directory contains the run-time system files used in the RAM volume.

The SDK directory contains all Software Developer’s Kit (SDK) components that third-party developers and GO developers need to build an application. It includes the INC, LIB, SAMPLE, DLL, and UTIL directories.

**System Distribution**

The \PENPOINT\BOOT directory contains files and directories used to distribute and cold-load the PenPoint operating system. BOOT has this structure:

```
\BOOT (These files are loaded at boot time)
   \penpoint.os
   \ppboot.exe (PC only)
   \boot.dlc
   \environ.ini
   \sysapp.ini
   \app.ini
   \dll.ini (PC only)
   \service.ini
   \syscopy.ini
   \DLL
      \system dll files
      \DLL (PC only)
         \debug system dll files
   \APP
      \NBAPP
      \NBTOC
         \A System Application
         \A System Application
      \DB (PC only)
         \db.ini
   \DOC
   \HELP
   \NOTEBOOK
```

PENPOINT.OS is the actual operating system.

BOOT.DLC describes the system DLL and executable files. .DLC files are described in Chapter 111, Dynamic Link Libraries. The ENVIRON.INI file contains preferences and environment variables for running PenPoint.

SYSAPP.INI is a list of system application directories to be loaded at boot time.

APP.INI is an additional list of application directories to be loaded when the PenPoint computer is bootstrapped. Application developers and OEMs shouldn’t have to alter the directories listed in SYSAPP.INI, unless you are removing system applications.

If you want to make an application available at boot time, you should add its directory to APP.INI. When developing applications it is a good idea to add the application to APP.INI, rather than have to go through the installer each time you boot the PenPoint computer.

SERVICE.INI contains a list of service directories from which to load services at boot time.

SYSCOPY.INI lists the files and directories to copy into the running system at boot time. This includes both source and destination specifications.

The DLL directory contains system DLL files.
The APP directory contains system application directories, which contain .EXE files for system applications. These directories include DTAPP, which contains the desktop application, and BROWSER, which contains the browser application.

Installable Entities

Installable (and deinstallable) entities are contained in three directories under \PENPOINT. The installer knows to look in these directories for the installable entities: fonts, handwriting prototypes, and applications. A basic version of these directories is shipped with the standard operating system distribution diskette. Additional resources (such as new fonts) can be distributed separately.

The QINSTALL file is only present in installation volumes. It is an optional file that directs the installer to pop up a quick installer sheet when the user connects a volume containing QINSTALL to the PenPoint computer. QINSTALL is described in greater detail later in this chapter in the section titled “Quick Install.”

The PREFS directory contains installable system preferences. It always includes the standard set of preferences in the file named GENERIC.

The FONT directory contains installable fonts.

The HWXPROT directory contains handwriting prototypes. The handwriting prototypes shipped with PenPoint have been gathered from many people. End-users add further information to the HWXPROT directory when they run the Handwriting Training application; the application stores handwriting prototypes in this area.

The APP directory contains directories for installable applications and is described in the following section.

The SERVICE directory contains directories for installable services and is described after the section on installable applications.

Installable Applications

The \PENPOINT\APP directory contains directories for installable applications. Each application is in its own directory, named with the user-visible name of the application.
Each application directory contains:

- The application resource file (APP.RES).
- AW.INI and SERVICE.INI files that list all the other applications and services that must be loaded to run this application. When the installer finds that one of the required applications or services is not installed, it prompts the user to insert the correct distribution disk and installs that application or service.
- All the .DLL, .EXE, and .DLC files that an application will use (and that are not part of the base operating system).
- Subdirectories for stationery templates (STATNRY), help templates (HELP), and miscellaneous application files (MISC). The MISC subdirectory can contain files and directories common to all instances of an application, such as a common graphics logo.

Installable Services

The \\PENPOINT\SERVICE directory contains directories for installable services. Each service is in its own directory, named with the user-visible name of the service.

\SERVICE (Installable services)
  \ Service
    \ .dll files
    \ .dli file
    \ .SERVICE.res
    \ INST
      \ Service State Node
      \ Service State Node

Each service directory contains:

- A SERVICE.RES file that contains resources required for this service.
- The .DLL and .DLC files that a service needs that are not part of the base operating system.
- An INIT subdirectory that contains service state nodes. These state nodes can be either:
  - Files containing preconfigured instances of the service.
  - Directories containing preconfigured instances of the service and other resource or data files.
- A MISC subdirectory that contains other information used by the service.

The Run-Time System

The operating system makes use of the file system at run time to maintain the dynamic state of the system. The run-time file system is also a safe storage for data across a warm boot. All run-time files are stored in \\PENPOINT\SYS.
PENPOINT.RES is a resource file that contains resources for the running system.

The HWXPROT, FONT, PREFS, APP, and SERVICE directories contain the currently installed handwriting prototypes, fonts, preferences, applications, and services.

Each installed application has a directory under the APP directory, which contains the application resource file and any other global application data. The application's instance 0 runs out of this directory.

Each installed service has a directory (under the SERVICE directory), which contains an INIT directory. The INIT directory for a particular service contains state files for the service and directories as required by the service.

Clients can get lists of these installed resources by sending a message to the well-known UIDs: theInstalledFonts, theInstalledHWXProtos, theInstalledApps, and theInstalledServices.

The DOC directory contains the current notebook (NOTEBOOK, or whatever the user names it) and its sections. A directory that contains a notebook is organized into sections; each section is a directory in the notebook directory. Each section directory contains directories for each of the documents in that section (these directories are created by clsAppDir).

Each document directory always contains three files (in addition to any files created by the document's application). The three files are:

- **DOC.RES**, which is the document's copy of the application resource file.
- **DOCSTATE.RES**, which is the file to which the application files the document's instance data.
- **DOC.LNK**, which contains a list of UUIDs used to keep track of links to other documents (such as GoTo Buttons, embedded applications, bookmarks, and so on). This file is explained in Chapter 14, The Application Class, of Part 2: Application Framework.
Your Own Internal Development

When you develop your own PenPoint application, you will probably want to maintain your files in some proximity to the \PENPOINT directories. This discussion grows from PenPoint development practices used at GO Corporation.

The organization allows you to keep your development separate from the \PENPOINT\SDK directories that you use during development and the other \PENPOINT directories that you use during testing and debugging. It also allows you to keep your applications close enough to the \PENPOINT directories that you can build your applications such that your executables and DLL files can be stored directly into the \PENPOINT\APP directory.

While you do not have to structure your development in any particular way, this works well.

The \Your Company directory has the following organization:

```
\ \Volume Name
 \ \Your Company
 \ \ \yourLIB
 \ \ \yourINC
 \ \ \A Project
 \ \ \ \A Project
```
The yourLIB and yourINC subdirectories contain private .LIB and .H files used for your own components and applications that you use to develop system software and applications.

The individual project directories contain the files needed to build individual parts of PenPoint and applications.

**Organization of Distribution Volumes**

To make installation consistent and as simple as possible for end-users, PenPoint provides an application installer that performs application installation, deinstallation, deactivation, and a number of other tasks. The PenPoint Installer can automatically present an application or service for installation by reading an automatic install control file in the distribution volume.

If you use the PenPoint Installer, the files in your distribution volume must be organized according to the directories described in this chapter. This section supplements the general discussion of file organization by describing the contents of the distribution directories.

**PENPOINT.DIR Files**

File and directory names under PenPoint can be up to 32 upper and lowercase characters long and can include many special characters. However, DOS file and directory names are limited to eight uppercase characters plus a three character extension; the filename and extension can use a handful of special characters. In order to create the complex PenPoint names on DOS volumes, PenPoint stores extended information about files and directories in a file named PENPOINT.DIR. The PENPOINT.DIR file also contains file and directory attributes, which are used by PenPoint and applications.

There is one PENPOINT.DIR file per directory; each PENPOINT.DIR file contains information about the files and directories stored in that directory.

**A Quick Look at STAMP**

The STAMP utility allows you to mark a DOS file or directory with a PenPoint name. You also use STAMP to assign attributes to files or directories (you must STAMP all your application executable and DLL files to identify them as applications).

When you create long names or assign attributes to a file or directory, the PenPoint file system creates a PENPOINT.DIR file in the directory that contains the file or directory that you modified. PENPOINT.DIR contains an entry for each file that has a long name or other attributes. You can use the GDIR command (in PENPOINT\SDK\UTIL\DOS) to examine the contents of the PENPOINT.DIR file.

STAMP adds or modifies information in a PENPOINT.DIR file. If the PENPOINT.DIR file does not exist, STAMP creates a new one; if an entry for the file or directory being stamped doesn't exist, STAMP creates a new entry.
The STAMP utility (and other utilities that you can use to manipulate PENPOINT.DIR files) is documented in “DOS Utilities User’s Guide” in the PenPoint Development Tools volume. The PENPOINT.DIR file is described in detail in Part 7: File System in this volume.

**Application Directories**

To create an application distribution volume, create a \PENPOINT directory in the root of a volume. Under the PENPOINT directory, create an APP directory. Under \PENPOINT\APP, create a directory for your application.

The name of this directory is the name of your application that the user sees on the screen. The directory name must be unique in the entire world of PenPoint applications. The directory name can be a PenPoint name; use the STAMP utility to name it. However, the name or an application directory cannot be longer than 28 characters; the reason for this limit is explained below.

Thus, a distribution volume for an application named “Time Management” on a volume named “Time Management Dist” would have the following structure:

```
\Time Management Dist
 \PENPOINT
   \APP \Time Management
```

We will use this example through the rest of the discussion of application directories.

**Files in the Application Directory**

The application directory contains:

- An APPRES file, which contains the icons and any other resources used by your applications.
- A .DLC file (if your application uses DLLs).
- .EXE and .DLL files for your application.

When the Installer finds your application directory it first looks for a .DLC file. A .DLC file should have the same PenPoint name as the application directory, but with the extension .DLC. You need a .DLC file only when your application requires DLLs in addition to its executable file.

If the Installer doesn’t find a .DLC file, it looks for an executable file. The executable file should have the same name as the application directory, but with a .EXE extension.

The maximum size of a PenPoint file or directory name is 32 characters. Because these extensions (.DLC or .EXE) add four characters to the application name, the maximum size of an application name is 28 characters.
With the files in the application directory, the PENPOINT directory has this organization:

```
\PENPOINT
  \APP
    \\Time Management
       \app.res
       \database engine.dll
       \alarm clock.dll
       \Time Management.exe
       \Time Management.dlc
```

The TIME MANAGEMENT.DLC file contains:

```
TiManagementInc-Time Management_exe-V1(0)  Time Management.exe
TiManagementInc-database_engine_dll-V1(0)  database engine.dll
TiManagementInc-alarm_clock_dll-V1(0)      alarm clock.dll
```

All of the required executable and DLL files are in the application directory.

**Stationery**

To provide users with preconfigured stationery, create a directory named STATNRY under your application directory. For each stationery document that you provide, create a subdirectory in the STATNRY directory. The subdirectory should contain the file or files required by your application to store a document.

It is up to your application to understand the contents of the document subdirectory. Usually a document subdirectory contains a single APP.RES file, which contains the field instance data for a document.

When installing your application, the installer creates a section for your application in the Stationery notebook. If your application directory contains a STATNRY directory, the installer copies the document directories and their files to your application's section in the Stationery notebook. Each document directory copied by the installer is marked with the well-known UID of your application.

If your application directory does not contain a STATNRY directory, the installer creates a section for your application in the Stationery notebook. If the stationery bit is set in the application manager metrics flags, the installer creates an empty document from your application in that directory. (For more information on application manager metrics, see Part 2: Application Framework.

The name of each document subdirectory in the STATNRY directory becomes the name of a document in the Stationery notebook.

Use the STAMP utility to:

- Give a PenPoint name to the document subdirectories.
- Set the stationery menu attributes of the stationery document (whether the stationery appears in the stationery menu).
- Set the NoLoad attribute for the stationery document. If NoLoad is true, the installer will not load the stationery document into the Stationery notebook.
When we add the STATNRY directory, the PENPOINT directory on our application
distribution volume has this organization:

```
\PENPOINT
  \APP
  \Time Management
  \app.res
  \database engine.dll
  \alarm clock.dll
  \Time Management.exe
  \Time Management.dlc
  \STATNRY
  \Personal Scheduler
  \app.res
  \WorkGroup Scheduler
  \app.res
```

When the user opens the Stationery notebook, it contains two documents
belonging to the Time Management application: Personal Scheduler and
WorkGroup Scheduler.

**Accessories**

An accessory is similar to a piece of stationery. An accessory is simply an instance
of an application. Accessories appear in the Accessory notebook.

To provide accessories, create a directory named ACCESSRY under your application
directory. For each accessory that you provide, create a subdirectory in the
ACCESSRY directory. The subdirectory should contain the file or files required by
your application to store a document.

It is up to your application to understand the contents of the document
subdirectory. Usually a document subdirectory contains a single DOCSTATE.RES
file, which contains the file instance data for a document.

When installing your application, the installer creates a section for your
application in the Accessory notebook. If your application directory contains an
ACCESSRY directory, the installer copies the document directories and their files to
your application's section in the Accessory notebook. Each document directory
copied by the installer is marked with the well-known UID of your application.

If the accessory bit is set in the application manager metrics flags, the installer
creates a directory for your application in the ACCESSRY directory and creates an
empty document from your application in that directory. (For more information
on application manager metrics, see *Part 2: Application Framework*.

The name of each document subdirectory in the ACCESSRY directory becomes the
name of an accessory.

**Help**

To provide users with help tutorials for your application, create a directory named
HELP under your application directory. For each help document that you provide,
create a subdirectory in the help directory. If the installer finds a HELP directory, it
creates a section for your application in the Help notebook and loads the contents
each subdirectory as documents in that section. The name of each subdirectory
in the HELP directory becomes the name of a help topic in the Help notebook.
Each subdirectory can contain either:

- An APP RES file for a PenPoint application.
- A single, specially named text file that contains plain ASCII text or RTF text.

If the subdirectory contains an APP RES file, it assumes that an application that can read the file will be available to PenPoint. Usually the application is MiniText, but you can provide a more sophisticated help application and store its documents in this form.

If the subdirectory contains a text file, MiniText will import the text. However, the file must have one of these two names:

- HELP.TXT This file contains plain ASCII text.
- HELPRTF This file contains RTF-format text.

When MiniText finds a HELP... file, it displays the file in “help” mode (that is, no menu bar, different point size, and other minor changes).

When we add the HELP directory, the PENPOINT directory on our application distribution volume has this organization:

```
PENPOINT
    |-- APP
       |-- Time Management
           |-- app.res
           |-- database engine.dll
           |-- alarm clock.dll
           |-- Time Management.exe
           |-- Time Management.dlc
           |-- STATNRY
               |-- Personal Scheduler
                   |-- app.res
                   |-- WorkGroup Scheduler
                   |-- app.res
           |-- HELP
               |-- Address Book Help
                   |-- app.res
                   |-- Address Book.rtf
               |-- scheduler Help
                   |-- app.res
                   |-- Scheduler Help
                   |-- scheduler.rtf
```

When the user opens the Help notebook, it contains two documents belonging to the Time Management application: Address Book Help and Scheduler Help.

**Application Global Data**

If your application has non-resource data that it needs to access, you should create a MISC directory.

When the installer finds a MISC directory in the application distribution directory, it creates a MISC directory in your application's run-time directory, where it copies the files.

Your application should be able to determine the location of its application directory (by sending msgAppMgrGetMetrics to your application class). From the application directory, you can create a path to the MISC directory.

Some types of data do not fit easily into the resource model, especially those that consume a large amount of space.
CHAPTER 110 / PENPOINT FILE ORGANIZATION
Organization of Distribution Volumes

When an application reads a resource, it gets a copy of the data. If the amount of data is large, the application might have problems finding room to store the data.

The alternative is to store the data in a file in the MISC directory; when your application needs to access the data, you create a memory-mapped handle on the file. That way the only memory cost is in the file handle.

Good examples of data you might want to put in the MISC directory is a zip code directory, a tax table, or other fixed data you want to distribute to your users. A zip code directory would be far too large and unwieldy to store as a resource, but when it is stored as a file all documents can create memory-mapped handles on the file and access the data.

When we add the MISC directory, the PENPOINT directory on our application distribution volume has this organization:

```
\PENPOINT
  \APP
    \app.res
    \database engine.dll
    \alarm clock.dll
    \Time Management.exe
    \Time Management.dlc
    \STANDBY
      \Personal Scheduler
      \app.res
      \WorkGroup Scheduler
      \app.res
    \HELP
      \Address Book Help
      \address book.rtf
      \Scheduler Help
      \scheduler.rtf
  \MISC
    \Zip Codes
```

All time management documents can access the zip codes stored in the MISC directory.

**Service Directories**

A service directory is quite similar to an application directory. To create a service distribution volume, create a \PENPOINT directory in the root of a volume (if one doesn’t exist already). Under the PENPOINT directory, create a SERVICE directory. Under \PENPOINT\SERVICE, create a directory for your service.

The name of the service directory is the name of your service that PenPoint shows to your users. The directory name must be unique in the entire world of PenPoint services. The directory name can be a PenPoint name of up to 28 characters.

Thus, a distribution volume for an service named “CD Driver” on a volume named “CD Driver Dist” would have the following structure:

```
\CD Driver Dist
  \PENPOINT
    \SERVICE
      \CD Driver
```
Files in the Service Directory

The service directory contains:

- A SERVICE.RES file.
- A .DLC file.
- DLL files for your service.
- An optional INST subdirectory.
- An optional MISC subdirectory.

When the installer finds a service directory it looks for a .DLC file. The .DLC file should have the same PenPoint name as the service directory, but with the extension .DLC.

With the files in the application directory, the PENPOINT directory has this organization:

```
\PENPOINT
  \SERVICE
    \CD Driver
      \CD-AUDIO.dll
      \CD Driver.dlc
```

The CD DRIVER.DLC file contains:

```
TiManagementInc-CD_AUDIO_dll-V1(0) CD-AUDIO.dll
TiManagementInc-CD_ROM_dll-V1(0) CD-ROM.dll
```

All of the required executable and DLL files are in the service directory.

The INST Directory

In applications, you can provide users with preconfigured documents in the form of stationery. For services, you can provide users with preconfigured services in the form of service state nodes. These state nodes can be either:

- Files containing preconfigured instances of the service.
- Directories containing preconfigured instances of the service and other resource or data files.

To provide preconfigured services, create an INST directory in your service directory. The INST directory can contain any number of service state nodes.

When the installer finds an INST directory, it copies the contents of the directory into an INST directory for the installed service.

When we add the INST directory, the PENPOINT directory on our service distribution volume has this organization:

```
\PENPOINT
  \SERVICE
    \CD Driver
      \CD-AUDIO.dll
      \CD-ROM.dll
      \CD Driver.dlc
      \INST
        \Macintosh Format
        \MS-DOS Format
```
**The Quick Installer**

You can configure a disk volume so that PenPoint pops up an installation sheet automatically when the volume becomes available (usually when the user inserts the volume in a floppy drive).

To enable quick installation, open the Connections notebook to the disks sheet and open the disk (if it isn’t open already). Tap on Disk... in the Options menu for the disk you want to modify.

Change Quick Installer? to yes, then select the Initial View, which specifies which installer is first displayed by the quick installer (Directory, Applications, Services, Handwriting, and so on).

In Figure 110-2 the disk TIMGMT is open in the Connections notebook and the user has tapped on Initial View.
Multiple Applications and Multiple Volumes

A distribution volume can contain a mixture of applications and services (provided they will all fit on a single volume).

The PenPoint Installer does not currently support applications or services that span more than one volume. If you have such an application, you must control installation yourself from the application monitor (instance 0 of your application).

Upgrading

Currently users upgrade applications or services by deinstalling the old version and installing the new version.
Chapter 111 / Dynamic Link Libraries

A linker joins together a number of object modules to form a single executable file. This allows you to incorporate object files from many sources. It also allows you to maintain a single, shared copy of the object file, which all applications link with. However, each executable file that you create by linking object modules contains a copy of the object code; if 10 applications link with the same object file, there are 10 copies of that object file in the resulting 10 executable files. In the PenPoint computer, where memory is at a premium, this replication is wasteful and undesirable.

The solution is to create an executable file that can access a single, shared object code module only when it is required. PenPoint provides this solution by using Dynamic Link Library files (DLL files).

You create a DLL file by compiling a source file with options that create a DLL file rather than an object file.

You reference a DLL file by calling a routine or referencing a class that is external to your object file. Before you link your object file, you define all DLL routines and classes in a DLL.DEF file.

For more information on the DLL.DEF file, see the *PenPoint Application Writer's Guide* and the *Microsoft Operating System/2 Programmer's Toolkit* manual.

**References to DLL Files**

Clients can refer to DLL files in two ways:

- A client makes a reference to a routine in the DLL file (explicit referencing).
- A client makes a reference to a class in the DLL file (implicit referencing).

A client can reference a DLL file either from within its own executable file or from within another DLL file.

Explicit references cause the linker to make a dynamic link entry in the referencing executable or DLL file. There can be circular explicit references. An explicitly referenced DLL file must be loaded when the linker entry is processed.

Implicit references must be determined by the developer, because the linker cannot make entries for them. Most of the DLL file dependencies in PenPoint are due to implicit references.

An implicitly referenced DLL file must be loaded before an object of the referenced class is created.
**DLL File Issues**

The PenPoint Application Monitor ensures that only one copy of a particular DLL file is loaded on the PenPoint computer. This is not as simple as you might expect because:

- PenPoint has to know which applications require which DLL files.
- If more than one version of a DLL file exists, PenPoint must determine which version or versions can be loaded.
- PenPoint must know when it can unload a DLL file because the applications that referred to it are no longer present.

In order to determine which DLL files are loaded (and avoid duplicating DLL files in memory), PenPoint needs to reliably identify DLL files. When unloading DLL files, PenPoint needs to determine whether any applications still refer to the DLL file; if the DLL file is still referenced, it shouldn’t be unloaded. You cannot guarantee that the application that caused PenPoint to load a DLL file is the only application that refers to that DLL file.

Finally, PenPoint must have a scheme that deals with different versions of the DLL files that make up both our system software and applications. This scheme needs to take versioning of classes into account as well.

This chapter discusses the mechanisms used by PenPoint to ensure that the correct DLL files are loaded. This chapter is essential to application installation because you, the application developer, are responsible for providing PenPoint with the correct information in the correct places. Without correct information, the PenPoint application installation software will not be able to install your application properly.

**Identifying DLLs**

Each application has its own set of DLL files, which live in the application’s directory on disk. This directory must contain all non-operating system DLL files that an application needs (including third-party DLL files and company DLL files that are common to a suite of applications).

Although the application directories on disk might contain duplicate DLL files, PenPoint ensures that only one copy of a particular DLL file is installed in PenPoint computer memory (where it is shared by all applications that need it).

Each DLL file has a unique **dll-id** name. A DLL file is named by the developer that created it. Executable files are also named in this manner. The **dll-id** is given in the LIBRARY field of the DLL.DEF file that was used to build the module. The **dll-id** is a text string (up to 32 characters) of the form:

```
companyName-moduleName-majorVersion(minorVersion)
```

The dashes and parentheses are part of the syntax and are required; string parsing depends on them. The fields can contain letters, numbers, and the underscore character (_).
companyName is the name of the company that created the DLL or executable file. For example, the company name for GO Corporation is GO.

moduleName is the name of the DLL or executable file. Be sure that DLL files and executable files have different moduleNames. GO recommends that you call the executable file moduleName_exe and the DLL file moduleName_dll.

majorVersion is the version number of the DLL file. minorVersion is an optional minor version number; it is ignored by the operating system when it determines whether a DLL is already loaded in the PenPoint computer.

### DLC Files

A .DLC file lists all the explicit and implicit DLL file references made by the application executable file; the .DLC file also names the application executable file itself. The .DLC file allows the operating system to preload implicit references and to locate explicit references.

Application developers must create a .DLC control file for each application that uses DLL files. If an applications has a single executable file that has the same name as the application directory, the application does not need a .DLC file.

The .DLC file must have the same name as the application directory and the executable file. For example, an application with the PenPoint name Graph It Right would be stored in the directory \PENPOINT\APP\Graph It Right; the executable file would be called Graph It Right.EXE and the .DLC file would be called Graph It Right.DLC.

Each line in a .DLC file contains a pair or names: a dll-id and a file system path to the corresponding DLL file or executable file. All paths are relative to the application’s directory. For example, grapher might have the following format:

```
GO-forms_dll-Vl(2) forms.dll
GO-Grapher_exe-Vl grapher.exe
```

The operating system reads the .DLC file when it loads an application; it ensures that all the DLL files in the .DLC file are loaded before it brings in the executable file.

The operating system will use the dll-id to determine if a DLL file is already loaded, and not load the new DLL file if so. This rule also applies to executable files, but there will be very few cases where an executable file is already loaded.

Applications can also load optional DLL files if desired. These are DLL files that aren’t part of a executable file load, such as a database engine. The developer must create a .DLC file for optional DLL files if they reference other DLL files.

### Sharing DLL Files

When the application monitor loads a new application, it checks the dll-ids in the .DLC file against the list of currently loaded dll-ids. The comparison includes the company name, the DLL name, and the major version number. (It is too
dangerous to automatically run an application with a DLL file that has a different major version number.)

The application monitor does not compare minor version numbers. Different minor version numbers must be guaranteed to be compatible by the developer.

If the dll-ids match, the DLL has been loaded already; the operating system does not load the DLL but increments a reference count for the dll-id.

If the dll-ids do not match, the application monitor loads the DLL from the path specified in the .DLC file.

This scheme guarantees that an application that was written to run with a particular DLL file will always run with that DLL file.

When an application is deinstalled, the application monitor again opens the corresponding .DLC file and compares its dll-ids against the currently loaded dll-ids. If the Application Monitor finds a match, it decrements the dll-id reference counter. If the reference counter becomes zero, the application monitor can deinstall the DLL file.

**Versions**

An already loaded DLL file is used only if there is an match between dll-ids through the major version number.

From a developer’s perspective, when you add new functionality to a DLL file, you should revise the dll-id’s major version number.

- If the new functionality is backwards compatible, you should upgrade the version number in the class UID.
- If the new functionality is not backwards compatible, you must create a new class ID.

**DLL Processes**

A DLL file can optionally contain a DLLMain() routine, which usually contains code to install the DLL’s classes. When a DLL file is loaded, PenPoint creates a process and transfers control to the DLLMain(). These classes (and any other memory that the DLL file creates) are owned by the DLL file’s process. A DLL process will dispatch messages, so a DLL file can also create well-known objects at init time.

The process, all memory associated with the DLL file, and all classes owned by the DLL file are destroyed when the DLL file’s reference count goes to zero and it is deinstalled.

However, this utility comes at some expense; a process uses approximately 37K bytes of memory. If you structure your application so that it uses several distributed DLLs, each of which creates its own process, you can run short of memory fairly quickly.
When you create a DLL file, you have three options:

- Create a DLL file that doesn’t have a `DLLMain()`, contains code for one or more classes, and which exports initialization functions for one or more classes. When your application needs a class, it calls the exported function to initialize the class. Your application uses a private well-known UID to identify the class.

- Create a DLL file that has a `DLLMain()` and contains code for more than one class. All of the classes use global well-known UIDs to identify the classes. All the classes share the same memory and process.

- Create a DLL file that has a `DLLMain()` and code for only one class. The class is identified with a global well-known UID.

### Operating System DLL Files and Versions

The operating system is somewhat different from applications. The operating system is defined as those DLL files and executable files that are loaded into the machine at boot time and are always present. There can only be one version of the operating system in the machine at any time, and upgrading the operating system is done by a cold boot. Applications should be able to run on many different operating system versions. Most importantly, applications written for earlier versions of the operating system should run on later versions.

Operating system DLL files also have dll-ids, but these dll-ids have no major or minor version numbers. This is so explicit references that show up in application executable or DLL files will not be affected by operating system upgrades.

The version number for operating system classes can be incremented when new functionality is introduced. However, operating system classes must be strictly backwards compatible. A new class number must be used when backwards compatibility is broken. The class manager will always search for an exact match when a class is referenced, but will accept a higher-versioned class if it can’t find an exact match.

Applications can query class version on a per-class basis. We will also provide an overall version number for the operating system in the system resource file. An application can specify the minimum operating system version it will run under at installation time.

### DLL Files and MAKE Files

All PenPoint DLL files that are not part of the system that is loaded at boot time (in `BOOT.DLC`) are distributed in `\PENPOINT\SDK\DLL`. The `DLL_TYPE=DISTRIBUTED`, builds a project into `\PENPOINT\SDK\DLL`. For example, if project `Grafpapr` specifies

```
DLL_TYPE = DISTRIBUTED
```

in its makefile, `MAKE` creates the file `\PENPOINT\SDK\DLL\GRAFPAPR\GRAFPAPR.DLL`. 
Applications that depend on a distributed .DLL file should include a DISTRIBUTED_DLLS = line in their makefile. This line should contain all the distributed DLL files that should be included with the app. For example:

```
DISTRIBUTED_DLLS = grafpapr\grafpapr.dll atp\flap.dll
```

If an application needs a .DLC file, the .DLC file should be part of the application project directory. The default name for a .DLC file is PROJECT_NAME.DLC. The .DLC file is copied into the application directory when the application is built.
Chapter 112 / Installation Managers

clsInstallMgr provides applications with the facilities to manage installable items. An item can be any sort of resource. Each type of installable item has its own installer.

An installer is an instance of clsInstallMgr, or a instance of a subclass of clsInstallMgr. The installer handles all common operations used to maintain installable items.

The installer application uses installers to install and deinstall several sets of items, such as applications and fonts. Fonts are a straight-forward use of the installable item manager. Applications require a much more complex use of the installable item manager.

Installer Concepts

The install manager class (clsInstallMgr) describes messages used by installers to install, deinstall, and maintain installable items.

Unless you are writing your own installer, you probably don’t need to send or handle most clsInstallMgr messages. However, there is a core set of clsInstallMgr messages that are used by many applications. These messages are:

- msgIMInstall
- msgIMDeinstall
- msgIMGetInstalledList
- msgIMGetState
- msgIMSetCurrent
- msgIMGetCurrent
- msgIMGetName
- msgIMFind

Additionally, if your application needs to keep track of installed items, you can observe an installation manager.

clsInstallMgr also serves as the superclass for a number of specialized installers. Some types of installable items require their own installer class. These classes are subclasses of clsInstallMgr:

- clsAppInstallMgr maintains applications.
- clsFontInstallMgr maintains fonts.
- clsServiceInstallMgr maintains services.

Each of these services and their messages are described at the end of this chapter.
Installers

For each of the installer classes, there is a well-known instance of that class. That instance is the installer for that type of item.

The instance of clsAppInstallMgr is theInstalledApps; the instance of clsServiceInstallMgr is theInstalledServices; the instance of clsFontInstallMgr is theInstalledFonts.

Other types of items do not require specialized messages for installation. The installers for these groups are simply instances of clsInstallMgr. These installers are:

- theInstalledHWXProtos, which maintains the handwriting samples.
- theInstalledPDicts, which maintains the spelling dictionaries.
- theInstalledPrefs, which maintains the user preferences.

Additionally, the class clsInstallMgr also serves as a central repository of information about all installed items. If you use a subclass to maintain information about an installed item, the same information is also maintained by clsInstallMgr.

Installed Item Database

clsInstallMgr and its subclasses use the file system to keep a database of the installed items. Each item is represented by a handle on a file or directory. When PenPoint installs an installable item, it creates a directory or file in the appropriate \PENPOINT subdirectory. Application directories are stored in \PENPOINT\APP; font files are stored in \PENPOINT\FONT, and so on. The handle is the mechanism by which the install manager keeps track of the installed items.

Representing installed items with files or directories is a big win for items that are files or directories. When the item is a file or directory, the install manager's handle is a handle on that actual item. However, if an installable item is not a file, there must be an extra level of indirection.

The installers use the attributes in the file system node to store information about the item, such as the item's identifier. An item's identifier has different meanings for different items. For applications, the identifier is the application class's well-known UID; for fonts the identifier is the font ID number.

Usually the installer creates an initial set of item handles from the contents of its directory. You can override this behavior by setting the createInitial style bit to false; when createInitial is false, the installer will not create item handles from its directory.

Controlling Items

clsInstallMgr provides several attributes that pertain to items, and an API for getting and setting these items.
clsInstallMgr provides an optional API to make an item current, and an API for getting and setting that attribute. An item can also be marked as being in use. The current item is considered to be in use. Items that are in use cannot be deinstalled.

A notion of whether a item differs from when it was installed is also provided.
Clients should mark items as modified when they change them by sending msgIMSetModified. The installer will remember the time and date that the item was modified.

Clients access installable managers by an ObjectCall() interface. clsInstallMgr can accommodate simultaneous access by multiple clients if the shared style bit is set true (the default). This causes it to use the semaphore for all of its operations. This semaphore is available to subclasses through msgIMGetSema, and should be used to protect all subclass messages if multiple clients will be accommodated. clsInstallMgr also sets objCapCall on by default.

**Observing Installation Managers**

Usually most application developers do not need to use clsInstallMgr messages. However, if you want to monitor the coming and going of things such as fonts or handwriting samples, you can observe an installer.

clsInstallMgr monitors the item directory and sends notification whenever an installable item is added, removed, or when a different item becomes current. clsInstallMgr first sends observer messages to self, which allows subclasses to intercept them. If the send to self returns stsOK, clsInstallMgr sends the messages to observers.

A subclass of clsInstallMgr can prevent notification of observers by returning stsIMNoNotify. A subclass can turn notification generation off entirely by sending msgIMSetNotify to its ancestor. msgIMSetNotify takes a BOOLEAN value as its argument. If the value is true, it turns notification off.

A client can request the current notification state by sending msgIMGetNotify to self. The message takes a pointer to the BOOLEAN value that will receive the current notification state.

Clients can make themselves observers of a installation manager by sending msgAddObserver to that installation manager. For instance, to observe the handwriting installation manager, you would send msgAddObserver to theInstalledHWX.
Table 112-1 lists the notification messages described by clsInstallMgr.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgIMNameChanged</td>
<td>P_IM_NOTIFY</td>
<td>The name of an item has changed.</td>
</tr>
<tr>
<td>msgIMCurrentChanged</td>
<td>P_IM_CURRENT_NOTIFY</td>
<td>The current item has changed.</td>
</tr>
<tr>
<td>msgIMInUseChanged</td>
<td>P_IM_INUSE_NOTIFY</td>
<td>An item’s inUse attribute has changed.</td>
</tr>
<tr>
<td>msgIMModifiedChanged</td>
<td>P_IM_MODIFIED_NOTIFY</td>
<td>An item’s modified attribute has changed.</td>
</tr>
<tr>
<td>msgIMInstalled</td>
<td>P_IM_NOTIFY</td>
<td>A new item was installed.</td>
</tr>
<tr>
<td>msgIMDeinstalled</td>
<td>P_IM_DEINSTALL_NOTIFY</td>
<td>An item has been deinstalled.</td>
</tr>
</tbody>
</table>

**Handling msgIMNameChanged**

Observers receive **msgIMNameChanged** when the name of an item has changed. The message passes a pointer to an IM_NOTIFY structure that contains:

- `manager` The UID of the manager that sent the notification.
- `handle` The handle on the item that changed.

**Handling msgIMCurrentChanged**

Observers receive **msgIMCurrentChanged** when another item becomes the current item. The message passes a pointer to an IM_CURRENT_NOTIFY structure that contains:

- `manager` The UID of the manager that sent the notification.
- `newHandle` The handle of the new current item.
- `oldHandle` The handle of the old item.

**Handling msgIMInUseChanged**

Observers receive **msgIMInUseChanged** when a item’s inUse attribute has changed. The message passes a pointer to an IM_INUSE_NOTIFY structure that contains:

- `manager` The UID of the manager that sent the notification.
- `handle` The handle on the item that changed.
- `inUse` A BOOLEAN value that indicates the item’s new inUse state.

**Handling msgIMModifiedChanged**

Observers receive **msgIMModifiedChanged** when an item’s modified attribute has changed. The message passes a pointer to an IM_MODIFIED_NOTIFY structure that contains:

- `manager` The UID of the manager that sent the notification.
- `handle` The handle on the item that changed.
- `modified` A BOOLEAN value that indicates the item’s new modified state.
Handling msgIMInstalled
Observers receive msgIMInstalled when an installable item has been installed or a deactivated item was reinstalled. The message passes a pointer to an IM_NOTIFY structure, as described above in “Handling msgIMNameChanged.”

Handling msgIMDeinstalled
Observers receive msgIMDeinstalled when an installable item has been deinstalled. The message passes a pointer to an IM_DEINSTALL_NOTIFY structure.

Using clsInstallMgr Messages
The messages defined by clsInstallMgr are used by installers to manipulate installed items and to get information about the installation managers. These messages are defined in INSTLMGR.H.

Table 112-2 lists the messages defined by clsInstallMgr.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_IM_NEW</td>
<td>Creates a new install manager.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_IM_NEW</td>
<td>Initializes the IM_NEW structure to default values.</td>
</tr>
<tr>
<td>msgIMGetStyle</td>
<td>P_IM_STYLE</td>
<td>Passes back the current style settings.</td>
</tr>
<tr>
<td>msgIMSetStyle</td>
<td>P_IM_STYLE</td>
<td>Sets the current style.</td>
</tr>
<tr>
<td>msgIMGetCurrent</td>
<td>P_IM_HANDLE</td>
<td>Passes back the current item’s handle.</td>
</tr>
<tr>
<td>msgIMSetCurrent</td>
<td>IM_HANDLE</td>
<td>Sets the current item.</td>
</tr>
<tr>
<td>msgIMSetInUse</td>
<td>P_IM_SET_INUSE</td>
<td>Changes an item’s in use setting.</td>
</tr>
<tr>
<td>msgIMSetModified</td>
<td>P_IM_SET_MODIFIED</td>
<td>Changes an item’s modified setting.</td>
</tr>
<tr>
<td>msgIMGetName</td>
<td>P_IM_GET_SET_NAME</td>
<td>Gets the name of a item.</td>
</tr>
<tr>
<td>msgIMSetName</td>
<td>P_IM_GET_SET_NAME</td>
<td>Sets the name of a item.</td>
</tr>
<tr>
<td>msgIMGetVersion</td>
<td>P_IM_GET_VERSION</td>
<td>Gets the version string for this item.</td>
</tr>
<tr>
<td>msgIMGetList</td>
<td>P_LIST</td>
<td>Passes back a list of all the items on this install manager.</td>
</tr>
<tr>
<td>msgIMGetState</td>
<td>P_IM_GET_STATE</td>
<td>Gets the state of a item.</td>
</tr>
<tr>
<td>msgIMGetSize</td>
<td>P_IM_GET_SIZE</td>
<td>Returns the size of an item.</td>
</tr>
<tr>
<td>msgIMInstall</td>
<td>P_IM_INSTALL</td>
<td>Installs a new item.</td>
</tr>
<tr>
<td>msgIMDeinstall</td>
<td>P_IM_DEINSTALL</td>
<td>Deinstalls an item.</td>
</tr>
<tr>
<td>msgIMDup</td>
<td>P_IM_DUP</td>
<td>Creates a new item that is a duplicate of an existing one.</td>
</tr>
</tbody>
</table>
Managing Installable-Item Managers

The following sections describe how to create and alter installable managers (instances of clsInstallMgr).

Creating an Installable-Item Manager

To create an installable manager, send msgNewDefaults and msgNew to clsInstallMgr. The message takes a pointer to an IM_NEW structure that contains an OBJECT_NEW_ONLY structure and:

- **style** Style specifiers for the installable manager. Currently the only style is **shared**, which specifies whether the installable manager will be accessed by multiple clients. If **shared** is true (the default) clsInstallMgr creates a semaphore for the manager. However, semaphores are an expensive system resource; if you know that no other clients will access the installable manager, you can specify false, which does not create a semaphore.

- **locator** A file system locator that indicates the directory that contains the items. The directory must exist before you create the installable manager object.
Getting and Setting the Style of an Installable-Item Manager

When you create an instance of clsInstallMgr, you can specify the installable item's style. Currently, the style only indicates whether the installable item is accessed by one or many clients. By default, the installable manager is shared, however, you can get and set the style by sending msgIMGetStyle and msgIMSetStyle to an instance of clsInstallMgr.

Both messages require an IM_STYLE structure that contains a single member: a BOOLEAN value (shared) that indicates whether the item is shared or not. If shared is true, the installable manager can be shared by more than one client.

As described earlier, when an installable manager is shared, it must create a semaphore. When you set shared to false, the semaphore is released, freeing system resources.

Getting the Item Directory

To get a directory handle of the directory observed by an installable manager, send msgIMGetDir to the installable manager. The only argument for the message is a pointer to the location that receives the handle.

Managing Installable Items

The following sections discuss how to install, deactivate, reactivate, copy, update, and delete installable items.

Installing an Installable Item

To add an installable item from an external volume to a item directory, send msgIMInstall to the appropriate installable manager. The message takes a pointer to an IM_INSTALL structure that contains:

- locator A locator that specifies where the installable item is stored.
- updateOK An IM_INSTALL_EXIST value that specifies what to do if the installable item exists already. The possible IM_INSTALL_EXIST values are:
  - imExistUpdate Copy a new item over an existing item.
  - imExistReactivate Deactivate the existing item, and activate the new item.
  - imExistGenError Return stsIMAAlreadyInstalled.
  - imExistGenUnique Generate a different name for the new item.
  - imExistIncRefCount Increment the reference count of the existing item.
- listAttrLabel An FS_ATTR_LABEL structure that describes the attribute list for the handle on the new item. This field is used internally by PenPoint, and should always be null.
- listHandle A file system handle that will get the attributes. This field is used internally by PenPoint, and should always be null.

If the message is successful, it returns a handle on the installed item (handle).
Duplicating an Installed Item

To duplicate an installable item, send msgIMDup to the installable manager. The message takes an IM_DUP structure that specifies:

- **handle** The handle of the item to duplicate.
- **pName** The name of the new item. If pName is NULL, clsInstallMgr creates a unique name. If pName is not NULL, it must be a valid item name.

When the message completes, the structure returns the handle of the new item (newHandle).

If a item with pName exists already, the message returns stsIMAlreadyInstalled.

Deleting an Installable Item

To delete an installable item, send msgIMDeinstall to the installable manager. The item cannot be the current item. The message takes a pointer to an IM_DEINSTALL structure that specifies:

- **handle** The UID of the item to delete.

You must use msgIMDelete to delete an installable item from the file system. The handle on the item node and the installable manager protect items from any other form of destruction. Messages that attempt to delete a file system node that is used by the installable manager return stsFSNodeBusy.

Altering Installable Item Attributes

The following sections describe messages that alter the attributes of individual installable items.

Getting and Setting the Current Installable Item

To make an installable item the current item, send msgIMSetCurrent to the installable manager. The only argument to the message is the UID of the item to make current. To clear the current item (so that no item in the item directory is current) use objNull as the object UID.

To get the handle of the current item, send msgIMGetCurrent to the installable manager. The only argument for the message is a pointer to the OBJECT location that receives the UID of the current object. If there is no current object, the message returns objNull.

Changing an Installable Item’s Name

To change the name of an installed item, send msgIMSetName to the installable manager. The message takes an IM_SET_NAME structure that contains:

- **handle** The handle of the item to be changed.
- **pName** The new name for the item. The name must be a valid item name.

To get the current name for a item, use msgIMGetState.
Getting Information About an Installable Item

The messages in the following sections get information about installable items.

Finding an Installable Item by Name

If you know the name of an installable item but don’t know its handle (its UID), you can send msgIMFind to the appropriate installable manager. The message takes a pointer to an IM_FIND structure that specifies the name of the item to look up (pName).

If the item is found, the IM_FIND structure returns the UID of the item handle (handle).

If the item is not found, the message returns stsNoMatch.

Getting a List of Items

To get a list of items (installed and deactivated) in a item directory, send msgIMGetList to the installable manager. The only argument for the message is a pointer to a LIST that receives the item handles. The message obtains space for the list from your default process heap. It is your responsibility to free the LIST object.

Getting the Attributes of an Installable Item

To find out the name and attribute state of an installable item, send msgIMGetString to the installable manager. The message takes a pointer to an IM_GET_STATE structure that specifies:

handle The handle of the item to get information on.
pName A pointer to the string that receives the name of the item. If you specify pNull in pName, the message will not return the item name.

The message uses the IM_GET_STATE structure to return:
current A BOOLEAN value that receives the indication whether the item is the current item.
modified A BOOLEAN value that receives the indication whether the item has been modified or not.

Getting the Size of an Installable Item

Before installing an item, it is a good idea to know its size so that you can determine whether there is enough room in the PenPoint computer for the item. To get the size of the item, send msgIMGetSize to the installable manager. The message takes a pointer to an IM_GET_SIZE structure, which contains the handle of the item (handle).

When the message returns, the IM_GET_SIZE structure contains the size of the item in the size field.
Advanced clsInstallMgr Topics

The following messages should only be used by clients that subclass clsInstallMgr.

Using the Semaphore

When an installable manager object is created with shared set to true, it uses a semaphore to synchronize client access. If you subclass clsInstallMgr, you will need to use the same semaphore to synchronize your access to a item directory.

Send msgIMGetSema to a specific installable manager to get the manager's semaphore identifier. The message takes a pointer to an OS_FAST_SEMA value that receives the semaphore identifier. Note that this is only the identifier of the semaphore; not access to the item directory.

clsInstallMgr uses the semaphore whenever it receives a message that concerns the item directory (such as msgIMInstall). clsInstallMgr requests the semaphore before it first accesses the item directory and clears it only after the last time it accesses the item directory.

When a subclass of clsInstallMgr must access the item directory, it should get the semaphore identifier from the installable manager. The subclass must use the OSFastSemaRequest to request access to the item directory and OSFastvSemaClear to release the semaphore.

At some time, the subclass might need to call ancestor (clsInstallMgr) while it holds the semaphore. When invoked by a call ancestor, clsInstallMgr can request the same semaphore. PenPoint grants the semaphore because the request came from the same task that is currently holding the semaphore. PenPoint maintains a count of the number of semaphore requests from the same task. When PenPoint receives OSSemaClear, it decrements the count.

Code Installation Manager

The code installation manager, clsCodeInstallMgr, is a subclass of clsInstallMgr that performs installation tasks common to both applications and services. The capabilities added by clsCodeInstallMgr are that it can:

- Install an application or service.
- Return a list of the classes of all the currently installed applications or services.
- Get the application or service class of an item handle.
- Find the item handle for a specific application or service class.

The clsCodeInstallMgr messages are defined in CODEMGR.H. Table 112-3 lists the messages defined by clsCodeInstallMgr.
Installing an Application or Service

To install an application, send msgCIMLoad to theInstalledApps or theInstalledServices. This message must be sent with the ObjectSend() macro, not ObjectCall(). The message takes a pointer to an CIM_LOAD structure, which contains a file handle on the item to load.

clsCodeInstallMgr does not handle msgCIMLoad; it is up to the classes that inherit from clsCodeInstallMgr (clsApplInstallMgr and clsServiceInstallMgr) to handle msgCIMLoad.

Application Installation Manager

clsApplInstallMgr is a subclass of clsCodeInstallMgr used to install applications. clsApplInstallMgr adds the ability to get or set the mask class. The mask class enables the PenPoint notebook to prompt the user when they attempt to turn to a document for which the application is no longer installed.

The clsApplInstallMgr messages are defined in APPIMGR.H. Table 112-4 lists the messages defined by clsApplInstallMgr.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAIMGetMaskClass</td>
<td>P_CLASS</td>
<td>Passes back the mask class.</td>
</tr>
<tr>
<td>msgAIMSetMaskClass</td>
<td>CLASS</td>
<td>Sets the mask class.</td>
</tr>
</tbody>
</table>
If the user deinstalls an application while there are documents for that application in the PenPoint computer, the installer substitutes a mask application class in place of the application class for each document. By default, PenPoint uses the placeholder application (defined by clsMaskApp) as the mask application. The placeholder's only function is to respond to Application Framework activation messages by displaying a page that informs the user that the application has been deinstalled.

For more information about clsMaskApp, see Part 2: Application Framework.

PenPoint allows you to substitute your own mask application in place of the placeholder application.

### Service Installation Manager

The service installation manager, theInstalledServices, maintains the installed and deinstalled services in PenPoint. theInstalledServices is an instance of clsServiceInstallMgr; clsServiceInstallMgr is a subclass of clsInstallMgr. The service installation manager differs from clsInstallMgr in that it must handle metrics for the service classes.

The clsServiceInstallMgr messages are defined in SERVIMGR.H. Table 112-5 lists the messages defined by clsServiceInstallMgr.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgSIMGetMetrics</td>
<td>P_SIM_GET_METRICS</td>
<td>Gets the specified service class's metrics.</td>
</tr>
</tbody>
</table>

### Font Installation Manager

The font installation manager, theInstalledFonts, maintains the installed and deinstalled fonts in PenPoint. theInstalledFonts is an instance of clsFontInstallMgr; clsFontInstallMgr is a subclass of clsInstallMgr. The font installation manager differs from clsInstallMgr in that it must identify fonts and maintains the notion of the system font.

### Font Identification

Fonts are identified in four ways:

- A font file handle.
- The name of a font file.
- A short font id.
- A string (or long) font id.

A font file handle is a file handle on to the actual font file. Most of the font installation manager interface uses these handles.
The font file name is the user-visible name for the font. You can get the font file name with the messages `msgIMGetName` (which uses a handle) and `msgFIMGetNameFromId` (which uses an ID).

A short font ID is a pre-defined, 16-bit value that identifies a specific font. It is a compact, specific reference for a particular font that appears in the window system API.

A string font ID is a 4-character version of a short font ID.

You can get a list of all the font handles in the system by sending the superclass message `msgIMGetList` to `theInstalledFonts`. This list includes fonts which are deactivated. You can use the superclass message `msgIMGetActiveList` to get a list of the active fonts.

To get a pruned list of the active fonts that is appropriate for end-user display, send `msgFIMGetInstalledIdList` to `theInstalledFonts`.

`clsFontInstallMgr` can:
- Get a list of the short IDs of all the installed fonts.
- Get a font's short and long IDs.
- Set the font's file ID.
- Find a font handle, given a short ID.
- Return a font name, given a short ID.

### clsFontInstallMgr Messages

The `clsFontInstallMgr` messages are defined in `FONTMGR.H`. Table 112-6 lists the messages defined by `clsFontInstallMgr`.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>msgFIMGetId</code></td>
<td><code>P_FIM_GET_SET_ID</code></td>
<td>Gets the short and long font IDs, given a handle.</td>
</tr>
<tr>
<td><code>msgFIMSetId</code></td>
<td><code>P_FIM_GET_SET_ID</code></td>
<td>Set the font file's ID.</td>
</tr>
<tr>
<td><code>msgFIMFindId</code></td>
<td><code>P_FIM_FIND_ID</code></td>
<td>Finds a font handle given a short ID.</td>
</tr>
<tr>
<td><code>msgFIMGetNameFromId</code></td>
<td><code>P_FIM_GET_NAME_FROM_ID</code></td>
<td>Passes back font name given an short ID.</td>
</tr>
<tr>
<td><code>msgFIMGetInstalledIdList</code></td>
<td><code>P_FIM_GET_INSTALLED_ID_LIST</code></td>
<td>Passes back a list of the short IDs of all installed fonts.</td>
</tr>
</tbody>
</table>

`clsFontInstallMgr` does not understand these superclass messages:
- `msgIMGetCurrent`
- `msgIMSetCurrent`
- `msgIMDup`

`clsFontInstallMgr` does not send `msgIMCurrentChanged`. 
Getting and Setting a Font's ID

If you have a font handle and want the font’s short and long IDs, send msgFIMGetId to theInstalledFonts. The message takes pointer to a FIM_GET_SET_ID structure that contains the handle on the font file (handle).

When the message completes successfully, it returns stsOK, and passes back:

- **id** An FIM_SHORT_ID value that contains the short ID.
- **longId** An FIM_LONG_ID value that contains the long ID.

Usually you do not need to change a font’s ID. This message allows third parties to create tools that edits font IDs.

To assign an ID to a font file, send msgFIMSetId to theInstalledFonts. The message takes a pointer to an FIM_GET_SET_ID structure that contains:

- **handle** The handle of the font file to which you want to assign the new IDs.
- **id** An FIM_SHORT_ID value that contains the new short ID. If id is 0, the value in the long ID is used.
- **longId** An FIM_LONG_ID value that contains the new long ID.

Finding a Font Handle

If you have a font’s short ID and need to know the handle on the font file, send msgFIMFindId to theInstalledFonts. The message takes a pointer to an FIM_FIND_ID structure, which contains an FIM_SHORT_ID value that specifies the font’s ID (ID).

If the message completes successfully, it returns stsOK. If the font ID is not found, it returns stsNoMatch.

Getting a Font Name

To get a font’s name, given its short ID, send msgFIMGetNameFromId to theInstalledFonts. The message takes a pointer to an FIM_GET_NAME_FROM_ID structure, which contains:

- **id** An FIM_SHORT_ID value that contains the font’s short ID.
- **pName** A pointer to the string that will receive the font’s name.

If the message completes successfully, it returns stsOK and copies the name to the array specified by pName.

Getting a List of Installed Fonts

To get a list of the installed fonts, send msgFIMGetInstalledIdList to theInstalledFonts. The message takes a pointer to an FIM_GET_INSTALLED_ID_LIST structure that contains:

- **prune** An FIM_PRUNE_CONTROL value that describes how the list should be pruned. The prune value specifies:
  - **fimNoPruning** Don’t prune the list.
fimPruneDupFamilies  Remove duplicates from font families.

fimPruneSymbolFonts  Remove symbol fonts.

When the message completes successfully, it returns stsOK and passes back a list object. The list contains the short IDs of all the installed fonts. The list is pruned, according to the prune field, so that it is useable as a user pick list. For example, if the client specified fimPruneDupFamilies and both Helvetica and Helvetica Bold are in the system, only Helvetica is on this list.

You must destroy the list object when you are finished using it.

You can use the superclass message msgIMGetList to get a list of all the font file handles. There are handles for both installed and deactivated fonts.
Chapter 113 / The Auxiliary Notebook Manager

The auxiliary notebooks are separate notebooks defined by PenPoint™ that provide system functions. The auxiliary notebooks include the In box and Out box notebooks, the Help notebook, the Stationery notebook, the Connections notebook, the Settings notebook, and also include the keyboard and the clock.

The auxiliary notebook manager allows clients to open auxiliary notebooks and to perform other management tasks.

Topics covered in this chapter:

- The auxiliary notebooks used by the PenPoint operating system.
- The messages used to control auxiliary notebooks.
- Creating sections and documents in auxiliary notebooks.
- Deleting sections and documents from auxiliary notebooks.
- Modifying the stationery menu.

Auxiliary Notebook Concepts

There are two types of items on the PenPoint bookshelf:

- Data items, such as notebooks and documents.
- System items, such as the Settings notebook, the In box, Out box, and so on.

All system items on the Bookshelf (the auxiliary notebooks) are controlled through the auxiliary notebook manager (theAuxNotebookMgr), which is the only instance of clsAuxNotebookMgr.

The Auxiliary Notebooks

While using PenPoint, you have probably encountered these items at one time or another. The file AUXNBGMGR.H defines tags for each of these auxiliary notebooks. Table 113-1 lists these tags and their purpose.
Auxiliary Notebooks

Some of these "auxiliary notebooks" are not notebooks at all. They actually are instances of specialized applications. The auxiliary notebooks that actually behave like notebooks are: the In box, Out box, Stationery, and Help notebooks.

### Auxiliary Notebooks and the File System

For each auxiliary notebook, there is a directory in \PENPOINT\SYS\DOC that contains the documents (and, if supported, sections) in that notebook. For example, the Help notebook contains directories for each installed application. When the user installs an application, the installer:

1. Creates a directory for that application in the Help notebook directory.
2. Uses clsAuxNotebookMgr messages to create a section in the Help notebook.
3. Copies the help templates from the HELP directory to the new section.

While it is important to understand the relationship between the auxiliary notebooks and the file system, clients should not access the auxiliary notebooks through the file system. Rather, clients should send messages to theAuxNotebookMgr to manipulate the auxiliary notebooks. This allows PenPoint to change the file-system name or location of these notebooks at any time in the future.

### Back Up Considerations

Auxiliary notebooks contain documents that are maintained by PenPoint system software. These notebooks provide their own mechanisms to back up their data. If you are writing a backup program for PenPoint, do not back up documents in the auxiliary notebooks.

---

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Auxiliary Notebook</th>
</tr>
</thead>
<tbody>
<tr>
<td>anmSettingsNotebook</td>
<td>The Settings UI. Used to configure PenPoint settings.</td>
</tr>
<tr>
<td>anmHelpNotebook</td>
<td>The Help notebook. Used to access help on topics.</td>
</tr>
<tr>
<td>anmStationeryNotebook</td>
<td>The Stationery notebook. Used to store stationery.</td>
</tr>
<tr>
<td>anmInboxNotebook</td>
<td>The In box notebook. Used to import and store arriving documents.</td>
</tr>
<tr>
<td>anmOutboxNotebook</td>
<td>The Out box notebook. Used to queue and output documents.</td>
</tr>
<tr>
<td>anmAccessories</td>
<td>The accessories. Used for central storage of accessories and other documents.</td>
</tr>
</tbody>
</table>
Auxiliary Notebook Manager Messages

The messages and defines for clsAuxNotebookMgr are defined in the file AUXNBMGR.H. Table 113-2 lists the messages defined by clsAuxNotebookMgr.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgANMOpenNotebook</td>
<td>P_ANM_OPEN_NOTEBOOK</td>
<td>Activate and optionally open an auxiliary notebook.</td>
</tr>
<tr>
<td>msgANMGetNotebookPath</td>
<td>P_ANM_GET_NOTEBOOK_PATH</td>
<td>Returns the base path of one of the auxiliary notebooks.</td>
</tr>
<tr>
<td>msgANMCreateSect</td>
<td>P_ANM_CREATE_SECT</td>
<td>Create a section in one of the auxiliary notebooks.</td>
</tr>
<tr>
<td>msgANMCreateDoc</td>
<td>P_ANM_CREATE_DOC</td>
<td>Create a document in one of the auxiliary notebooks.</td>
</tr>
<tr>
<td>msgANMMoveInDoc</td>
<td>P_ANM_MOVE_COPY_DOC</td>
<td>Move a document into an auxiliary notebook.</td>
</tr>
<tr>
<td>msgANMCopyInDoc</td>
<td>P_ANM_MOVE_COPY_DOC</td>
<td>Copy a document into an auxiliary notebook.</td>
</tr>
<tr>
<td>msgANMDelete</td>
<td>P_ANM_DELETE</td>
<td>Delete a section or document in one of the auxiliary notebooks.</td>
</tr>
<tr>
<td>msgANMDeleteAll</td>
<td>P_ANM_DELETE_ALL</td>
<td>Delete all the nodes that are identified by 'ID'.</td>
</tr>
<tr>
<td>msgANMGetNotebookUUID</td>
<td>P_ANM_GET_NOTEBOOK_UUID</td>
<td>Returns the UUID of one of the auxiliary notebooks.</td>
</tr>
</tbody>
</table>

Generalized Auxiliary Notebook Manager Messages

Clients can use the following three messages (msgANMOpenNotebook, msgANMGetNotebookPath, and msgANMSystemInited) with all auxiliary notebooks.

Opening an Auxiliary Notebook

When you open an auxiliary notebook, it is displayed on the screen—much like opening a document causes it to be displayed on screen.

To open an auxiliary notebook send msgANMOpenNotebook to theAuxNotebook-Mgr. The only argument for the message is an ANM_AUX NOTEBOOK value that specifies the notebook to open.

Getting the Path to an Auxiliary Notebook

All auxiliary notebooks are stored in the directory \EPENPOINT\SYS\DTMGR. To get the complete path to an auxiliary notebook, send msgANMGetNotebookPath to theAuxNotebook-Mgr. The message takes a pointer to an ANM_GET_NOTEBOOK_PATH structure that contains:
notebook An ANM_AUX_NOTEBOOK value that specifies the notebook for which you want the path.

pLocator A pointer to the buffer to receive the path name. If the notebook does not exist, the message pLocator is set to pNull and the message returns stsOK.

Specialized Auxiliary Notebook Manager Messages

The messages described here apply only to the Help notebook, In box, Out box, and the accessories.

Creating Auxiliary Notebook Sections

To create a section in the Help notebook or In box and Out box notebooks, send msgANMCreateSect to theAuxNotebookMgr. The message takes a pointer to an ANM_CREATE_SECT structure, which contains:

- **notebook** An ANM_AUX_NOTEBOOK value that specifies the notebook in which to create the section.
- **sectClass** The class of the section.
- **pPath** A pointer to a path that indicates the location of the new section in the auxiliary notebook. You can get the path to an auxiliary notebook by sending msgANMGetNotebookPath to theAuxNotebookMgr. If the pointer is null, the message creates the section in the top level of the notebook.
- **pName** A pointer to a string that contains the name of the new section.
- **sequence** A sequence number that identifies where to insert the section.
  The new section is inserted in front of the document originally identified by the sequence number.
- **pBookmarkLabel** A pointer to a string that contains the bookmark label for the section. If the pointer is null, no bookmark is created.
- **exist** An ANM_EXIST_BEHAVIOR value that specifies what to do if the section exists or doesn't exist.
- **pDestPath** A pointer to the string that receives the path to the created section. If this pointer is null, the message won't return the path.
- **id** A U32 value used to identify contents of the section. When the installer deinstalls an application, it uses this identifier to locate and remove all related entries in the Stationery notebook, Help notebook, and so on.

If the message completes successfully, it returns stsOK.

You cannot send msgANMCreateSect to accessories.
Creating Auxiliary Notebook Documents

To create a document in the Help notebook, In box, Out box, or accessories, send msgANMCreateDoc to the AuxNotebookMgr. The message takes a pointer to an ANM_CREATE_DOC structure, which contains:

- **notebook** An ANM_AUX_NOTEBOOK value that specifies the notebook in which to create the document.
- **docClass** The class of the document.
- **pPath** A pointer to a path that indicates the location of the new document in the auxiliary notebook. You can get the path to an auxiliary notebook by sending msgANMGetNotebookPath to the AuxNotebookMgr. If the pointer is null, the message creates the document in the top level of the notebook.
- **pName** A pointer to a string that contains the name of the new document.
- **sequence** A sequence number that identifies where to insert the section.
  The new document is inserted in front of the document originally identified by the sequence number.
- **pBookmarkLabel** A pointer to a string that contains the bookmark label for the document. If the pointer is null, no bookmark is created.
- **exist** An ANM_EXIST_BEHAVIOR value that specifies what to do if the document exists or doesn't exist.
- **putInMenu** A BOOLEAN value that specifies whether a stationery document should be listed in the stationery menu.
- **pDestPath** A pointer to the string that receives the path to the document.
  If this pointer is null, the message won't return the path.
- **id** A U32 value used to identify contents of the document. When the installer deinstalls an application, it uses this identifier to locate and remove all related entries in the Stationery notebook, Help notebook, and so on.

If the message completes successfully, it returns **stsOK**.

Moving and Copying Documents to an Auxiliary Notebook

To move or copy documents into the Help notebook, In box, Out box, or accessories, send msgANMMoveInDoc or msgANMCopyInDoc to the AuxNotebookMgr. Both messages take a pointer to an ANM_MOVE_COPY_DOC structure that contains:

- **notebook** An ANM_AUX_NOTEBOOK value that specifies the notebook that will contain the document.
- **source** A locator to the document that will be moved or copied.
- **pPath** A path to the destination of the document within the auxiliary notebook. The path is relative to the auxiliary notebook's directory (which is a subdirectory of \PENPOINT\SYS\DTMGR). If you specify pNull, the document is moved or copied to the top level of the notebook.
defaultClass A class UID for the document if it isn't already stamped with a class.

sequence A sequence number that identifies where to insert the document. The document is inserted in front of the document originally identified by the sequence number.

pBookmarkLabel A pointer to a string that contains the bookmark label for the document. If the pointer is null, no bookmark is created.

exist An ANM_EXIST_BEHAVIOR value that specifies what to do if the document exists or doesn't exist.

forceInMenu A BOOLEAN value that specifies whether a stationery document should be listed in the stationery menu, regardless of any local attributes.

pDestPath A pointer to the string that receives the path to the created document. If this pointer is null, the message won't return the path.

id A U32 value used to identify contents of the document. When the installer deinstalls an application, it uses this identifier to locate and remove all related entries in the Stationery notebook, Help notebook, and so on.

Deleting an Auxiliary Notebook Section or Document

To delete a section or document from the Help notebook, In box, Out box, or accessories, send msgANMDelete to theAuxNotebookMgr. The message takes a pointer to an ANM_DELETE structure that contains:

notebook An ANM_AUX_NOTEBOOK value that specifies the notebook that contains the section or document to delete.

pPath The path to the section or document to delete.

To delete all nodes that have a specific application ID, send msgANMDeleteAll to theAuxNotebookMgr. The message takes a pointer to an ANM_DELETE_ALL structure that contains:

notebook An ANM_AUX_NOTEBOOK value that specifies the notebook that contains the documents to be deleted.

id A U32 value that identifies the application.

Modifying the Stationery Menu

clsAuxNotebookMgr defines two messages that clients can use to add and remove documents from the stationery menu.

Adding a Document to the Stationery Menu

To add a document that is in the Stationery notebook to the stationery menu, send msgANMAddToStationeryMenu to theAuxNotebookMgr. The message takes a pointer to an ANM_MENU_ADD_REMOVE structure that contains the directory index of document to add to the menu (document).
Removing a Document to the Stationery Menu

To remove a document from the stationery menu, send msgANMRemoveFromStationeryMenu to the AuxNotebookMgr. The message takes a pointer to an ANM_MENU_ADD_REMOVE structure, as described above in msgANMAddToStationeryMenu.
Chapter 114 / The System Class

The system class, clsSystem, provides information about the running PenPoint™ operating system. This chapter discusses these topics:

- The PenPoint booting sequence.
- The file system paths.
- The system messages.

**Concepts**

clsSystem manages PenPoint booting. There is only one instance of clsSystem, which has the well-known identifier theSystem. All clients send clsSystem messages to theSystem.

**Booting Sequence**

After the PenPoint machine interface layer (MIL) is loaded, clsSystem manages booting the PenPoint operating system. The SYS_BOOT_PROGRESS enum defines symbols for completion of each of the major steps in the boot sequence. Table 114-1 lists these enums in the correct boot sequence.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>sysKernelComplete</td>
<td>The PenPoint kernel is loaded.</td>
</tr>
<tr>
<td>sysSystemDllsComplete</td>
<td>The system DLLs listed in BOOT.DLC are loaded.</td>
</tr>
<tr>
<td>sysSystemAppsInstalled</td>
<td>The system applications listed in SYSAFP.INI are loaded.</td>
</tr>
<tr>
<td>sysInitialAppInstalled</td>
<td>The initial application (the Bookshelf) is installed.</td>
</tr>
<tr>
<td>sysBookshelfItemsCreated</td>
<td>The items on the Bookshelf notebooks are created.</td>
</tr>
<tr>
<td>sysServicesInstalled</td>
<td>The services listed in SERVICE.INI are installed.</td>
</tr>
<tr>
<td>sysAppsInstalled</td>
<td>The applications listed in APP.INI are installed.</td>
</tr>
<tr>
<td>sysInitialAppRunning</td>
<td>The initial application is running.</td>
</tr>
<tr>
<td>sysBootComplete</td>
<td>Booting is complete.</td>
</tr>
</tbody>
</table>

The initial application is the bookshelf application. Note that the items that appear on the bookshelf are not placed in the bookshelf until the bookshelf application is running (sysInitialAppRunning).

Your application or service might want to perform specific actions as soon as it is installed. However, if it depends on another component that is loaded later in the boot sequence, it can use theSystem to find out when booting has reached a particular step.
File System Paths

As part of managing booting, clsSystem defines a number of constants for file system paths to frequently used locations. Rather than adding your own string literals to your applications, you should use these constants.

For example, if you want to create a string that identifies where PenPoint applications live, do not use:

```c
strcpy(pFoo, "PENPOINT\APP");
```

Instead, you should use:

```c
strcpy(pFoo, sysBaseDir "\" sysInstallableAppDir);
```

Table 114-2 lists the file system path constants and their meanings.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>sysBaseDir</td>
<td>PENPOINT</td>
</tr>
<tr>
<td>sysInstallableFontDir</td>
<td>FONT</td>
</tr>
<tr>
<td>sysInstallablePrefDir</td>
<td>PREFS</td>
</tr>
<tr>
<td>sysInstallableHWXProtDir</td>
<td>HWXPROT</td>
</tr>
<tr>
<td>sysInstallableGestureDir</td>
<td>GESTURE</td>
</tr>
<tr>
<td>sysInstallablePDictDir</td>
<td>PDICT</td>
</tr>
<tr>
<td>sysInstallableAppDir</td>
<td>APP</td>
</tr>
<tr>
<td>sysInstallableServiceDir</td>
<td>SERVICE</td>
</tr>
<tr>
<td>sysBootDir</td>
<td>BOOT</td>
</tr>
<tr>
<td>sysQuickInstall</td>
<td>QINSTALL</td>
</tr>
<tr>
<td>sysRuntimeRootDir</td>
<td>SYS</td>
</tr>
<tr>
<td>sysSysAppFile</td>
<td>SYSAPI.INI</td>
</tr>
<tr>
<td>sysAppFile</td>
<td>APP.INI</td>
</tr>
<tr>
<td>sysSysServiceFile</td>
<td>SYSSERV.INI</td>
</tr>
<tr>
<td>sysServiceFile</td>
<td>SERVICE.INI</td>
</tr>
<tr>
<td>sysCopyFile</td>
<td>SYSCOPY.INI</td>
</tr>
<tr>
<td>sysResFile</td>
<td>PENPOINT.RES</td>
</tr>
<tr>
<td>sysMILResFile</td>
<td>MIL.RES</td>
</tr>
<tr>
<td>sysLiveRoot</td>
<td>Bookshelf</td>
</tr>
<tr>
<td>sysLoaderDir</td>
<td>LOADER</td>
</tr>
<tr>
<td>sysDefaultInitialApp</td>
<td>Bookshelf</td>
</tr>
</tbody>
</table>

Default Initial App (in PENPOINT\BOOT\APP)

Just as a reminder, you can use the `BootVolume` to identify the volume from which the PenPoint operating system was booted. You can use the `SelectedVolume` to identify the volume that contains the PenPoint run-time information.

### The System Messages

Table 114-3 lists the messages defined by `clsSystem` in the file `SYSTEM.H`.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>msgSysGetBootState</code></td>
<td><code>P_SYS_BOOT_STATE</code></td>
<td>Passes back the current booting stage.</td>
</tr>
<tr>
<td><code>msgSysGetRuntimeRoot</code></td>
<td><code>P_OBJECT</code></td>
<td>Passes back a dir handle onto the root of the Penpoint runtime area.</td>
</tr>
<tr>
<td><code>msgSysGetLiveRoot</code></td>
<td><code>P_SYS_GET_LIVE_ROOT</code></td>
<td>Passes back an appDir handle onto the root of a volume's live document area.</td>
</tr>
<tr>
<td><code>msgSysIsHandleLive</code></td>
<td><code>P_SYS_IS_HANDLE_LIVE</code></td>
<td>Determines if a filesystem handle is within the live document area.</td>
</tr>
<tr>
<td><code>msgSysCreateLiveRoot</code></td>
<td><code>P_SYS_CREATE_LIVE_ROOT</code></td>
<td>Create a new live root on a volume.</td>
</tr>
<tr>
<td><code>msgSysGetVersion</code></td>
<td><code>P_U16</code></td>
<td>Passes back the system version number.</td>
</tr>
<tr>
<td><code>msgSysGetSecurityObject</code></td>
<td><code>P_OBJECT</code></td>
<td>Gets the current security object.</td>
</tr>
<tr>
<td><code>msgSysSetSecurityObject</code></td>
<td><code>P_SYS_SET_SECURITY_OBJECT</code></td>
<td>Sets the current security object.</td>
</tr>
<tr>
<td><code>msgSysGetCorrectiveServiceLevel</code></td>
<td><code>P_STRING</code></td>
<td>Gets the corrective service level.</td>
</tr>
<tr>
<td><code>msgSysSetCorrectiveServiceLevel</code></td>
<td><code>P_STRING</code></td>
<td>Sets the corrective service level.</td>
</tr>
</tbody>
</table>

### Boot Progress Messages

There are two ways your application can find out the progress of system booting:

- It can make itself an observer of the `System`. When the boot state changes, it will receive `msgBootStateChanged`.
- It can send `msgSysGetBootState` to `theSystem` to find out the current stage.

Both messages pass a pointer to a `SYS_BOOT_STATE` structure, which contains:

- **booted** A `BOOLEAN` value that indicates whether booting has completed (true) or not (false).
- **progress** A `SYS_BOOT_PROGRESS` value (described above) that indicates the current boot stage.
- **type** A `SYS_BOOT_TYPE` value that indicates the boot type. There are two types of boots: cold-boot (`sysColdBoot`) and warm-boot (`sysWarmBoot`). When booting on a PC, the boot type will be...
sysColdBoot: when booting tablet hardware, the boot type will always be sysWarmBoot.

initialAppClass  A UID that indicates the class of the initial application.

System Directory Messages

As described in Chapter 110, PenPoint defines specific directories for many of its system files. clsSystem defines several messages that allow you to access these files and directories.

You can send msgSysGetRuntimeRoot to theSystem to get a directory handle on the root of the PenPoint runtime area.

If a PenPoint volume has a bookshelf (in \PENPOINT\SYS\DOC), this is said to be the "live" area for documents on the volume. You can use msgSysGetLiveRoot to access the live area on a volume. The live area can be on any PenPoint volume. To find out if a handle is on a file or directory within the live area, send msgSysIsHandleLive to theSystem.
Part 13 / Writing PenPoint Services
Chapter 115 / Introduction

This manual describes how to write services for the PenPoint™ operating system. PenPoint services are separately installable, non-application DLLs that provide system extensions such as database engines, e-mail backends, and PenPoint device drivers.

**Intended Audience**

This manual is written for people who are designing and implementing PenPoint services.

Writing a service requires a good knowledge of PenPoint, its object-oriented design, and its message passing mechanism.

We expect that you are familiar with the PenPoint operating system and object-oriented programming concepts, proficient in the C programming language, and—if you are writing a device drive—that you have some familiarity with hardware interfaces or data communications protocols.

**Layout of This Manual**

This chapter, Chapter 115, provides an overview to services, describes the intended audience for this manual, describes the organization of the manual, and provides pointers to other sources of information.

Chapter 116, Service Concepts, describes the concepts that you need to know before writing a service, but doesn’t describe how to implement these concepts.

Chapter 117, Programming Services, describes the actions that your service must take in response to a message.

Chapter 118, Distributing Your Service, describes how to organize your service so that it works correctly with the installer architecture.

Chapter 119, Test Service Examples, provides listings of the sample services that are distributed in \PENPOINT\SDK\SAMPLE.

**Other Sources of Information**

Chapter 94, Using Services, in *Part 10: Connectivity* describes services from the client’s point of view. To write an application that uses or tests your service, you will need to understand this information.

If you are writing a device driver, you will need documentation for the device with which you intend to communicate.
Chapter 116 / Service Concepts

This section discusses the concepts that you need to know before writing a service. Chapter 117 describes how to implement these concepts.

Service Manager Architecture

The PenPoint service architecture enables the PenPoint™ operating system to manage installable and deinstallable non-application facilities. The need for a service manager architecture was dictated by two conflicting needs in the PenPoint operating system:

• PenPoint does not depend on a disk being available, so memory is usually limited.
• PenPoint machines are portable and may require device drivers for several different devices (for example, different locations might have different printers).

If memory is tight, users probably only want to load the services that they need. If the user goes to an office that has a different printer, the PenPoint service architecture allows the user to deinstall the current printer driver and load the printer driver for the new device.

The service manager:

• Manages non-application facilities.
• Allows arbitrary categories of services.
• Manages finding, binding, and opening services.
• Provides notification for addition, removal, connection, and disconnection of services.
• Gracefully handles the case where a requested service is not available.

Generally, services:

• Provide installable and deinstallable, non-application system extensions.
• Can target other services to form chains.
• Do not require their target to be present when they are created (delayed binding).
• Can have owners.
Applications, Components, and Services

There are three types of programs that you might write for the PenPoint operating system: applications, components, and services.

Applications

Applications are PenPoint classes that inherit from clsApp. Generally, applications:
- Provide the interface through which users interact with the system.
- Provide some computational work.
- Use services and components to perform more sophisticated tasks.
- Respond to the PenPoint Application Framework messages.

Components

Components are DLLs that provide specific application capabilities. Components can be seen as building blocks that developers can use to add functionality to their application.

Components can provide a user interface (but don’t have to). Components do not inherit from clsApp. Typical examples of components are:
- clsTextView
- clsMiniNote.

Services

PenPoint services are installable, non-application DLLs that provide system extensions. Services inherit from clsService. While components can be used by any number of users, most services require controlled access; the access to services is controlled by the service managers.

The philosophy behind the services architecture is that no part of PenPoint should expect any particular hardware to be present. Other system extensions provided by services are:
- Database engines
- E-mail backends
- Installable file systems.

Not all services provide a user interface. Services also do not respond to the PenPoint Application Framework messages.

Services in PenPoint

PenPoint uses services to provide system extensions. This section outlines the various services provided by PenPoint. You can use these services and their organization to model how your services relate to PenPoint. You can also use these services to model how your services present their user interface (do they have an option card? do they require a configuration accessory?).
MIL Services

PenPoint MIL services perform the functions usually performed by device drivers in other operating systems. The MIL services handle requests to communicate directly with ports. (Not all services are directly connected to devices; services can be chained to act like a protocol stack.)

The MIL services that communicate directly with device ports are usually loaded at boot time (depending on the configuration of the machine). Two examples of port MIL services are:

- **Serial Port** Controls the serial port or ports.
- **Parallel Port** Controls the parallel port or ports.

Other services and MIL services that communicate directly with optional devices can be loaded at boot time or can be installed by the user. Two examples of non-port services are:

- **DotMatrix** Handles dot-matrix printer requests from a printer window device and communicates printer commands through a parallel port.
- **Modem** Handles modem requests from applications and communicates AT commands through the serial port.

Other Services

Other types of services in PenPoint are the DOS and Macintosh file systems and the replaceable shape matcher.

Classes Used by Services

When developing a service, you must know about three classes:

- **clsService** The service class.
- **clsInstallMgr** The install manager class.
- **clsServiceManager** The service manager class, which inherits from clsInstallMgr.

The Service Class and Service Instances

A service class implements the functions for a particular type of device or function. For each device or function configured in a running PenPoint system, there is an instance of a service class.

An instance of a service can be created:

- By the service's DLLMain at install time.
- By the service installer at install time from pre-configured instance state data.
- Dynamically at the user's request (through a user interface, such as the Printers).

Each service class is a subclass of clsService and is implemented as a separate DLL.
A service instance receives messages from its service manager and, when opened, from its opener. Most of the messages sent by the service manager are defined by clsService. The service instance does little work with these messages, but passes them up to its superclass, clsService.

The service instance does most of its work when receiving requests from its openers. A service instance usually interacts with its openers by handling messages from the openers. However, the service can also tell its openers the entry points to specific procedural interfaces (if any). This allows openers to access time-critical services without the overhead of object calls.

### The Service Manager Class

A service manager maintains a list of instances that share the same minimum API. There can be instances of different service subclasses on the same service managers. Each service instance must belong to at least one service manager; service instances can be listed on more than one service manager.

When a client wants to use a service, the client communicates with the service manager that has the needed service on its list. Service instances can be on more than one service manager list.

Service managers are instances of clsServiceManager. Most service managers are created by the PenPoint operating system at boot time.

Figure 116-1 shows a service manager and its service instances.

When writing a client that uses services, you use the messages defined by clsServiceManager to find, express interest in, and open a service instance. Clients can observe individual services, or can observe a service manager.
The Service Installation Manager Class

The service installation manager handles installation and deinstallation of service classes. For each installed service, the service installation manager creates and manages a directory in the RAM file system.

There is only one service installation manager in the system, `theInstalledServices`, which is an instance of `clsServiceInstallMgr`.

When getting information about installed services, you use messages defined by `clsServiceInstallMgr` and its ancestor, `clsInstallMgr`.

Open Service Object Class

When a client opens a service, the service manager passes back a service instance UID to the client. The client then communicates with the service by sending messages to the UID.

While most services allow only one opener per service instance, some services allow more than one client to access the same service instance. When designing multiple-access services, you (the service developer) must decide whether the service should give the UID of the service instance to each opener or if the service should give a separate, placeholder UID to each opener.

But where do you get these UIDs? PenPoint provides a class, the open service object class (`clsOpenServiceObject`), that you can subclass. When a client opens your service, you create an instance of the subclass and give the UID of the subclass back to the client. `clsOpenServiceObject` inherits from `clsStream`.

The advantage of giving separate UIDs to each opener is that your service can use the open service object subclass to maintain the state of each of the openers individually.

Service Overview: Installation to Use

There are four parts to the life cycle of a service:

- Installing the service class.
- Creating a service instance.
- Accessing the service instance.
- Responding to service messages.

This section presents an overview of a service, beginning with service installation to the service responding to client messages.

Installing a Service Class

There are three ways that service classes can be installed:

- By a cold-boot of an SDK version of PenPoint (the service is listed in the SERVICE.INI file).
- By the quick installer (the volume specifies quick install from the services; requires some user interaction).
By the user (the user installs a service from the Connections notebook or the Settings notebook).

When installation is initiated, the InstalledServices finds the service on disk, copies the service DLL files into the loader data base, and copies non-executable files to the RAM file system.

theInstalledServices then calls the service's DLLMain() routine, which creates the service class by subclassing clsService. The DLLMain() routine also creates any other classes required by the service.

Creating Service Instances

Once the DLLMain routine creates the service class, a client can send msgNew to the service class to create a service instance. The arguments to msgNew for a service include a list of service managers that will manage this service instance and a target for the service instance (if any).

Usually the service class does little with msgNew, but passes it to its superclass. clsService creates the new service instance and adds the instance to the lists maintained by the specified service managers.

Depending on the service flags, clsService can also bind the service instance to its target when it is created.

If a service has a fixed number of service instances, the service can create its own instances from its DLLMain() routine. Otherwise, the service should provide some application or accessory through which users can create and destroy instances of a service, rename service instances, or specify or change the target of a service. (Service targets are explained later in this chapter.)

Using a Service Instance

When the service instance appears on one or more service manager lists, a client can locate, bind to, and then open the service instance:

1. The client finds the service instance by sending msgIMFind to the appropriate service manager, specifying the name of the service instance. msgIMFind passes back a handle on the service instance.
2. The client binds to the service handle by sending msgSMBind to the service manager. This makes the client an observer of the service instance.
3. The client sends msgSMSetOwner to the service manager. This allows the client to access exclusive-access services.
4. The client sends msgSMOpenDefaults to the service manager, so that the service instance can provide default values for the open pArgs (if any).
5. The client opens the service instance by sending msgSMOpen to the service manager. If the open is successful, the service manager passes back the UID of the service instance or the open service handle. A client should open a service instance only when it is ready to send or receive data.
6 The client uses the service instance to perform some action. Usually this consists of sending `msgStreamRead` or `msgStreamWrite` to the service instance.

7 The client should close the service instance (by sending `msgSMClose` to the service manager) as soon as it has finished sending or receiving data.

8 Finally, the client unbinds from the service instance by sending `msgSMUnbind` to the service manager. The client must close the service before unbinding from it.

Chapter 94, Using Services, in *Part 10: Connectivity* describes these steps in much greater detail.

### Responding to Service Messages

For some messages that a client sends to a service manager, the service manager sends a corresponding message to the appropriate service instance. Most of these messages inform the service instance that a request was made and allow the service instance (or its superclass) to veto the request or provide some information.

When the client sends `msgSMBind` to the service manager, the service manager sends `msgSvcBindRequested` to the service instance; when the client sends `msgSMOpen` to the service manager, the service manager sends `msgSvcOpenRequested` to the service instance, and so on.

When the service instance receives `msgSvcOpenRequested`, it must allow `clsService` to handle the message, then, if the service instance allows multiple openers, it creates an instance of its open handle class and passes that UID to the client.

When a client has opened a service instance, it sends messages directly to the service instance to perform some work. The service class can define methods for messages created by its ancestors (such as `msgStreamRead` and `msgStreamWrite`) or it can define methods for its own messages.

### Services and the File System

Each service has its own directory. The name of the service directory is the name of the service. In installation volumes, service directories are in the `\OPENPOINT\SERVICE` directory. In a running system, service directories are in the `\OPENPOINT\SYS\SERVICE` directory.

Each service instance is represented by a file system node. The node is usually a file, but can be a directory if the service desires. A service uses its file system node to save its state information. For example, a service that uses the serial port can save the baud rate and other serial communication parameters.

When a service is installed and instances of the service are created, its service manager creates a file system handle on the node for the service. Each service manager maintains a list of open file system handles for the service nodes it is responsible for.
The service instance nodes live in a subdirectory of the `\PENPOINT\SYS\SERVICE\INST`.

### Services on Distribution or Boot Disks

In a distribution or boot volume, service directories are in the `\PENPOINT\SERVICE` directory. A service directory can contain the following files, as shown in the directory hierarchy below:

- One or more service DLL files.
- A `.DLC` file (if there is more than one DLL file) that lists the DLLs required by this service.
- An optional `INIT.DLL` file that is loaded, run, and unloaded during service installation.
- An optional service resource file, `SERVICE.RES`, which is similar to an application's `APP.RES` file.
- An optional `MISC` directory (similar to an application's `MISC` directory), which can contain static data files that are common to all service instances.
- An optional ` INST` directory, which contains one or more **service state nodes** (a node is either a file or a directory) containing instance data for saved service instances. The service install manager uses this data to create pre-configured service instances when the service is installed.
- An optional `SERVICE.INI` file that specifies additional services that must be installed when this service is installed.
- An optional `APP.INI` file that specifies applications that must be installed when this service is installed.

---

If there is only one DLL file and no `.DLC` file, the DLL file must have the same name as the service (with a `.DLL` extension). For example, the FLAP service directory contains FLAP.DLL.

If there is more than one DLL file, the `.DLC` file must have the same name as the service (with a `.DLC` extension). For example, the PCL service directory contains PCL.DLC (along with PCL.DLL, OBXSERV.DLL, CLSBND.DLL, and CLSPRN.DLL).
CHAPTER 116 / SERVICE CONCEPTS
Exclusive and Multiple Access Services

Services in the Selected Volume

When a service is installed, its service directory is created in \PENPOINT\SYS\SERVICE. The service directory contains:

- An INST directory that contains one or more service state nodes (a node is either a file or a directory). Each service state node contains instance data for a specific service instance.
- An optional MISC directory that contains static data files common to all instances of the service.
- The optional SERVICE.RES file that contains the service's UI components, quick help resources, and other read-only resources.

Services in File System at Run Time

When clsService creates a new instance of a service class, it creates a service state node (either a file or a directory) in the INST directory of the service's directory (that is, in \PENPOINT\SYS\SERVICE\MyService\INST). The service instance uses this state node to store any state that needs to survive a warm boot. There is no explicit save and restore sequence for services. A service must update its state node whenever its state changes.

Exclusive and Multiple Access Services

Exclusive access services allow only one client to access their instances at a time; multiple access services can be accessed by more than one client at a time. Exclusive access services usually represent a physical device, such as a serial port or a printer. Multiple access services usually represent an abstract entity that allows simultaneous access, such as a network or a database.

Exclusive Access Services

There are two ways of specifying that a service instance has exclusive access:

- A service instance can allow only one opener at a time (specified by a flag).
- A service instance can be opened only by its owner.

Each service instance has an owner field. This field can be NULL, meaning the service has no owner. Each service instance also has a flag that, when true, specifies that only the owner can open the service.

When a client wants to become the owner of a service instance (or when a third-party wants to change the owner of a service), it sends msgSMSetOwner to a service manager for the service instance. The service manager then negotiates with
the current owner, the new owner, and the service itself to ensure that all are prepared to change ownership. The protocol for changing ownership is explained in Chapter 117.

**Multiple Access Services**

There are two different types of multiple access services:

- Shared services, where each opener of a service instance receives the UID of the service instance; all openers send requests to the same UID.

- Multiuser services, where each opener of a service instance receives the UID of a unique open service object; each opener sends its requests to the UID of its own open service object.

Any client that has the UID of a shared service can send messages to the service instance at any time, even while the service is handling another request. You must use semaphores to protect access, where necessary.

Multiuser services provide a convenient place to store caller-specific state. When a client opens a multiuser service, clsService sends msgNew to a a subclass of clsOpenServiceObject (the subclass was created by the service class when it was installed). When a client closes a multi-user service, clsService sends msgDestroy to the open service object.

**Targeting and Chaining Services**

Service instances can bind and open other service instances or targets.

When the service instance is first created, the default behavior is to attempt to bind to its target.

When a client opens a service, a style bit in the service’s metrics can tell the service to automatically open its target.

When a service becomes a client to its target all client observer notifications and ownership messages from the target are also sent to the client service. Information about ownership is also propagated down the target chain.

Services that communicate directly with the hardware do not have a target.

**Service Connection**

Services support the notion of knowing whether they are connected or not. When a device is connected to a port, the service can communicate with the device. Each service has a state bit that indicates whether it is connected or not.

Some services are able to detect when a connection is made or is broken; other services cannot detect connections. The service writer specifies whether a service can detect connections when creating a service instance. As a third alternative the service writer can tell the service to take on the connection state of its target. Usually services that deal directly with a hardware interface specify auto-detect or non-auto-detect; all other services follow their target’s connection state.
The connection state is important to services that have been enabled. To relieve the user of having to turn to the Out box notebook and check Enabled every time the PenPoint computer is attached to a printer, the user can leave Enabled checked all the time for an auto-detecting device. A good example is a printer on a TOPS network: when the PenPoint computer is connected to the TOPS network, the service detects that a connection has been made and starts to send data to the port.

Similarly, if connection is broken, an auto-detecting service will be able to stop its current action and prompt the user to reestablish the connection or cancel the operation.

Most hardware services can detect whether their hardware is connected or disconnected. Each service has a state bit which says whether it is connected or not. When the hardware changes connection state, the service sends a notification message to itself. This notification message then notifies all clients that are bound to that service.

Non-hardware services automatically change their connection state when their targets change connection state. Thus, connection state propagates up from the hardware to all services that are bound to that hardware.

A hardware service can also decide that it cannot detect hardware connect. In this case the service is always marked connected.
Chapter 117 / Programming Services

This chapter describes the code that you must write to implement a service.

Object-Oriented Architecture

Before you write a PenPoint™ service, you must be familiar with the object oriented design of PenPoint. It would help if you have written an application or component for PenPoint already, particularly one that uses the service manager to access a service.

The object oriented design of PenPoint requires a service to be a subclass of clsService. clsService provides methods for most messages that your service will receive from a service manager. Your service handles many clsService messages by calling ancestor.

A client uses a service manager to bind to and open an instance of your service. Once the client has opened your service instance the client sends messages directly to your service, and your service begins to perform its work.

Design Decisions

Chapter 116 described the concepts of services and their organization. In describing the service concepts, the chapter presented many options that are available with PenPoint services. When you design your service, you must answer these questions:

- Is the service an exclusive or multiple access service?
- If the service allows multiple accessors, is the service shared (all clients use the same UID) or is it multiuser (clients use an instance of an open service object class)?
- Is the service targeted? If not, is it a hardware service?
- What kind of state data will the server need? Should the state node be a file or should it be a directory?
- What does the user need to do to create instances of your service?
- Which service manager will manage your service, or do you need to create a new service manager?

Using the Template Services

To make the job of writing a service easier, GO provides two template services, clsTestSvc and clsMilSvc, which you can modify to implement your own service. The files used to implement clsTestSvc are in the directory \PENPOINT\SDK\SAMPLE\TESTSVC; the files for clsMilSvc are in \PENPOINT\SDK\SAMPLE\MILSVC.
If you are writing a multiple access, multi-user service, GO also provides a
template open service object class, clsTestOpenObject, which is a subclass of
clsOpenServiceObject. The files for this class (OPENOBJ.H and OPENOBJ.C)
are in the PENPOINT\SDK\SAMPLE\TESTSVC directory.

The rest of this chapter uses the sample service files for many of its examples; you
should have them available to you while you read. If you do not have the sample
service files, please contact GO Developer Technical Support.

**Service Installation**

When your service is installed, the service installer locates your service's directory
and looks in the directory for a DLL or .DLC file with the same name.

To create your **DllMain**, you start with the INIT.C file in TESTSVC.

```c
STATUS EXPORTED DllMain(void)
{
    TEST_SVC_NEW       testSvcNew;
    STATUS            s;
    // Install classes.
    StsRet(TstsvcSymbolsInit(), s);
    StsRet(ClsTestServiceInit(), s);
    StsRet(ClsTestOpenObjectInit(), s); // Only needed if multi-user.
    // Let system know about our service and set global service
    // characteristics. THIS IS REQUIRED!
    StsRet(InitService(pNull,
                      clsTestService, // Service class.
                      false, // Autocreate instances? Set to false
                      if all instances are statically
                      created in DllMain();
                      0, // Global service type. This would be
                      set to a tag if you are using type
                      categorization (ie. printers, e-mail)
                      svcPopupOptions, // Flags. Or-in options.
                      0, // Reserved. Always set to 0.
                      0), s); // Reserved. Always set to 0.
    // Create any static service instances. YOU MUST SET autoCreate in
    // InitService() TO FALSE IF YOU DO THIS! If you know exactly how many
    // service instances there will be, this is where you should create them.
    // Alternatively, they can be dynamically created elsewhere, after
    // DllMain() runs. Beware of process ownership issues if you create
    // services dynamically. You must ensure that the process which owns
    // the dynamically created service instance will be around for the
    // lifetime of the service instance!
    ObjCallWarn(msgNewDefaults, clsTestService, &testSvcNew);
    testSvcNew.svc.pServiceName = "Testing, testing..."; // Name of this instance
    testSvcNew.svc.target.svc.manager = theTransportHandlers; // Target svc manager
    strcpy(testSvcNew.svc.target.svc.pName, "Netware"); // Target name
    ObjCallRet(msgNew, clsTestService, &testSvcNew, s);
    return stsOK;
} // DllMain
```

The following sections describe issues to consider while coding your **DllMain**
function.
Calling Your Service Initialization Routines

The first step of DLLMain() is to install your service class and any other classes required by your service.

```c
// Install classes.
StsRet(TstsvcSymbolsInit(), s);
StsRet(CIsTestServiceInit(), s);
StsRet(CIsTestOpenObjectInit(), s); // Only needed if multi-user.
```

The template service calls two initialization routines:

- **ClsTestServiceInit()** Initializes and installs your service class.
- **ClsTestOpenObjectInit()** (optional) Initializes and installs your class for open service objects.

You can add other initialization routines for other classes that you require. The following sections describe the routines for the service class and the open service object class in detail.

Initializing Your Service Class

As with an application or component, you must create your class by sending msgNew to clsClass.

```c
/**********************************************************
ClstServiceInit
***********************************************************/
STATUS FAR PASCAL ClsTestServiceInit(void)
{
    STATUS s;
    CLASS_NEW new;

    // Create the service class.
    ObjCallWarn(msgNewDefaults, clsClass, &new);
    new.object.uid   = clsTestService;
    new.cls.pMsg     = clsTestServiceTable;
    new.cls.ancestor = clsService;
    new.cls.size     = SizeOf(INSTANCE_DATA);
    new.cls.newArgsSize = SizeOf(TEST_SVC_NEW);
    ObjCallRet(msgNew, clsClass, &new, s);

    return s;
} // ClsTestServiceInit
```

The ClsTestServiceInit function:

- Sends msgNewDefaults to clsClass.
- Sets the service's well-known UID, message table UID, ancestor, and size of data.
- Sends msgNew to clsClass.

If necessary, the ClsTestServiceInit() function could also initialize any data used by the service.
**Initializing an Open Service Object Class**

If a service allows multiple openers, there are two ways to give clients an identifier for the service, by giving all openers the same UID (called a shared service) or by giving each opener an instance of an open service object class (called a multi-user service).

Multiuser services provide a convenient place to store caller-specific state.

If you implement a multiuser service, you must create your own open service object class by subclassing clsOpenServiceObject. The template service creates its open service object class by calling ClsTestOpenObjectInit(), which is defined in OPENOBJ.C.

```pascal
STATUS FAR PASCAL ClsTestOpenObjectInit(void)
{
    STATUS s;
    CLASS_NEW new;

    // Create the service class.
    ObjCallWarn(msgNewDefaults, clsClass, &new);
    new.object.uid = clsTestOpenObject;
    new.cls.pMsg = clsTestOpenObjectTable;
    new.cls.ancestor = clsOpenServiceObject;
    new.cls.size = SizeOf(INSTANCE_DATA);
    new.cls.newArgsSize = SizeOf(TEST_OPEN_OBJECT_NEW);
    ObjCallRet(msgNew, clsClass, &new, s);

    return s;
} // ClsTestOpenObjectInit
```

**Calling Other Class Initialization Routines**

If your application requires private classes in addition to the service class itself, you should call their initialization routines soon after creating your service class (to make your initialization procedure orderly, if for no other reason).

**Calling InitService**

InitService informs theInstalledServices about the service and sets global service characteristics. Other than creating the service class, sending msgSvcClassInitService is one of the most important functions in a service’s DLLMain(). msgSvcClassInitService takes a pointer to an SVC_INIT_SERVICE structure that contains:

- **autoCreate** A BOOLEAN value that specifies whether the service manager should automatically create an instance for each state node at installation and warm boots. If this parameter is true, InitService locates the service state nodes in the service’s INST directory and creates instances of the service from the nodes.
serviceType A U32 that contains the service type (defined in SVCTYPES.H). This is usually 0.

initServiceFlags A set of flags that specify how the installer should treat this service. You can specify more than one flag by ORing flags. The flags are:

svcNoShow Don't show this service in the installer. If this flag is set, the user can't configure or deinstall the service.

svcPopupOptions Automatically pop up the global service option card when this service is installed. If the service is being reactivated, the option card will not be displayed.

svcNoLoadInstances When the service is installed, don't copy in the state files from the INST directory.

svcCreatePrivateServiceMgr Create a private service manager for instances of this class. All instances of this class will automatically be added to the private service manager. See SVC_CLASS_METRICS for UID of the private service manager.

svcFullEnvironment Generate a complete process environment in the DLLMain() process. In PenPoint 1.0, this means creating the ProcessResList. Also, a service resource file handle will be created even if the service resource file is empty. A complete process environment takes up significant memory. Only turn this bit on if you need it.

This is the InitService call from INIT.C for the template service.

```c
STATUS EXPORTED DLLMain(void)
{
    SVC_INIT_SERVICE initService;
    STATUS s;
    StsRet(ClsTestServiceInit(), s);
    memset(initService.spare, 0, sizeof(initService.spare));
    initService.autoCreate = true;
    initService.serviceType = 0;
    initService.initServiceFlags = 0;
    ObjCallRet(msgSvcClassInitService, clsTestService, &initService, s);
    return stsOK;
} // DLLMain
```

**Static and Dynamic Service Instances**

If you know exactly how many service instances there will be, you should create them soon after calling InitService (by sending msgNewDefaults and msgNew to your service class, described below). If you create static service instances, the InitService flag autoCreate must be false.

Alternatively, you can create service instances dynamically after DLLMain() runs. However, you must beware of process ownership issues if you create services dynamically. You must ensure that the process that owns the dynamically created service instance will be around for the lifetime of the service instance.
This is the template service code for creating a static service instance:

```c
ObjCallWarn(msgNewDefaults, clsTestService, &testSvcNew);
testSvcNew.svc.pServiceName = "Testing, testing...");  // Name of this instance
testSvcNew.svc.target.svc.manager = theTransportHandlers;  // Target svc manager
strcpy(testSvcNew.svc.target.svc.pName, "Netware");  // Target name
ObjCallRet(msgNew, clsTestService, &testSvcNew, s);
```

If you don’t know how many instances of your service will be created, you must
create them dynamically. You must ensure that the process owner of the service
instance remains alive for the life of the service instance; if the process owner is
destroyed, the service instance is also destroyed.

### Creating Service Instances

You create new instances of your service by sending `msgNewDefaults` and
`msgNew` (or `msgObjectNew`) to your service class. Because your service class
inherits from `clsService`, the _NEW structure that your service class takes must
include the SVC_NEW_ONLY structure, which is defined by `clsService`.

```c
SVC_NEW_ONLY contains eight arguments:
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>An SVC_TARGET structure that specifies the initial target instance for this</td>
</tr>
<tr>
<td></td>
<td>service instance. The SVC_TARGET structure contains:</td>
</tr>
<tr>
<td></td>
<td>target.manager          The UID of the target’s manager object. If the service</td>
</tr>
<tr>
<td></td>
<td>is a MIL service for a hardware device, set this field to objNull.</td>
</tr>
<tr>
<td></td>
<td>target.pName            A pointer to a string that contains the name of the</td>
</tr>
<tr>
<td></td>
<td>target service instance.</td>
</tr>
<tr>
<td>pServiceName</td>
<td>The name of this service instance. This is the name of the</td>
</tr>
<tr>
<td></td>
<td>service state node for this instance and is the name that the user sees.</td>
</tr>
<tr>
<td>style</td>
<td>A SVC_STYLE structure that specifies the attributes of the service</td>
</tr>
<tr>
<td></td>
<td>instance. The possible values for style are:</td>
</tr>
<tr>
<td></td>
<td>waitForTarget            Specifies that the service will wait until its target</td>
</tr>
<tr>
<td></td>
<td>appears. When the target appears the service will attempt to bind to it.</td>
</tr>
<tr>
<td></td>
<td>If you do not specify waitForTarget and the specified target doesn’t exist,</td>
</tr>
<tr>
<td></td>
<td>msgNew will fail.</td>
</tr>
<tr>
<td></td>
<td>exclusiveOpen            Specifies that only one client is allowed to open</td>
</tr>
<tr>
<td></td>
<td>the service instance at a time.</td>
</tr>
<tr>
<td></td>
<td>autoOwnTarget            Specifies that the service instance should attempt</td>
</tr>
<tr>
<td></td>
<td>to set itself as the owner of its target when it receives msgSvcChangeOwner-</td>
</tr>
<tr>
<td></td>
<td>Requested.</td>
</tr>
<tr>
<td></td>
<td>autoOpen                 Specifies that the service instance should open and</td>
</tr>
<tr>
<td></td>
<td>close its target when the service instance is opened and closed.</td>
</tr>
<tr>
<td></td>
<td>autoMsgPass              Specifies that all messages not used by the service</td>
</tr>
<tr>
<td></td>
<td>should be passed to its target.</td>
</tr>
<tr>
<td></td>
<td>checkOwner               Specifies that a client must own this service instance</td>
</tr>
<tr>
<td></td>
<td>before it can open it.</td>
</tr>
</tbody>
</table>
autoOption  Specifies that clsService should forward all option sheet messages to the target of this service instance.

connectStyle  Specifies whether the service instance can detect hardware connections (svcAutoDetect) or cannot detect connections (svcNoAutoDetect), or whether it should follow the connection state of its target (svcFollowTarget).

openClass  Specifies the open service object class for multiple access, multi-user services that pass back a UID from msgSMOpen. If there is no open service object class, you must specify objNull.

handleClass  The handle class used to access the service state node. If the service state node is a file, you should specify clsFileHandle; if the service state node is a directory, you should specify clsDirHandle.

fsNew  An FS_NEW structure that contains msgNew arguments for the handle class specified in handleClass. The structure must be initialized if the handleClass is anything other than clsFileHandle.

fsNewExtra  An array of 25 U32 values that you can use to pass extra fsNew data.

pManagerList  A pointer to an array of service manager UIDs that will administer this service instance. clsService adds the service instance to each service manager. A service instance can be administered by more than one service manager.

numManagers  The number of managers in the pManagerList array.

Services and Tasks

Just like any other object, a service instance is owned by some task. However, because objCapCall is always true for service instances, all service instances must be callable from outside the owning task. This affects two aspects of services:

• Where the service instance stores its data.
• Which task owns the service instance.

Service Data

Service instances must never store data in a local heap; they should either store data in their instance data or in global heaps. (See the PenPoint Architectural Reference, Part 1: Class Manager for more information on instance data and storage.)

Owning Task

The service's owning task must remain active for the life of the service instance.

If you create service instances in DLLMainO (either static service instance or dynamically from saved state nodes), the service instance is owned by the task executing the DLLMainO (the main task of the service). A service's main task remains active until the service is deactivated or deinstalled.
However, if a service instance is created by a transient user interface task such as a document or a tool, the service instance will become invalid when the user turns away from the document or closes the tool. You have two alternatives:

- Keep the creating task around for the lifetime of the service instance.
- Use msgObjectNew, rather than msgNew, to create the service instance.

msgObjectNew creates an object in the context of the object to which the message is sent. You must use ObjectSend(), not ObjectCall(), to send msgObjectNew and any pointers in the pArgs for msgObjectNew must point to global memory.

We suggest that you send msgObjectNew to the resource file handle for the service class (because the resource file handle is owned by the service’s main task). Your creating task can find the resource file handle by sending msgSIMGetMetrics to the !InstalledServices, specifying the service class in its pArgs. The message passes back the handle in metrics.resFile. Send msgObjectNew to this handle.

Deinstalling Services

Deinstallation of services is a two-phase process.

In the first phase, the installer sends msgSvcClassTerminateOK to the service class being deinstalled. The service class immediately passes the message to its superclass, which sends msgSvcDeinstallRequested to all instances of the service. To approve the deinstallation, a service instance returns stsOK; any return value other than stsOK is a veto. If the call ancestor returns stsOK, the service class determines whether it can allow the termination. Again, to veto the termination, the service class returns a value other than stsOK.

If any instances veto the termination, the installer sends msgSvcClassTerminateVetoed to the service class. The service class passes the message to its superclass, which sends msgSvcDeinstallVetoed to all instances of the service. This message lets them know that the termination will not happen.

If all service instances and the service class approved the termination, the installer sends msgSvcClassTerminate to the service class. Again, the service class passes the message to its superclass, which sends msgDestroy to all instances of the service.

Finally, when the call ancestor for msgSvcClass Terminate returns, the service installer destroys the service class.

Messages Sent to Your Service Class

This section describes the messages that your service class must handle. In addition to msgNewDefaults and msgNew, clsService defines a number of class messages that relate to termination negotiation, service home information, and loading service instances. Usually a service will pass these messages to its superclass without any further action.
Handling msgNewDefaults

When a client sends msgNewDefaults to your service class, you should first route the message to your ancestor, clsService (usually this is done in the method table, see METHOD.TBL in the TESTSVC directory). This code fragment shows how clsService handles msgNewDefaults:

```c
pArgs->svc.style.waitForTarget = true; // Almost always true.
pArgs->svc.style.exclusiveOpen = false; // Exclusive open service?
pArgs->svc.style.autoOwnTarget = true; // Usually true if exclusive open.
pArgs->svc.style.autoOpen = false; // Usually true if exclusive open.
pArgs->svc.style.autoMsgPass = false; // Usually false.
pArgs->svc.style.autoOption = false; // True if exclusive open and you want clients to use owner stuff.
pArgs->svc.style.autoOption = false; // Usually true if your service does option sheets.
pArgs->svc.style.connectStyle = svcFollowTarget; // Only change this if you don't have a target.
pArgs->svc.handleClass = clsFileHandle; // Change this if you want your state handle to be something other than a file handle.
```

The defaults provided by clsService are general; when your service handles msgNewDefaults you should specify the style flags and the service manager or managers for the service.

If a subclass wants to change the handleClass, fsNew, or fsNewExtra parameters, it must also send msgNewDefaults to the handle class after it sends msgNewDefaults to its ancestor (most services will not need to do this):

```c
ObjCallWarn(msgNewDefaults, pArgs->svc.handleClass, &(pArgs->svc.fsNew));
```

Handling msgNew

As with msgNewDefaults, you should route msgNew to your ancestor (clsService). If you look at METHOD.TBL in the TESTSVC directory, you will see that the method table for clsTestService doesn't even list msgNew. Because there is no entry for msgNew, the class manager automatically routes the message to the class's ancestor.

Handling msgSvcClassTerminateOK

When the installer deinstalls a service, it sends msgSvcClassTerminateOK to the service instances class. The service class must call ancestor first, which allows clsService to send msgSvcDeinstallRequested to all service instances.

If all of the service instances return stsOK, the call ancestor returns stsOK to the service class. Then it is up to the service class to approve the termination. If any of the instances return a value other than stsOK, the call ancestor returns that status value and the service class should return with that status value.

msgSvcClassTerminateOK passes a pointer to an OBJECT. This value contains nothing, but if a service instance vetoes the termination, the superclass will return
the UID of the instance in the OBJECT. Typically, if your service class vetoes the termination, it should store its own UID in this location.

**Handling msgSvcClassTerminateVetoed**

When a service instance or the service class vetoes the termination, the installer sends msgSvcClassTerminateVetoed to the service class. The service class immediately passes the message to its superclass. clsService sends msgSvcDeinstallVetoed to all instances of the service class.

The message passes a pointer to an SVC_TERMINATE_VETOED structure, which contains two arguments:

- **vetoer** Specifies the object that vetoed the deinstallation.
- **status** Specifies the status value that the vetoer used.

**Handling msgSvcClassTerminate**

If the request to deinstall the service was not vetoed, the installer sends msgSvcClassTerminate to the service class. The service class passes the message to its superclass. clsService then sends msgDestroy to all instances of the service class.

There are no arguments for this message.

**Handling msgFree**

When you receive msgFree, you should save any stateful objects, free all objects that you created, then call ancestor. The method table defined in TESTSYC\METHOD.TBL specifies objCallAncestorAfter for msgFree.

You shouldn't delete the service instance state node as part of msgFree. If you received msgFree as part of a service deactivation, the service instance should be able to restore itself from its state node.

This is a code fragment from TESTSYC.C:

```c
117.6.6
111.6.4
117.6.6
****************************************************
TestSvcFree
Free any resources that this service created. Don't delete the state file or free the state file handle!
****************************************************/
MsgHandlerWithTypes(TestSvcFree, P_ARGS, P_INSTANCE_DATA)
{
    MsgHandlerParametersNoWarning;
    return stsOK;
}
// TestSvcFree
```

**Handling msgSvcClassLoadlnstance**

If an application needs to load a service state instance at a time other than initialization or warm-boot, it can send msgSvcClassLoadInstance to the service class.

The message passes a pointer to an SVC_LOAD_INSTANCE structure, containing an FS_LOCATOR (source), which indicates the location of the service state node.
clsService copies the state node specified by source to the INST directory for the service and starts an instance of the service stored in the state node (much like warm-boot or service installation). If a service instance with the name of the state node exists already, clsService first removes the existing service instance.

Usually subclasses of clsService do not process this message, unless they need to change the behavior when the service instance exists already.

Messages Sent by Service Managers

Before a client opens a service instance and has direct access to the service, the client must communicate with the service instance through its service managers. The client uses messages defined in clsSvcManager to get service information about the service, attempt to change owner, or attempt to change the target of the service.

When the service manager receives one of the clsSvcManager messages, it sends a corresponding clsService message to the service instance.

The next three sections describe the clsService messages and how you handle them. The first section describes messages for which you perform some action and then pass to clsService. The second section describes messages that clsService uses to either inform your service of an event or ask for confirmation. The third section lists the messages that you do not need to handle; you just allow clsService to handle these messages.

Service Manager Requests for Information

Although clsService does most of the work, you must be prepared to do some work for the messages listed in Table 117-1.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgFree</td>
<td>OBJECT</td>
<td>Frees the service instance.</td>
</tr>
<tr>
<td>msgSvcGetFunctions</td>
<td>P_SVC_GET_FUNCTIONS</td>
<td>Passes back a pointer to a table of function entry points.</td>
</tr>
<tr>
<td>msgSvcGetMetrics</td>
<td>P_SVC_GET_SET_METRICS</td>
<td>Passes back the current configuration metrics.</td>
</tr>
<tr>
<td>msgSvcSetMetrics</td>
<td>P_SVC_GET_SET_METRICS</td>
<td>Sets the configuration metrics.</td>
</tr>
</tbody>
</table>

Handling msgSvcGetMetrics

Clients use msgSvcGetMetrics to get metrics from your service instance. msgSvcGetMetrics passes a pointer to an SVC_GET_SET_METRICS structure, which the service instance uses to send back metrics information. The structure contains two arguments:

pMetrics A pointer to a buffer intended to hold the metrics for this service instance.

len A U16 value that specifies the size of the buffer specified in pMetrics.
Clients usually send `msgSvcGetMetrics` twice. The first time, the client specifies a zero length for the size of the metrics buffer (len); the service instance passes back only the size of its metrics. The client then sends `msgSvcGetMetrics` again, this time specifying the len value that it received in the first call. The service instance writes its metric information to the buffer specified by the client.

If your service has metrics that clients can access, it must be prepared to handle `msgSvcGetMetrics`. Because clients might file this information, you should also incorporate some form of version identification into your metrics.

This example shows how a typical service responds to `msgSvcGetMetrics`:

```c
//****************************************************************************
SvcDemoGetMetrics
****************************************************************************/

STATUS_LOCAL SvcDemoGetMetrics(
    OBJECT self,
    P_ARGS pArgs,
    P_INSTANCE_DATA pInst)
{
    P_SVC_GET_SET_METRICS pSvcMetrics;
    pSvcMetrics = (P_SVC_GET_SET_METRICS)pArgs;
    if (pSvcMetrics->len == 0)
    {
        pSvcMetrics->len = sizeof(DEMO_METRICS);
        return stsOK;
    }
    else
    {
        if (pSvcMetrics->len < sizeof(DEMO_METRICS))
        { return stsFailed; }
        *((P_DEMO_METRICS)(pSvcMetrics->pMetrics)) = pInst->demoMetrics;
        return stsOK;
    }
}
```

**Handling `msgSvcSetMetrics`**

Clients use `msgSvcSetMetrics` to set your service instance's metrics. `msgSvcSetMetrics` passes a pointer to an SVC_GET_SET_METRICS structure, which contains two arguments:

- **pMetrics** A pointer to a buffer that contains the new metrics data.
- **len** A U16 value that specifies the size of the buffer specified in `pMetrics`.

If your service has client-writable instance data, you must respond to this message. Because clients might attempt to restore data that has been filed for a time, you must have some form of version checking.
This example shows how a service can respond to msgSvcSetMetrics:

```
/****************************************************************************
SvcModemSetMetrics
Set modems metrics.
****************************************************************************/
STATUS LOCAL SvcModemSetMetrics(
    OBJECT self,
    P ARGS pArgs,
    P INSTANCE DATA pInst)
{
    P.MODEM.METRICS pMdmMetrics;
    P.SVC.GET_SET_METRICS pSvcMetrics;
    STATUS s;
    pSvcMetrics = (P_SVC_GET_SET_METRICS)pArgs;
    pMdmMetrics = (P_MODEM.METRICS)pSvcMetrics->pMetrics;
    if (pSvcMetrics->len < sizeof(MODEM.METRICS))
    {
        return stsFailed;
    }
    StsRet(ModemRestoreModem(pIrist, self), s);
    pInst->modemMetrics.mdmFAXResolution = pMdmMetrics->mdmFAXResolution;
    pInst->modemMetrics.mdmFAXEncoding = pMdmMetrics->mdmFAXEncoding;
    pInst->modemMetrics.mdmFAXLineWidth = pMdmMetrics->mdmFAXLineWidth;
    pInst->modemMetrics.mdmFAXMaxBaudRate = pMdmMetrics->mdmFAXMaxBaudRate;
    return stsOK;
}
```

**Messages from Service Managers**

The service instance receives the messages listed in Table 117-2 from the service manager when it has received a request from a client. Usually the messages ask the service instance for permission to continue.

When you receive most of these messages, you must pass the message to clsService; if clsService doesn’t veto the request, you can determine if you should allow the request.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgSvcBindRequested</td>
<td>P_SVC_BIND</td>
<td>Client asked to bind to this service.</td>
</tr>
<tr>
<td>msgSvcUnbindRequested</td>
<td>P_SVC_BIND</td>
<td>Client asked to unbind from this service.</td>
</tr>
<tr>
<td>msgSvcOpenDefaultsRequested</td>
<td>P_SVC_OPEN_CLOSE</td>
<td>Client wants open pArgs initialized.</td>
</tr>
<tr>
<td>msgSvcOpenRequested</td>
<td>P_SVC_OPEN_CLOSE</td>
<td>Client asked to open this service.</td>
</tr>
<tr>
<td>msgSvcCloseRequested</td>
<td>P_SVC_OPEN_CLOSE</td>
<td>Client asked to close this service.</td>
</tr>
<tr>
<td>msgSvcQueryLockRequested</td>
<td>pNull</td>
<td>Client asked to QueryLock this service.</td>
</tr>
<tr>
<td>msgSvcQueryUnlockRequested</td>
<td>pNull</td>
<td>Client asked to QueryUnlock this service.</td>
</tr>
<tr>
<td>msgSvcDeinstallRequested</td>
<td>pNull</td>
<td>Client asked to destroy this service instance.</td>
</tr>
<tr>
<td>msgSvcDeinstallVetoed</td>
<td>P_SVC_DEINSTALL_VETOED</td>
<td>Deinstallation process was vetoed.</td>
</tr>
<tr>
<td>msgSvcSaveRequested</td>
<td>P_FS_FLAT_LOCATOR</td>
<td>Client asked to save this instance to external media.</td>
</tr>
</tbody>
</table>
**Handling msgSvcBindRequested**

When the service manager receives msgSMBind from a client, it sends msgSvcBindRequested to the specified service instance. This offers the service instance the chance to veto the bind request.

The message passes a pointer to an SVC_BIND structure, which contains two arguments:

- **caller**  The UID of the object making the request.
- **manager**  The UID of the service manager through which the caller is making the request.

You must always pass this message to clsService. By default, clsService returns stsOK.

When clsService returns the message, you can do one of two actions:

- Refuse the bind by returning stsFailed.
- Accept the bind by returning stsOK.

If you return stsOK, the service manager adds the client to the observer list for your service's handle.

**Handling msgSvcUnbindRequested**

When a client sends msgSMUnbind to the service manager, the service manager notifies the specified service instance by sending it msgSvcUnbindRequested. This offers the service instance the opportunity to veto the request.

The message passes the service a pointer to an SVC_BIND structure, described above in msgSvcBindRequested.

You must always pass this message to clsService. Usually you do not need to do any further handling for this message, simply return the status returned by clsService.

If you return stsOK, the service manager removes the client from the observer list for your service instance.
Handling msgSvcOpenRequested

When a client sends msgSMOpen to the service manager, the service manager sends msgSvcOpenRequested to the specified service instance. The message passes a pointer to an SVC_OPEN_CLOSE structure, which contains four arguments:

- **caller** The UID of the client that made the request.
- **manager** The UID of the service manager that took the request.
- **pArgs** A pointer to a buffer that contains service-specific open arguments.
  - If your service requires additional arguments, the client can pass its arguments using this structure. The client can also ask your service to provide defaults for these arguments by sending msgSMOpenDefaults to the service manager, which then sends msgSvcOpenDefaultsRequested to your service instance.
- **service** An OBJECT element in which you can store the UID of an open service object, if any.

You must pass this message to clsService. clsService handles most of the checks specified by the service flags:

- Is the service locked?
- If checkOwner is TRUE, is the opener the owner?
- If exclusiveOpen is TRUE, does any other client have this service open?

clsService also determines whether the service class is currently being deinstalled. If openClass is not null, clsService creates a new instance of the open class. If autoOpen is true, clsService opens the service's target.

If your service instance needs to refuse the open request, it can return stsFailed. If your service instance accepts the open request, it should return stsOK. If you return stsOK, the service manager adds the client to the list of clients that have your service instance open. You or other clients can get the list of openers on a service by sending msgSMOpenList to the service manager.

Handling msgSvcOpenDefaultsRequested

Before a client sends msgSMOpen to a service manager, it must request the default pArgs for msgSMOpen by sending msgSMOpenDefaults to the service manager, specifying the service instance. When the service manager receives msgSMOpenDefaults, it sends msgSvcOpenDefaultsRequested to your service instance. The message passes a pointer to an P_SVC_OPEN_CLOSE structure, as described in msgSvcOpenRequested.

If your service instance can provide defaults for any of its open arguments, it should add its defaults to the structure indicated by pArgs.

If your service is a multiple access, multi-user service, you do not have to do anything more for this message other than send it to your superclass. clsService which sends msgNewDefaults to your open service object class (indicated by metrics.openClass). (The METHOD.TBL file in the TESTSVC directory does not
specify an ancestor call for msgSvcOpenDefaultsRequested; if you use an open
service object class, you must remember to modify its method table entry.)

If your service is a single access service, provide the defaults and return; clsService does
not have a method for this message, so your service does not have to call its ancestor.

    /********************************************************************************
    TestSvcOpenDefaultsRequested
    A client wants to get the defaults for the pArgs field of his open
structure.
    The client sent msgSMOpenDefaults to a service manager.
    *********************************************************************************/
MSG_HANDLER TestSvcOpenDefaultsRequested(
    const MESSAGE msg,
    const OBJECT self,
    const P_SVC_OPEN_CLOSE pArgs,
    const CONTEXT context,
    const P_INSTANCE_DATA pInst)
{
    msg; self; pArgs; context; pInst;
    // If you are a multi-user service (openClass <> objNull) then
    // you probably don’t need to handle this message. Superclass behavior
    // is to send msgNewDefaults to your openClass.
    // Single-user services that support the concept of a defaulted open
    // pArgs should initialize pArgs->pArgs here.
    return stsOK;
} // TestSvcOpenDefaultsRequested

Handling msgSvcCloseRequested
When a client sends msgSMClosed to a service manager, the service manager sends
msgSvcCloseRequested to the specified service instance. The message passes a pointer
to an SVC_OPEN_CLOSE structure, as described above in msgSvcOpenRequested.

This message informs the service that a particular object no longer has it open. The
service manager removes the service instance from its open list before sending this
message.

The service instance cannot veto this request, however, it should perform any
necessary cleanup and then pass the message to its superclass.

    /********************************************************************************
    TestSvcCloseRequested
    A client is finished interacting with the service.
    The client sent msgSMClose to a service manager.
    *********************************************************************************/
MSG_HANDLER TestSvcCloseRequested(
    const MESSAGE msg,
    const OBJECT self,
    const P_SVC_OPEN_CLOSE pArgs,
    const CONTEXT context,
    const P_INSTANCE_DATA pInst)
{
    msg; self; pArgs; context; pInst;
// If this is a multi-user service then clsService will automatically
// send msgDestroy the openClass instance created at open time.
// pArgs->pArgs becomes the pArgs for the msgDestroy.

// If this is a single-user service you might need to update instance
// data state at this time.

return stsOK;

} // TestSvcCloseRequested

Handling msgSvcQueryLockRequested

So that clients can access a service without opening it, clients can send
msgSMQueryLock to a service manager, specifying a service instance to lock.
When a client sends msgSMQueryLock to a service manager, the service manager
sends msgSvcQueryLockRequested to the specified service instance. The message
has no arguments.

If your service instance receives msgSvcQueryLockRequested, it must pass the
message to its superclass.

A client cannot read and write data using a service instance while it has a query
lock; the client must open the service to do so.

If exclusiveOpen is true, a query lock counts as an open. If a client has a service
instance open and exclusiveOpen is true, another client cannot access the service
instance. If a client has a query lock on the service instance and exclusiveOpen is
ture, it has exclusive access to that service instance.

Handling msgSvcQueryUnlockRequested

To release a query lock, a client sends msgSMQueryUnlock to the service
manager. The service manager then sends msgSvcQueryUnlockRequested to the
specified service instance. The message has no arguments.

If your service instance receives msgSvcQueryUnlockRequested, it must pass the
message to its superclass.

Handling msgSvcDeinstallRequested

When theInstalledServices deinstalls a service or when a service application needs
to destroy a service instance, it sends msgIMDeinstall to a service manager for
each instance of the service. The service manager checks whether the service
instance is in use. If the service instance is in use, the service manager refuses the
message with stsFailed.

If the service instance is not in use, the service manager sends msgSvcDeinstallRequested
to the specified service instance. The message has no arguments.

If your service instance receives msgSvcDeinstallRequested, it should perform any
clean up that it needs and returns stsOK. A service instance does not have to pass
this message to the superclass, but if it does pass msgSvcDeinstallRequested, it
must also pass msgSvcDeinstallVetoed to the superclass.
If all instances of the service return \texttt{stsOK}, the \texttt{InstalledServices} sends \texttt{msgFree} to each instance. A service instance cannot veto \texttt{msgFree}. \texttt{msgFree} causes the service instance to be removed from all service managers. Between returning \texttt{stsOK} and receiving \texttt{msgFree}, the service instance must not accept any new clients.

**Handling \texttt{msgSvcDeinstallVetoed}**

If any instance vetos the deinstallation, the \texttt{InstalledServices} sends \texttt{msgSvcDeinstallVetoed} to each service that received \texttt{msgSvcDeinstallRequested}. The message passes no arguments.

When your service instance receives \texttt{msgSvcDeinstallVetoed}, it can accept new clients.

If you passed \texttt{msgSvcDeinstallRequested} to the superclass, you must also pass \texttt{msgSvcDeinstallVetoed} to the superclass.

**Handling \texttt{msgSvcSaveRequested}**

When a client sends \texttt{msgSMSave} to a service manager and specifies this service, or when the entire service is being saved to home, the service manager sends \texttt{msgSvcSaveRequested} to your service instance. The message passes a pointer to an \texttt{FS_FLAT_LOCATOR} structure, which specifies the parent directory in which this service instance should be saved.

Your service instance should make sure that your state file is up to date and then send this message to the superclass. \texttt{clsService} saves the state file and the current target. If a node with the same name as the service instance already exists, \texttt{clsService} overwrites the destination. If the \texttt{pArgs} is \texttt{pNull}, the service instance is saved to its home. The service instance’s home is in the INST directory in the home directory of the service class.

Alternatively, your service instance could save its state data and anything else, and not pass the message to the superclass.

**Handling \texttt{msgSvcClientDestroyedEarly}**

If a client terminates while it is bound to a service instance, has a service instance open, or owns a service instance, \texttt{msgSvcClientDestroyedEarly} is sent to the service instance. Service instances that maintain per-client state must handle this message and perform their own cleanup.

The message passes the UID of the client that was destroyed.

You must pass this message to your superclass. \texttt{clsService} cleans up the service instance by sending \texttt{msgSMUnbind}, \texttt{msgSMClose} and \texttt{msgSMSetOwner} to self as appropriate.
Change Ownership Protocol Messages

When a client sends msgSMChangeOwner to a service manager, the service manager negotiates with the current owner of the service instance, the new owner of the service instance, and the service instance itself. Note that the client that sends msgSMChangeOwner does not have to be one of the owners; it can be a separate application, such as Printers, which tells the current owners of a device to change.

The negotiation occurs in this sequence:

1. The service manager sends msgSvcOwnerAcquireRequested to the new owner. The new owner can veto the change by returning anything other than stsOK or stsNotUnderstood. If the new owner vetoes the change, msgSMChangeOwner returns with a failure status.

2. The service manager sends msgSvcOwnerReleaseRequested to the current owner. The old owner can veto the change by returning anything other than stsOK or stsNotUnderstood. If the old owner vetoes the change, msgSMChangeOwner returns with a failure status.

3. If the current owner approves the change and it has the service instance open, it must close the service instance before returning stsOK.

4. The service manager checks to see if the service instance is still open. If it is, the service manager abandons the change owner operation by returning stsSvcInUse.

5. The service manager sends msgSvcChangeOwnerRequested to the service instance. This informs the service instance that its owner is about to change, and allows the service instance to veto the change (by returning a status other than stsOK or stsNotUnderstood). If the service instance vetoes the change, msgSMChangeOwner returns with a failure status.

6. The service manager sends msgSMOwnerChanged to all clients that are bound to the service instance and to all clients that are observing the service manager that lists this service.

7. The service manager sends msgSvcOwnerReleased to the old owner.

8. The service manager sends msgSvcOwnerAcquired to the new owner.

Clients can also use msgSMSetOwnerNoVeto, which is like msgSMSetOwner, but it does give the current and new owners the chance to veto the change (it doesn't send msgSvcReleaseRequested to the current owner and msgSvcAcquireRequested to the new owner).

Handling msgSvcOwnerReleaseRequested

When a service manager receives a msgSMChangeOwner, it sends a msgSvcOwnerReleaseRequested to the client that currently owns the service instance. The message asks the client if it is willing to relinquish ownership of a specific service instance.
instance. The message passes a pointer to an SVC_OWNED_NOTIFY structure, which contains:

- **ownedService**  The UID of the service instance whose owner is changing.
- **oldOwner**  The UID of the current owner.
- **newOwner**  The UID of the new owner.

When you receive this message, you must pass it to your superclass. If clsService vetoes the request (by returning a status other than stsOK or stsNotUnderstood), you should return with that status.

If you want to release ownership of the service instance, return stsOK. If you need to remain the owner of the service instance, return any status other than stsOK or stsNotUnderstood.

**Handling msgSvcOwnerAcquireRequested**

If the current owner of the service instance does not veto msgSvcOwnerReleaseRequested, the service manager sends msgSvcOwnerAcquireRequested to the client that is proposed as the new owner. This message asks the client if it is willing to take ownership of the service instance. The message passes a pointer to an SVC_OWNED_NOTIFY structure, as described above in msgSvcOwnerReleaseRequested.

When you receive this message, you must pass it to your superclass. If clsService vetoes the request (by returning a status other than stsOK or stsNotUnderstood), you should return with that status.

If you want to take ownership of the service instance, return stsOK. If you don't want to become the owner of the service instance, return any status other than stsOK or stsNotUnderstood.

**Handling msgSvcChangeOwnerRequested**

If the owner of the service instance does not veto msgSvcOwnerAcquireRequested, the service manager sends msgSvcChangeOwnerRequested to the service instance. This message informs the service instance you that its owner owner is about to change and allows it to veto the change. The message takes a pointer to an SVC_OWNED_NOTIFIED structure, which contains three arguments:

- **ownedService**  The UID of the service instance whose owner is changing.
  
  This should be self.

- **oldOwner**  The UID of the current owner.

- **newOwner**  The UID of the new owner.

If you want to allow the ownership change, return stsOK. If you don't want to allow the change, return any status other than stsOK or stsNotUnderstood.

If you don't veto the request, you must pass this message to your superclass. If clsService vetos the request (by returning a status other than stsOK or stsNotUnderstood), you should return with that status.
Handling msgSvcOwnerReleased

When a client is no longer the owner of a service instance, a service manager sends msgSvcOwnerReleased to the client. The message takes a pointer to an SVC_OWNED_NOTIFY structure, as described above in msgSvcChangeOwnerRequested.

You must send this message to your superclass.

If you need to preserve any owned state information for the service instance while you own it, you should get it now (with msgSvcGetMetrics) and save it in your state file.

You can manipulate the service as its owner until you return; the ownership change actually takes place when you return from this message.

Handling msgSvcOwnerAcquired

When a client becomes the owner of a service instance, a service manager sends msgSvcOwnerAcquired to the client. The message takes a pointer to an SVC_OWNED_NOTIFY structure, as described above in msgSvcChangeOwnerRequested.

You must send this message to your superclass.

You can manipulate the service as its owner as soon as you receive this message.

If you need to restore any saved state information for the service instance, you should do it here (usually with msgSvcSetMetrics).

Messages Handled By clsService

Your service instance should not handle the messages listed in Table 117-3; they should be handled by clsService only.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgSvcGetStyle</td>
<td>P_SVC_STYLE</td>
<td>Returns current style settings.</td>
</tr>
<tr>
<td>msgSvcSetStyle</td>
<td>P_SVC_STYLE</td>
<td>Changes style settings.</td>
</tr>
<tr>
<td>msgSvcAutoDetectingHardware</td>
<td>P_BOOLEAN</td>
<td>Is the hardware that this service ultimately talks to auto-detecting?</td>
</tr>
<tr>
<td>msgSvcGetHandle</td>
<td>P_OBJECT</td>
<td>Returns a handle to the service’s state node.</td>
</tr>
<tr>
<td>msgSvcOpenTarget</td>
<td>P_SVC_OPEN_TARGET</td>
<td>Attain access to the target service for data transfer.</td>
</tr>
<tr>
<td>msgSvcCloseTarget</td>
<td>P_SVC_OPEN_CLOSE_TARGET</td>
<td>Give up data transfer access to the target service.</td>
</tr>
<tr>
<td>msgSvcGetTarget</td>
<td>P_SVC_GET_TARGET</td>
<td>Returns current target.</td>
</tr>
<tr>
<td>msgSvcSetTarget</td>
<td>P_SVC_SET_TARGET</td>
<td>Change the target.</td>
</tr>
<tr>
<td>msgSvcAddToManager</td>
<td>P_SVC_ADD_TO_MANAGER</td>
<td>Add this service instance to a service manager.</td>
</tr>
<tr>
<td>msgSvcRemoveFromManager</td>
<td>P_SVC_REMOVE_FROM_MANAGER</td>
<td>Removes this service instance from a service manager.</td>
</tr>
<tr>
<td>msgSvcGetConnected</td>
<td>P_SVC_GET_SET_CONNECTED</td>
<td>Gets the connected state of this service.</td>
</tr>
</tbody>
</table>

Table 117-3

clsService Responsibility Messages

continued
### Messages Sent to Open Services

Once a client has opened a service, the client sends messages directly to the service instance. These messages tell the service to perform its functions.

Some services require clients to use the clsStream messages to send or receive data from the service (remember that clsService inherits from clsStream). Other services cannot use the clsStream messages. These services must define their own messages in their header file and METHOD.TBL file.

In either case, the service class must define methods for the messages from clients.

### Open Service Objects

If you are writing a multiple access, multi-user service, you must create a class that can provide UIDs to each of the service openers. You must do this by creating a subclass of clsOpenServiceObject when you initialize your service class. When you create your service instances, the arguments to msgNew specify the open service object class that your service uses.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgSvcGetConnected</td>
<td>P_SVC_GET_SET_CONNECTED</td>
<td>Sets connection state of self.</td>
</tr>
<tr>
<td>msgSvcGetModified</td>
<td>P_SVC_GET_SET_MODIFIED</td>
<td>Gets the modified state of this service.</td>
</tr>
<tr>
<td>msgSvcSetModified</td>
<td>P_SVC_GET_SET_MODIFIED</td>
<td>Sets modified state of self.</td>
</tr>
<tr>
<td>msgSvcGetMyOwner</td>
<td>P_OBJECT</td>
<td>Gets the current owner of this service, if any.</td>
</tr>
<tr>
<td>msgSvcGetOwned</td>
<td>P_OBJECT</td>
<td>Gets the service that is owned by this service,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if any.</td>
</tr>
<tr>
<td>msgSvcGetName</td>
<td>P_SVC_GET_NAME</td>
<td>Gets the name of this service instance.</td>
</tr>
<tr>
<td>msgSvcGetClassMetrics</td>
<td>P_SVC_CLASS_METRICS</td>
<td>Gets metrics for the service class that controls</td>
</tr>
<tr>
<td></td>
<td></td>
<td>this instance.</td>
</tr>
<tr>
<td>msgSvcPropagateMsg</td>
<td>P_OBJ_NOTIFY_OBSERVERS</td>
<td>Propagates a service-specific message.</td>
</tr>
<tr>
<td>msgSvcGetBindList</td>
<td>P_SVC_GET_LIST</td>
<td>Gets a list of all the callers that have bound</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to this service.</td>
</tr>
<tr>
<td>msgSvcGetOpenList</td>
<td>P_SVC_GET_LIST</td>
<td>Gets a list of all the callers that have opened</td>
</tr>
<tr>
<td></td>
<td></td>
<td>this service.</td>
</tr>
<tr>
<td>msgSvcGetOpenObjectList</td>
<td>P_SVC_GET_LIST</td>
<td>Gets a list of the open objects which were</td>
</tr>
<tr>
<td></td>
<td></td>
<td>returned for each open.</td>
</tr>
<tr>
<td>msgSvcGetManagerList</td>
<td>P_SVC_GET_LIST</td>
<td>Gets a list of all the service managers that this</td>
</tr>
<tr>
<td></td>
<td></td>
<td>service is on.</td>
</tr>
<tr>
<td>msgSvcGetManagerHandleList</td>
<td>P_SVC_GET_LIST</td>
<td>Gets a list of the svc mgr handles that this</td>
</tr>
<tr>
<td></td>
<td></td>
<td>service is represented by.</td>
</tr>
<tr>
<td>msgSvcGetDependentAppList</td>
<td>P_SVC_GET_LIST</td>
<td>Gets a list of the InstalledApps handles for all</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dependent apps.</td>
</tr>
<tr>
<td>msgSvcGetDependentServiceList</td>
<td>P_SVC_GET_LIST</td>
<td>Gets a list of the InstalledServices handles for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>all dependent services.</td>
</tr>
</tbody>
</table>
The section “Initializing an Open Service Object Class” describes how to initialize the open service object class and gives an example showing how to subclass clsOpenServiceObject.

To the client, the behavior of exclusive- and multiple-access services is no different. The client sends msgSMOpen to a service manager; if the service is available, the service manager passes back a service UID in the service field of the SM_OPEN_CLOSE structure. The client does not know whether the service is multiple access or exclusive access, nor should it have to know.

**clsService Does Most of the Work**

When you use a subclass of clsOpenServiceObject in a service, clsService actually performs most of the work for you.

When your service passes msgSvcOpenDefaultsRequested to your superclass, clsService determines that you have a open service object class, allocates a _NEW structure for the class, and initializes the _NEW structure by sending msgNewDefaults to the open service object class.

Later, when your service passes msgSvcOpenRequested to your superclass, clsService creates an instance of your open service object class by sending it msgNew. clsService then passes back the UID of the new open service object to the opener.

Because clsService creates the open service object in response to a client opening a specific service instance, the open service object is said to be associated with a service instance.

**What clsOpenServiceObject Does**

Like clsService, clsOpenServiceObject is a subclass of clsStream. Openers of the service send messages to the open service object. The open service object performs some local handling, and forwards all clsService messages to the service instance with which it is associated.

clsOpenServiceObject defines a single message, msgOSOGGetServiceInstance, which allows its subclasses to get the UID of the service instance with which the open service object is associated.

**Subclassing clsOpenServiceObject**

Your open service object class performs front end work for your service and maintains per-client state for the service. For instance, an open service object class might maintain the client's current position in a database service; another open service object class might maintain the client's session information for a network service.

It is hard to specify exactly where to draw the line between what the open service object class does, and what the service does. You must determine what is most efficient for your particular service.
When clsService creates the open service object, it creates it in the context of the opener's process. The open service object is local to the client; only the client can call the open service object. If performance is not an issue, the service can use ObjectSend to pass messages to the open service object; if performance is an issue, the open service object can make itself callable by the service.

Because the open service object is local to the client, it can use OSProcessHeap() to allocate its data structures. On the other hand, services must never use OSProcessHeap() to allocate data structures.
Chapter 118 / Distributing Your Services

This chapter describes what you must do to make your service available to the user to install. It also describes what the user must do to install your service.

What You Must Do

So that a user can install your service correctly, you must use the PenPoint file organization correctly.

The PenPoint file organization for services was described in Chapter 116. As a review, your distribution disk should have this structure:

```
\Distribution Volume
  \PENPOINT
    \SERVICE
      Service Directory
        \.dll files
        \.dli file
        \.INI file
        \SERVICE.RES
        \MISC
          \Static Data Files
            \INST
              \Service State Node
                \Service State Node
              \APP.INI
              \SERVICE.INI
```

Providing Preconfigured Instances

Where appropriate, you should provide preconfigured instances of your service.

The best way to save a preconfigured instance is to run PenPoint with the B 800 debugger flag set. This flag allows you to view the file system for the running PenPoint system in the Connections notebook.

Create the instance, then open the system disk to \PENPOINT\SYS\SERVICE\Your Service\INST and copy the appropriate service state node to \PENPOINT\SERVICE\Your Service\INST.

Providing Demo Apps

If your service will be used programmatically, provide a demo application that shows how to access your service.

It is probably not a good idea to place the demo application source files in the \PENPOINT directory. Rather, create a directory in the root of your distribution volume, and store the source files there.

What ever you do, make sure you tell your potential clients where to find the source.
What the User Must Do

When users obtain your service, they need to know the following information:

- How to install your service.
- How to use your service.
- What additional files (such as utilities or service instances) you shipped with your service.

Of course, the best way to provide this information is to document it. Tech Note #8 from GO Developer Technical Support recommends some procedures for documenting PenPoint applications. Most of this information will be directly applicable to services.
Chapter 119 / Test Service Examples

This chapter describes the example services in TESTSVC, BASICSVC, and MILSVC.

- **TESTSVC** is a general service that you can use it as a template from which to build most services.
- **BASICSVC** is a simple service that performs the minimum work that a service needs to do.
- **MILSVC** provides the foundations for a device driver service that uses the PenPoint machine interface layer (MIL).
The following sections list the files used to build TESTSVC. These files are also in the directory \PENPOINT\SDK\SAMPLE\TESTSVC. The sections are:

- METHOD.TBL
- TESTSVC.H
- TESTSVC.C
- OPENOBJ.H
- OPENOBJ.C

**METHOD.TBL**

/*****************************************************************************
 File: method.tbl
(C) Copyright 1992 by GO Corporation, All Rights Reserved.
 You may use this Sample Code any way you please provided you do not resell the code and that this notice (including the above copyright notice) is reproduced on all copies. THIS SAMPLE CODE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL, OR INDIRECT DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS SAMPLE CODE.

$Revision: 1.19 $
$Date: 31 Jan 1992 08:40:34 $

Contains the method tables for the Test Service.

Note: The method table for clsTestService has two handlers for each of these four methods: msgOptionAddCards, msgOptionProvideCardWin, msgOptionRefreshCard, and msgOptionApplyCard. One handler is what we typically think of as a message handler -- it handles messages sent to instances of a class. The other one is slightly different. It handles messages sent to the class itself.

You specify that a handler is for messages sent to a class by OR'ing in objClassMessage with objCallAncestorBefore, After, and so on.

You declare a class's message handler in the normal way. However, remember that self is a class, and that pInst does not point to the instance data of any of its instances. Instead, it points to some random chunk of memory. So, do not look at the instance data in a class's message handler.

With services, the msgOption<...> messages are sent to the service class if the user draws a check mark over the service in the Installer, and when the user initially installs the service if svcPopUpOptions is or-ed in to the InitService() flags. For more information, please see the comment titled "5. Advanced Features" in the service.h public include file.

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

```c
#include <option.h>
#include <testsvc.h>
#include <openobj.h>

MSG_INFO clsTestServiceMethods [] = {
    msgNewDefaults, "TestSvcNewDefaults", objCallAncestorBefore,
    msgInit, "TestSvcNew", objCallAncestorBefore,
    msgFree, "TestSvcFree", objCallAncestorAfter,
    msgSvcGetMetrics, "TestSvcGetMetrics", 0,
    msgSvcSetMetrics, "TestSvcSetMetrics", 0,
    msgSvcOpenRequested, "TestSvcOpenRequested", objCallAncestorBefore,
    msgSvcOpenDefaultsRequested, "TestSvcOpenDefaultsRequested", objCallAncestorBefore,
    msgSvcCloseRequested, "TestSvcCloseRequested", objCallAncestorBefore,
    msgOptionAddCards, "TestSvcAddCards", objCallAncestorAfter,
    msgOptionProvideCardWin, "TestSvcProvideCardWin", objCallAncestorAfter,
    msgOptionRefreshCard, "TestSvcRefreshCard", objCallAncestorAfter,
    msgOptionApplyCard, "TestSvcApplyCard", objCallAncestorAfter,
    msgSvcClassTerminate, "TestSvcTerminate", objClassMessage,
    objCallAncestorBefore, 0
};

MSG_INFO clsTestOpenObjectMethods [] = {
    msgNewDefaults, "TestOpenObjectNewDefaults", objCallAncestorBefore,
    msgInit, "TestOpenObjectNew", objCallAncestorBefore,
    msgFree, "TestOpenObjectFree", objCallAncestorAfter,
};

CLASS_INFO classInfo[] = {
    "clsTestServiceTable", clsTestServiceMethods, 0,
    "clsTestOpenObjectTable", clsTestOpenObjectMethods, 0,
};
```
This file contains the API definition for clsTestService.
clsTestService inherits from clsService.

typedef struct TEST_SVC_NEW_ONLY {
  U32 unused1i;
  U32 unused2i;
  U32 unused3i;
} TEST_SVC_NEW_ONLY, *P_TEST_SVC_NEW_ONLY;

#define testServiceNewFields serviceNewFields 
  TEST_SVC_NEW_ONLY testSvc;

typedef struct TEST_SVC_NEW {
  testServiceNewFields 
  TEST_SVC_NEW_ONLY testSvc;
} TEST_SVC_NEW, *P_TEST_SVC_NEW;

******************************************************************************
*** Your messages here! */

***** Class Option Tags ****/
#define tagTestSvcOptionCard MakeTag(clsTestService, 10)
#define tagTestSvcGlobalOptionCard MakeTag(clsTestService, 11)

***** Quick Help Tags ****/
#define hlpTestSvcOptionCard MakeTag(clsTestService, 20)
#define hlpTestSvcGlobalOptionCard MakeTag(clsTestService, 21)

******************************************************************************

msgNew takes P_TEST_SVC_NEW, returns STATUS
category: class message
Set defaults:

/**** Messages ****/

******************************************************************************

This file contains the class definition and methods for clsTestService.
This is a canonical service, designed to give service writers a head start. Stubs for the messages that are typically subclassed by a service
are provided. Note that not every subclass-responsibility message is given; only those that are most often used. For example, a handler for msgSvcBindRequested isn’t provided since it is very rarely subclassed.

Notes:

Remember that service instances have capCall turned on. This means that they can be called from a process besides their owning process. If it is possible for more than one caller to access the same service instance simultaneously, think through the concurrency issues! You might have to use semaphores to protect areas which can’t handle being called simultaneously by multiple folks.

#include <string.h>
#include <stdlib.h>
#include <debug.h>
#include <list.h>
#include <fs.h>
#include <os.h>
#include <osheap.h>
#include <opttable.h>
#include <option.h>
#include <servmgr.h>
#include <method.h>
#include <testsvc.h>

II TKTables for instance and class option cards.
static TK_TABLE_ENTRY optionCard[] = {
    {"Instance Option1:" , 0, 0, 0, 0, 0, 0, clsButton},
    {"Instance Option2:" , 0, 0, 0, 0, 0, 0, clsButton},
    {pNull}
};

static TK_TABLE_ENTRY globalOptionCard[] = {
    {"Global Option1:" , 0, 0, 0, 0, 0, clsButton},
    {"Global Option2:" , 0, 0, 0, 0, 0, clsButton},
    {pNull}
};

// The service manager(s) we should be put on.
static UID managers = {thePrinterDevices};

typedef struct INSTANCE_DATA {
    U32 unused1;
    U32 unused2;
} INSTANCE_DATA, *P_INSTANCE_DATA;

TestSvcNewDefaults

Set defaults. This is where the service characteristics that are common to all instances of your service subclasses are specified.

*******************************************************************************

MsgHandlerWithTypes(TestSvcNewDefaults, P_TEST_SVC_NEW, P_INSTANCE_DATA)
{
    MsgHandlerParametersNoWarning;

    // These are the default values from clsService.
    pArgs->svc.style.exclusiveOpen = false; // Exclusive open service?
    pArgs->svc.style.autoOption = false;   // Set to true to pass option
         // sheet messages to your target.
    pArgs->svc.handleClass = clsFileHandle; // Change this if you want your
         // state handle to be something
         // other than a file handle.

    // The next two lines are only needed if you change svc.handleClass.
    ObjCallWarn(msgNewDefaults, pArgs->svc.handleClass, &(pArgs->svc.fsNew));
    pArgs->svc.fsNew.fs.exist = fsExistOpen | fsNoExistCreate;

    // Service managers that this instance should go on. Often statically
    // defined, as in this example. See managers definition above.
    pArgs->svc.pManagerList = &managers;
    pArgs->svc.numManagers = 1;

    // The setting of openClass determines whether the service instance
    // object is passed back on open (openClass = objNull), or a new
    // instance of the openClass is created and passed
    // back (openClass = clsYourOpenObject). openObject classes should
    // be subclasses of clsOpenServiceObject.
    pArgs->svc.style.openClass = objNull;

    return stsOK;
}

} // TestSvcNewDefaults

TestSvcNew

Create a new Test Service.

1. Set up your instance data.
2. Deal with your state file. Your state file will either contain stuff
   (if you are being recreated due to a warm boot or having been copied
   in from disk) or be empty (if this is the first time). You must call
   ancestor before dealing with your state file.
Notes:
If autoCreate in msgSvcClassInitService was true then the system will send msgNewDefaults and msgNew for each state file it finds.

******************************************************************************
MsgHandlerWithTypes(TestSvcNew, P_TEST_SVC_NEW, P_INSTANCE_DATA)
{
    INSTANCE_DATA inst;
    OBJECT stateHandle;
    STATUS s;

    // Get my state file.
    ObjCallJmp(msgSvcGetHandle, self, &stateHandle, s, objectWrite);
    s = stsOK;

    objectWrite:
    StsWarn(ObjectWrite(self, ctx, &inst));
    return s;
} // TestSvcNew

******************************************************************************
TestSvcFree
Free any resources that this service created. Don't delete the state file or free the state file handle!
******************************************************************************
MsgHandlerWithTypes(TestSvcFree, P_ARGS, P_INSTANCE_DATA)
{
    MsgHandlerParametersNoWarning;
    return stsOK;
} // TestSvcFree

******************************************************************************
TestSvcGetMetrics
If this service supports client accessible metrics then this message must be handled. Be sure to include some form of version identification with the metrics.
******************************************************************************
MsgHandlerWithTypes(TestSvcGetMetrics, P_SVC_GET_SET_METRICS, P_INSTANCE_DATA)
{
    MsgHandlerParametersNoWarning;
    if (pArgs->len == 0) {
        // pArgs->len = myMetricsLength;
    } else {
        // Copy out my metrics to pArgs->pMetrics.
    }
    return stsOK;
} // TestSvcGetMetrics

******************************************************************************
TestSvcSetMetrics
If this service supports client accessible metrics then this message must be handled. Be able to handle old versions of the metrics...
******************************************************************************
MsgHandlerWithTypes(TestSvcSetMetrics, P_SVC_GET_SET_METRICS, P_INSTANCE_DATA)
{
    MsgHandlerParametersNoWarning;
    return stsOK;
} // TestSvcSetMetrics

******************************************************************************
TestSvcOpenDefaultsRequested
A client wants to get the defaults for the pArgs field of his open structure.
The client sent msgSMOpenDefaults to a service manager.
******************************************************************************
MsgHandlerWithTypes(TestSvcOpenDefaultsRequested, P_SVC_OPEN_CLOSE, P_INSTANCE_DATA)
{
    MsgHandlerParametersNoWarning;

    // If you are a multi-user service (openClass <> objNull) then
    // you probably don't need to handle this message. Superclass behavior
    // is to send msgNewDefaults to your openClass.

    // Single-user services that support the concept of a defaulted open
    // pArgs should initialize pArgs->pArgs here.
The client sent msgSMClose to a service manager.

*********************************************
MsgHandlerWithTypes(TestSvcCloseRequested, P_SVC_OPEN_CLOSE, P_INSTANCE_DATA)
{
    // SVC_OPEN_CLOSE_TARGET closeTarget;
    // STATUS s;
    MsgHandlerParametersNoWarning;
    // If this is a multi-user service then clsService will automatically
    // send msgDestroy the openClass instance created at open time.
    // pArgs->pArgs becomes the pArgs for the msgDestroy.
    // If this is a single-user service you might need to update instance
    // data state at this time.
    // If auto-open is false, this service will probably want to close its
    // target.
    closeTarget.pArgs = ...;
    ObjCallRet(msgSvcCloseTarget, self, &closeTarget, s);
    return stsOK;
}

} // TestSvcCloseRequested

/*****************************/
TestSvcAddCards

The client sent msgSMOpen to a service manager.

***********************************************/
MsgHandlerWithTypes(TestSvcOpenRequested, P_SVC_OPEN_CLOSE, P_INSTANCE_DATA)
{
    // SVC_OPEN_CLOSE_TARGET openTarget;
    // SVC_GET_TARGET getTarget;
    // STATUS s;
    MsgHandlerParametersNoWarning;
    // If this is a multi-user service then clsService will automatically
    // create an instance of the openClass you specified in
    // msgNewDefaults. You might want to specify or modify the open
    // request's pArgs (pArgs->pArgs). These become the pArgs for the msgNew
    // to your openClass, and is how you can communicate with your
    // openClass.
    // If this is a single-user service you might need to update instance
    // data state at this time. Note that you don't have handle checking for
    // exclusive access; clsService does this all for you.
    // If auto-open is false, this service will probably want to open its
    // target.
    openTarget.pArgs = ...;
    ObjCallRet(msgSvcOpenTarget, self, &openTarget, s);
    return stsOK;
}

} // TestSvcOpenRequested

/*****************************/
TestSvcCloseRequested

A client wants interact directly with the service, typically to move data.
The client sent msgSMOpen to a service manager.

***********************************************/
MsgHandlerWithTypes(TestSvcAddCards, P_OPTION_TAG, P_INSTANCE_DATA)
{
    // OPTION_CARD optCard;
    // STATUS s;
    MsgHandlerParametersNoWarning;
    // Add service instance option cards. This routine should be used if this
    // service provides options cards for its instances.
    optCard.tag = tagTestSvcOptionCard;
    optCard.pName = "Testing, testing, testing...";
    optCard.win = objNull;
    optCard.client = self;
    ObjCallRet(msgOptionAddCard, pArgs->option, &optCard, s);
    return stsOK;
}

} // TestSvcAddCards

/*****************************/
TestSvcProvideCardWin

A client is finished interacting with the service.
Provide service instance option cards. This routine should be used if this service provides options cards for its instances.

```c
MsgHandlerWithTypes(TestSvcProvideCardWin, P_OPTION_CARD, P_INSTANCE_DATA)
{
    OPTION_TABLE_New optTable;
    STATUS s;

    MsgHandlerParametersNoWarning;

    if (pArgs->tag == tagTestSvcOptionCard) {
        ObjCallWarn(msgNewDefaults, clsOptionTable, &optTable);
        optTable.win.flags.style = wsShrinkWrapWidth | wsShrinkWrapHeight;
        optTable.win.helpId = hlpTestSvcOptionCard;
        optTable.tkTable.pEntries = optionCard;
        ObjCallRet(msgNew, clsOptionTable, &optTable, s);
        pArgs->win = optTable.object.uid;
    }
    return stsOK;
}
```

Refresh the instance option cards.

```c
MsgHandlerWithTypes(TestSvcRefreshCard, P_OPTION_CARD, P_INSTANCE_DATA)
{
    MsgHandlerParametersNoWarning;
    return stsOK;
}
```

Apply the instance option cards.

```c
MsgHandlerWithTypes(TestSvcApplyCard, P_OPTION_CARD, P_INSTANCE_DATA)
{
    MsgHandlerParametersNoWarning;

    if (pArgs->tag == tagTestSvcOptionCard) {
        return stsOK;
    }
}
```

Perform any operations required when the entire service is being deinstalled. Note that all instances have already been destroyed.

```c
MsgHandlerArgType(TestSvcTerminate, P_ARGS)
{
    MsgHandlerParametersNoWarning;
    return stsOK;
}
```
Apply the global option cards.
******************************************************************************

MsgHandlerArgType(TestSvcClassApplyCard, P_OPTION_CARD)
{
    MsgHandlerParametersNoWarning;
    if (pArgs->tag != tagTestSvcGlobalOptionCard) {
        return stsOK;
    }
    // Apply our global option cards here...
    return stsOK;
}

******************************************************************************

DLLMain()
Install our service. This routine is called once when the service is first installed on PenPoint.
******************************************************************************

STATUS EXPORTED DLLMain(void)
{
    SVC_INIT_SERVICE initService;
}
Install classes.

StsRet(ClsTestServiceInit(), s);  // Only needed if multi-user service.

Let system know about our service and set global service
characteristics. THIS IS REQUIRED!
memset(initService.spare, 0, sizeof(initService.spare));
initService.autoCreate = false;
initService.serviceType = 0;
initService.initServiceFlags = svcPopupOptions;  // Flags. Or-in options.
ObjCallRet(msgSvcClassInitService, clsTestService, &initService, s);

Create any static service instances. Set autoCreate above to false
if you do this. If you know exactly how many service instances
there will be, this is where you should create them.
Alternatively, they can be dynamically created elsewhere, after
DLLMain() runs. Beware of process ownership issues if you create
services dynamically. You must ensure that the process which owns
the dynamically created service instance will be around for the
lifetime of the service instance!
ObjCallWarn(msgNewDefaults, clsTestService, &svcNew);
svcNew.svc.pServiceName = "Test Instance";  // Name of this instance
svcNew.svc.target.manager = theTransportHandlers;  // Target svc manager
strcpy(svcNew.svc.target.pName, "Netware");  // Target name
ObjCallRet(msgNew, clsTestService, &svcNew, s);

return stsOK;
}

DllMain

This file contains the API definition for clsTestOpenObject.
clsTestOpenObject inherits from clsOpenServiceObject.
Provides a template for a multi-user service's open object class.

#ifndef OPENOBJ_INCLUDED
#define OPENOBJ_INCLUDED

#include <openserv.h>
#endif

#define testOpenObjectNewFields \
openServiceObjectNewFields \
TEST_OPEN_OBJECT_NEW_ONLY testOpenObject;

typedef struct TEST_OPEN_OBJECT_NEW_ONLY {
    U32   unused1;
    U32   unused2;
    U32   unused3;
} TEST_OPEN_OBJECT_NEW_ONLY, *P_TEST_OPEN_OBJECT_NEW_ONLY;

#define testOpenObjectNewFields \
openServiceObjectNewFields \
TEST_OPEN_OBJECT_NEW_ONLY testOpenObject;

typedef struct TEST_OPEN_OBJECT_NEW {
    testOpenObjectNewFields \
} TEST_OPEN_OBJECT_NEW, *P_TEST_OPEN_OBJECT_NEW;

#ifndef OPENSERV_INCLUDED
#define OPENSERV_INCLUDED
#include <openserv.h>
#endif

#define msgNewDefaults takes P_TEST_OPEN_OBJECT_NEW, returns STATUS
category: class message
Set defaults:
/*
typedef struct TEST_OPEN_OBJECT_NEW {
    U32   unused1;
    U32   unused2;
    U32   unused3;
} TEST_OPEN_OBJECT_NEW, *P_TEST_OPEN_OBJECT_NEW;

#define msgNew Defaults takes P_TEST_OPEN_OBJECT_NEW, returns STATUS
category: class message

Your messages here! */
#endif /* OPENSERV_INCLUDED */
This file contains the class definition and methods for clsTestOpenObject. This is a stub for an open service Object class. This class is only used for multi-user services that wish to pass a new uid to each opener.

Notes:

There’s really not a whole lot to the default behavior for this class. Basically just put your stuff here.

```c
typedef struct INSTANCE_DATA {
    U32 unused1;
    U32 unused2;
} INSTANCE_DATA, *P_INSTANCE_DATA;
```

```c
 MagHandlerWithTypes(TestOpenObjectNewDefaults, P_TEST_OPEN_OBJECT_NEW, P_INSTANCE_DATA)
 { 
     MagHandlerParametersNoWarning;
     return stsOK;
 } // TestOpenObjectNewDefaults
```

```c
 TestOpenObjectNew

 MagHandlerWithTypes(TestOpenObjectNew, P_TEST_OPEN_OBJECT_NEW, P_INSTANCE_DATA)
 { 
     INSTANCE_DATA inst;
     STATUS s;
     MagHandlerParametersNoWarning;
     s = stsOK;
     StsWarn(ObjectWrite(self, ctx, &inst));
     return s;
 } // TestOpenObjectNew
```

```c
 TestOpenObjectFree

 MagHandlerWithTypes(TestOpenObjectFree, P_ARGS, P_INSTANCE_DATA)
 { 
     MagHandlerParametersNoWarning;
     return stsOK;
 } // TestOpenObjectFree
```

```c
 ClsTestOpenObjectInit

 Install our class.

 STATUS PASCAL ClsTestOpenObjectInit(void)
 { 
     STATUS s;
 } // ClsTestOpenObjectInit
```
The following sections list the files used to build BASICSVC. These files are also in the directory \PENPOINT\SDK\SAMPLE\BASICSVC. The sections are:

- METHOD.TBL
- BASICSVC.H
- BASICSVC.C

**METHOD.TBL**

Specifies class information for BASICSVC.

```c
classInfo[] = {
    "clsBasicServiceTable", clsBasicServiceMethods, 0, 0
};

MSG_INFO clsBasicServiceMethods [] = {
    msgNewDefaults, "BasicSvcNewDefaults", objCallAncestorBefore, 0
};
```

**BASICSVC.H**

Describes the BASICSVC service.

```c
BASICSVC

This file contains sample code for a minimal service, clsBasicService. This is the absolute minimum required to make a service. Add your methods to this framework.

$Revision: 1.3 $
$Date: 31 Jan 1992 08:05:18$

Contains the method tables for clsBasicService.

#include <service.h>
```

This file is reproduced on all copies. THIS SAMPLE CODE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL, OR INDIRECT DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS SAMPLE CODE.

$Revision: 1.3 $
$Date: 31 Jan 1992 08:05:22 $

This file contains sample code for a minimal service, clsBasicService.

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

$Revision: 1.3 $
$Date: 31 Jan 1992 08:05:18$

Contains the method tables for clsBasicService.

#include <service.h>
```
This file contains sample code for a minimal service, clsBasicService. This is the absolute minimum required to make a service. Add your methods to this framework.

initService.initServiceFlags = 0; // Flags. Or-in options.
ObjCallRet(msgSvcClassInitService, clsBasicService, &initService, s);

// Create an instance.
ObjCallWarn(msgNewDefaults, clsBasicService, &svcNew);
svcNew.svc.pServiceName = "Basic Instance"; // Name of this instance.
ObjCallRet(msgNew, clsBasicService, &svcNew, s);
return stsOK;

} //DllMain

/*****************************************************************************
BasicSvcNewDefaults
Set defaults. This is where the service characteristics that are common to all instances of your service subclasses are specified.
******************************************************************************/
MsgHandlerWithTypes(BasicSvcNewDefaults, P_SVC_NEW, P_IDATA)
{
    MsgHandlerParametersNoWarning;
    // What service manager(s) should instances go on?
    pArgs->svc.nurnManagers = 1;
pArgs->svc.pManagerList = &managers;
    // What are the exclusivity requirements for this service?
pArgs->svc.style.exclusiveOpen = true;
    return stsOK;
}

} // BasicSvcNewDefaults

/*****************************************************************************
ClsBasicServiceInit
Install our class.
******************************************************************************/
STATUS PASCAL ClsBasicServiceInit(void)
{
    STATUS s;
    CLASS_NEW new;
    // Create the service class.
    ObjCallWarn(msgNewDefaults, clsClass, &new);
    new.object.uid = clsBasicService; // SUBSTITUTE YOUR CLASS HERE!
    new.cls.pMsg = clsBasicServiceTable;
    new.cls.ancestor = clsService;
    new.cls.size = 0;
    new.cls.newArgsSize = SizeOf(SVC_NEW);
    // Let the system know about our service.
    memset(initService.spare, 0, sizeof(initService.spare));
    initService.autoCreate = false;
    initService.serviceType = 0;
    // StsSet(ClsBasicServiceInit(), s);
}
ObjCallRet(msgNew, clsClass, &new, s);
return s;

} // ClsBasicServiceInit

The following sections list the files used to build MILSVC. These files are also in the directory \PENPOINT\SDK\SAMPLE\MILSVC. The sections are:

METHOD.TBL
MILSVC.H
MILSVC.C
MILSVC0.H
MILSVC0.C

METHOD.TBL

******************************************************************************
File: method.tbl
(C) Copyright 1992 by GO Corporation, All Rights Reserved.
You may use this Sample Code any way you please provided you do not resell the code and that this notice (including the above copyright notice) is reproduced on all copies. THIS SAMPLE CODE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL GO CORPORATION BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL, OR INDIRECT DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS SAMPLE CODE.

$Revision: 1.16 $
$Date: 13 Mar 1992 16:37:10 $

method.tbl contain the method tables for class clsTestMILService.
******************************************************************************

#include <mildev.h>
#include <milser.h>
#include <milsvc.h>

MSG_INFO clsTestMILSvcServiceMethods[] =
{
    msgNewDefaults, "TestMILSvcNewDefaults", objCallAncestorBefore, 0,
    msgInit, msgFree, "TestMILSvcNew", objCallAncestorBefore, objCallAncestorAfter, 0,
    msgSvcGetMetrics, msgSvcSetMetrics, "TestMILSvcGetSvcMetrics", "TestMILSvcSetSvcMetrics", 0,
    msgSvcOpenRequested, msgSvcQueryLockRequested, msgSvcCloseRequested, msgSvcQueryUnlockRequested, "TestMILSvcOpen", objCallAncestorBefore, 0,
    msgTestMILSvcInitialize, msgTestMILSvcStatus, "TestMILSvcHWInitialize", "TestMILSvcGetHWStatus", 0,
    msgTestMILSvcAutoLineFeedOn, msgTestMILSvcAutoLineFeedOff, "TestMILSvcAutoLineFeedOn", "TestMILSvcAutoLineFeedOff", 0,
    msgTestMILSvcGetTimeDelays, msgTestMILSvcSetTimeDelays, "TestMILSvcGetTimeDelays", "TestMILSvcSetTimeDelays", 0,
    msgTestMILSvcCancelPrint, msgFSFlush, "TestMILSvcCancelPrintBuffer", "TestMILSvcFlush", 0,
    msgStreamWrite, msgMILSvcAreYouConnected, "TestMILSvcAreYouConnected", 0,
    msgMILSvcConnectionStateResolved, "TestMILSvcConnectionStateResolved", objCallAncestorAfter, 0,
    msgMILSvcStartConnectionProcessing, "TestMILSvcPenpointBooted", 0,
    msgMILSvcPowerOff, msgTestMILSvcDoConnection, "TestMILSvcPowerOff", "TestMILSvcDoConnection", 0,
};

CLASS_INFO classInfo[] =
{
    "clsTestMILSvcServiceTable", clsTestMILSvcServiceMethods, 0,
};
MILSV.C.H

/******************************************************************************
File: milsvc.h

(C) Copyright 1992 by GO Corporation, All Rights Reserved.

You may use this Sample Code any way you please provided you
do not resell the code and that this notice (including the above
copyright notice) is reproduced on all copies. THIS SAMPLE CODE
IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, AND GO CORPORATION
EXRESSLY 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: 1.16 $
$Date: 12 Mar 1992 12:32:32 $

This file contains the API definition for clsTestMILService.
clsTestMILService inherits from clsMILService.

This mil service provides the interface between the parallel printer
mil device and the rest of Penpoint. This interface allows for the
configuring of the parallel printer mil device and for printing using
the parallel printer mil device. The pport mil service will typically
only be accessed by printer drivers since they are responsible for
rendering an image for printing.

You access this mil service by using the standard service access techniques.
These techniques are described in servmgr.h.
The pport mil service is a member of the 'theParallelDevices' and
'thePrinterDevices' service managers.

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

#ifndef MILSV_C_INCLUDED
#define MILSV_C_INCLUDED
#endif

#ifndef MO_INCLUDED
#include <go.h>
#endif

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

#ifndef MIL_SERVICE_INCLUDED
#include <milsvrv.h>
#endif

typedef OBJECT TEST_MIL_SVC, *P_TEST_MIL_SVC;

#define stsTestMILSvcBusy
#define stsTestMILSvcOutOfPaper
#define stsTestMILSvcOffLine
#define stsTestMILSvcNoPrinter
#define stsTestMILSvcPrinterErr

MakeStatus(clsTestMILService, 1)
MakeStatus(clsTestMILService, 2)
MakeStatus(clsTestMILService, 3)
MakeStatus(clsTestMILService, 4)
MakeStatus(clsTestMILService, 5)

typedef struct TEST_MIL_SVC_METRICS
{
  U16 version;
  U16 devFlags;
  U16 unitFlags;
  U32 initDelay;
  U32 interruptTimeOut;
} TEST_MIL_SVC_METRICS, *P_TEST_MIL_SVC_METRICS;

#define msgTestMILSvcStatus MakeMsg(clsTestMILService, 3)

typedef struct TEST_MIL_SVC_STATUS
{
  U16 testMILSvcStatus;
} TEST_MIL_SVC_STATUS, *P_TEST_MIL_SVC_STATUS;

#define testMILSvcStsBusy flag7 // printer is busy
#define testMILSvcStsAcknowledge flag6 // printer acknowledged char.
#define testMILSvcStsEndOfPaper flag5 // printer out of paper
#define testMILSvcStsSelected flag4 // printer online
#define testMILSvcStsIOError flag3 // printer error occurred
#define testMILSvcStsInterruptHappened flag2 // printer interrupt occurred

typedef struct TEST_MIL_SVC_STATUS
{
  U16 testMILSvcStatus;
}
msgTestMILSvcInitialize takes P_NULL, returns STATUS
initializes the printer.

The printer is initialized by asserting the control
line "Initialize" to the printer for initDelay microseconds.

#define msgTestMILSvcInitialize MakeMsg(clsTestMILService, 4)

msgTestMILSvcAutoLineFeedOn takes P_NULL, returns STATUS
inserts a line feed after each carriage return.

The auto line feed signal to the printer is set active.

#define msgTestMILSvcAutoLineFeedOn MakeMsg(clsTestMILService, 5)

msgTestMILSvcAutoLineFeedOff takes P_NULL, returns STATUS
disables inserting a line feed after each carriage return.

The auto line feed signal to the printer is set inactive.

#define msgTestMILSvcAutoLineFeedOff MakeMsg(clsTestMILService, 6)

msgTestMILSvcGetTimeDelays takes P_TEST_MIL_SVC_TIME_DELAYS, returns STATUS
gets the initialization and interrupt time out intervals.

The initialization time period is the time the initialization pulse
is asserted to the printer in microseconds. The interrupt time out
interval is the maximum time the printer will assert busy before being
ready to accept another character in milliseconds.

#define msgTestMILSvcGetTimeDelays MakeMsg(clsTestMILService, 7)

typedef struct TEST_MIL_SVC_TIME_DELAYS {
  U32 initDelay; // initialization delay
  U32 interruptTimeOut; // interrupt time out
} TEST_MIL_SVC_TIME_DELAYS, *P_TEST_MIL_SVC_TIME_DELAYS;

msgTestMILSvcSetTimeDelays takes P_TEST_MIL_SVC_TIME_DELAYS, returns STATUS
sets the initialization and interrupt time out intervals.

Neither value can be zero. It's best to get the present
values before changing the time intervals.

#define msgTestMILSvcSetTimeDelays MakeMsg(clsTestMILService, 8)

msgTestMILSvcCancelPrint takes P_NULL, returns STATUS
cancels the printing of the buffer currently being printed.

#define msgTestMILSvcCancelPrint MakeMsg(clsTestMILService, 9)

msgNew takes P_TEST_MIL_SVC_NEW, returns STATUS
creates a new test mil service object.

#define testMILSvcNewFields MilServiceNewFields

typedef struct TEST_MIL_SVC_NEW {
  MilServiceNewFields
} TEST_MIL_SVC_NEW, *P_TEST_MIL_SVC_NEW;

STATUS EXPORTED CIsTestMILSvcInit(void);

#endif // MILSVC_INCLUDED

MILSVC.C

**************************************************************************
File: milsvc.c

(C) Copyright 1992 by GO Corporation, All Rights Reserved.
This file contains the class definition and methods for clsTestMILService.

****************************************************************************

#include <go.h>
#include <clsmgr.h>
#include <system.h>
#include <os.h>
#include <ospriv.h>
#include <debug.h>
#include <milserv.h>
#include <stream.h>
#include <string.h>
#include <list.h>
#include <servmgr.h>
#include <dvparall.h>
#include <method.h>
#include <milsvc.h>
#include <milsvc0.h>

/status EXPORTED XTestMILSvcGetReqBlocks(P_INSTANCE_DATA pInst);
/status EXPORTED TestMILSvcGetReqBlocks(P_INSTANCE_DATA pInst)
{    return (STATUS) (OSSupervisorCall(XTestMILSvcGetReqBlocks, &pInst, 1));
}
/status EXPORTED XTestMILSvcInitialize(P_INSTANCE_DATA pInst);
/status EXPORTED TestMILSvcInitialize(P_INSTANCE_DATA pInst)
{    return (STATUS) (OSSupervisorCall(XTestMILSvcInitialize, &pInst, 1));
}
/status EXPORTED XTestMILSvcGetMetrics(P_TEST_MIL_SVC_METRICS pMetrics,
P_INSTANCE_DATA pInst);
/status EXPORTED TestMILSvcGetMetrics(P_TEST_MIL_SVC_METRICS pMetrics,
P_INSTANCE_DATA pInst)
{    Unused(pInst);
    return (STATUS) (OSSupervisorCall(XTestMILSvcGetMetrics, &pMetrics, 2));
}
/status EXPORTED XTestMILSvcGetMetrics(P_TEST_MIL_SVC_METRICS pMetrics,
P_INSTANCE_DATA pInst);
/status EXPORTED TestMILSvcGetMetrics(P_TEST_MIL_SVC_METRICS pMetrics,
P_INSTANCE_DATA pInst)
{    Unused(pInst);
    return (STATUS) (OSSupervisorCall(XTestMILSvcGetMetrics, &pMetrics, 2));
}
/status EXPORTED XTestMILSvcPrint(P_STREAM_READ_WRITE pStream,
P_INSTANCE_DATA pInst);
/status EXPORTED TestMILSvcPrint(P_STREAM_READ_WRITE pStream,
P_INSTANCE_DATA pInst)
{    Unused(pInst);
    return (STATUS) (OSSupervisorCall(XTestMILSvcPrint, &pStream, 2));
}
/status EXPORTED XTestMILSvcGetStatus(P_TEST_MIL_SVC_STATUS pTestMILSvcStatus,
P_INSTANCE_DATA pInst);
/status EXPORTED TestMILSvcGetStatus(P_TEST_MIL_SVC_STATUS pTestMILSvcStatus,
P_INSTANCE_DATA pInst)
STATUS EXPORTED TestMILSvcGetStatus(P_TEST_MIL_SVC_STATUS pTestMILSvcStatus, P_INSTANCE_DATA pInst)
{
    Unused(pInst);
    return (STATUS)(OSSupervisorCall(XTestMILSvcGetStatus, &pTestMILSvcStatus, 2));
}

STATUS EXPORTED XTestMILSvcSetlnterrupt(P_INSTANCE_DATA plnst);
STATUS EXPORTED TestMILSvcSetlnterrupt(P_INSTANCE_DATA plnst)
{
    return (STATUS)(OSSupervisorCall(XTestMILSvcSetlnterrupt, &plnst, 1));
}

STATUS EXPORTED XTestMILSvcODestroy(P_INSTANCE_DATA plnst);
STATUS EXPORTED TestMILSvcODestroy(P_INSTANCE_DATA plnst)
{
    return (STATUS)(OSSupervisorCall(XTestMILSvcODestroy, &plnst, 1));
}

STATUS EXPORTED XTestMILSvcCancelPrint(P_INSTANCE_DATA plnst);
STATUS EXPORTED TestMILSvcCancelPrint(P_INSTANCE_DATA plnst)
{
    return (STATUS)(OSSupervisorCall(XTestMILSvcCancelPrint, &plnst, 1));
}

STATUS EXPORTED XTestMILSvcStartConnectionDetection(P_INSTANCE_DATA plnst);
STATUS EXPORTED TestMILSvcStartConnectionDetection(P_INSTANCE_DATA plnst)
{
    return (STATUS)(OSSupervisorCall(XTestMILSvcStartConnectionDetection, &plnst, 1));
}

STATUS EXPORTED XTestMILSvcStopConnectionDetection(P_INSTANCE_DATA plnst);
STATUS EXPORTED TestMILSvcStopConnectionDetection(P_INSTANCE_DATA plnst)
{
    return (STATUS)(OSSupervisorCall(XTestMILSvcStopConnectionDetection, &plnst, 1));
}

/******************************************************************************
void LOCAL TestMILSvcSwitchForConnection(P_INSTANCE_DATA pInst)
This routine sends the message msgTestMILSvcDoConnection, a private message, to self in process context. See msgTestMILSvcDoConnection for more details.
*******************************************************************************/
void LOCAL TestMILSvcSwitchForConnectionDetection(P_INSTANCE_DATA pInst)
{
    OS_TASK_ID taskId;

    ObjectCall(msgOwner, plnst->self, &taskId);
    ObjectSendTask(msgTestMILSvcDoConnection, pInst->self, &taskId, SizeOf(OS_TASK_ID), taskId);
}

******************************************************************************
MsgHandlerWithTypes(TestMILSvcDoConnection, P_ARGS, P_INSTANCE_DATA)
This routine handles the message msgTestMILSvcDoConnection, a private message. This message was self sent to change process to testMILSvc’s original process context. This process keeps the continuous mil request from terminating when the process whose context the service set connected message was sent. We don’t know if a sub-task will be created for the continuous request. We don’t need to do this for terminating connection detection.
*******************************************************************************/
MsgHandlerWithTypes(TestMILSvcDoConnection, P_ARGS, P_INSTANCE_DATA)
{
    P_INSTANCE_DATA pInst;

    MsgHandlerParametersNoWarning;
    pInst = *(P_INSTANCE_DATA *)pData;
    return TestMILSvcStartConnectionDetection(pInst);
}

} // TestMILSvcDoConnection
MsgHandlerWithTypes(TestMILSvcPenpointBooted, P_ARGS, P_INSTANCE_DATA)

This routine handles the message msgMILSvcConnectionProcessing. This message is received when the PenPoint operating system is fully booted. We wait to start connection processing until the system is booted and all services are running. When a connection is detected, all services will have a chance to respond to the queries of the conflict group manager.

****************************************************************************
MsgHandlerWithTypes(TestMILSvcPenpointBooted, P_ARGS, P_INSTANCE_DATA)
{
P_INSTANCE_DATA pInst;

MsgHandlerParametersNoWarning;

pInst = *(P_INSTANCE_DATA *)pData;

// Start connection detection if we have auto detecting hardware
if (pInst->connectStyle == svcAutoDetect)
{
  pInst->connected = false;
  // Start connection detection */
  TestMILSvcSwitchForConnectionDetection(pInst);
}
return stsOK;
} /* TestMILSvcPenpointBooted */

****************************************************************************
MsgHandlerWithTypes(TestMILSvcNewDefaults, P_ARGS, P_INSTANCE_DATA)
{
MsgHandlerParametersNoWarning;

/* Since a mil service targets a mil device not another
   * service, any style bit dealing with a target should be
   * false. These bits are 'waitForTarget', 'autoOwnTarget',
   * 'autoOpen', 'autoMsgPass', and 'autoOption'.
   * Since our mil service can only be opened by one client,
   * 'exclusiveOpen' and 'checkOwner' style bits should be true. */
pArgs->svc.style.waitForTarget = false;
pArgs->svc.style.exclusiveOpen = true;
pArgs->svc.style.autoOwnTarget = false;
pArgs->svc.style.autoOpen = false;
pArgs->svc.style.autoMsgPass = false;
pArgs->svc.style.checkOwner = true;
pArgs->svc.style.autoOption = false;
pArgs->svc.style.openClass = objNull;

/* Indicate which service managers we're to be placed on */
pArgs->svc.numManagers = SizeOf(logicalParallelPorts) / SizeOf(UID);
pArgs->svc.pManagerList = logicalParallelPorts;
return stsOK;
} /* TestMILSvcNewDefaults */

*****************************************************************************
BOOLEAN LOCAL IsAnybodyConnected(P_INSTANCE_DATA pInst)

This routine scans the conflict group we're in checking if a mil service in the conflict group is connected.
*****************************************************************************
BOOLEAN LOCAL IsAnybodyConnected(P_INSTANCE_DATA pInst)
{
LIST_ENTRY le;
OBJECT list;
U16 n;
SM_QUERY_LOCK query;
STATUS s;
SVC_GET_SET_CONNECTED serviceConnected;

// Create a list of mil services in our conflict group
ObjCallRet(msgIMGetList, pInst->conflictGroup, &list, s);

// Get the number of mil services in our conflict group
ObjectCall(msgListNumItems, list, &n);

// Query each mil service in our group
for (le.position = 0; le.position < n; le.position++)
Get the next mil service in our group
ObjCallWarn(msgListGetItem, list, &le);
query.handle = (OBJECT)le.item;
ObjCallWarn(msgSMQuery, pInst->conflictGroup, &query);

Get the connected state of the mil service
ObjCallWarn(msgSvcGetConnected, query.service, &serviceConnected);
if (serviceConnected.connected != false)
{
    // if connected - we're done
    break;
}

Destroy list
ObjCallWarn(msgDestroy, list, pNull);

return connected state
return serviceConnected.connected;

IsAnybodyConnected

****************************************************
MsgHandlerWithTypes(TestMILSvcNew, P_TEST_MIL_SVC_NEW, P_INSTANCE_DATA)
This routine handles the message msgInit. This routine creates a
new testMILSvc object. ‘msgMILSvcGetDevice’ is called to obtain the
mil device we are to use for printing. Connection detection is
not started until the PenPoint is booted. A check is provided
here in case we’re loaded after the system is booted. In which
case, we start connection detection only if no other mil service
in our conflict group is in the connected state. It’s normal for
all mil services other than the mil service being connected to
terminate their connection detection functions when the service
is connected.

*******************************************************************************
MsgHandlerWithTypes(TestMILSvcNew, P_TEST_MIL_SVC_NEW, P_INSTANCE_DATA)
{
    MIL_SVC_DEVICE device;
    OBJECT myself;
    P_INSTANCE_DATA pInst;
    TEST_MIL_SVC_TIME_DELAYS timeDelays;
    STATUS s;
    SYS_BOOT_STATE sysBootState;

    MsgHandlerParametersNoWarning;

    /* Create self. */
    myself = pArgs->object.uid;
    // Get mil device we are to use
    ObjCallRet(msgMILSvcGetDevice, myself, &device, s);
    Dbg(Debugf("TEST_MIL_SVC: creating instance for logicalId %d: unit %d",
                device.logicalId, device.unit));

    // Allocate our instance data
    StsRet(OSHeapBlockAlloc(osProcessSharedHeapId,
                SizeOf(INSTANCE_DATA),
                &pInst), s);

    // Store pointer to our instance data
    ObjectWrite(myself, ctx, &pInst);

    // Initialize our instance data
    memset(pInst, 0, SizeOf(INSTANCE_DATA));
    // Save who we are and who our mil device is
    pInst->self = myself;
    pInst->logicalId = device.logicalId;
    pInst->conflictGroup = device.conflictGroup;
    pInst->unit = device.unit;

    // Get whether we can detect connection
    pInst->connectStyle = pArgs->svc.style.connectStyle;

    // Initialize print in progress, connected and open flag
    pInst->printInProgress = false;
    pInst->connected = false;
    pInst->open = false;

    // Allocate necessary ring 0 memory
    StsRet(TestMILSvcGetReqBlocks(pInst), s);

    // Enable interrupt
    StsRet(TestMILSvcSetInterrupt(pInst), s);

    // Initialize parallel printer device timeouts
    timeDelays.initDelay = 500000;
    timeDelays.interruptTimeOut = 30000;
    ObjCallRet(msgTestMILSvcSetTimeDelays, self, &timeDelays, s);
    StsRet(TestMILSvcGetMetrics(&(pInst->testMILSvcMetrics), pInst), s);

    // Get the booted state of the system
    ObjCallRet(msgSysGetBootState, theSystem, &sysBootState, s);

    // Don't start if we don't have auto connecting hardware
    if (pInst->connectStyle == svcAutoDetect)
    {
        // Save who we are
        myself = pArgs->object.uid;
        // Get mil device we are to use
        ObjCallRet(msgMILSvcGetDevice, myself, &device, s);
        Dbg(Debugf("TEST_MIL_SVC: creating instance for logicalId %d: unit %d",
                    device.logicalId, device.unit));

        // Allocate our instance data
        StsRet(OSHeapBlockAlloc(osProcessSharedHeapId,
                    SizeOf(INSTANCE_DATA),
                    &pInst), s);

        // Store pointer to our instance data
        ObjectWrite(myself, ctx, &pInst);

        // Initialize our instance data
        memset(pInst, 0, SizeOf(INSTANCE_DATA));
        // Save who we are and who our mil device is
        pInst->self = myself;
        pInst->logicalId = device.logicalId;
        pInst->conflictGroup = device.conflictGroup;
        pInst->unit = device.unit;

        // Get whether we can detect connection
        pInst->connectStyle = pArgs->svc.style.connectStyle;

        // Initialize print in progress, connected and open flag
        pInst->printInProgress = false;
        pInst->connected = false;
        pInst->open = false;

        // Allocate necessary ring 0 memory
        StsRet(TestMILSvcGetReqBlocks(pInst), s);

        // Enable interrupt
        StsRet(TestMILSvcSetInterrupt(pInst), s);

        // Initialize parallel printer device timeouts
        timeDelays.initDelay = 500000;
        timeDelays.interruptTimeOut = 30000;
        ObjCallRet(msgTestMILSvcSetTimeDelays, self, &timeDelays, s);
        StsRet(TestMILSvcGetMetrics(&(pInst->testMILSvcMetrics), pInst), s);

        // Get the booted state of the system
        ObjCallRet(msgSysGetBootState, theSystem, &sysBootState, s);

        // Don't start if we don't have auto connecting hardware
        if (pInst->connectStyle == svcAutoDetect)
If the system is booted and no other mil service is connected
start connection detection
if (sysBootState.booted != false && IsAnybodyConnected(pInst) == false)
{
    TestMILSvcSwitchForConnectionDetection(pInst);
}
return s;

/* TestMILSvcNew */

/*****************************************************************************
MsgHandlerWithTypes(TestMILSvcFree, P_ARGS, P_INSTANCE_DATA)

This routine handles the message msgFree. This routine destroys
a testMILSvc object. We destroy ourself by terminating connection
detection and freeing all system resources we've allocated.
*****************************************************************************/

MsgHandlerWithTypes(TestMILSvcFree, P_ARGS, P_INSTANCE_DATA)
{
    P_INSTANCE_DATA pInst;
    STATUS s;
    MsgHandlerParametersNoWarning;
    pInst = *(P_INSTANCE_DATA *)pData;
    if (pInst != 0)
    {
        StsRet(TestMILSvcStopConnectionDetection(pInst), s);
        StsRet(TestMILSvcODestroy(pInst), s);
        StsRet(OSHeapBlockFree(pInst), s);
    }
    return stsOK;
} /* TestMILSvcFree */

/*****************************************************************************
MsgHandlerWithTypes(TestMILSvcGetSvcMetrics, P_SVC_GET_SET_METRICS, P_INSTANCE_DATA)

This routine handles the message msgSvcGetMetrics. This routine
gets the parallel printer port metrics.
*****************************************************************************/

MsgHandlerWithTypes(TestMILSvcGetSvcMetrics, P_SVC_GET_SET_METRICS, P_INSTANCE_DATA)
{
    P_INSTANCE_DATA pInst;
    MsgHandlerParametersNoWarning;

    // Get pointer to our instance data
    pInst = *(P_INSTANCE_DATA *)pData;

    /*
     * If the len field of the metrics structure passed in is zero
     * return the size of data area needed to store the metrics;
     * otherwise, if the length field equals the size of the
     * metrics, return the metrics.
     */
    if (pArgs->len == 0)
    {
        return pArgs->len = SizeOf(TEST_MIL_SVC_METRICS);
    }
    else if (pArgs->len != SizeOf(TEST_MIL_SVC_METRICS))
    {
        return stsBadParam;
    }
    else
    {
        memcpy(pArgs->pMetrics, &pInst->testMILSvcMetrics, SizeOf(TEST_MIL_SVC_METRICS));
    }
} /* TestMILSvcGetMetrics */

/*****************************************************************************
MsgHandlerWithTypes(TestMILSvcSetSvcMetrics, P_SVC_GET_SET_METRICS, P_INSTANCE_DATA)

This routine handles the message msgSvcSetMetrics. This routine
sets the parallel printer port metrics.
*****************************************************************************/

MsgHandlerWithTypes(TestMILSvcSetSvcMetrics, P_SVC_GET_SET_METRICS, P_INSTANCE_DATA)
{
    P_INSTANCE_DATA pInst;
    MsgHandlerParametersNoWarning;

    pInst = *(P_INSTANCE_DATA *)pData;

    /*
     * If the len field of the metrics structure passed in is zero
     * return the size of data area needed to store the metrics;
     * otherwise, if the length field equals the size of the
     * metrics, return the metrics.
     */
    if (pArgs->len == 0)
    {
        return pArgs->len = SizeOf(TEST_MIL_SVC_METRICS);
    }
    else if (pArgs->len != SizeOf(TEST_MIL_SVC_METRICS))
    {
        return stsBadParam;
    }
    else
    {
        memcpy(pArgs->pMetrics, &pInst->testMILSvcMetrics, SizeOf(TEST_MIL_SVC_METRICS));
    }
} /* TestMILSvcSetMetrics */
// Only set the metrics if the size of the data area is correct
if (pArgs->len != SizeOf(TEST_MIL_SVC_METRICS))
{
    return stsBadParam;
}
// Call the ring 0 code to set the mil device metrics
return TestMILSvcSetMetrics(pArgs->pMetrics, pInst);
} /* TestMILSvcSetMetrics */

/******************************
MsgHandlerWithTypes(TestMILSvcOpen, P_ARGS, P_INSTANCE_DATA)
This routine handles the message msgSvcOpenRequested/ 
msgQueryLockRequested. Either of these messages is received when 
a client wishes to open our mil service. We save our open state 
since when we receive messages from our client they don't go through 
the service open checks.
**********************************************************************/

MsgHandlerWithTypes(TestMILSvcOpen, P_ARGS, P_INSTANCE_DATA)
{
    P_INSTANCE_DATA pInst;
    MsgHandlerParametersNoWarning;

    pInst = *(P_INSTANCE_DATA *)pData;
    pInst->open = true;
    return stsOK;
} /* TestMILSvcOpen */

/******************************
MsgHandlerWithTypes(TestMILSvcClose, P_ARGS, P_INSTANCE_DATA)
This routine handles the message msgSvcCloseRequested/ 
msgSvcQueryUnlockRequested. Either of these messages is received when 
a client wishes to close our mil service.
**********************************************************************/

MsgHandlerWithTypes(TestMILSvcClose, P_ARGS, P_INSTANCE_DATA)
{
    P_INSTANCE_DATA pInst;
    MsgHandlerParametersNoWarning;

    pInst = *(P_INSTANCE_DATA *)pData;
    pInst->open = false;
    return stsOK;
} /* TestMILSvcClose */

/******************************
MsgHandlerWithTypes(TestMILSvcHWInitialize, P_ARGS, P_INSTANCE_DATA)
This routine handles the message msgTestMILSvcHWInitialize. This routine 
initializes the printer attached by toggling the initialize signal to 
the printer.
**********************************************************************/

MsgHandlerWithTypes(TestMILSvcHWInitialize, P_ARGS, P_INSTANCE_DATA)
{
    P_INSTANCE_DATA pInst;
    MsgHandlerParametersNoWarning;

    pInst = *(P_INSTANCE_DATA *)pData;
    
    if (pInst->connected == false || pInst->open == false)
    { 
        return stsFailed;
    }
    // Must call the mil device from ring 0
    return TestMILSvcInitialize(pInst);
} /* TestMILSvcHWInitialize */

/******************************
MsgHandlerWithTypes(TestMILSvcGetHWStatus, P_TEST_MIL_SVC_STATUS, 
P_INSTANCE_DATA)
This routine handles the message msgTestMILSvcGetStatus. The routine 
gets the contents of the parallel port status register.
**********************************************************************/

MsgHandlerWithTypes(TestMILSvcGetHWStatus, P_TEST_MIL_SVC_STATUS, 
P_INSTANCE_DATA)
{
    P_INSTANCE_DATA pInst;
    MsgHandlerParametersNoWarning;

    pInst = *(P_INSTANCE_DATA *)pData;
Must call the mil device from ring 0
return TestMILSvcGetStatus(pArgs, pInst);

} /* TestMILSvcGetHWStatus */

******************************************************************************
MsgHandlerWithTypes(TestMILSvcAutoLineFeedOn, P_ARGS, P_INSTANCE_DATA)

This routine handles the message msgTestMILSvcAutoLineFeedOn. The routine
sets the auto line feed signal to the printer active.
******************************************************************************

MsgHandlerWithTypes(TestMILSvcAutoLineFeedOn, P_ARGS, P_INSTANCE_DATA)
{
    P_INSTANCE_DATA pInst;
    MsgHandlerParametersNoWarning;
    
    pInst = *(P_INSTANCE_DATA *)pData;
    // Set the auto line feed flag in our metrics
    pInst->testMILSvcMetrics.unitFlags |= parallelAutoLineFeed;
    // Set the auto line feed flag in the mil devices parameters
    return TestMILSvcSetMetrics(&(pInst->testMILSvcMetrics), pInst);
} /* TestMILSvcAutoLineFeedOn */

******************************************************************************
MsgHandlerWithTypes(TestMILSvcAutoLineFeedOff, P_ARGS, P_INSTANCE_DATA)

This routine handles the message msgTestMILSvcAutoLineFeedOn. The routine
sets the auto line feed signal to the printer active.
******************************************************************************

MsgHandlerWithTypes(TestMILSvcAutoLineFeedOff, P_ARGS, P_INSTANCE_DATA)
{
    P_INSTANCE_DATA pInst;
    MsgHandlerParametersNoWarning;
    
    pInst = *(P_INSTANCE_DATA *)pData;
    // Clear the auto line feed flag in our metrics
    pInst->testMILSvcMetrics.unitFlags &= ~parallelAutoLineFeed;
    // Clear the auto line feed flag in the mil devices parameters
    return TestMILSvcSetMetrics(&(pInst->testMILSvcMetrics), pInst);
} /* TestMILSvcAutoLineFeedOff */

******************************************************************************
MsgHandlerWithTypes(TestMILSvcGetTimeDelays, P_ARGS, P_INSTANCE_DATA)

This routine handles the message msgTestMILSvcGetTimeDelays. This routine
gets the time delays used by the mil device. 'initDelay' is the width
in microseconds of the initialization pulse used to hardware initialize
the printer. 'interruptTimeOut' is the time in milliseconds to wait
for an interrupt to occur after writing a character to the printer.
******************************************************************************

MsgHandlerWithTypes(TestMILSvcGetTimeDelays, P_ARGS, P_INSTANCE_DATA)
{
    P_INSTANCE_DATA pInst;
    MsgHandlerParametersNoWarning;
    
    pInst = *(P_INSTANCE_DATA *)pData;
    // Return the time periods from our metrics
    pArgs->initDelay = pInst->testMILSvcMetrics.initDelay;
    pArgs->interruptTimeOut = pInst->testMILSvcMetrics.interruptTimeOut;
    return stsOK;
} /* TestMILSvcGetTimeDelays */

******************************************************************************
MsgHandlerWithTypes(TestMILSvcSetTimeDelays, P_ARGS, P_INSTANCE_DATA)

This routine handles the message msgTestMILSvcSetTimeDelays. This routine
sets the time delays used by the mil device. 'initDelay' is the width
in microseconds of the initialization pulse used to hardware initialize
the printer. 'interruptTimeOut' is the time in milliseconds to wait
for an interrupt to occur after writing a character to the printer.
******************************************************************************

MsgHandlerWithTypes(TestMILSvcSetTimeDelays, P_ARGS, P_INSTANCE_DATA)
{
    P_INSTANCE_DATA pInst;
    MsgHandlerParametersNoWarning;
    
    pInst = *(P_INSTANCE_DATA *)pData;
    // Set the time periods from our metrics
    pArgs->initDelay = pInst->testMILSvcMetrics.initDelay;
    pArgs->interruptTimeOut = pInst->testMILSvcMetrics.interruptTimeOut;
    return stsOK;
} /* TestMILSvcSetTimeDelays */
pInst = *(P_INSTANCE_DATA *)pData;

// Neither time period can be zero
if (pArgs->initDelay == 0 || pArgs->interruptTimeOut == 0)
{
    return stsBadParam;
}
else
{
    // Set our metrics to the new values
    pInst->testMILSvcMetrics.initDelay = pArgs->initDelay;
    pInst->testMILSvcMetrics.interruptTimeOut = pArgs->interruptTimeOut;

    // Tell the mil device of the new values
    return TestMILSvcSetMetrics(&(pInst->testMILSvcMetrics), pInst);
}

} /* TestMILSvcSetTimeDelays */

/*************************************************************************
**                         MsgHandlerWithTypes                          
**                        (TestMILSvcPrintBuffer, P_ARG, P_INSTANCE_DATA) 
**                      */

MsgHandlerWithTypes(TestMILSvcPrintBuffer, P_ARG, P_INSTANCE_DATA)

This routine handles the message msgStreamWrite. The routine causes a
client's buffer to be sent to the printer connected. An error is
returned and the buffer is not printed if we are not open and no printer
is connected.

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

**************************************************************************
**                         MsgHandlerWithTypes                          
**                        (TestMILSvcCancelPrintBuffer, P_ARG, P_INSTANCE_DATA) 
**                      */

MsgHandlerWithTypes(TestMILSvcCancelPrintBuffer, P_ARG, P_INSTANCE_DATA)

This routine handles the message msgTestMILSvcCancelPrintBuffer. The
routine cancels the printing of the buffer currently being printed.

**************************************************************************
**                         MsgHandlerWithTypes                          
**                        (TestMILSvcFlush, P_ARG, P_INSTANCE_DATA) 
**                      */

MsgHandlerWithTypes(TestMILSvcFlush, P_ARG, P_INSTANCE_DATA)

This routine handles the message msgFSFlush. This routine allows a
client to send a flush to our mil service without causing an error.

**************************************************************************
**                         MsgHandlerWithTypes                          
**                        (TestMILSvcAreYouConnected, 
**                          P_MIL_SVC_ARE_YOU_CONNECTED, 
**                          P_INSTANCE_DATA) 
**                      */

MsgHandlerWithTypes(TestMILSvcAreYouConnected, 
                    P_MIL_SVC_ARE_YOU_CONNECTED, 
                    P_INSTANCE_DATA)

This routine handles the message msgMILSvcAreYouConnected. The
message is sent by our conflict group manager when a mil service in
our conflict group reports connected. More than one mil service can report connected. The valid responses are msYES, 
msMaybe, or msNO. 'msYES' indicates we are certain our device is connected. 'msMaybe' indicates something is connected, but we are not sure if it is our device. 'msMaybe' can also indicate we don't know if anything is connected. 'msNO' indicates our device is not connected. The mil device for the printer can determine absolutely if a printer is connected.
This message is sent by our conflict manager when a mil service’s connection state changes. This message indicates which mil service on the conflict group is being set connected. The logical id of the corresponding mil device is used to indicate who received the connection. A logical id of maxU16 indicates a mil service was disconnected. When another mil service receives connection, we should terminate our connection detection function to prevent our connection function from interfering with the operation of the other mil service and mil device. We restart our connection detection function when the other mil service is disconnected.

****************************************************************************
MsgHandlerWithTypes(TestMILSvcAreConnected, P_MIL_SVC_ARE_CONNECTED, P_INSTANCE_DATA)
{
    P_INSTANCE_DATA     pInst;
    MsgHandlerParametersNoWarning;

    pInst = *(P_INSTANCE_DATA *)pData;

    /*
     * Answer maybe if we can't determine
     * if anything is connected. If only one
     * other mil service indicates it is connected
     * with msYes, then that mil service will get
     * the connection. If more than one mil service
     * indicates it is connected with msYes or msMaybe,
     * then a dialog box will be displayed with all the
     * mil services indicating connected. The user
     * then selects which mil service will actually be
     * connected. If we're the only mil service indicating
     * connected with msMaybe, then we will be connected by
     * default.
     */
    if (pInst->connectStyle == svcNoAutoDetect)
    {
        *pArgs = msMaybe;
    }
    else if (pInst->connected == false)
    {
        Dbg(Debugf("TEST_MIL_SVC: responding to 'msgMILSvcAreConnected' "
                    "with msNo");)
        *pArgs = msNo;
    }
    else
    {
        Dbg(Debugf("TEST_MIL_SVC: responding to 'msgMILSvcAreConnected' "
                    "with msYes");)
        *pArgs = msYes;
    }
    return stsOK;
} /* TestMILSvcAreYouConnected */

***************************************************************************************
MsgHandlerWithTypes(TestMILSvcConnectionStateResolved, P_ARGS, P_INSTANCE_DATA)
{
    P_INSTANCE_DATA     pInst;
    MsgHandlerParametersNoWarning;

    pInst = *(P_INSTANCE_DATA *)pData;

    /*
     * Start our connection detection function if
     * no other mil service is connected, if our
     * mil device can detect connection, and if
     * our connection detection function is not
     * already started.
     * Terminate our connection detection function
     * if another mil service in our conflict group
     * is connected and if we have a connection
     * detection function started.
     * Otherwise if we are connected, perform any
     * initialization operations.
     */
    if ((U16)*pArgs == maxU16)
    {
        if (pInst->connectStyle == svcAutoDetect && pInst->pRBConnect == 0)
        {
            TestMILSvcSwitchForConnectionDetection(pInst);
        }
    }
    else if ((U16)*pArgs != pInst->logicalId)
    {
        if (pInst->connectStyle == svcAutoDetect && pInst->pRBConnect != 0)
        {
            TestMILSvcStopConnectionDetection(pInst);
            pInst->pRBConnect = 0;
        }
else if ((UL16)pArgs == pInst->logicalId)
{
}
return stsOK;
} /* TestMILSvcConnectionStateResolved */

(MsgHandlerWithTypes(TestMILSvcPowerOff, P_ARGS, P_INSTANCE_DATA)

This message handles the message msgMILSvcPowerOff. This message is received when the power to the tablet is about to be shut off. We perform any clean up operations here. There is a corresponding message, msgMILSvcPowerOn, which can be handled when power is applied. At which time, we can perform any operations necessary when power is applied. When power is shut off we want to cancel printing if a buffer is currently being printed.

**********************************************
MsgHandlerWithTypes(TestMILSvcPowerOff, P_ARGS, P_INSTANCE_DATA)
{
    P_INSTANCE_DATA pInst;
    MsgHandlerParametersNoWarning;
    pInst = *(P_INSTANCE_DATA *)pData;
    return TestMILSvcCancelPrint(pInst);
} /* TestMILSvcPowerOff */

ClsTestMILSvcServiceInit (void)

Install the class.

**********************************************
STATUS EXPORTED ClsTestMILSvcServiceInit(void)
{
    CLASS_NEW classNew;
    STATUS s;

    /* Create the class. */
    ObjCallWarn(msgNewDefaults, clsClass, &classNew);
    classNew.cls.pMsg = clsTestMILSvcServiceTable;
    classNew.cls.ancestor = clsMILService;
    classNew.cls.size = SizeOf(INSTANCE_DATA);
    classNew.cls.newArgsSize = SizeOf(TEST_MIL_SVC_NEW);
    ObjCallRet(msgNew, clsClass, &classNew, s);
    return (stsOK);
} /* ClsTestMILSvcServiceInit */

DLLMain

Initialize milsvc.dll

**********************************************
STATUS EXPORTED DLLMain(void)
{
    STATUS s;
    SVC_INIT_SERVICE initService;

    // Install classes
    StsRet(ClsTestMILSvcServiceInit(), s);
    // Let system know about our service
    // and set global service characteristics
    memset(initService.spare, 0, sizeof(initService.spare));
    initService.autoCreate = true;
    initService.serviceType = 0;
    initService.initServiceFlags = svcNoShow;
    ObjCallRet(msgSvcClassInitService,
               clsMILParallelPortDevice,
               &initService, s);
    return stsOK;
} /* DLLMain */

MILSVCO.H

File: milsvc0.h

(C) Copyright 1992 by GO Corporation, All Rights Reserved.

You may use this Sample Code any way you please provided you do not resell the code and that this notice (including the above
This file contains the private interface definition for clsTestMILService. clsTestMILService is a subclass of clsMILService.

```c
/*   Private Parallel Port Class Messages   */
/*****************************************************************************/

msgTestMILSvcStatus takes P_NULL, returns STATUS
starts connection detection.

We switch to our process context to start connection detection. We don't need to do this when connection detection is terminated.

#define msgTestMILSvcDoConnection MakeMsg(clsTestMILService, 100)

/*   Instance Data   */

typedef struct INSTANCE_DATA
{
    // Who we are
    OBJECT self; // our instance
    U16 logicalId; // mil device logical id
    U16 unit; // mil device unit number
    UID conflictGroup; // conflict group we're on
    U16 connectStyle; // is connection detection supported
    BOOLEAN testMILSvcMetrics; // our metrics
    BOOLEAN open; // are we open for business
    BOOLEAN connected; // are we connected
    BOOLEAN printInProgress; // are we in mil printing
    // Request blocks
    P_MIL_REQUEST_BLOCK pRBMisc; // used for single stage req.
    P_MIL_REQUEST_BLOCK pRBPrint; // used for printing
    P_MIL_REQUEST_BLOCK pRBConnect; // used for connection
    // Pointer to ring 0 buffer for printing
    P_U8 printBuffer;
} INSTANCE_DATA, *P_INSTANCE_DATA;
#endif

MILSVCO.C

*******************************************************************************
File: milsvcO.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: 1.0 $  
$Date: 12 Mar 1992 12:33:00 $
This file contains the ring 0 code for clsTestMILService.

*******************************************************************************
#include <clsmgr.h>
#include <debug.h>
#include <list.h>
#include <servmgr.h>
#include <service.h>
#include <stream.h>
#include <drvmgr.h>
#include <os.h>
```
#include <string.h>
#include <dvpall.h>
#include <milsvc.h>
#include <milsvc0.h>

#include <string.h>
#include <dvpall.h>
#include <milsvc.h>
#include <milsvc0.h>

1****************************************************
**********************
pRBInit, pRBPrn, pRBSet, pRBStatus, and pRBAttach

Shorthand version of corresponding variable on right

pRBInit -- used in printer initialization routine
pRBPrn -- used in print routine
pRBSet -- used in get/set device parameters routines
pRBStatus -- used in the get printer status routine
pRBAttach -- used in the printer attachment routines
1****************************************************
**********************1

define pRBInit ((P_MIL_RB_PUB_PARALLEL_INIT_PRINTER)(pInst->pRBMisc))
define pRBPrn ((P_MIL_RB_PUB_PARALLEL_WRITE)(pInst->pRBPrint))
define pRBSet ((P_MIL_RB_PUB_PARALLEL_DEVICE_PARAMETERS)(pInst->pRBMisc))
define pRBStatus ((P_MIL_RB_PUB_PARALLEL_DEVICE_PARAMETERS)(pInst->pRBMisc))
define pRBAttach ((P_MIL_RB_PUB_BASE_ATTACHMENT_CONT)(pInst->pRBConnect))

1****************************************************
**********************1

STATUS EXPORTED0 TestMILSvcGetReqBlocks(P_INSTANCE_DATA pInst)

Allocates ring0 system resources.

STATUS EXPORTED0 TestMILSvcGetReqBlocks(P_INSTANCE_DATA pInst)
{
  STATUS s;

  /*
  * Our instance data is used by the attachment callback
  * procedure. Locking it in memory insures that it will
  * be in memory when we need it. We aren't required to do
  * this since the attachment routine is a timed not an interrupt
  * function. Otherwise page faults can occur to bring pages
  * swapped to disk back into memory.
  */
  OSMemLock(pInst, SizeOf(INSTANCE_DATA));

  // Allocate a request block for all the single stage mil requests
  StsRet(DrvMILRequestBlockCreate(pInst->logicalId,
                         (PP_UNKNOWN)(&(pInst->pRBMisc))), s);

  // Allocate a request to perform printing
  StsRet(DrvMILRequestBlockCreate(pInst->logicalId,
                         (PP_UNKNOWN)(&(pInst->pRBPrint))), s);

  // Initialize the unit field in the request block to our unit number
  pInst->pRBPrint->unit = pInst->unit;
  pInst->pRBMisc->unit = pInst->unit;

  /*
  * Allocate a 512 byte buffer (size arbitrary).
  * This buffer must be allocated/locked such that it
  * in memory at all times. The printing mil request
  * is a stage on interrupt function; therefore, the
  * buffer to be printed must be in memory at all times.
  * A page fault to bring in a non-memory resident page
  * cannot occur within an interrupt routine.
  */
  StsRet(OSHeapBlockAlloc(osProcessSharedHeapId, 512,
                         *(pInst->printBuffer), s));
  return stsOK;
}

1****************************************************
**********************1

STATUS EXPORTED0 TestMILSvcODestroy(P_INSTANCE_DATA pInst)

Free all ring0 resources used by this service

STATUS EXPORTED0 TestMILSvcODestroy(P_INSTANCE_DATA pInst)
{
  BOOLEAN enable;
  LIST conflictGroupItems;
  STATUS s;
  UI6 numConflictItems;

  // Unlock memory used by this service
  OSMemUnlock(pInst, SizeOf(INSTANCE_DATA));

  // Free request blocks
  StsRet(DrvMILRequestBlockFree((PP_UNKNOWN)(pInst->pRBMisc)), s);
  StsRet(DrvMILRequestBlockFree((PP_UNKNOWN)(pInst->pRBPrint)), s);

  // Free our print buffer
  if (pInst->printBuffer != 0)
    
    // Unlock our print buffer
    OSMemUnlock(pInst->printBuffer, 512);

  // Free our print buffer
  StsRet(OSHeapBlockFree(pInst->printBuffer), s);

  // If we are the last mil service on our

  return stsOK;
}
// conflict group, disable our interrupt.
ObjCallRet(msgIMGetList, pInst->conflictGroup, &conflictGroupItems, s);
ObjCallRet(msgListNumItems, conflictGroupItems, &numConflictItems, s);
if (numConflictItems == 1)
{
    enable = false;
    OSIntMask(pInst->pRBMisc->logicalId, &enable);
}
return ObjectCall(msgDestroy, conflictGroupItems, pNull);
} /* TestMILSvcDestory */

/*****************************
STATUS EXPORTED0 TestMILSvcSetInterrupt(P_INSTANCE_DATA pInst)
Enables parallel port interrupt
**************************************************************************/
STATUS EXPORTED0 TestMILSvcSetInterrupt(P_INSTANCE_DATA pInst)
{
    BOOLEAN enable;
    enable = true;
    return OSIntMask(pInst->pRBMisc->logicalId, &enable);
} /* TestMILSvcSetInterrupt */

*****************************************************************************
void EXPORTED0 TestMILSvcAttachmentCallBack(P_MIL_COMMON_DATA pCommonData,
P_MIL_RB_PUB_BASE_ATTACHMENT_CONT pRB)

The connection detection call back routine. This routine is called by the mil connection detection function whenever a printer is attached or detached.
*****************************************************************************
void EXPORTED0 TestMILSvcAttachmentCallBack(P_MIL_COMMON_DATA pCommonData,
P_MIL_RB_PUB_BASE_ATTACHMENT_CONT pRB)
{
    P_INSTANCE_DATA pInst;
    SVC_GET_SET_CONNECTED svcGetSetConnected;
    Unused(pCommonData);

    // Get our instance data pointer from the request block
    pInst = (P_INSTANCE_DATA)(pRB->callerDataU32A);
    if (pRB->event == milDevAttachedEvent)
    {
        // We're attached
        pInst->connected = true;
        Dbg(Debugf("TEST_MIL_SVCO: printer connected to device %d", pRB->logicalId);
    }
    else if (pRB->event == milDevDetachedEvent)
    {
        // We're detached
        pInst->connected = false;
        Dbg(Debugf("TEST_MIL_SVCO: printer not connected to device %d", pRB->logicalId));
    }

    // Send message to ourself indicating new connection state
    svcGetSetConnected.connected = pInst->connected;
    ObjectPost(msgSvcSetConnected, pInst->self, &svcGetSetConnected,
        SizeOf(SVC_GET_SET_CONNECTED));
} /* TestMILSvcAttachmentCallBack */

*****************************************************************************
STATUS EXPORTED0 TestMILSvcStartConnectionDetection(P_INSTANCE_DATA pInst)
Start the parallel mil devices connection function. The connection function is a continuous staged on time function. When the connection function runs, it checks for a printer connected to the parallel port. When a printer is attached or detached, the attachment call back procedure is called with the attachment state.
*****************************************************************************
STATUS EXPORTED0 TestMILSvcStartConnectionDetection(P_INSTANCE_DATA pInst)
{
    STATUS s;

    // Initially indicate we are disconnected
    pInst->connected = false;

    /*
    * Allocate a request block for the attachment function
    * We allocate the request block here since the cancel
    * request frees the request block
    */
    StsRet(DrvMILRequestBlockCreate(pInst->logicalId,
        (P_PP_UNKNOWN)((pInst->pRBConnect)), s);

    /* Fill in the appropriate fields of the request block
    * our unit number
    * function code for attachment
    * user data (our instance data)
    * the call back function to use
    */
pRBAttach->unit = pInst->unit;
pRBAttach->functionCode = milBaseAttachmentCont;
pRBAttach->callerDataU32A = (U32)pInst;
pRBAttach->pAsyncEventFunc =
    (P_MIL_ASYNC_EVENT_FUNC) TestMILSvcAttachmentCallBack;

    // Start the attachment function
    return DrvMILRequest(pInst->pRBConnect);
} /* TestMILSvcStartConnectionDetection */

/******************************************************************************/
STATUS EXPORTED0 TestMILSvcStopConnectionDetection(P_INSTANCE_DATA pInst)
Terminates the connection detection mil request.
*******************************************************************************/

STATUS EXPORTED0 TestMILSvcStopConnectionDetection(P_INSTANCE_DATA pInst)
{
    return DrvMILCancelRequest(pInst->pRBConnect);
} /* TestMILSvcStopConnectionDetection */

/******************************************************************************/
STATUS EXPORTED0 TestMILSvcGetMetrics(P_TEST_MIL_SVC_METRICS pArgs,
P_INSTANCE_DATA pInst)
Get the mil device parameters portion of testMILSvc metrics
*******************************************************************************/

STATUS EXPORTED0 TestMILSvcGetMetrics(P_TEST_MIL_SVC_METRICS pArgs,
P_INSTANCE_DATA pInst)
{
    STATUS s;

    /* You must reset the request block before */
    /* reusing an existing request block. */
    DrvMILRequestBlockReset(pInst->pRBMisc);

    // Request the device parameters from the parallel port mil device
    pRBSet->functionCode = milParallelGetDevParameters;
    StsRet(DrvMILRequest(pInst->pRBMisc), s);

    // Copy parameters to metrics structure
    pArgs->devFlags = pRBSet->parallelDevParms.parallelDevFlags;
    pArgs->unitFlags = pRBSet->parallelUnitParms.parallelUnitFlags;
    pArgs->initDelay = pRBSet->parallelTimeIntervals.initDelay;
    pArgs->interruptTimeOut = pRBSet->parallelTimeIntervals.interruptTimeOut;
    return stsOK;
}

/******************************************************************************/
STATUS EXPORTED0 TestMILSvcSetMetrics(P_TEST_MIL_SVC_METRICS pArgs,
P_INSTANCE_DATA pInst)
Set the mil device parameters from the testMILSvc metrics
*******************************************************************************/

STATUS EXPORTED0 TestMILSvcSetMetrics(P_TEST_MIL_SVC_METRICS pArgs,
P_INSTANCE_DATA pInst)
{
    STATUS s;

    // interrupt time out and init delay cannot be zero
    if (pArgs->interruptTimeOut == 0 || pArgs->initDelay == 0)
    {
        return stsBadParam;
    }
    else
    {
        /* You must reset the request block before */
        /* reusing an existing request block. */
        DrvMILRequestBlockReset(pInst->pRBMisc);

        // Fill in the proper fields in the request block
        pRBSet->functionCode = milParallelSetDevParameters;
        pRBSet->parallelDevParms.parallelDevFlags = pArgs->devFlags;
        pRBSet->parallelUnitParms.parallelUnitFlags = pArgs->unitFlags;
        pRBSet->parallelTimeIntervals.initDelay = pArgs->initDelay;
        pRBSet->parallelTimeIntervals.interruptTimeOut =
            pArgs->interruptTimeOut;

        // Perform the request to set the parameters
        if ((s = DrvMILRequest(pInst->pRBMisc)) == stsOK)
        {
            // If successful copy new metrics to instance data
            memcpy(&(pInst->testMILSvcMetrics), pArgs,
                SizeOf(TEST_MIL_SVC_METRICS));
        }
        else
        {
            // Otherwise get old metrics from mil device
            TestMILSvcGetMetrics(&(pInst->testMILSvcMetrics), pInst);
        }
    }
    return s;
}
TestMILSvcSetMetrics

Have the parallel printer mil device initialize the printer by asserting the initialize hardware signal to the printer. The initialize signal is asserted for initDelay microseconds.

TestMILSvcInitialize(P_INSTANCE_DATA pInst)

Have the parallel printer mil device initialize the printer by asserting the initialize hardware signal to the printer. The initialize signal is asserted for initDelay microseconds.

TestMILSvcInitialize(P_INSTANCE_DATA pInst)

Have the parallel printer mil device initialize the printer by asserting the initialize hardware signal to the printer. The initialize signal is asserted for initDelay microseconds.

TestMILSvcGetStatus(P_TEST_MIL_SVC_STATUS portStatus,
P_INSTANCE_DATA pInst)

Get the status of the printer connected to the parallel port. All parallel printer functions return the printer status. We use the get device function since it gets the status without affecting the printer. The get printer status returns the contents of the parallel port status register.

TestMILSvcGetStatus(P_TESTS_MIL_SVC_STATUS portStatus,
P_INSTANCE_DATA pInst)

Get the status of the printer connected to the parallel port. All parallel printer functions return the printer status. We use the get device function since it gets the status without affecting the printer. The get printer status returns the contents of the parallel port status register.

TestMILSvcGetStatus(P_TEST_MIL_SVC_STATUS portStatus,
P_INSTANCE_DATA pInst)

Get the status of the printer connected to the parallel port. All parallel printer functions return the printer status. We use the get device function since it gets the status without affecting the printer. The get printer status returns the contents of the parallel port status register.

U8 LOCAL HexToAscii(U8 num)

#define HighU4(x) (((x) >> 4) & 0x0f)
#define LowU4(x) ((x) & 0x0f)

void LOCAL DisplayBuffer(P_U8 pBuffer, U16 bufferSize)

for (count = 0, i = 0, count < bufferSize; count++)

outbuf[i++] = HexToAscii(HighU4(pBuffer[count]));
outbuf[i++] = HexToAscii(LowU4(pBuffer[count]));
outbuf[i++] = '\0';
if (i > 77) {
    outbuf[i] = '\0';
    Debugf(outbuf);
    i = 0;
}
if (i) {
    outbuf[i] = '\0';
    Debugf(outbuf);
}
} /* DisplayBuffer */
#endif

>Status EXPORTED0 TestMILSvcPrint(P_STREAM_READ_WRITE pStream,
                                   P_INSTANCE_DATA pInst)

Sends the contents of the buffer specified in the pStream structure
to the printer to be printed. The buffer is copied to our local
buffer before printing. It is copied to our buffer to insure that
the data is available when the interrupt routine is executed.

>Status EXPORTED0 TestMILSvcPrint(P_STREAM_READ_WRITE pStream,
                                   P_INSTANCE_DATA pInst)

    U16 count;
    U16 i, j;

    Dbg(Debugf("TEST_MIL_SVC0: Printing");)

    // initialize the count we have printed variable to zero
    count = 0;
    do {
        /*
        * copy 512 bytes of the client buffer
        * or the entire client buffer to our
        * local buffer
        */
        j = count;
        for (i = 0; i < 512 && j < pStream->numBytes; i++, j++)
            pInst->printBuffer[i] = ((P_U8)(pStream->pBuf)[j];
    } /* You must reset the request block before
        * reusing an existing request block.
        */
    DrvMILRequestBlockReset(pInst->pRBPrint);

    if (j)
    {
        outbuf[i] = '\0';
        Debugf(outbuf);
    }
} /* DisplayBuffer */

>Status EXPORTED0 TestMILSvcPrint(P_STREAM_READ_WRITE pStream,
                                   P_INSTANCE_DATA pInst)

    U16 count;
    U16 i, j;

    Dbg(Debugf("TEST_MIL_SVC0: Printing");)

    // initialize the count we have printed variable to zero
    count = 0;
    do {
        /*
        * copy 512 bytes of the client buffer
        * or the entire client buffer to our
        * local buffer
        */
        j = count;
        for (i = 0; i < 512 && j < pStream->numBytes; i++, j++)
            pInst->printBuffer[i] = ((P_U8)(pStream->pBuf)[j];
    } /* You must reset the request block before
        * reusing an existing request block.
        */
    DrvMILRequestBlockReset(pInst->pRBPrint);

    // call the mil
    StsWarn(DrvMILRequest(pInst->pRBPrint));

    // has the printing been cancelled?
    if (pInst->printInProgress == false)
    {
        /* yes - exit
         * break;
        */
    }

    // indicate we are not in the mil
    pInst->printInProgress = false;

    // update count of total characters printed
    count += pRBPrn->countPrinted;

    // continue until an error or the entire buffer is printed
    while (pRBPrn->countPrinted == pRBPrn->bufferSize &&
           count < pStream->numBytes);

    // indicate number of characters printed
    pStream->count = count;

    #ifdef DEBUG
    // if an error occurred display buffer being printed
    if (pRBPrn->status != STS_OK)
    {
        Debugf("count printed is %d -- count requested %d",
                pRBPrn->countPrinted,
                pRBPrn->bufferSize);
    }

    Debugf(DisplayBuffer(pInst->printBuffer, pRBPrn->bufferSize));
    Debugf("pStream->count is %d -- pStream->numBytes is %d",
            pStream->count,
            pStream->numBytes);
pStream->numBytes);
#endif

return pRBPrn->status;
)
/* TestMILSvcPrint */

/****************************************************
 STATUS EXPORTED0 TestMILSvcCancelPrint(P_INSTANCE_DATA pInst)
 Cancel print request.
 **************************************************

 STATUS EXPORTED0 TestMILSvcCancelPrint(P_INSTANCE_DATA pInst)
 {
     STATUS s;

     // If print mil reques in progress, cancel request
     if (pInst->printInProgress != false)
     {
         // call mil with request block to be cancelled
         StsRet(DrvMILCancelRequest(pInst->pRBPrint), s);

         // indicate request has been cancelled
         pInst->printInProgress = false;
     }
     return stsOK;
 }
/* TestMILSvcCancelPrint */
16-bit character  
string functions, 111–114  
composition, 114  
support, 110–114  
features, 110  
types, 111  

8259 programmable interrupt controller (PIC), 275  

80386  
protected mode, 102  
ring structure, 103  

AB_MGR_ID structure, 329  
AB_MGR_NOTIFY structure, 330  

Access  
intentions, 62  
protocols, 257  

Accessing services, 258–259  
binding to a service, 259  
opening service, 259  
service managers, 261  
predefined, 258–259  

see also Services  

Accessories, 393  
documents, 377  
ACCESSRY directory, 393  

Active In box service, 312  

Adding  
address book entry, 328  
document to stationary menu, 426  
items to transaction, 203  
list items, 129  
network protocols, 251  
rows to tables, 222–223  
transfer types, 173  

see also Installing  

ADDR_BOOK_ATTR structure, 320–321  
ADDR_BOOK_ENTRY structure, 322, 327  
allocation of, 328  
ADD_BOOK_QUERY_ATTR  
structure, 426–327  

AddrBookStreetId, address book identifier, 321  

changing information in, 328  
closing, 326  
concepts, 317  
defined, 317–318  
entry  
adding, 328  
attribute identifiers, 321  
attributes, 320–322  
deleting, 328  
groups, 322  
organization, 320  
service addresses, 322  
GO, application, 323–324  
messages, 324–325  
msgSendServGetAddrDesc and, 332  
opening, 326  
operation participants, 318  
organization, 320–322  
protocols, 318–320  
registering, 329  
searching, 326–328  
sendable services protocol uses, 331  
system, 329–330  
deactivating, 330  
defined, 318  
theAddressBookMgr and, 318  
unregistering, 329  
writing, 328–330  
Address book manager protocol, 320  
function, 318–319  
Address book protocol, 319  
function, 318  
Address descriptors, 331–332  
getting, 333  
Address window, 332  
creating, 333  
filling, 333–334  
Agents, resource, 345  
Alarm services, 103  
ANM_CREATE_DOC structure, 425  
ANM_CREATE_SECT structure, 424  
ANM_DELETE_ALL structure, 426  
ANM_DELETE structure, 426  
ANM_GET_NOTEBOOK  
structure, 424–424  
ANM_MOVE_COPY_DOC  
structure, 425–426  
appAttrClass, 148  
AP directory, 384, 386  
contents, 386  
directory contents, 386–387  
APP.INI file, 387  
service directory and, 444  
AppleTalk  
protocol, 301–302  
changing size of ATP packets  
and, 302  
name, 302–304  
options, 301–302  
zones, 304  
services, 250  
AppleTalk transport protocol  
(ATP), 253, 297  
changing packet size, 302  
Application directories, 391–395  
accessories, 393  
creating, 391  
files in, 391–392  
global data, 394–395  
help, 393–394  
stationary, 392–393  
see also Applications  
Application distribution cassette, 375  
Application global data, 394–395  
Application installation manager, 415–416  
Application monitor, 378, 380  
checking dll-ids, 401–402  
DLL files and, 400  
multiple volumes and, 398  
Applications, 438  
drivers/devices and, 246  
installable, 386–387  
installing, 377–378, 415  
multiple volumes and, 398  
ports and, 245  
upgrading, 398  
AppMonitorMain(), 377–378  
ASCII  
metrics transfer, 175  
text file creation, 180  
AT command, modem, 286–287  
set, 290–293  
At-least-once-delivery, datagram, 296  
Atom identifier, 15  
Atoms, 37–38  
defined, 37  
for nil string, 37  
preset, 38  
ATP_OPTIONS structure, 301–302  
Attributes, 7–9  
address book entry, 320–322  
identifiers, 321  
aruments, 16–18  
changing, 8, 19–20  
character, 8–9, 16–17  
clearing, 20  
default, 7  
file system, 55, 77  
client defined, 77–78  
getting and setting, 76–80  
getting values, 78  
length of values, 79  
lists of, 76–77  
setting values, 79  
zero value, 77  
getting and setting, 16–20  
initializing, 19  
installable item, 412  
label macros, 77–78
INDEX

local, 7
modifying, 19–20
node, 54–55
client-defined, 54
file-system, 55
flags, 79–80
paragraph, 9, 17–18
value types, 76

Auto-answer mode, modem, 284
Auto shutdown preference, 365
Auto suspend preference, 364
Auxiliary notebook manager, 421–427
messages, 423
generalized, 423–424
specialized, 424–426
Auxiliary notebooks, 380, 421–422
back up considerations, 422
concepts, 421–422
creating, documents, 425
creating, sections, 424
deleting section/document, 426
file system and, 422
getting paths to, 423–424
list of, 422
moving/copying documents to, 425–426
opening, 423
tags, 421–422
see also Notebook

Backslashes, in path names, 66
BASICSVC service, 485–487
BASICSVC.H, 485–487
declared, 475
METHOD.TBL, 485
Baud rate, setting, 269
Bell preference, 365
Binding, 247
to local transport address, 301
to service, 259
Block, 7
Blocking protocol, 168–169
deadlocks, 169
defined, 168
Boolean operators, table, 225
Boot
disks, services on, 444
progress messages, 431–432
sequence, 429
symbols, 429
volume, 43
BOOT directory, 384
structure, 385
BREAK signal, 272
Browser, 124
changing, client, 144
class, 137–145
concepts, 137–138
creating, 138
object, 140
defined, 137
examples, 137
expanding and collapsing sections with, 143
file export mechanism, 147
file import mechanism, 147
getting and setting, metrics, 143–144
integrating, into application, 138
menu bar, 138
menu messages, 145
navigating with, 144
notification messages, 145
reading and writing, state, 143
refreshing, data, 142
selection, 140–141
table of contents and, 137–138
TOC, 148
user columns, 145–146
BROWSER_CREATE_DOC structure, 141–142
BROWSER_METRICS structure, 144
BROWSER_NEW structure, 140
Buffered data, 265
Buffers
flushing, 85
input and output, 271
input, 265
status, 271
output, 265
status, 271
Busy clock, 193
delay and reference count, 194
Busy manager, 193–194
function, 124
BYTEBUF_DATA structure, 208
Byte buffer
data, 207, 208
objects, 207–209
concepts, 207
creating, 208
notification of observers, 209
resetting, 208–209
BYTEBUF_NEW_ONLY structure, 208
Byte position
file handle, 61
setting current, 135–136
Carrier state, modem, 284
CHAR8, 111
CHAR16, 111
CHAR, 111
Character box height preference, 366
Character box width preference, 366
Characters
16-bit, 110–114
attributes of, 8–9, 16–17
deleting, 11
font masks, 17
getting range of, 14
getting single, 14
inserting, 12
reading, in text data objects, 14
scanning ranges of, 15–16
types of, 111
C language, for defining resources, 355–357
Classes
file system, 62
subclassing, 67
installation, 379–380
mask, 415–416
open service object, 441
service, 255–264, 439–441
installation, 441
service manager, 440
Text subsystem, 7
that respond to search messages, 198
writing, that can be searched, 196
see also specific classes
Client-defined
attributes, 54, 77–78
transfer protocols, 170
Closing
address book, 326
files, 46, 74–75
sample code, 46
parallel port, 277
serial port, 268
service, 262–263
socket handle, 299
see also Opening
clsABMgr, messages, 325
clsAddrBookApplication, 320
clsAddressBook, 319
clsAddressBookApplication, 317
messages, 324–325
clsApp, 148, 438
search and replace and, 195, 196
clsAppDir, 67
clsAppInstallMgr, 405, 415
instance of, 406
messages, 415
clsAppMask, 416
clsAppMonitor, 379
clsAuxNotebookMgr, 379, 380
messages, 423
stationary menu, 426–427
clsBrowser, 137
creating instance of, 137
function, 137
messages, 138–140
for changing displayed information, 142
for changing sort order, 142
class, 138
instance, 138–140
menu, 145
notification, 145
user columns and, 145–146
using, 138–144
clsByteBuf, 124, 207
messages, 208
notification of observers and, 209
resetting byte buffer object and, 209
clsCodeInstallMgr, 379, 414
in installing applications or services, 415
messages, 415
clsCommandBar, 138
clsDirHandle, 58
messages, 64
clsEmbeddedWin, 157
handles selection messages, 161
clsExport, 138
messages, 133
clsFileHandle, 58
clsStream and, 134
in creating resource file handle, 348
messages, 64
clsFileSystem, 58, 124
messages, 62–63
clsFontInstallMgr, 378, 380, 405
functions, 417
instance of, 406
messages, 417
clsGWin
Quick Help, 181–182
messages and, 187
clsHWXInstallMgr, 378
clsImport, 150
messages, 150
clsINBXService, 305
default behaviors, 310
existing Out box services and, 311
messages, 310, 314
Out box document response to, 310
writing own Out box service and, 310
clsOpenServiceObject, 441, 450, 452
clsService and, 471
function, 471
subclassing, 471–472
clsParallelPort, 275
messages, 276
structures, 275
clsPreferences, 361
clsQuickHelp, 182
messages, 187
using, 187–188
clsResFile, 67
messages, 347–348
using, 347–354
clsResList, 345
in creating resource list, 346
clsSelection, 155
instance, 155
message categories, 156
messages, 157–158
from clients to theSelectionManager, 158–159
clsSendableService, 319, 331
clsService, 255
handling msgNewDefaults, 457
handling of msgNew, 457
messages
change ownership protocol, 467–469
information, 459
notification, 461–462
responsibility, 469–470
msgSvcClassLoadInstance and, 459
msgSvcOpenRequested and, 463
object-oriented architecture and, 449
service instances and, 439–440
service manager messages and, 459
clsServiceInstallMgr, 405, 416, 441
instance of, 406
messages, 416
clsServiceMgr, 258, 260
concepts, 440
messages, 260
opening and closing service and, 262
clsSio, 124, 280
clsStream, 133
function, 124
messages, 133
services and, 470
writing agents and, 354
stream transfers and, 168
subclassing, 133
clsString, 124, 280
messages, 212
object, 211
clsSvcManager, 459
clsSystem, 429
messages, 431
paths for file system constants, 430
clsTable, 213
creating table object and, 216
data files and, 214
function, 124
library support routines, 213
messages, 217–218
information, 226
requesting new position from, 215
semaphore and, 217
clsTestOpenObject, 450
ClsTestOpenObjectInit routine, 451, 452
ClsTestServiceInit routine, 451
clsTestService method table, 457
dsTestSvc, 449

dsText, 198

embedded objects and, 20
messages, 12–13
for changing attributes, 7
observer, 21
Text subsystem and, 3

dsTexteditApp, 180

dsTextIP, 33

messages, 33

dstextView, 9
creating object of, 35
defined, 9
in insertion pad creation, 26
messages, 23–24
msgNewDefaults for, 24–25

clsTimer, 104

dstransport, 295

messages, 297

NBP and ZIP, 301
transport protocols and, 297
using, 297–301
for AppleTalk, 301–305

clsUndo messages, 202

using, 202–206

csXfer, 165, 166

in establishing transfer type, 171
functions, 170
stream transfer protocols
and, 168–169
transfer types, 166

csXferList, 171

csXferStream, 168–169

messages, 171

Cmpstext.H, 114

Code, installation manager, 414–416

Columns, table, 213
data types, 219
descriptors, 213
contents, 214
finding number, 226
getting description of, 227
getting number of, 227
see also Tables

Command mode, modem, 285

Communication
asynchronous, 297
connectionless, 296
connection-oriented, 296
conventions, 297
targeting, devices, 307

Compiling, resources, 359–360

Components, 438

ComposeText functions, 114

Concurrency considerations, 66–67

file location, 67

protecting file data, 66–67

serial I/O, 266
volume protection, 67

Configuration
data modem, 283–287
parallel port, 277
serial port, 268–271
data modem, 280

Connecting, volumes, 50

Connection
service, 258, 446–447
socket, 296

Connectionless communication, 296
see also Datagram, delivery

Connection-oriented communication, 296

Connections notebook, 250
installing services through, 256
quick installation and, 397
socket instance and, 298
see also Auxiliary notebooks

Connectivity, 245–250
adding network protocols and, 251
additional information on, 242
crcomputer, 244–245
facilities, 250
introduction to, 241
MIL services, 245–246
other services and, 246–249
principles of, 243
remote interfaces and, 251–253
service manager and, 250
services and interfaces, 249–250

Copy gesture, 165–166

Copying
beginning, operation, 160–161
documents to Auxiliary notebook,
425–426

nodes, 80–81
see also Moving

Counting
changes, 37
list items, 130
createInitial style bit, 406

Creating
address descriptors, 331–332
address windows, 333–334
application distribution volume, 391

Auxiliary notebook
documents, 425
sections, 424
browser, 138
object, 140

byte buffer object, 208
csModem object, 282–283
directories, 69–74

browser and, 141
indexes, 80

directory handles, 58, 60, 71

DLLMain, 450
document with browser, 141

file handles, 71–73
files, 69–74
handles, 69–70
help text, 180
installable-item managers, 410
lists, 129
mark, 196
nodes, 43
Quick Install disk, 397
receiver’s stream, 176–177

resource file handle, 348–349

resource lists, 346

sender’s stream, 177

service instances, 442, 454–455

stream objects, 134

string object, 212
table object, 220
tables, 221
temporary files and, 65
text data object, 7, 13
text insertion pads, 33
text views, 9, 24–26

see also Deleting; Removing

C run-time library, 109–114

16-bit character support, 110–114

ANSI standard C routines, 109

files, 109
time and date preferences, 110

ctype.h, 112

Current selection, 155
getting, text view, 30–31
Writing Paper application, 31

Data

application global, 394–395

buffered, 265
reading, 45
resource, 337, 342, 353

C language definition, 355
writing and updating, 349–350

sending and receiving via modem, 289

service, storage, 455
table
files, 214
getting, 223–224
setting, 223
transaction, 201
transfer type, 166
tags, 166–167
writing, 44–45
see also Data modem, interface; Text data object

Database

installed item, 406
using tables in, 217
Datagram delivery, 296
transaction services, 296
types of, 296
receiving, 300
sending, 299

Data modem
AT command set, 290–293
characteristics, 284
configuring, 283–287
auto-answer mode, 284
carrier state, 284
command and data modes, 285
dial type, 284
duplex mode, 286
MNP mode, 286
sending own AT commands, 286–287
speaker, 284–285
connection types, 285
direct communication with, 290–293
establishing connection with, 287
interface, 279–293
classModemAPI, 281
classModem messages, 281–290
concepts, 279–280
configuration, 280
direct communication with, 290–293
MNP data communication and, 289–290
reset settings, 283
sending and receiving data with, 289
waiting for connection with, 289

Data mode, modem, 285
Datasheets, 95
Date format preference, 367

Date/time services
alarm services, 103
current time, 104
object-oriented timer interface, 104–105
timer routines, 103
see also Time

Default attributes
changing, 8
text data objects, 7

Deinstalling, services, 256, 456

Deleting
address book entry, 328
Auxiliary notebook section/document, 426
directories, 75
with browser, 141
forcing, 75–76
files, 75
with browser, 141
forcing, 75–76
many characters, 11
resources, 352–353
table rows, 224
see also Removing

Destroying
lists, 131
text data objects, 7

Device
applications, drivers and, 246
connectivity strategy, 244
drivers. see MIL services
interface and, 249
object UID, 261
option sheet, 247
SCSI, 247
services and, 306–307
installing, 307
targeting communications, 307

Dial string modifiers, 287–288
defined, 287
function, 287–288

Dial type, modem, 284

Directories, 43
application, 391–395
concepts, 382
creating, 69–74
browser and, 141
defined, 52, 54
deleting, 75
with browser, 141
directory entries and, 54
forcing deletion of, 75–76
Help Notebook, 179–180
item, 376
locating, 56
mode flags, 71
names of, 70–71
renaming, 141
root, 52
handle, 60–61
service, 395–396
target, 59
changing, 86
see also PenPoint directory; specific directories

Directory entries, 54
reading, 87–89
all, 88
sorting, 88–89

Directory handles, 43, 59–61
creating, 58, 60, 71
directory nodes and, 59
instance messages, 64
locators and, 59
observing, 60
RAM, 61
target directory and, 59
using, 60
volume root, 60–61
well-known, 60
working, 61
see also Directories

Directory index, 56, 67
creating and using, 80–81

DIRENT.H, 114

Disconnecting, volumes, 50
Disk formats, 51

Display seconds preference, 367

Distributing, service, 473–474
Distribution disks, services on, 444
Distribution volumes, organization, 390–398
application directories, 391–395
multiple applications and volumes, 398

PENPOINT.DIR files, 390
quick installation, 397
service directories, 396–396
STAMP utility, 390–391
upgrading, 398
see also Volumes

DLC files, 401
services and, 444
see also DLL files

DLL directory, 385

DLL files, 399
creation options, 402–403
DLL files and, 401
DLLMain() routine and, 402–403
issues, 400
MAKE files and, 403–404
operating system, 403
references to, 399
service, 444
sharing, 401–402
unloading, 400
versions and, 402
see also DLC files
dll-id name, 400
application monitor and, 401–402
operating system DLL files, 403
sharing DLL files and, 401–402

DLLMain(), 379
creating, 450
DLL processes and, 402–403
owning task and, 455
service instance creation and, 442

DLLs, 399–404
identifying, 400–401
processes, 402–403
table class component, 213
see also DLL files

DLL_TYPE_DISTRIBUTED, 403

DOC directory, 388
contents, 388

Document accessory, 377
creating with browser, 141
In box, 313
Out box, 309–310
stationary, 377
wrapper, 310
Document menu (standard application), 331
DOS file system, 439
DotMatrix service, 439
DTR (data-terminal-ready) lines, 270
Duplex mode, modem, 286
DVHSPKT, 274
Dynamic Link Libraries. see DLLs
Dynamic ports, 296
Dynamic resource IDs, 343-344
defined, 343
DynResIdO macro, 345
Embedded objects
in views, 26-27
window, 161
Embedding objects, 20
Enumerating
list items, 130-131
resources, 351-352
Events
serial, 266
break status, 273
detecting, 272-273
mask indicators, 272
polling for, 273
Exactly-once, datagram delivery, 296
Exclusive access services, 445-446
defined, 445
Explicit locators, 56
EXPORT_DOC structure, 154
EXPORT_FORMAT structure, 153, 154
Exporting files, 147
application responsibilities, 150
clsExport messages, 152-154
msgExport, 154
msgExportGetFormats, 152-153
msgExportName, 153
export dialog, 149
export overview, 148-149
file export mechanism, 147
how export happens, 152
see also Importing files
EXPORT_LIST structure, 153
Facilities, for networking and connectivity, 250
FCNTL.H, 114
File attribute arguments, 76
File handles, 43, 61-62
access intentions, 62
byte position and, 61
creating, 71-73
file access control and, 62
instance messages, 64
locators and, 59
 translating file pointer into, 66
File import and export, 147-154
clsExport messages and, 152-154
clsImport messages and, 150-152
functions, 124
interface, 252-253
mechanisms, 147
see also File System
File pointer, 66
Files, 43
access control, 62
closing, 46, 74-75
with stdio, 66
creating, 69-74
deleted, 55
deleting, 75
with browser, 141
DLC, 401
DLL, 399
creation options, 402-403
DLC files and, 401
DLLMainO routine and, 402-403
issues, 400
MAKE files and, 403-404
operating system, 403
references to, 403
sharing, 401-402
unloading, 400
versions and, 402
forcing deletion of, 75-76
locations of, 67
MAKE, 403-404
memory-mapped, 55
mode flags, 72
names, checking, 70-71
opening, 46
with stdio, 66
organization of, 381-398
position and size of, 84-85
protecting data, 66-67
reading, 83
renaming types of, 151
renaming, 141
resource, 337, 342
compacting and flushing, 353
definition, 355
organization, 355-356
viewing contents of, 359, 360
table data, 214
temporary, using handles with, 65
writing, 83
see also File Handles
File system
accessing, 57-68
with stdio, 65-66
attributes, 54-55, 77
auxiliary notebooks and, 422
browser, 124
classes, 58
subclasing, 67
common, operations, 47
concurrency considerations and, 66-67
connectivity and, 244
developer’s quick start, 44-46
directories, 54
files, 55
functions, 43-44
handles, 43, 57-62
directory, 59-61
file, 61-62
functions, 57
locators and, 58-59
using with temporary files, 65
interface, 252
locators, 55-56
making, changes, 141-142
messages, 62-64
nodes, 52-54
accessing, 57
names, 53-54
service instance, 443
Notebook use of, 68
overview, 43-44
paths, 430-431
constants, 430
Penpoint comparison with other systems, 46-47
PENPOINT.DIR file and, 68
performing, operations, 43
principles and organization, 49-56
programmic interface provisions, 57
services and, 443-445
structure, 43
using, 69-91
changing target directory, 86
closing files, 74-75
comparing handles, 86-87
copying and moving nodes, 80-81
creating directories and files, 69-74
deleting files and directories, 75
ejecting floppies, 91
file position and size, 84-85
flushing buffers, 85
forcing deletion of files/directories, 75-76
getting and setting attributes, 76-80
getting path handle, 85-86
getting volume information, 90-91
handle mode flags, 87
making native node, 89-90
node existence determination, 83
observing changes, 89
reading and writing files, 83
reading directory entries, 87-89
renaming nodes, 83
setting/changing volume name, 91
INDEX

traversing nodes, 81–82
volume specific messages, 91
volumes, 49–52
see also File import and export
FIM_GET_INSTALLED_ID_LIST structure, 418–419
FIM_GET_NAME_FROM_ID structure, 418
FIM_GET_SET_ID structure, 418
FIXED numbers, 115
Fixed-point calculations, 115
functions, 116–117
numbers, 115
Flags
directory mode, 71
existence, 70
file mode, 72
FS_SEEK, 84
handle mode, 87
node attributes, 79–80
transaction item, 203–204
TV_STYLE, 25–26
Floating allowed preference, 365
Floppies, ejecting, 91
Flow control, 265–266
characters, 270
protocols, 265
specifying, 269
using, 272
Flushing
buffered output, 353
buffers, 85
input and output, 271
resource files, 353
streams, 136
FONT directory, 384, 386
Font installation manager, 416–419
Fonts
Gesture, 188–190
handle, 416
finding, 418
identification, 416–417
IDs, getting and setting, 418
installing, 378
list of installed, 418–419
name, 418
system, 363
user, 363
FS_CHANGE_INFO structure, 89
fsDirNewDefaultMode, 71
FS_DIR_NEW_MODE, 71
FS_EXIST constants, 70
fsExistGenUnique flag, 70
fsFileNewDefaultMode, 73
FS_FILE_NEW_MODE constants, 72
FS_FLAT_LOCATOR structure, 140, 141
FS_FORCE_DELETE structure, 75
FS_GET_PATH structure, 85–86
FS_GET_SET_ATTR structure, 76–77
getting values and, 78
setting values and, 79
FS_GET_VOL_METRICS structure, 90
FS_H, 69
attribute label macros, 77–78
FS_LOCATOR structure, 86
FS_MAKE_NATIVE structure, 90
FS_MOVE_COPY structure, 80
FS_NAMEValid() function, 53, 70
FS_NEW structure, 69–70, 71
FS_NODE_EXISTS structure, 83
FS_NODE_FLAGS structure, 79
fsNoExistCreateUnique flag, 70
FS_READ_DIR structure, 88
FS_SEEK structure, 84
flags, 84
FS_SET_HANDLE_MODE structure, 87
fsSharedMemoryMap, 73
fsTempFile, 65
FS_TRAVERSE structure, 81
FS_VOL_METRICS structure, 61
FxMakeFixedO routine, 115
GDIR command, 390
Gesture, font, 188–190
Gestures
adding, to help text, 190
adding, to Quick Help strings, 191
GO Address book, 323
Gesture timeout preference, 364
GO Address book, 323–324
gestures, 323
illustrated, 324
loading, 323
using, 323
see also Address book
Graphics subsystem, 5
Handle mode flags, 87
Handles
comparing, 86–87
creating, 69–70
file system, 57–62
directory handles, 69–61
file handles, 61–62
locators and, 58–59
using with temporary files, 65
finding, 263–264
freeing, 74–75
getting path of, 85
objects, 57
creating, 58
on parallel port, 251
on serial port, 251
parallel port, 276–277
serial, 268
data modem, 279–280
Hand preference, 363
Handwriting timeout preference, 364
Hardware RTS/CTS flow control, 265, 266
see also Flow control
Header files, 95
Heaps, 101–102
derived, 101
management, 101
size and characteristics of, 102
Help, 179–191
advanced topics, 187–191
concepts, 179–182
directory, 393–394
Gesture font and, 188–190
Help Notebook, 179–180
icon, 179
Quick Help, 181–182
resources, 183–187
templates, 387
text, 180
adding gestures to, 190
creating, 180
HELP directory, 387, 393–394
Help Notebook, 179–180, 393
contents, 394
creating help text and, 180
defined, 179
directories, 179–180
see also Auxiliary notebooks
High-speed packet I/O interface, 252, 273–274
notes, 274
on serial lines, 273
parallel cable connection
detection, 274
protocol variations, 274
HWXPROT directory, 384, 386
Identifying, DLLs, 400–401
IM_CURRENT_NOTIFY structure, 408
IM_DEINSTALL structure, 412
IM_DUP structure, 412
IM_GET_STATE structure, 413
IM_INSTALL structure, 411
IM_INUSE_NOTIFY structure, 408
IM_MODIFIED_NOTIFY structure, 408
IM_NEW structure, 410
IM_NOTIFY structure, 408
Implicit locators, 56
IMPORT_DOC structure, 151–152
Importing files, 147
application responsibilities, 150
msgImport messages, 150–152
msgImportQuery, 150–151
file import mechanism, 147
overview, 148
TOC browser and, 148
see also Exporting files

IMPORT_QUERY structure, 150–151

1M_SET_NAME structure, 412

In box, 305
concepts, 312–313
connectivity and, 244
documents, 313
general device concepts, 306–308
introduction, 305–306
installing, 307
messages, 313–316
passive, 312–313
sections, 306–307
see also Out box

Inbox Notebook. see Auxiliary notebooks
Index
directory, 56, 67
creating and using, 80
list object, 127
resource ID, 344
text, 27–29

Initialization routines, 451
class, 452
service, 451–452
InitService, 379, 452–453
call for template service, 453

INIT subdirectory, 387

Input buffer, 265
flushing, 271
status, 271
see also Buffers

Input line status, 270
Input pad style preference, 366
Input subsystem, 27–29

Inserting
concept, 12
text view in scroll window, 30

Insertion pads
text, 9
creating, 33
destroying, 33
embedding objects and, 26
messages, 33
using, 33
Inside AppleTalk, 301

Installable applications, 386–387
Installable entities, 386
Installable items, 411–412
altering, attributes, 412
changing, name, 412
deleting, 412
duplicating, 412
finding, 413
getting and setting current, 412
getting attributes of, 413
getting information about, 413
getting list of, 413
installing, 411
manager, 410–411
size of, 413

Installable services, 387
Installation
classes, 379–380
initiation, 376
process, 376–377
service, 378–379, 450–456

Installation API’s
classes, 375–380
service, 378–379, 450–456

Installation managers, 375–376,
379–380, 405–419
advanced clsInstallMgr topics and, 414
application, 415–416
code, 414–416
time, 416–419
installer concepts and, 405–407
observing, 407–409
service, 416
using clsInstallMgr messages and, 409–413

Installers, 405
classes, 405–407
defined, 406

Installing
applications, 377–378, 415
devices, 307
fonts and handwriting prototypes, 378
service class, 441–442
services, 307, 415
see also Adding

INST directory, 396
service directory and, 444, 445

Interfaces
connectivity and, 244
data modem, 279–293
devices and, 249
file import/export, 252–253
file system, 252
high-speed packet I/O, 252
modem, 253
networking, 253
parallel I/O, 251–252, 275–278
serial I/O, 251, 265–274

services and, 249–250
SoftTalk, 250
stream, 246

Intertask communication, 100–101
messages, 100–101
semaphores, 101

Intertask messages, 100–101
Class Manager messages and, 100
modes, 100
processing order, 100

INTL.H file, 111
Item directory, 376
Items
controlling, 406–407
installable, 411–412
altering, attributes, 412
changing, name, 412
deleting, 412
duplicating, 412
finding, 413
getting and setting current, 412
getting attributes of, 413
getting information about, 413
getting list of, 413
installing, 411
managers, 410–411
size of, 413

installed, database, 406
installing, 376–377
list
adding, 129
counting, 130
enumerating, 130–131
getting, 129
removing, 130
removing all, 130
replacing, 130
NotePaper data, 234–235
transaction data, 201
contents, 201
flags, 203–204

Kernel, 97
functions, 105–107
date and timer routines, 106
debugger entry routines, 106
display/screen device routines, 107
heap routines, 107
inter-task communications routines, 106
keyboard routines, 107
memory information routines, 106
miscellaneous routines, 107
task manager routines, 105–106
tone routines, 107
layer, 98
overview, 97–107
semaphores and, 101
services of, 97
summary, 105–107
task scheduler and, 99

LaserJet, 247
Line control, 269
Line height preference, 366
Link protocols, 253
List
accessing end of, 130
class, 127–131
concepts, 127
creating, 129
defined, 127
destroying, 131
items
adding, 129–130
counting, 130
enumerating, 130–131
getting, 129–130
removing, 129–130
removing all, 130
replacing, 129–130
object index, 127
positioning within, 129
resource, 337, 345–346
using, messages, 128
LIST_ENTRY structure, 129
LIST_ENUM structure, 130
LIST_FREE structure, 131
LIST_NEW structure, 129
Loading,
GO Address book, 323
Local area network (LAN), 295
Local attributes
changing, 8
text data objects, 7
Local disk volumes, 51
LocalTalk, 253
Locators, 55–56
explicit, 56
handles and, 58–59
implicit, 56

Machine Interface Layer (MIL), 98
see also MIL services
Macintosh file system, 439
MakeDynUUID, 80
MAKE files, 403–404
MakeTag() macro, 167
MakeWknResId() macro, 343
Mapping, file to memory, 73–74
Mark, creating, 196
MarkHandlerForClass() function, 197
Mask
class, 415–416
in node attribute flags, 79
text attribute messages, 16
Math run-time library, 115–117
programmatic interface, 115–117
Measurement, units of, 10
Memory
freeing, 88, 95
management, 101–103
80386 protected mode, 102
heaps, 101–102
privilege levels, 103
rings, 103
map size, 73
RAM, 52
Memory-mapped files, 55
function, 73
life cycle, 73–74
sharing, 73
Memory-resident volumes, 52
Menu bar, browser, 138
Messages
auxiliary notebook manager, 423
generalized, 423–424
specialized, 424–426
boot progress, 431–432
clsABMgr, 325
clsAddressBookApplication, 324–325
clsAppInstallMgr, 415
clsBrowser, 138–140
for displayed information, 142
menu, 145
notification, 145
for sort order, 142
clsByteBuf, 208
clsCodelInstallMgr, 415
clsDirHandle, 64
clsExport, 152
clsFileHandle, 64
clsFileSystem, 62–63
clsFontInstallMgr, 417
clsImport, 150
clsINBXService, 315
clsInstallMgr, 405
class, 409
instance, 409–410
notification, 408
subclass, 410
using, 409–413
clsObXService, 316
clsList, 128
functions, 127
clsMILAsyncSIODevice, 267
clsModem, 281–282
clsNotePaper, 231
clsNPData, 233–234
clsNPItem, 234–235
clsOBXService, 310, 314
clsParallelPort, 276
clsQuickHelp, 187
using, 187–188
clsResFile, 347–348
clsSelection, 157–158
clsService
change ownership protocol, 467–469
information messages, 459
notification messages, 461–462
responsibility, 469–470
clsServiceInstallMgr, 416
clsServiceMgr, 260
clsStream, 133
clsString, 212
clsSystem, 431
clsTable, 217–218
information, 226
clsTextIP, 33
clsTransport, 297
NBP and ZIP, 301
clsUndo, 202
clsXHRStream, 171
connection status, 247
file system, 62–64
intertask, 100–101
Out box
protocol, 308–309
response to, 310
search and replace, 198
classes that respond to, 198
to selection owners, 159–161
sendable services, 333–334
sent by service managers, 459–470
sent to open services, 470
sent to service class, 456–459
system directory, 432
text data, 12–13
observer, 21
text insertion pad, 33
text view, 23–24
theBusyManager, 193
to theSelectionManager, 161–162
theTimer, 104
volume specific, 91
see also specific messages
Methods, handling search and replace
functions, 197
Metrics
ASCII, transfer, 175
browser, 143–144
embedded window object, 161
NotePaper, 230
serial port settings and, 270–271
text, 14
transaction, 205
volume, 49–50
Microcom Network Protocol
(MNP), 286
INDEX

MIL services, 98, 439
binding, 247
connection management, 247–248
connection status messages, 247
connectivity, 245–246
other services and, 246–249
functions, 245
ports and, 246
programmatic interface, 246
stream interface and, 246
see also Parallel port; Serial port

MILSVC service, 487–506
defined, 475
METHOD.TBL, 487–489
MILSVCC.C, 489–499
MILSVCCO.C, 500–506
MILSVCCO.H, 499–500
MiniText, 394
file import/export, 253
MISC directory, 387, 394–395
data examples, 395
service directory and, 387, 444, 445
Miscellaneous application files, 387
MNP mode, modem, 286
data communication, 289–290
Modem
interface, 253
service, 439
see also Data modem
MODEM_AUTO_ANSWER_SET structure, 284
MODEM_DIAL structure, 287
MODEM_MNP_BREAK_TYPE_SET structure, 290
MODEM_MNP_FLOW_CONTROL_SET structure, 290
MODEM_MNP_MODE_SET structure, 286
MODEM_NEW_ONLY structure, 283
MODEM_SEND_COMMAND structure, 287
MODEM_SPEAKER_STATE_SET structure, 285
Modifying, attributes, 19–20
Move gesture, 165–166
Moving
beginning, operation, 160–161
documents to Auxiliary notebooks, 425–426
nodes, 80–81
see also Copying

MS-DOS
disk drive class, 247
FAT disk format, 51
creating PENPOINT.DIR and, 68
volume name, 51
msgABMgrActivate, 329
status values, 329–330
msgABMgrChanged, 330
msgABMgrClose, 326
msgABMgrDeactivate, 330
msgABMgrOpen, 326
msgABMgrRegister, 329
msgABMgrUnregister, 329
msgAddObserver, 49
observing installation managers and, 407
observing tables and, 220
system preferences and, 368
msgAddrBookAdd, 328
msgAddrBookDelete, 328
msgAddrBookGet, 327–328
msgAddrBookSearch, 326, 327
msgAddrBookSet, 328
msgANMAddToStationaryMenu, 426
msgANMCopyInDoc, 425
msgANMCreateDoc, 425
msgANMCreateSect, 424
msgANMDel ete, 426
msgANMDel eteAll, 426
msgANMGetNotebookPath, 423
msgANMInsertDoc, 425
msgANMO penNotebook, 423
msgANMRemoveFromStationaryMenu, 427
msgANMSys temInited, 423
msgAppInit, 261, 378
msgAppMgrActivate, 148, 152
msgAppMgrGetMetrics, 394
msgAppRestore, 261
msgAppSearch, 195, 196
msgAppUndo, 206
msgATRespPktSize, 302
msgBootStateChanged, 431
msgBrowserBookmark, 145
msgBrowserBy messages, 142
msgBrowserCollapse, 143
msgBrowserCreateDir, 141
msgBrowserCreateDoc, 141
msgBrowserDelete, 141
msgBrowserExpand, 143
msgBrowserGetBrowserWin, 137
getting internal display window and, 144–145
msgBrowserGetMetrics, 144
msgBrowserGetSelection, 143
msgBrowserGoto, 144
msgBrowserReadState, 143
msgBrowserRefresh, 142
msgBrowserRename, 141
msgBrowserRename, 141
msgBrowserSelectionDir, 140
msgBrowserSelectionName, 141
msgBrowserSelectionOff, 145
msgBrowserSelectionOn, 145
msgBrowserSelectionPath, 145
msgBrowserSelectionUUID, 141
msgBrowserSetClient, 144
msgBrowserSetMetrics, 143–144
msgBrowserSetSaveFile, 143
msgBrowserSetSelection, 141
msgBrowserShow messages, 142
setting metrics and, 144
msgBrowserUserColumnQueryState, 146
msgBrowserWriteState, 143
msgBusyDisplay, 193
in busy clock delay and reference count, 194
msgBusySetXY, 193
msgByteBufChanged, 209
msgByteBufGetBuf, 208
msgByteBufSetBuf, 208
msgCIMLoad, 415
msgDestroy
for closing memory-mapped file, 74
in closing multi-user service, 446
in destroying insertion pad object, 33
in destroying lists, 131
in freeing handle, 74–75
in freeing table, 228
msgExport, 149
responding to, 154
msgExportGetFormats, 148
responding to, 152–153
msgExportName, 153–154
msgFIMFindId, 418
msgFIMGetId, 418
msgFIMGetInstalledIdList, 417, 418
msgFIMGetNameFromId, 417, 418
msgFIMSetId, 418
msgFree, 67
in closing file, 46
in forced deletion, 75–76
in freeing stream, 177
in freeing table, 228
handling, 458
service instance and, 466
in unbinding application from service, 261
msgFSChanged, 89
msgFSCopy, 80
msgFSErase, 65, 75
msgFSEjectMedia, 91
msgFSFlush, 74, 353
in flushing buffers, 85
msgFSForceDelete, 66, 67, 75
msgFSGetAttr, 59
  for attribute manipulation, 76
  for getting attribute value length, 79
  getting values and, 78
msgFSGetHandleMode, 71, 73
  handle mode flags and, 87
msgFSGetInstalledVolumes, 60
  for list of volume objects, 90
msgFSGetPath, 59
msgFSGetSize, 85
msgFSGetVolMetrics, 49, 60
  call example, 91
  duplicate volume names and, 50
  for getting volume information, 90
msgFSMakeNative, 89–90
  call example, 90
msgFSMemoryMap, 73
msgFSMemoryMapFree, 74
msgFSMemoryMapGetSize, 74
msgFSMemoryMapSetSize, 73, 74
msgFSMove, 80
msgFSNodeExists, 83
msgFSReadDir, 87–88
msgFSReadDirFull, 88
msgFSRename, 83
msgFSSeek, 59
  in getting file position, 84–85
msgFSSetAttr, 78
  for attribute manipulation, 76
  for creating directory index, 80
  for renaming nodes, 83
  setting values and, 79
msgFSSetHandleMode, 65, 71, 73
  handle mode flags and, 87
msgFSsetSize, 85
msgFSSetTarget, 59, 67, 76
  call example, 86
  in changing target directory, 86
  directory position and, 88
msgFSGetVolName, 91
msgGetInstalledVolumes, 49
msgGetPath, 85
  example of, 86
msgIMCurrentChanged, 408
msgIMDelete, 412
msgIMDup, 412
msgIMFind, 261, 262, 413
  in locating parallel port, 276
  in locating serial port, 268
  in locating service, 442
  in locating socket service handle, 298
msgIMGetCurrent, 412
msgIMGetDir, 261, 413
msgIMGetName, 258, 262, 417
msgIMGetNotify, 407
msgIMGetSema, 407, 414
msgIMGetSize, 413
msgIMGetState, 412, 413
msgIMSetAttr, 85
  for attribute manipulation, 76
  for creating directory index, 80
  for renaming nodes, 83
  setting values and, 79
msgIMSetHandleMode, 65, 71, 73
  handle mode flags and, 87
msgIMSetCurrent, 412
msgIMSetModified, 407
msgIMSetNotify, 407
msgIMSetStyle, 411
msgINBXSvcPollDocuments, 313
msgListAddItem, 173
msgListAddItemAt, 129
msgListEnumItems, 130
msgListFindItem, 129
msgListFree, 131
msgListGetItem, 129
msgListNumItems, 127
  in counting items, 130
msgListRemoveItemAt, 129–130
msgListRemoveItems, 130
msgListReplaceItem, 129–130
msgMarkCreateToken, 196
msgMarkPositionAtSelection, 196
msgMarkSelectTarget, 198
msgMarkShowTarget, 198
msgModemAutoAnswerSet, 284, 289
msgModemCarrierStateSet, 285
msgModemCommandModeSet, 285, 287
msgModemConnected, 289
msgModemDial, 287
msgModemDiafTypeSet, 284
msgModemDisconnected, 289
msgModemDuplexSet, 286
msgModemHangup, 287
msgModemMNPBreakTypeSet, 289
msgModemMNPCompressionSet, 289
msgModemMNPFlowControlSet, 290
msgModemMNPModeSet, 286, 289
msgModemOffHook, 289
msgModemOnline, 285, 287, 289
msgModemReset, 283
msgModemRingDetected, 289
msgModemSendCommand, 286–287
msgModemSpeakerControlSet, 285
msgNPBCancel, 303
msgNBPConfirm, 304
msgNBPFind, 303
msgNBPRemove, 303
msgNew
  clsService handling of, 457
  creating browser and, 140
  creating byte buffer object and, 208
  creating clsModem object and, 282–283
  creating directory handle and, 58, 71
  creating directory index and, 80
  creating file handle and, 72
  creating handles and, 69
  creating installable-item manager and, 410
  creating lists and, 129
  creating resource file handle and, 348
  creating resource list and, 346
  creating service instances and, 442, 454
  creating stream object and, 134
  creating string object and, 212
  creating table object and, 220
  creating temporary file and, 65
  creating text data objects and, 7, 13
  creating text insertion pad and, 33
  creating text view object and, 5, 24
  limiting file access with, 67
  in opening file, 46
  opening multi-user service and, 446
  in specifying serial port handle, 281
  table data files and, 214
  temporary file flag with, 75
msgNewDefaults
  clsService handling of, 457
  for clsTextView, 24–25
  creating browser and, 140
  creating byte buffer object and, 208
  creating clsModem object and, 282–283
  creating directory handles and, 71
  creating handles and, 69–70
  creating installable-item manager and, 410
  creating resource file handle and, 348
creating resource list and, 346
creating service instances and, 454
creating stream object and, 134
creating string object and, 212
creating table object and, 220
creating text data objects and, 7, 13
creating text insertion pad and, 33
in creating text view object and, 5, 24
in opening file, 46
msgNotUnderstood, 159
msgObjectNew, 456
msgOBXDocOutputDone, 309
msgOBXSvcCopyInDoc, 309
msgOBXSvcLockDocument, 309, 310
msgOBXSvcMoveInDoc, 309
msgOBXSvcNextDocument, 309
msgOBXSvcOutputStart, 311
msgOBXSvcPollDocuments, 309
msgOBXSvcUnlockDocument, 309, 310
msgOptionAddCards, 318
msgOptionSheetAddCards, 330
msgOSOGetServiceInstance, 471
msgPPortAutoLineFeedOn/Off, 277
msgPPortCancelPrint, 278
msgPPortGetTimeDelays, 277
msgPPortInitialize, 277
msgPPortSetTimeDelays, 277, 278
msgPPortStatus, 278
msgPrefsPreferenceChanged, 368
msgQuickHelpOpen, 188
msgQuickHelpShow, 181
in displaying Quick Help text, 187
msgRemoveObserver, 220
msgResAgent, 353–354
msgResCompact, 352, 353
msgResDeleteResource, 352
msgResEnumResources, 351–352
msgResFlush, 353
msgResGetInfo, 349
msgResGetObject, 45
msgResNextDynResID, 344
msgResPutObject, 44, 207, 341
dynamic resource IDs and, 343–344
msgResReadData, 338, 349, 353
preferences resources and, 361
writing agents and, 354
msgResReadObject, 338, 350
msgResReadObjectWithFlags, 350–351
msgRestore, 45
message handler response to, 45
object resources and, 341, 342
msgResUpdateData, 349–350, 353
preferences resources and, 361
msgResWriteData, 349–350, 353
changing hand preference with, 363
changing screen orientation preference with, 363
changing scroll margin style preference with, 365
changing system and user fonts preference with, 363
preferences resources and, 361
writing agents and, 354
msgResWriteObject, 351
msgResWriteObjectWithFlags, 351
msgSave, 44–45
object resources and, 341
writing objects/data and, 44
msgSelBeginCopy, 159, 160
msgSelBeginMove, 159, 160
msgSelChangedOwners, 162, 163
restoring selection owners and, 162
msgSelDelete, 159
handling, 160
msgSelDemote, 159
handling, 160
preserving owner selection and, 162
msgSellsSelected, 159
msgSelOptions, 159
handling, 160
msgSelOptionTagOK, 160
msgSelOwner, 31
finding selection owners and, 161–162
selection transitions and, 157
msgSelPromote, 159
handling, 160
restoring selection owner and, 162
msgSelPromotedOwner, 163
msgSelSelect, 161
msgSelSetOwner, 157, 159, 162
clsEmbeddedWin and, 161
promoting/demoting and, 160
msgSelSetOwnerPreserve, 159, 162
clsEmbeddedWin and, 161
restoring selection owner and, 162
msgSelYield, 157, 159
handling, 160
restoring selection owner and, 162
msgSendServCreateAddrWin, 332, 333
msgSendServFillAddrWin, 332, 333–334
msgSendServGetAddrDesc, 332, 333
msgSendServGetAddrSummary, 332, 334
msgSetAttr, 66
msgSIMGetMetrics, 456
msgSioBaudSet, 269
data modem and, 280
msgSioBreakSend, 272
msgSioBreakStatus, 273
msgSioControlInStatus, 270
msgSioControlOutSet, 270
msgSioEventGet, 273
msgSioEventHappened, 266, 272
msgSioEventSet, 266, 272
msgSioEventStatus, 273
msgSioFlowControlCharSet, 270
msgSioFlowControlSet, 269
msgSioGetMetrics, 270–271
msgSioInputBufferFlush, 271
msgSioInputBufferStatus, 271
msgSioLineControlSet, 269
data modem and, 280
msgSioOutputBufferSet, 271
msgSioOutputBufferStatus, 271
msgSMAccess, 261
msgSMBind, 259, 261, 262
in binding parallel port, 276
in binding serial port, 268
in binding socket service handle, 298
data modem serial port, 280
service manager and, 462
msgSMChangeOwner, 467
msgSMClose, 263
in closing serial port, 268
in closing service instance, 443
in closing socket handle, 299
msgSMClosed, 464
msgSMConnectedChanged, 264
msgSMFindHandle, 263
msgSMOOpen, 261, 262, 263
data modem serial port, 280
in opening parallel port, 276
in opening serial port, 268
in opening service instance, 442
in opening socket service handle, 298
open service objects and, 471
service manager and, 463
msgSMOOpenDefaults, 262, 263, 463
socket service handle and, 298
msgSMOOpenList, 463
msgSMOwnerChanged, 467
msgSMQueryLock, 465
msgSMQueryUnlock, 465
msgSMSave, 466
msgSMSetOwner, 264, 442, 467
service instance owner and, 445
msgSMSetOwnerNoVeto, 264, 467
msgSMUnbind, 263
in unbinding from service instance, 443
msgSRGetChars, 197
msgSRNextChars, 197
msgSRPositionChars, 197
msgSRReplace, 198
msgStreamFlush, 136
msgStreamRead, 44, 45
  blocking protocol and, 168–169
  example of using, 83
  producer protocol and, 169
  in reading files, 83
  in reading streams, 134
  in reading with serial port, 271
  service instance and, 443
  to store state, 136
  stream transfers and, 168
msgStreamReadTimeOut, 134–135
  data modem and, 289
  reading with serial port and, 271
msgStreamSeek, 135
  passing back current position, 136
  setting current position, 136
msgStreamWrite, 44, 59
  blocking protocol and, 168–169
  example of using, 83
  producer protocol and, 169
  to save state, 136
  service instance and, 443
  stream transfers and, 168
  in writing files, 83
  in writing streams, 134
  in writing to parallel port, 278
  in writing with serial port, 271
msgStreamWriteTimeOut, 134–135
  data modem and, 289
  writing with serial port and, 271
msgStrObjChanged, 212
msgStrObjGetStr, 212
msgStrObjSetStr, 212
msgSvcBindRequested, 443
  handling, 463
msgSvcChangeOwnerRequested, 467
  handling, 468
msgSvcClassInitService, 452
msgSvcClassLoadInstance, 458–459
msgSvcClassTerminate, 456
  handling, 458
msgSvcClassTerminateOK, 456
  handling, 457–458
msgSvcClassTerminateVetoed, 456
  handling, 458
msgSvcClientDestroyedEarly, 466
msgSvcCloseRequested, 464–465
msgSvcDeinstallRequested, 456, 457
  msgSvcDeinstallVetoed and, 466
msgSvcDeinstallVetoed, 456, 458
  handling, 466
msgSvcDeinstallRequested and, 466
msgSvcGetMetrics, 459–460
  owned state information and, 469
  service response to, 460
msgSvcOpenDefaultsRequested, 463–464
msgSvcOpenRequested, 443
msgSvcOwnerAcquired, 467
  handling, 469
msgSvcOwnerAcquiredRequested, 468
msgSvcOwnerMessages, 264
msgSvcOwnerReleased, 467
  handling, 469
msgSvcOwnerReleaseRequested, 467
msgSvcQueryLockRequested, 465
msgSvcQueryUnlockRequested, 465–466
msgSvcSaveRequested, 466
msgSvcSetMetrics, 460–461
  saved state information and, 469
  service response to, 461
msgSvcUnbindRequested, 462
msgSysGetBootState, 431
msgSysGetLiveRoot, 432
msgSysGetRuntimeRoot, 432
msgTBLAddRow, 222
msgTBLBeginAccess, 214, 221
  observing tables and, 220
msgTBLColGetData, 223, 227
msgTBLColGetSetData, 223
msgTBLCompact, 224
msgTBLDeleteRow, 224
msgTBLEndAccess, 214, 228
msgTBLGetRowCount, 227
msgTBLGetState, 215, 227
msgTBLRowGetData, 223–224
msgTBLRowNumToRowPos, 226
msgTBLRowSetData, 223
msgTBLRowStateClear, 222
msgTBLRowStateRequest, 217, 222
msgTextAffected, 21
msgTextChangeAttrs, 19, 35
msgTextChangeCount, 37
msgTextClearAttrs, 19, 20
msgTextEmbedObject, 20
msgTextEnumEmbeddedObjects, 20
msgTextExtractObject, 20
msgTextGet, 14
msgTextGetAttrs, 18, 35
msgTextGetBuffer, 14
msgTextInitAttrs, 19
msgTextLength, 14
msgTextModified, 15
  counting changes and, 37
msgTextReplaced, 21
msgTextSpan, 15
msgTextViewAddIP, 26–27, 33
  circle-line gesture and, 26
  overriding behavior of, 27
msgTextViewCheck, 32
msgTextViewEmbed, 26
msgTextViewGetStyle, 32
msgTextViewResolveXY, 27–28
msgTextViewScroll, 29
msgTimerAlarmNotify, 104
msgTimerNotify, 104
msgTPBind, 301
msgTPSendFrom, 300
msgTPSendTo, 300, 299, 300
msgUndoAbort, 204–205
msgUndoAddltem, 203–204
  to aborting transaction, 204–205
msgUndoBegin, 202–203
msgUndoCurrent, 206
msgUndoEnd, 203, 204
msgUndoFreeItem, 206
msgUndoFreeItemData, 203
msgUndoGetMetrics, 205
msgUndoItem, 206
msgUndoLimit, 205
msgViewGetDataObject, 26
msgViewSetDataObject, 26
msgXferGet, 31
  in ASCII metrics transfers, 175
  in one-shot transfers, 167–168, 173
  in replying to one-shot transfers, 176
msgXferList, 171
  to list transfer types, 172–173
msgXferStreamConnect, 178
msgXferStreamFreed, 170, 177
msgXferStreamInit, 178
msgXferStreamSetAuxData, 170, 177
msgXferStreamWrite, 169
msgZIPGetMyZone, 304
msgZIPGetZoneList, 304
Multiple access services, 446
  defined, 445
Index

Name binding protocol (NBP), 302–303
Names
  AppleTalk protocol, 302–304
  looking, 303–304
  registering, 303
  removing, 303
  zone, 304
duplicate volume, 50
local disk volume, 51
memory-resident volume, 52
node, 53–54
remote volume, 51
NBP_CONFIRM structure, 304
NBP_LOOKUP structure, 303
NBP_REGISTER structure, 303
Networking
  facilities, 250
  interfaces, 253
Network protocols, adding, 251
Nodes, 43, 52–54
  accessing, 57
  attributes, 54–55
  behavior of, 67
  copying, 80–81
  creating, 43
  determining existence of, 83
  flags, 68
  locators and, 55–56
  making, native, 89–90
  moving, 80–81
  names of, 53
  paths, 55–56
  renaming, 83
  service state, 396, 444, 445
  traversing, 81–82
  call back routine, 82
  order of traversal and, 82
  quicksort routine, 82
  types of, 52
Notebook
  file system usage, 68
  organization, 44
  see also specific types of notebooks
NotePaper component, 229–235
  clsNotePaper view, 229
data, 232–234
data items, 234–235
defined, 229
messages, 231–232
metrics, 230
NOTEPAPER_METRICS data structure, 230
Notification
  messages, clsInstallMgr, 408
  observer, 163
  byte buffer object, 209
  string object, 212
  preference change, 368
  receiving connection state, 264
NULL-terminated strings, 355
resource managers and, 353
ObjectCall()
  address book and, 320
  installable manager access and, 407
  theSelectionManager and, 156
Object-oriented architecture, 449
ObjectPost(), 187
Object resources, 337, 341–342
  once and many modes for, 342
  reading, 350–351
  replaceable, 341
  writing, 351
Objects
  byte buffer, 207–209
  embedded in views, 26–27
  embedding text data object, 20
  open service, 441, 470–472
  reading, 45
  selection ownership, 155
  stream, 134
  string, 211–212
  text data, 7–21
  text view, 5
  writing, 44–45
ObjectSend(), 266
  address book and, 320
  in installing applications and services, 415
  intertask messages and, 100
  for msgObjectNew, 456
  open service objects and, 472
Observer messages, text data, 21
Observer notification, 163
  byte buffer object, 209
  string object, 212
Observing
  changes, 89
  installation managers, 407–409
  tables, 215, 220–221
One-shot transfer. see Transfer, one-shot
Opening
  address book, 326
  Auxiliary notebooks, 423
  files, 46
  sample code, 46
  parallel port, 276
  serial port, 268
  service, 262–263
  example, 263
  socket handle, 298
  see also Closing
Open service
  messages sent to, 470
  objects, 441, 470–472
  clsOpenServiceObject and, 471
  clsService and, 471
  subclasseing clsOpenServiceObject and, 471–472
see also Services
Operating system, DLL files and versions, 403
Organization
  distribution volumes, 390–398
  file, 381–398
  PenPoint, 382–389
  required, 381
OS_DATE_TIME structure, 110
OSErrorBeep(), 105
OSFastSemaRequest, 414
OSFastSemaClear, 414
OSGetTime routine, 110
OS.H, 105
  functions, 105–107
OSHeapBlockAlloc(), 95, 175
OSHeapBlockFree, 88, 95
OSHEAP.H, 105
  functions, 107
OS HIMM prefix, 100
OSProcessHeap, 472
oProcessHeapId handle, 102
OSProgramInstall(), 377, 378–379
OSSharedMemAlloc(), 175
OSSubtaskCreate(), 297
OSSupervisorCall() function, 103
OSTone(), 105
Out box, 305
  connectivity and, 244
documents in, 309–310
general device concepts, 306–308
  introduction, 305–306
  notebook, 308
  operation, 308
  protocol messages, 308–309
  service
    communication target, 307
    enabling and disabling, 307–308
    handling input and, 312
    installing, 307
    sections, 306
    working with existing, 311
    writing own, 310
  service messages, 313–316
  see also In box
Outbox Notebook. see Auxiliary notebooks
Output buffer, 265
  flushing, 271
  status, 271
  see also Buffers
Output manager. see thePrintManager; theSendManager
Output, operation phases, 311
Removing
class all list items, 130
document to stationary menu, 427
list items, 130
see also Deleting
Renaming
directories, 141
files, 141
Replaceable shape matcher, 439
Replacing
characters, 198
list items, 130
Required organization, 381
RES_AGENT structure, 354
RESAPPND utility, 360
RES_DUMP utility, 360
RES_ENUM structure, 352
RESFILE.H, 343
macros, 344–345
RES_FILE_NEW structure, 348
RES_ID value, 344
RES_INFO structure, 349
resInput array, 356
resInputFile, 359
Resource agents, 345, 353–354
reading and writing data resources
and, 353
writing own, 356
Resource compiler, 359–360
Resource definitions, 356
example, 356–357
structure, 356
Resource files, 337, 342
compacting, 353
definition, 355
deleting resource from, 348–349
flushing, 353
handle, 348–349
header, 342
organization, 355–356
viewing contents of, 359, 360
see also Resources
Resource IDs, 337, 338
dynamic, 343–344
system preferences and, 362
using, 344–345
well-known, 343
list, 344
Resource lists, 337, 345–346
creating, 346
Resources, 337–368
C language definition, 355–357
compiling, 359–360
RESAPPND utility, 360
RES_DUMP utility, 360
running source compiler, 359–360
data, 337, 342, 353
C language definition, 355
reading, 349
writing and updating, 349–350
defining Quick Help, 356–357
deleting, 352–353
developer's quick start, 337–338
enumerating, 351–352
identifying, 342–344
locating, 349
object, 337, 341–342
once and many modes for, 342
reading, 350–351
replaceable, 341
writing, 351
overview, 337
system preferences, 361
types, 343
resOutputFile, 359
RES_READ_DATA structure, 349
RES_READ_OBJECT structure, 350
RES_WRITE_DATA structure, 350
RES_WRITE_OBJECT structure, 351
Rings, 103
Root directory, 52
handle, 60–61
Rows, table, 213
adding, 222–223
element, 223
converting number to position, 226
deleting, 224
getting length of, 227
getting number of, 227
see also Tables
RTF
documents, 188
files, 180
strings, 191
RTS/CTS protocol, 252
RTS (request-to-send) lines, 270
Run-time system, 387–388
Screen orientation preference, 363
Scrolling
text view, 29
window, inserting text view in, 30
Scroll margin style preference, 365
SDK directory, 385, 389
contents, 389
Search and replace, 195–198
API function, 124
concepts, 195
highlighting text and, 198
messages, 198
protocol, 196
replacing characters and, 198
searching text and, 197
setting initial search position, 196
writing class and, 196–198
Searching
address book, 326–328
more information, 327–328
search query, 326–327
search result, 327
tables, 224–226
Selection
classes that handle, 157
current, 155
determination, 157
owners, 156
finding, 161–162
messages sent to, 159–161
preserving, 162
restoring, 162
setting, 159, 162
preserving, 156
transitions, 157
Selection manager, 155–163
client messages, 158–159
concepts, 155–157
determining selection, 157
function, 124
messages passes to, 161–162
messages sent to selection owners, 159–161
observer notification and, 163
preserving selection and, 156
selection classes, 157
messages, 157–158
selection ownership and, 156
selection transitions and, 157
SEL_OWNERS structure, 161–162, 163
Semaphores, 101
clsInstallMgr and, 414
counting, 101
locked, 101
table
example, 222
shared, 217
using, 222
Sendable services, 331–334
defined, 331
messages, 333–334
protocol, 331–332
function, 319
Send command, 308
Sender, 166
Serial driver, 265
Serial handle
defaults, 269
requesting and releasing, 268
Serial I/O interface, 251, 265–274
buffered data, 265
concepts, 265–266
concurrency issues, 266
events, 266
flow control, 265–266
high-speed packet I/O concepts, 273–274
interrupt driven I/O, 265
messages, using, 267–273
detecting events, 272–273
flow control, 272
reading/writing with serial port, 271
reinitializing serial port, 268
requesting/releasing serial handle, 268
sending BREAK, 272
serial port configuration, 268–271
Serial port, 246
closing, 268
configuration, 268–271
baud rate, 269
data modem, 280
DTR/RTS output, 270
flow control, 269
flow control characters, 270
input line status, 270
line control, 269
requesting settings, 270–271
data modem and, 279–280
configuring, 280
getting handle, 279–280
opening, 268
reading and writing with, 271
reinitializing, 268
requesting all settings, 270–271
see also Parallel port; Ports
Service addresses, 322
SERVICE directory, 384, 386, 395
contents, 387
files in, 396
INST directory and, 396
SERVICE.INI file, 256, 385, 387
service directory and, 444, 445
Services, 438, 438–439
accessing, 258–259
overview, 257
service manager and, 261
binding to, 259
broken connection and, 247
chaining, 446
classes, 255–264, 439–441
initializing, 451, 452
closing, 263
clsStream messages and, 470
connections, 258, 446–447
connection status, 248, 447
data storage, 455
defined, 250, 255
downloading, 256, 456
design decisions, 449
directory contents, 444
distributing, 473–474
documentation and, 474
providing demo application, 473
providing preconfigured instances, 473
DLL and DLC files and, 256
exclusive access, 445–446
file system and, 443–445
in file system at run time, 445
In box and Out box, 305
devices and, 306–307
enabling and disabling, 307–308
installing devices and, 307
passive and active, 312–313
sections, 306
installable, 387
installation, 378–379, 450–456
calling initialization routines, 451–452
calling InitService, 452–453
calling other class initialization routines, 452
creating service instances, 454–455
INDEX

static and dynamic service instances, 453–454

tasks and, 455–456

installation to use overview, 441–443

installing, 256

interfaces and, 249–250

layered, 248

messages sent to open, 470

~IL, 246–249, 439

see also Parallel port; Serial port

msgSvcGetMetrics response, 460

msgSvcSetMetrics response, 461

multiple access, 445, 446

multi-user, 446

msgSvcOpenDefaultsRequested and, 463

multiple openers and, 452

non-port, 439

on distribution/boot disks, 444

opening, 257, 259, 262–263

owner, 247

setting, 264

ownership, 257

setting, 264

preconfigured, 396

printer, 250

programming, 449–472

service manager and, 440

shared, 446

targeting, 258, 446

test examples, 475–506

BASICSVC, 485–487

~ILSVC, 487–506

TESTSVC, 476–485

in theSelectedVolume, 445

transaction, 296

requesting, 300

responding to, 301

unbinding, 263

upgrading, 398

writing, 435–472

see also Open service

Service sections, 306

Service State Nodes, 396, 444, 445

setbufO system service, 65

Settings notebook. see Auxiliary notebooks

SIO_CONTROL_IN_STATUS structure, 270

SIO_CONTROL_OUT_SET structure, 270

SIO_EVENT_HAPPENED structure, 272–273

SIO_EVENT_SET structure, 272

generating current, 273

SIO_FLOW_CONTROL_CHAR_SET structure, 270

SIO_FLOW_TYPE structure, 269

SIO_INIT structure, 268

SIO_INPUT_BUFFER_STATUS structure, 271

SIO_LINE_CONTROL_SET structure, 269

SIO_METRICS structure, 270–271

SIO_OUTPUT_BUFFER_STATUS structure, 271

SM_ACCESS structure, 261

SM_BIND structure, 262

in unbinding from service, 263

SM_CLOSE structure, 263

SM_CONNECTED_NOTIFY structure, 264

SM_FIND_HANDLE structure, 263–264

SM_OPEN_CLOSE structure, 262–263

accessing socket and, 298

SM_SET_OWNER structure, 264

Socket, 295–296

accessing, 298

closing, handle, 299

connection, 296

identifier, 296

transport address, 296

SoftTalk interface, 250, 295

Sorting, directory entries, 88–89

Sound routine, 105

Speaker, modem, 284–285

SR_GET_CHARS structure, 197

SR_REPLACE_CHARS structure, 198

STAMP utility, 390–391

stationary and, 392

Starting point, locator, 55

State

browse, 143

collection, notification, 264

Stationary

application, 392–393

documents, 377

removing document to, 427

menu

adding document to, 426

modifying, 426–427

templates, 387

Stationary Notebook, 393

see also Auxiliary notebooks

STATNRY directory, 387, 392–393

Status values, in accessing service, 261

stdio calls, 65

STDIO.H, 113

stdio run-time package

accessing file system with, 65–66

for node creation, 43

paths and, 66

theWorkingDir and, 61

translating between handles and FILE pointers, 65–66

using, 66

SVC_BIND structure, 462

SVC_GET_SET_MTRICS structure

msgSvcGetMetrics and, 459–460

msgSvcSetMetrics and, 460

SVC_INIT_SERVICE structure, 452–453

Streams

accessing, auxiliary data, 177

connecting, to producer, 178

creating objects, 134

receiver, 176–177

sender, 177

flushing, 136

freeing, 177

initializing, 178

objects, 134

operations, 133

reading, 134

with timeout, 134–135

writing, 134

with timeout, 134–135

STREAM_ISK SEEK structure, 135

Stream transfers, 168–170

blocking protocol, 168–169

producer protocol, 168, 169

STRING.H, 111–112

String objects, 211–212

concepts, 211

creating, 212

getting, 212

messages, 212

notification of observers, 212

resetting, 212

Strings

16-bit function, 111–114

composition functions, 114

date and time, 110

STROBJ_NEW_ONLY structure, 212

Subclassing

cclsOpenServiceObject, 471–472

cclsStream, 133

file system classes, 67

Subtasks, 98–99

characteristics, 99

sibling, 99

see also Tasks

SVC_BIND structure, 462

SVC_GET_SET_MTRICS structure

msgSvcGetMetrics and, 459–460

msgSvcSetMetrics and, 460

SVC_INIT_SERVICE structure, 452–453
Table object
access concurrency characteristics, 216
accessing, 216
creating, 220
current state, 215
observing, 215
Table of contents
bookmark check box, 145
browser and, 137–138
creating, 138
Tables, 213
accessing, 214
beginning, 221
ending, 228
adding rows to, 222–223
Boolean operators for, 225
creating, 221
defined, 213
defining, 218–219
describing, 213–214
files, 214
freeing, 228
locating records in, 215
messages for, 217–218
information, 226
observing, 215, 220–221
positioning in, 215
searching, 224–226
semaphores and, 222
shared, 215–217
access to table object, 216
concurrency, 216–217
ownership, 216
state, 227
using, in database, 217
see also Columns, table; Rows, table
TA_CHAR_ATTRS structure, 16–17
TA_CHAR_MASK structure, 16–17
Tag argument, 16
atomChar, 16
atomPara, 17
atomParaTabs, 18
tagPrAutoShutdown, 365
tagPrAutoSuspend, 364
Tags, for data transfer types, 166–167
TAGS subdirectory, 389
TA_MANY_TABS structure, 18
TA_PARA_ATTRS structure, 17
TA_PARA_MASK structure, 17
Target
directory, 59
changing, 86
service, 258
Targeting
communication devices, 307
services, 446
Task management, 98–99
priority level, 99
processes, 98
software task scheduler, 99
subtasks, 98–99
Tasks, 98
80386 protected mode and, 102
family, 99
priority level, 99
privilege level, 103
services and, 455–456
software, scheduler, 99
see also Subtasks
TBL_BEGIN_ACCESS structure, 220, 221
TBL_BOOL_OP, 225
TBL_COL_DESC structure, 218–219
TBL_COL_GET_SET_DATA structure
getting data and, 223
setting data and, 223
TBL_CONVERT_ROW_NUM structure, 226
TBL_CREATE structure, 218
TBL_FIND_ROW structure, 224–225
TBL_FREE_BEAHV values, 220
TBL_GET_SET_Row structure
getting data and, 224
setting data and, 223
TBL_GET_STATE structure, 227–228
TBL_NEW structure, 220
TBL_Row_POS value, 215
TBL_STRING structure
getting data and, 224
setting data and, 223
TBL_TYPES, 219
TD_METRICS structure, 14
TD_NEW structure, 13
Template services, 449–450
TESTSVC service, 476–485
declared, 475
METHOD.TBL, 476
OPENOBJ.H, 483–485
TESTSVC.C, 477–483
TESTSVC.H, 477
Text
character encoding, 8
insertion pads, 9
creating, 33
destroying, 33
messages, 33
using, 33
length, 14
metrics, 14
style, getting and setting, 32
see also Text subsystem; Text views
Text attribute arguments, 16–20
character attributes, 16–17
paragraph attributes, 17–18
paragraph tabs, 18
TEXT_BUFFER structure, 14, 15
TEXT_CHANGE_ATTRS structure, 19–20
Text class
   hierarchies, 4
   see also clsText
TextCreateTextScrollWin function, 9, 30
Text data object, 7–9
   altering, 15
   attributes, 7–9
   creating, 7, 13
   embedding objects, 20
   functions, 11–12
   getting and setting attributes, 16–20
   getting and setting text metric, 14
   messages, 12–13
   observer messages, 21
   organization of, 8
   reading characters in, 14
   scanning ranges of characters in, 15–16
   text length and, 14
   using, 11–21
TextDeleteMany function, 11
Text editor, help text and, 180
TEXT_EMBED_OBJECT structure, 20
TEXT_GET_ATTRS structure, 18–19
Text index, 27–29
TEXT_INDEX type variables, 8
TextlnsertOne function, 12
TEXT_SPAN_AFFECTED structure, 21
TEXT_SPAN structure, 15–16
Text subsystem
   atoms, 37–38
   classes, 7
   comparison with graphics subsystem, 5
   for creating object of clsTextView, 35
   developer’s quickstart, 5
   features, 4–5
   interaction with Input subsystem, 27–29
      obtaining text index, 27–29
      processing input Xlist/gesture, 29
   overview, 3–4
   paragraph data attributes, 9
   sample code, 35–36
   text data object and, 7–9
   text insertion pads, 9
   text views, 9
   units of measurement, 10
   see also Text
Text views, 9
   checking consistency of, 32
   creating, 9, 24–26
   embedding objects in, 26–27
   getting and setting text style and, 32
   getting current selection and, 30–31
   getting viewed object’s UID, 26
   inserting in scroll window, 30
   interacting with Input subsystem, 27–29
   messages, 23–24
   scrolling, 29
   using, 23–32
   xRegions, 29
      illustrated, 28
   yRegions, 28
      illustrated, 28
   see also Text
theAddressBookMgr, 318
   address book manager protocol and, 320
   address book protocol and, 319
   changing information and, 328
   getting more information and, 327
   observing, 330
   option sheet protocol and, 330
   system address book and, 329–330
theAuxNotebookMgr, 380, 421
   file system and, 422
theBootVolume, 431
theBusyManager, 193
   in busy clock delay and reference count, 194
   example, 193
   messages, 193
   using, 193–194
theFileSystem, 49
 observer, 89
theHighSpeedPacketHandlers, 252, 273
theInstalledApps, 377, 388, 406
theInstalledFonts, 388, 406
   function, 416
theInstalledHWX, 378
theInstalledHWXProtos, 388, 406
theInstalledPDicts, 406
theInstalledPreferences, 361–362
theInstalledPrefs, 406
theInstalledServices, 256, 388, 406
   function, 416
   in service deinstallation, 465
   in service installation, 442
theParallelDevices, 251, 275
   in closing port, 277
   function, 276
   in opening port, 276
thePrinterDevices, 257
thePrintManager, 308
 output service, 308
theQuickHelp, 181
   advanced topics, 187
   object, 182
theSearchManager, 196
   creating mark and, 196
   replacing characters and, 198
   searching text and, 197
theSelectedVolume, 431
   services in, 445
theSelectionManager, 31, 155
   messages from clients to, 158–159
   messages passed, 161–162
   messages sent to selection owners, 159–161
   observer notification and, 163
   ownership and, 155–156
   preserving selection and, 156
   promoting/demoting and, 160
   responsibilities, 155
   setting owner and, 159
theSendableServices, 319, 331
theSendManager, 308
   output service, 308
theSerialDevices, 251, 257, 265
   data modem and, 279–280
   in requesting handle, 268
theSystem, 429
theSystemPreferences, 110, 361
   at boot time, 362
   preference change notification and, 368
theSystemResFile, 346
theTimer, 103, 104
   messages, 104
theTransportHandlers, 298
theUndoManager, 200
   aborting transaction and, 204
   adding items and, 203
   beginning transaction and, 202–203
   ending transaction and, 204
   messages, 202
   transaction history size and, 205
   transaction metrics and, 205
theWorkingDir, 61
   paths and stdio and, 66
   stdio operations and, 65
Time
   current, 104
   formats, 110
   strings, 110
   system, 110
   see also Date/time services
Time and date preference, 366
Time format preference, 367
TIME.H, 114
Time Management application, 391–392
Timer
   interface, 104–105
   routines, 103
TOC browser, 148
   exporting and, 152
   importing and, 148
TOPS SoftTalk (Sitka), 253, 295
TP_NEW_ONLY structure, 298
TP_RECVFROM structure, 300
TP_SENDRECVTO structure, 300
TP_SENDTO structure, 299
Transaction, 199
  aborting, 204–205
  adding items to, 203–204
  beginning, 202–203
  data, 201
  ending, 204
  history, changing size of, 205
  metrics, getting, 205
  services, 296
  requesting, 300
  responding to, 301
  undoing, 206
Transfer
  ASCII metrics, 175
  buffer
    fixed-length, 174
    structures, 167
    types, 173–174
    variable-length, 174–175
  client-defined, 170
  defined, 167
  functions and messages, 170–171
  one-shot, 167–168
  defined, 167
  performing, 173–174
  replying to, 176
  protocols, 167–170
  stream, 168–170
  defined, 167
  performing, 176–178
  types, 166
    adding to list, 172–173
    establishing, 171–173
    listing, 172
    requesting, 172
    searching list, 173
Transfer class, 165–178
  concepts, 165–167
  establishing transfer type, 171–173
  functions, 124
  messages and, 170–171
  performing one-shot transfers and, 173–176
  performing stream transfers and, 176–178
  transfer protocols, 167–170
Transport
  address, 296
  binding to, 301
  protocols, 253
  service types, 296
Transport API, 295–304
  concepts, 295–297
  using CTransport, 297–301
  for AppleTalk, 301–304
Traverse
  call back routine, 82
  ordering of, 82
  quicksort routine, 82
  tsAlignEdge, 29
  TV_Embed_Metrics structure, 26
  TV_New structure, 9, 25
  inserting text view in scrolling window and, 30
  style flag, 30
  TV_Resolve structure, 27
  TV_Scroll structure, 29
  TV_Style flags, 25–26
  Twips, 10

UID
  getting viewed object and, 26
  open service object class and, 441
Undo
  command, 199
  history, 201
  items, 199, 201
  deallocating buffer and, 201
UNDO_ITEM structure, 203,
  Undo manager, 199–206
  concepts, 199–201
  deallocating buffer and, 201
  function, 124
  instance, 200
  messages, 202
  using, 202–206
UNDO_METRICS structure, 205
UNIXD.H, 114
Unrecognized character preference, 368
User column, browser, 145–146
User interface, displaying, 332
Utility classes, 123
  features, 124
  see also specific classes

Values
  attribute, 78
  getting length of, 79
  setting, 79
  types of, 76
  zero, 77

Versions, DLL file, 402
  operating system, 403
Volume connectivity strategy, 244
Volumes, 43, 49–52
  concepts, 49
  connecting and disconnecting, 50
  defined, 49
  directory structure on, 53
  distribution, 390–398
  general structure, 384
  getting information about, 90–91
  list of, 49
  messages specific to, 91
metrics, 49–50
  information for, 90
  multiple, 398
  names, 51
    duplicate, 50
    local disk, 51
    memory-resident, 52
    remote, 51
    setting/changing, 91
  protection of, 67
  traversing, 82
  types, 50–52
    local disk, 51
    memory-resident, 52
    remote, 51–52
  uses of, 381
Volume structure, 383

WATCOM C run-time library. see
  C run-time library
Well-known list resource IDs, 344
  defined, 343
  index, 344
Well-known ports, 296
Well-known resource IDs, 343
  defined, 343
Window
  address, 332
  creating, 333
  filling, 333–334
  Quick Help, 181
  example, 186
  opening, 188
  WknItemResIdO macro, 344
  WknListResIdO macro, 344
  WknObjResIdO macro, 345
  WknResIdO macro, 345
  Wrapper document, 310
Writing
  address book, 328–330
    agents, own, 354
    browser state, 143
    data resource, 349–350
    resource agents and, 353
  files, 83
  object resource, 351
  objects and data, 44–45
  to parallel port, 278
  with serial port, 271
  services, 435–472
  streams, 134
    with timeout, 134–135
  see also Reading
Writing Paper application, 31
Writing style preference, 364
XferAddIds() function, 171-172
to add transfer type to list, 172
parameters, 173
prototype for, 172-173
XFER.ASCII_METRICS structure, 31
in transfer, 175
XFER.BUF structure, 174-175
XFER.CONNECT structure, 178
XFER.FIXED_BUF structure, 174
XFER.H, 173-174
XferListSearch() function, 171
parameters, 173
prototype, 173
in searching transfer type list, 173
XferMatch(), 171
to list transfer types, 172
parameters to, 172
prototype for, 172
Xfer mechanism, 30-31
XferStreamAccept, 177
XferStreamConnect() function, 176
arguments, 176
function, 176-177
XON/XOFF flow control, 265-266
see also Flow control

\Your Company directory, 384
organization, 389
subdirectories, 390

ZIP_GETZONES structure, 304
Zone protocol, AppleTalk, 304
Zooming allowed preference, 365
Your comments on our software documentation are important to us. Is this manual useful to you? Does it meet your needs? If not, how can we make it better? Is there something we're doing right and you want to see more of?

Make a copy of this form and let us know how you feel. You can also send us marked up pages. Along with your comments, please specify the name of the book and the page numbers of any specific comments.

Please indicate your previous programming experience:
- MS-DOS
- Mainframe
- Minicomputer
- Macintosh
- None
- Other ________

Please rate your answers to the following questions on a scale of 1 to 5:

1. Poor 2 3 Average 4 5 Excellent

How useful was this book?  
Was information easy to find?  
Was the organization clear?  
Was the book technically accurate?  
Were topics covered in enough detail?  

Additional comments:

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

Your name and address:

Name

Company

Address _____________________________

City _____________________________ State _____________ Zip _____________

Mail this form to:

Team Manager, Developer Documentation
GO Corporation
919 E. Hillsdale Blvd., Suite 400
Foster City, CA 94404–2128

Or fax it to: (415) 345-9833
Operating Systems

PenPoint™ Architectural Reference, Volume II

PenPoint Architectural Reference, Volume II, together with Volume I, provides a comprehensive description of the Application Programmatic Interface (API) for the PenPoint operating system. This volume describes the supplemental classes and functions that provide many different capabilities to PenPoint applications. The topics discussed in this volume include the:

- Text subsystem
- Installation APIs
- File system
- PenPoint services architecture
- Connectivity
- PenPoint resources
- PenPoint operating system kernel
- Miscellaneous other classes

The PenPoint operating system is a compact, 32-bit, fully object-oriented multitasking operating system designed expressly for mobile, pen computers. GO Corporation designed the PenPoint operating system as a productivity tool for people who may never have used computers before, including salespeople, service technicians, managers and executives, field engineers, insurance agents, and adjustors, and government inspectors.

Other volumes in the GO Technical Library are:

- *PenPoint Application Writing Guide* provides a tutorial on writing PenPoint applications, including many coding samples.
- *PenPoint User Interface Design Reference* describes the elements of the PenPoint Notebook User Interface, sets standards for using those elements, and describes how PenPoint uses the elements.
- *PenPoint Development Tools* describes the environment for developing, debugging, and testing PenPoint applications.
- *PenPoint Architectural Reference, Volume I* presents the concepts of the fundamental PenPoint classes.
- *PenPoint API Reference, Volume I* provides a complete reference to the fundamental PenPoint classes, messages, and data structures.
- *PenPoint API Reference, Volume II* provides a complete reference to the supplemental PenPoint classes, messages, and data structures.

**GO Corporation** was founded in September 1987 and is a leader in pen computing technology for mobile professionals. The company's mission is to expand the accessibility and utility of computers by establishing its pen-based operating system as a standard.

**GO Corporation**

919 East Hillsdale Blvd.
Suite 400
 Foster City, CA 94404

Addison-Wesley Publishing Company