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.
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(msgNew, defaults, clsAppMgr, &new, s, Error);
new.object.uid = clsTttApp;
new.object.key = 0;
new.cls.pMsg = clsTttAppTable;
new.cls.ancestor = clsApp;
new.cls.size = SizeOf(P_TTT_APP_INST);
new.cls.newArgsSize = SizeOf(APP_NEW);
new.appMgr.flags.stationery = true;
new.appMgr.flags.accessory = false;
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.
## Part 1 / Class Manager
1 / Introduction 5
2 / Class Manager Concepts 9
3 / Creating a New Class 31
4 / Manipulating Objects 49

## Part 2 / PenPoint Application Framework
5 / Introduction 67
6 / Application Environment Concepts 75
7 / Application Concepts 81
8 / Life Cycles 95
9 / Embedded Documents 117
10 / Mark Concepts 129
11 / Printing 135
12 / The Application Manager Class 145
13 / The Application Monitor Class 151
14 / The Application Class 157
15 / The View Class 173
16 / The Application Directory Handle Class 177
17 / Container Application Classes 185
18 / Embedded Window Class 189
19 / Application Window Class 195
20 / The Mark Class 199

## Part 3 / Windows and Graphics
21 / Introduction 209
22 / Window System Concepts 215
23 / The Window Class 227
24 / Window Device Classes 255
25 / Graphics Concepts 265
26 / The Drawing Context Class 281
27 / The Picture Segment Class 317
28 / Bitmaps and TIFFs 329
29 / ImagePoint Rendering Details 335

## Part 4 / UI Toolkit
30 / Introduction 349
31 / Concepts and Terminology 357
32 / Toolkit Ancestors 367
33 / Border Windows 371
34 / Layout Classes 381
35 / Controls 399
36 / Labels 409
37 / Buttons 417
38 / Toolkit Tables 425
39 / Menus and Menu Buttons 445
40 / Scrollbars 453
41 / List Boxes 463
42 / Fields 475
43 / Notes 487
44 / Frames 497
45 / Frame Decorations 507
46 / Option Sheets 511
47 / Icons 523
48 / Trackers and Grab Boxes 527
49 / Progress Bars 531

## Part 5 / Input and Handwriting Translation
50 / Introduction 545
51 / Developer's Quick Start 555
52 / Event Processing 563
53 / Input Subsystem API 567
54 / Pen Events 575
55 / Keyboard Events 581
56 / Using clsIP 585
57 / Using clsSPaper 589
58 / Using the Translation Classes 597
59 / Using Scribbles 607
60 / Using Xlists 611
61 / Using Gesture Windows 617

## Index
619
Part 1 / Class Manager
Chapter 1 / Introduction

About Object-Oriented Programming 1.1 5
An Overview of This Part 1.2 6
Further Literature on Object-Oriented Programming 1.3 7

Chapter 2 / Class Manager Concepts

Identifiers 2.1 9
Administration of Well-Known UIDs 2.1.1 10
Messages 2.1.2 11
Status Values 2.1.3 11
Tags 2.1.4 12
Macros for Working with UIDs 2.1.5 12
Sending Messages 2.2 13
Return Values 2.2.1 14
Setting Up the Message Argument Structure 2.2.2 14
Creating a New Object 2.3 15
New Object Argument Structures 2.3.1 15
Using msgNewDefaults 2.3.2 17
Modifying Argument Data 2.3.3 17
Using msgNew 2.3.4 18
Creating an Object with Default Values 2.3.5 18
Handling Message Status 2.4 18
Other Ways to Send Messages 2.5 19
ObjectSend() 2.5.1 20
Functions Related to ObjectSend() 2.5.2 21
ObjectPost() 2.5.3 21
Using the Message-Sending Macros
  DEBUG Warning Macros 2.6.1 23
  Error-Checking Macros 2.6.2 23
  ObjectCall() Macros 2.6.3 24
Using Keys 2.7 24
Capabilities
  Owner Capability 2.8.1 26
  Freeing Capability 2.8.2 26
  ObjectSend() Capability 2.8.3 26
  ObjectCall() Capability 2.8.4 26
  Observable Capability 2.8.5 27
  Inheritance Capability 2.8.6 27
  Scavenging Capability 2.8.7 27
  Creation Capability 2.8.8 28
  Creation Notification 2.8.9 28
  Mutation Capability 2.8.10 28
  Checking Capabilities 2.8.11 29
  Changing Capabilities 2.8.12 29

Chapter 3 / Creating a New Class 31

Overview 3.1 31
  The Parts of a Class 3.1.1 31
  Installation Summary 3.1.2 33
Design Considerations 3.2 33
Creating Objects 3.3 34
Instance Data 3.4 34
  Memory Protection 3.4.1 34
  Allocating Instance Data 3.4.2 34
  Accessing Your Instance Data 3.4.3 35
  Maintaining Dynamic Instance Data 3.4.4 35
  Saving and Restoring Instance Data 3.4.5 35
Creating a Header File 3.5 36
Ancestor Calls 3.6 36
  ObjectCallAncestor() 3.6.1 36
  ObjectCallAncestorCtx() 3.6.2 36
Creating the Methods 3.7 37
  Declaring Entry Points for Methods 3.7.1 37
  Message Parameters 3.7.2 38
  Method Declaration Macros 3.7.3 39
  Operations of a Method 3.7.4 41
Creating a Method Table 3.8 41
  Method Table Overview 3.8.1 41
  Creating a Method Table Definition File 3.8.2 42
  Compiling a Method Table 3.8.3 45
Installing a Class in PenPoint 3.9 47
  The Class Initialization Routine 3.9.1 47

Chapter 4 / Manipulating Objects 49

Copying Objects 4.1 49
  Using msgCopy 4.1.1 49
Observing Objects 4.2 50
  Adding an Observer 4.2.1 50
  Adding an Observer with Position 4.2.2 51
  Removing an Observer 4.2.3 51
  Getting Observers from a List 4.2.4 51
Notifying Observers 4.3 52
  Posting to Observers 4.3.1 53
  Example of Observer Notification 4.3.2 53
Getting Object and Class Information 4.4 54
  Confirming an Object's Class 4.4.1 54
  Confirming an Object's Ancestor 4.4.2 55
  Getting an Object's Class 4.4.3 55
  Getting the Owner of an Object 4.4.4 55
  Getting a Class's Class 4.4.5 55
  Checking an Object for Validity 4.4.6 55
  Checking an Object's Version Number 4.4.7 56
  Getting Notification of Object Creation 4.4.8 56
Properties 4.5 56
    Creating a Property 4.5.1 56
    Retrieving Properties 4.5.2 57
Object Destruction 4.6 57
    Destroying an Object 4.6.1 57
    Handling Object Free Protocol 4.6.2 57
    Handling Failures During msgInit and msgRestore 4.6.3 59
Scavenging 4.7 60
Mutating Objects 4.8 60

List of Figures
3-1 Method Table and Class Implementation 32
3-2 Method Table Files and Build Sequence 43

List of Tables
2-1 UID Scope and Type 10
2-2 Object Capability Flags 25
3-1 Method Declaration Macros 39
3-2 MSG_INFO Option Flags 44
4-1 Observer Messages 50
4-2 Object and Class Information Messages 54

List of Examples
2-1 Sending a Message to an Object 15
2-2 Creating a New Object 18
2-3 Handling the Message Return Status 19
3-1 Using ObjectCallAncestorCtx() 37
3-2 Creating a New Class 48
4-1 Notifying Observers 53
Chapter 1 / Introduction

This chapter provides a brief introduction to the concepts and terminology used in object-oriented programming, and it introduces the PenPoint Class Manager subsystem within that context. The PenPoint Application Writing Guide provides additional information on how to use the PenPoint Class Manager.

About Object-Oriented Programming

This section introduces some of the basic terms used throughout this Part to describe object-oriented programming techniques. If you have no prior experience with object-oriented programming, read the section titled “Getting Started with Classes” in the PenPoint Application Writing Guide.

Your PenPoint program uses functional units called objects. There are objects that represent windows, scroll bars, lists, text views, etc. Objects communicate with each other by sending and receiving messages.

Each object receives messages and responds to them in a particular way. For example, a menu object responds to the message msgWinShow by displaying itself on the screen.

An object’s behavior is determined by the class that it belongs to. A class is a factory for creating objects. The class contains the code that handles the messages received by an object. Thus, clsMenu defines the manner in which menu objects respond to msgWinShow. The code that the object executes in response to a message is called a message handler.

When a class creates an object, that object is said to be an instance of the class. The special PenPoint class clsClass is a factory for creating objects which are themselves classes. Thus there are two types of objects: classes and instances. Both can receive and process messages, but only classes can create objects.

A class doesn’t have to define all of its behavior from scratch. It can inherit behavior from another class. For example, because a menu object is defined as a type of window, clsMenu is defined as a subclass of clsWin. This means that clsMenu inherits from clsWin, and in this case it is clsWin which contains most of the code to display a menu on the screen.

In fact the behavior of a class is inherited from all of its ancestors. clsTkTable is actually a subclass of clsTableLayout, which is a subclass of clsBorder, which is a subclass of clsEmbeddedWin, clsGWin, clsWin, and clsObject (the fundamental class). clsMenu, as a subclass of clsTkTable, inherits the behavior of all these classes.
When an object receives a message, the class that created the object handles the message. As part of handling the message, the class can choose to pass the message to its ancestor. The ancestor can also pass the message to its ancestor, and so on up the hierarchy of inheritance, up to clsObject.

The PenPoint™ operating system has a wide range of built-in classes that generate the instances your application requires: windows, scrollbars, lists, data views, and text objects. The PenPoint Class Diagram shows all the built-in PenPoint classes and their hierarchy of inheritance.

If a built-in class doesn’t provide quite the functionality that you need, you can create your own class which inherits from it and has additional or modified functionality. For example, your application is a class that you design, which inherits many capabilities from the superclass of applications, clsApp. A running instance of your application is an object created by your customized class.

An Overview of This Part

This part describes the PenPoint Class Manager. In addition, the following manuals in the PenPoint Software Developer’s Kit describe the programming environment for PenPoint:

- PenPoint Application Writer’s Guide
- UI Design Reference

After a general examination of object-oriented programming in PenPoint, the functional elements of the Class Manager subsystem are covered. The root classes of the API Class Hierarchy, clsObject and clsClass, are described along with explanations of how to use them to create the objects that will comprise your application. The chapters of this part cover the following information:

- Chapter 1 (Introduction, this chapter) provides a brief introduction to the Class Manager and an overview of this part of the PenPoint Architectural Reference.
- Chapter 2 (Class Manager Concepts) describes the concepts necessary for understanding the PenPoint class manager and classes in general. The sections describe the PenPoint object identifiers, how to send massages to objects, how to create new objects, macros that aid you in passing messages to objects, and properties that objects can have.
- Chapter 3 (Creating a New Class) describes what you have to do to create a new class. The chapter includes descriptions of instance data, method tables, entry points for methods, method parameters, and how to install a class in PenPoint.
- Chapter 4 (Manipulating Objects) describes the facilities provided by the class manager for controlling objects. This chapter includes discussions on copying objects, observing objects, getting information about objects, object properties, destroying objects, and scavenging objects.
Further Literature on Object-Oriented Programming

This manual attempts to introduce object-oriented programming concepts at the same time it introduces the PenPoint Class Manager and API Class Hierarchy. If you are unfamiliar with object-oriented programming, you should read “Getting Started with Classes” in the PenPoint Application Writing Guide. Here is some recommended general reading on object-oriented programming.


Chapter 2 / Class Manager Concepts

This chapter describes many of the Class Manager features that you will use when writing applications for PenPoint. It covers sending messages, creating objects, destroying objects to free resources, designing a class, and setting object capabilities.

Topics covered in this chapter:

- Identifiers for PenPoint objects
- Sending messages to objects
- How to create a new object
- Other functions that you can use to send messages to objects
- Macros that you can use to send messages to objects
- Object capabilities
- Object keys.

Identifiers

To create an object or send a message in PenPoint, you must identify the object or the class of the message using a 32-bit unique identifier (UID).

There are two types of UIDs: well-known and dynamic. Well-known UIDs are defined by you, GO, and other PenPoint developers at compile time. You typically use well-known UIDs to identify classes, but they can also be used to identify shared objects. Dynamic UIDs are created by the Class Manager at run time; typically, you use dynamic UIDs to identify objects (instances of classes).

Well-known UIDs include an administered value. When you define a class, you create a well-known UID that uses this value. All other identifiers related to that class (for example, messages, tags, and status values) use the same administered value. Chapter 4, Manipulating Objects, discusses the macros that you can use to both create these identifiers and extract information from these identifiers.

A well-known UID contains flags that specify the scope of the UID, i.e., whether it is known to all tasks in PenPoint (global), known to tasks in a particular process (process global), or known only to the task that uses it (private).

A global well-known UID allows all processes in the system to access the same object by using the same identifier. For example, clsWin is a global well-known UID identifying the window class and theFileSystem is a global well-known UID identifying the file system object. Any process that refers to theFileSystem will reference the same file system object.
A process-global well-known UID allows each process in the system to reference different objects with a single identifier. This is useful for objects that exist in each process, but that must have the same identifier. For example, theWorkingDir is a process-global well-known UID identifying the process working directory. A process that refers to theWorkingDir will reference its own working directory object. Other processes that refer to theWorkingDir will reference different working directory objects.

A private well-known UID belongs to the application developer. For example, a component used only by your application should be identified by a private well-known UID.

Dynamic object UIDs are allocated by the Class Manager during the creation of an instance, and returned to the client for subsequent references. All dynamic UIDs have global scope. After the object referenced by a dynamic UID is freed, that UID may refer to a different dynamic object at a later time.

The scope of an object's UID specifies the maximal possible access to the object. The actual access permitted by the object may be further restricted by the values of the objCapCall and objCapSend capabilities for the object (described later in this chapter).

<table>
<thead>
<tr>
<th>Scope</th>
<th>Object Type</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>Well-known</td>
<td>Identifying system-wide objects, such as the FileSystem.</td>
</tr>
<tr>
<td>Process-Global</td>
<td>Well-known</td>
<td>Identifying per-task objects, such as the WorkingDir.</td>
</tr>
<tr>
<td>Private</td>
<td>Well-known</td>
<td>Identifying objects specific to a task, such as an application-specific class.</td>
</tr>
<tr>
<td>Global</td>
<td>Dynamic</td>
<td>Identifying objects created by a client.</td>
</tr>
</tbody>
</table>

You use the MakeWKNI() macro to create well-known UIDs in your source code. For example, the code fragment below creates a value that is assigned to the symbol clsFileSystem. When the program creates the class clsFileSystem, it uses this value as the class identifier. You refer to this class with the clsFileSystem constant in your application code.

```
#define clsFileSystem MakeWKNI(62, 1, wknGlobal)
```

In brief, the arguments to MakeWKNI() specify:

- The administered value
- A version number
- The access for the UID.

### Administration of Well-Known UIDs

GO Corporation maintains a register of well-known UIDs to ensure that they are unique. If you do not register a global or process-global well-known UID, your class UIDs and other identifiers might collide with identifiers created by another
developer. Because the administered portion of the UID is used in other identifiers (such as messages, status values, and tags), you only need to register one UID per class.

You must contact GO Developer Support in order to obtain unique UID numbers.

PenPoint defines several well-known global and well-known private UIDs that are reserved for development and testing. These UIDs are defined in UID.H and have the symbols wknGDTa through wknGDTk for the global UIDs and wknLDTa through wknLDTg for private UIDs. You must not use these UIDs in code released outside your development organization.

Well-known private UIDs are private to a process. In other words, you can use well-known private UIDs in your application without concern for whether other applications use the same UIDs. You administer them yourself, choosing values that are unique within your application to pass to MakeWKN().

**Messages**

Like objects, messages are also identified by 32-bit constants. Messages identifiers share the administered part of the UID of the class that defines the message. Because message identifiers are associated with their class, you avoid the problem of identifier collisions. This example defines the message msgWinGetMetrics that is associated with the class, clsWin:

```
#define msgWinGetMetrics MakeMsg(clsWin, 5)
```

In brief, the arguments to MakeMsg() specify: 1) the class to which the message belongs and 2) a message number that is unique for messages that belong to this class. The macro extracts the administered portion of the class identifier and combines it with the message number to create the message identifier. The message number must be between 0 and 254, inclusive. Message number 255 is reserved for use by the Class Manager.

**Status Values**

A status value is a constant value returned by a method when it has completed a message's request. Your application source code uses the Class Manager macro MakeStatus() to create status values. The following code fragment defines a status value associated with the root class, clsObject:

```
#define stsScopeViolation MakeStatus(clsObject,5)
```

In brief, the arguments to MakeStatus() specify: 1) the class to which the status value belongs and 2) a status number that is unique for status values that belong to this class. The macro combines the administered portion of the class identifier with the status number to create the status value constant. The status number must be between 0 and 255, inclusive.

The sign-bit in the status value indicates whether the status value is an error status or a non-error status. This allows the client code to do a quick check on the value...
to look for error conditions (without having to match a series of status values).
You can create non-error status values with the **MakeWarning()** macro.

```c
#define stsAlreadyAdded MakeWarning(clsObject, 2)
```

The Class Manager defines **stsOK** as the usual non-error status that is returned by
a message handler.

```c
#define stsOK MakeWarning(0, 0)
```

Instead of testing for exact equality with **stsOK**, you should test whether a status
is less than **stsOK**. If it is less, an error condition was encountered; if it is greater
than or equal, the message handler successfully completed (the message sending
macros described in the section on Error-Checking Macros perform this test
for you). The message handler may return a status greater than **stsOK** to advise
the caller of an interesting condition which is not an error. For example, an
enumerator may return **stsTruncatedData** if passed a buffer smaller than the
number of items available for enumeration.

### Tags

Another useful 32-bit constant is a tag. You use **tags** to identify well-known
constants that are used by different modules in your source code. You use tags
most commonly to identify option sheets, option cards, and Quick Help strings.
This example defines a tag value for an option sheet used by **clsTttView**.

```c
#define tagTttViewOptionSheet MakeTag(clsTttView, 0)
```

In brief, the arguments to **MakeTag()** specify: 1) the class to which the tag belongs
and 2) a tag ID that is unique for tags that belong to this class. The macro
combines the administered portion of the class identifier with the tag ID to create
the tag constant. The tag ID must be between 0 and 255, inclusive.

By using tags, you are guaranteed that an identifier is unique for the set of all
classes.

### Macros for Working with UIDs

The macros described above (**MakeWKN()**, **MakeStatus()**, **MakeMsg()**, and
**MakeTag()**) are defined in the **GO.H** file. **GO.H** defines other Class Manager
macros that you can use to extract information from UIDs:

- **TagNum()** extracts the tag ID from a tag.
- **WKNVer()** extracts the version number from a well-known UID.
- **WKNAdmin()** extracts the administered value from a well-known UID.
- **WKNScope()** extracts the scope from a well-known UID.
- **WKNValue()** extracts the administrated value and the scope information
  from a well-known UID.
In addition, CLSMGR.H defines several macros to test the type of a UID:

- **ObjectIsDynamic()** returns \textbf{true} if the object is dynamic, otherwise \textbf{false}.
- **ObjectIsWellKnown()** returns \textbf{true} if the object is a well-known, otherwise \textbf{false}.
- **ObjectIsWKNO** is a synonym for **ObjectIsWellKnown()**.
- **ObjectIsGlobal()** returns \textbf{true} if the object is global, otherwise \textbf{false}.
- **ObjectIsLocal()** returns \textbf{true} if the object is local (that is process-global or private well-known), otherwise \textbf{false}.
- **ObjectIsGlobalWKNO** returns \textbf{true} if the object is a global well-known, otherwise \textbf{false}.
- **ObjectIsProcessGlobalWKNO** returns \textbf{true} if the object is a process-global well-known, otherwise \textbf{false}.
- **ObjectIsPrivateWKNO** returns \textbf{true} if the object is a private well-known, otherwise \textbf{false}.

### Sending Messages

You send messages to tell the instances to do things: to tell a window to resize itself or to tell a table to send back an item from a certain row and column address. Sending messages to objects is the primary mechanism for control and data flow in PenPoint.

The Class Manager provides a set of C functions and macros that send messages to objects. The functions take arguments that describe the target object, the message being sent, and a pointer to a structure that can contain additional argument data.

The most commonly used function is **ObjectCall()**, whose function prototype follows:

```c
STATUS GLOBAL ObjectCall(
    MESSAGE msg,
    OBJECT object,
    P_ARGS pArgs
);
```

- **msg** is the identifier for the message being sent.
- **object** is the identifier for the object to which the message is being sent.
- **pArgs** is the pointer to a structure that contains additional arguments to accompany the message. In some cases this argument is used to convey a 32-bit value, rather than a pointer.

The **ObjectCall()** function passes the parameters to the method that handles the message.

When you read more about how to create a class, in Chapter 3, Creating a New Class, you will understand more about how **ObjectCall()** locates the entry point.
Return Values

When the method completes it returns a STATUS value. The ObjectCall() function returns the STATUS value to its caller (your program).

Negative status values indicate errors conditions, positive values indicate non-error conditions. Although some PenPoint classes use status codes to return data in addition to indicating a condition, this technique is discouraged. If you need to return data, it is best to pass it back in the message argument.

Because checking for return values is so common, the Class Manager provides several macros that send a message and check the returned status value. “Using the Message-Sending Macros,” later in this chapter, discusses these macros.

Setting Up the Message Argument Structure

The third argument in the ObjectCall() function (and in most message-sending functions) is a pointer to argument data. Each message requires a specific argument structure; not all argument structures are the same. The message description in the header files and in the PenPoint API Reference specifies the argument structure required by each message.

For example, the clsTextData message msgTextModify takes as its argument P_TEXT_BUFFER, a pointer to a TEXT_BUFFER data structure. You can examine the file TXTDATA.H (where clsTextData is defined) to find the description of the TEXT_BUFFER structure:

```c
typedef struct TEXT_BUFFER {
    TEXT_INDEX first;        // In
    TEXT_INDEX length;       // In
    TEXT_INDEX buflen;       // In
    P_CHAR buf;              // In:Out via *buf
    TEXT_INDEX bufUsed;      // Out
} TEXT_BUFFER, *P_TEXT_BUFFER;
```

The PenPoint header files label the members of the argument structures with the words In, Out or In:Out. In denotes arguments sent to the called object, but which are not modified. Out denotes arguments sent back by the called object. In:Out denotes members used to send data in both directions. If a member isn't labelled with one of these words, it is implied to be In.

Of course, the argument structure can itself contain pointers to buffers allocated by your program. The receiving object can also copy data to these buffers. In the example above, the buf field is a pointer to an array of CHAR values used to store text.

Usually message-sending between objects is done within the same task; passing information between tasks or processes requires slightly more forethought (for example, you must create buffers for data in shared memory).

Example 2-1 shows how to send msgTextModify to an instance of clsTextData.
One of the things an instance of clsTextData does is maintain a text value. Object-oriented programming techniques restrict you from knowing the internal representation of the text data, so you cannot modify it with a simple assignment. Instead, you send the message msgTextModify to the clsTextData instance, with a pointer to a TEXT_BUFFER as an argument. The TEXT_BUFFER carries the text value that you want the clsTextData instance to maintain.

This example shows how to prepare a TEXT_BUFFER argument with the string "Hello World" as its text value, and then to send msgTextModify to an instance of clsTextData with the prepared TEXT_BUFFER as an argument. For simplicity, the example assumes that the clsTextData object is defined and created elsewhere.

```c
TEXT_BUFFER textBuf;
OBJECT textDataObject;
STATUS s;

textBuf.first = 0;    // Modify textDataObject chars starting with first char
textBuf.length = 0;  // Insert rather than overwriting existing text
textBuf.buf = "Hello World";      // the new text

s = ObjectCall(msgTextModify, textDataObject, &textBuf);  // send message
```

If the status code returned in s is stsOK, you can use the textBuf structure to examine the values passed back from the message. Part 6: Text discusses clsTextData specifics such as these return values.

### Creating a New Object

Thus far we have discussed objects without really discussing how an object is created. This section describes how you create objects.

In brief, to create an object, you:

1. Declare the argument structure for the new object.
2. Send msgNewDefaults to the class for the object to initialize the argument structure to default values.
3. Modify the default values if required.
4. Create the object by sending msgNew to the class of the object you want to create, with a pointer to the argument structure as an argument.

When msgNew completes successfully, the argument structure contains the UID of the newly created object. For purposes of sending messages, the UID is the new object.

### New Object Argument Structures

To create an object that belongs to a particular class, you must find the description of the class in either the PenPoint API Reference or in the header files. There you will find the argument structure required by msgNewDefaults and msgNew.

The argument structures for a new object always end with _NEW. For example, the argument structure for a note object (an instance of clsNote) is NOTE_NEW. If you look in the header file for clsNote (NOTE.H), you will find the type definition for NOTE_NEW:

```c
typedef struct NOTE_NEW {
    noteNewFields
} NOTE_NEW, FAR *P_NOTE_NEW;
```
The symbol `noteNewFields` is defined immediately before the `NOTE_NEW` type definition:

```c
#define noteNewFields
  frameNewFields \ 
  NOTE_NEW_ONLY note;
```

`clsNote` is a subclass of `clsFrame`. The above `#define` says that `noteNewFields` consists of the fields defined by the symbol `frameNewFields`, and a new field called `note`, containing a `NOTE_NEW_ONLY` structure. That is, the fields of a `NOTE_NEW` structure are everything that `frameNewFields` is, plus a structure containing the added data for a `clsNote` object.

The structure for `NOTE_NEW_ONLY` is described immediately above the `#define` for `noteNewFields`:

```c
typedef struct {
    NOTE_METRICS metrics;
    P_CHAR pTitle;
    P_UNKNOWN pContentEntries;  // used to create the content
    P_TK_TABLE_ENTRY pCmdBarEntries;  // used to create the command bar
    U32 spare;  // reserved
} NOTE_NEW_ONLY, *P_NOTE_NEW_ONLY;
```

This structure describes the `clsNote`-specific arguments that pertain to the object created by `msgNew`. The object contains metrics, a pointer to a title string, a pointer to note text, a pointer to the command bar entries, and a spare `U32` for future use by GO Corporation.

**More About the NewFields #define**

The `msgNew` and `msgNewDefaults` argument structure used for a particular class contains structures defined by each of the classes from which the class inherits. The `#define` in the example above defined the symbol `noteNewFields` which contained the symbol `frameNewFields` plus a `NOTE_NEW_ONLY` structure. This gives `clsNote` all the data structure of its superclass, `clsFrame`, in addition to the `clsNote`-specific data. If you go to the header file for `clsFrame`, you will find that `frameNewFields` is defined as the symbol `shadowNewFields` and a `FRAME_NEW_ONLY` structure. You can continue to trace this inheritance back to the structure `OBJECT_NEW_ONLY`, which is defined by the fundamental class, `clsObject`.

2.3.1.1
The PenPoint API Reference displays the expanded form of the _NEW structure for each class. The expanded form of NOTE_NEW is:

```c
typedef struct NOTE_NEW {
    OBJECT_NEW_ONLY object;
    WIN_NEW_ONLY win;
    GWIN_NEW_ONLY gWin;
    EMBEDDED_WIN_NEW_ONLY embeddedWin;
    BORDER_NEW_ONLY border;
    CSTM_LAYOUT_NEW_ONLY customLayout;
    SHADOW_NEW_ONLY shadow;
    FRAME_NEW_ONLY frame;
    NOTE_NEW_ONLY note;
} NOTE_NEW, far *P_NOTE_NEW;
```

Thus, when you declare a structure using the type NOTE_NEW, the structure contains the NOTE_NEW_ONLY structure and all the structures defined by clsNote's ancestors.

It is important to understand the inheritance of the class from which you are creating an object, because all of the structures defined by the _NEW structure affect the object created by msgNew. msgNewDefaults sets most of the fields defined by these structures to useful default values.

### Using msgNewDefaults

You send msgNewDefaults to a class, passing a pointer to the uninitialized _NEW structure, so that the class can set its default values for the argument structure. All classes respond to msgNewDefaults. When you define your own classes, you must design them to handle msgNewDefaults by setting class-specific structures to useful default values; in particular, you should set the default values of your _NEW_ONLY structure.

The datasheets in the PenPoint API Reference and the header file for the class normally list the default values that msgNewDefaults assigns to the _NEW structure.

### Modifying Argument Data

To obtain some non-default behavior from the new object you are going to create, you must modify some of the default _NEW values that msgNewDefaults sets. Furthermore, for some _NEW fields, it is impossible to establish a useful default value; you must set such fields explicitly.

In most cases, the default values are sufficient. The header file or data sheet for a class should document the default values as well as which, if any, of the fields you must set explicitly.
Using msgNew

After you send msgNewDefaults to set default values for the _NEW structure (and after modifying those defaults if necessary), you send msgNew to the class that creates the object. The argument to msgNew is a pointer to the _NEW structure that you have initialized.

Example 2-2 shows how to create a new instance of clsNote.

Example 2-2
Creating a New Object

This example shows how to create a new object, in this case an instance of clsNote. The example does the following:

1. Sends msgNewDefaults to clsNote, with a pointer to a NOTE_NEW structure as an argument, to set the NOTE_NEW structure to default values.
2. Modifies some of the default NOTE_NEW values.
3. Sends msgNew to clsNote, with the initialized and modified NOTE_NEW structure as an argument, to create a new instance of clsNote.

```c
NOTE_NEW noteNew;
TK_TABLE_ENTRY tkEntry[2];
STATUS s;
s = ObjectCall(msgNewDefaults, clsNote, &noteNew);
//
noteNew.note.metrics.flags = nfAutoDestroy | nfDefaultAppFlags;
noteNew.note.pContentEntries = tkEntry;
//
s = ObjectCall(msgNew, clsNote, &noteNew);
```

clsNote responds to msgNew by manufacturing a new instance of clsNote, a note object. The Class Manager assigns a dynamic UID to the new instance, and places this value in note.object.uid.

Creating an Object with Default Values

Sometimes there is no reason to change any of the default values that msgNewDefaults sets. In these cases, rather than send msgNewDefaults followed immediately by msgNew, you can use msgNewWithDefaults.

msgNewWithDefaults is a convenient way to get the effect of sending msgNewDefaults followed immediately by msgNew.

Handling Message Status

When the message handler for msgNew completes, it returns a status value indicating whether clsNote succeeded in creating the new instance. If the message completes successfully, it returns stsOK. You can now use the UID value in note.object.uid as a target for messages; the UID identifies the new object.

If a message does not complete successfully, it returns a status other than stsOK. Your code must be prepared to handle such conditions. Example 2-3 shows the code structure required to handle return values other than stsOK.
Other Ways to Send Messages

Thus far we have limited our discussion to the `ObjectCall()` function for sending a message to an object. `ObjectCall()` executes code in the task that calls `ObjectCall()`. If your code sends a message with `ObjectCall()`, your execution thread goes from your code, to the code of the message handler, then back to your code—all executing in your code's task. This is why you normally use `ObjectCall()` to send `msgNew` to create a new object. By using `ObjectCall()`, you cause the instance-creation code to run in your task. Your task owns the object because it was created in your task.

However, the PenPoint™ operating system is multi-tasking. It supports many different tasks; each task gets a share of CPU cycles. Every active document in PenPoint is a separate task (thus embedded documents run in separate tasks from their parent documents). The PenPoint operating system itself uses several tasks. Your own application architecture may involve multiple tasks.

`ObjectSend()` works like `ObjectCall()` except that the message handler code runs in the task that owns the object rather than in your task. Your task waits for the `ObjectSend()` to return; while awaiting the return, your task will continue to dispatch messages sent to objects owned by your task. This prevents a deadlock condition from arising when you send to an object owned by another task, which in turn sends to an object owned by your task.

`ObjectPost()` works like `ObjectSend()` in that the message handler code runs in the object's owning task. However, `ObjectPost()` returns immediately, allowing your task to continue its execution. At some later time, the message handler for the posted message executes in the recipient object's task.
**ObjectSend()**

The multitasking aspect of the PenPoint operating system makes it likely that you will at some time or another wish to send messages to objects that are not in your task. To send a message to an object in another task, you use the `ObjectSend()` function (or one of the related functions described later in this chapter). When you send a message with `ObjectSend()`, the execution thread goes from your code running in your task, to the message handler code running in the object’s task, then back to your code running in your task.

There are two common reasons you would want to execute code in a different task:

- You want to make use of an object owned by another task. Using `ObjectSend()` is required when the object you want to send the message to is owned by another task and does not have `objCapCall` enabled.

- You want to protect your task from errors. If you use `ObjectCall()` to send a message to an object in another task, buggy code could terminate your task. If you send the same message with `ObjectSend()`, the error occurs in the recipient object’s task, leaving your task free to continue after the other task terminates.

When you send a message with `ObjectSend()`, your code’s task is suspended while waiting for the message handler to return. Your task resumes operation when the message handler returns.

`ObjectSend()` does not pass back updated message argument data. If your code requires information from any of the `Out` fields of the argument structure, it should use `ObjectSendUpdate()` (described below).

The function prototype for `ObjectSend()` is:

```c
STATUS EXPORTED ObjectSend(
    MESSAGE msg,
    OBJECT object,
    P_ARGS pArgs,
    // In only: Not updated
    SIZEOF lenArgs
);
```

`ObjectSend()` has one more argument than `ObjectCall()`. In `lenArgs`, you must specify the size, in bytes, of the data indicated by `pArgs`.

`ObjectSend()` usually copies the argument data to the address space of the task that owns the object (using `lenArgs` to determine how many bytes to copy) and then executes an `ObjectCall()` in that task. When the method returns, `ObjectSend()` returns the status value to the calling task.

If `lenArgs` is zero, the `pArgs` pointer is passed without copying the data to which it points. In this case, the data indicated by `pArgs` must be globally accessible from any task.

If the object is owned by the calling task, the `ObjectSend()` becomes a normal `ObjectCall()`. `pArgs` is not copied, but is used and modified directly.
Functions Related to ObjectSend()

The functions ObjectSendUpdate() and ObjectSendU32() behave similarly to ObjectSend(), but have been modified slightly to handle a different set of arguments or to provide different information.

ObjectSendUpdate()

ObjectSendUpdate() works just like ObjectSend() except that it copies the modified argument structure (the lenArgs bytes that pArgs points to) back to your task. This is useful when your task needs to gain access to Out fields that the message handler sets. The prototype for ObjectSendUpdate() is identical to that of ObjectSend(), except that pArgs is In/Out rather than just In:

```c
STATUS EXPORTED ObjectSendUpdate( 
    MESSAGE msg, 
    OBJECT object, 
    P_ARGS pArgs, // In/Out: Updated 
    SIZEOF lenArgs 
); 
```

ObjectSendU32()

ObjectSendU32() is is the same as ObjectSend(), except that it assumes that the lenArgs equals zero. ObjectSendU32() is slightly more efficient than ObjectSend() when you know that the argument structure to which pArgs points is globally accessible. In addition, since the data indicated by pArgs must be globally accessible, ObjectSendU32() causes the pArgs to be updated, providing the caller access to the Out fields of the pArgs.

The prototype for ObjectSendU32() is:

```c
STATUS EXPORTED ObjectSendU32( 
    MESSAGE msg, 
    OBJECT object, 
    P_ARGS pArgs // Out: updated 
); 
```

ObjectPost()

You use ObjectPost when you want to send a message to an object, but you don’t want the object to handle the message immediately. The two main reasons for using ObjectPost() are:

- You are sending a notification message to observers and aren’t concerned about when the object receives it (observers and notification messages are described in Chapter 4, Manipulating Objects).
- You want to destroy an object, but want to allow it to complete its current work before it receives the destruction message.

You can use ObjectPost() to send messages to an object in any task, including your own.
ObjectPost() places the message in the input subsystem’s input queue and returns immediately. The status value returned by ObjectPost() merely indicates whether the message was successfully added to the input queue. The message does not pass back any of the Out values in the argument structure to which pArgs points.

The PenPoint input subsystem consists of a global input queue that receives messages from device drivers (primarily the pen device driver) and other clients. When a message reaches the front of the input queue, the input subsystem uses ObjectSend() to send the message to the destination object. The destination object processes the message in its own low-level message queue. Therefore, any message sent by the input subsystem executes in the object's task, but it doesn't execute until the object's task returns to its top-level dispatch loop.

When the method returns, the status code is returned to the input subsystem, which can then handle the next message at the top of the input queue. In other words, messages in the input queue are handled synchronously; no object receives another message from the input queue until the object has completed its current work. Although messages sent with ObjectPost() are not input messages, they are processed synchronously by virtue of being posted to the input queue.

The function prototype for ObjectPost() is:

```c
STATUS EXPORTED ObjectPost ( 
    MESSAGE msg, 
    OBJECT object, 
    P_ARGS pArgs, 
    SIZEOF lenArgs 
); 
```

A related message-sending function, ObjectPostU32(), has the same relationship to ObjectPost() that ObjectSend32() has to ObjectSend(). That is, ObjectPost32() works just like ObjectPost() except that it assumes a lenArgs equal to zero. Therefore, you do not need to specify lenArgs in the function call. Also, as with ObjectSendU32(), ObjectPostU32() does not copy the argument structure to the task of the recipient object, so the data indicated by pArgs must be globally accessible. In addition, since the actual handling of the message can be indefinitely delayed, care must be taken to avoid having pArgs address memory that becomes invalid before the message handler gets to execute.

The prototype for ObjectPostU32() is:

```c
STATUS EXPORTED ObjectPostU32 ( 
    MESSAGE msg, 
    OBJECT object, 
    P_ARGS pArgs 
); 
```
Using the Message-Sending Macros

The Message-Sending Macros serve two purposes:

- Conditional macros report errors when compiled with the /DDEBUG compilation flag.
- Error-handling macros test the message return value and take special action based on whether the status indicates an error.

DEBUG Warning Macros

The names of the DEBUG-conditional macros are based on the names of the associated message-sending functions. For example, if you compile the ObjCallWarn() macro without the /DDEBUG flag, it compiles to the ObjectCall() function. If you use the /DDEBUG flag, ObjCallWarn() compiles to the ObjectCallWarning() function which logs errors. The other DEBUG-conditional macros are named according to the same pattern. For example, ObjSendWarn() compiles to ObjectSend() or ObjectSendWarning(), and ObjPostWarn() compiles to ObjectPost() or ObjectPostWarning().

Error-Checking Macros

Because most messages return a status value, you need to write code to check return values when you use the message-sending functions. To make your code more readable, CLSMGR.H defines a variety of macros that provide useful shortcuts for sending messages to an object and handling return values.

For example, ObjCallJmp() is a macro that calls ObjCallWarn(), tests the returned status value, and jumps to labelled error-handling code if the status is negative (recall that a negative status indicates an error). Take as an example the following use of ObjCallJmp():

ObjCallJmp(msgNoteShow, noteNew.object.uid, pNull, s, error);

This calls ObjCallWarn() as follows:

s = ObjCallWarn(msgNoteShow, noteNew.object.uid, pNull);

If the method returns a negative status, ObjCallJmp() jumps to the label error. The status value is returned in s.
The name of a message-sending macro indicates which message-sending function it invokes. For example, `ObjCallJmp()` invokes `ObjCallWarn()`. The letters (such as `Jmp`) after the name of the function indicate what the macros do. All of the message-sending functions have the following associated macros:

- `...Jmp()` macros jump to a specified label if the function returned an error code. Use this form to jump to error-handling code before returning. Typical error-handling activities include freeing allocated memory and destroying objects.
- `...Ret()` macros simply execute `return(s)`, where `s` is the returned status value, when the status is negative.
- `...OK()` macros return `true` if the function returns a non-negative status code.

**ObjectCall() Macros**

CLSMGR.H defines two further macros for `ObjectCall()` and `ObjectCallAncestor()` only.

- `...Failed()` macros return `true` if the function returns a negative status value.
- `...Chk()` macros also return `true` if the function returns a negative status, but they don't use the DEBUG-conditional macros and therefore never log errors.

**Using Keys**

Two fields in the `OBJECT_NEW_ONLY` structure are useful for controlling access to objects:

- `key` allows you to specify a key value when creating the object. If a client attempts to destroy (or make substantial modifications to) the object, it must use the same key value in its message. If protecting an object isn't particularly important, you can leave the default value (which is `objWKNKey`, defined as `(OBJ_KEY) 0`).
- `cap` argument allows you to establish and restrict the capabilities of the object. Capabilities are described in the next section.

Classes can control access to objects through keys. Several messages that request basic object operations such as `msgDestroy` require the use of a key (unless the capability flags that control these operations are specifically enabled). When you define messages for your own classes, you can require keys for your messages.

A key is a 32-bit value; the class manager stores a key as part of the instance data for an object. When you send a message that requires a key, you send the key in the argument structure for the message. The method that handles the message gets the key value from the argument structure and compares it to the key value in the instance data. If the values match, the class allows the requested operation to take place. If the values do not match, the method returns `stsProtectionViolation`.

The actual key value depends on how you are going to use the object and how you anticipate others might try to affect the object:
• If you want exclusive control over each object, you can use the rand() function to create a non-zero key value that is likely to be unique. You must remember this key value in some way.

• If you want any instance of your program to be able to affect objects that it creates, you can create a 32-bit identifier or use some other common value (such as your class’s method table identifier).

Capabilities

In the discussion on setting up the arguments to msgNew, we touched briefly on the object capabilities.

Capabilities are a set of flags that control some fundamental properties of objects and classes. Before the class manager actually delivers a message to an object, it checks the object’s capabilities. For example, the object capability objCapCall indicates whether an object can receive messages sent with the ObjectCall() function.

The only way to bypass a capability flag that prevents you from doing something is to use the correct key value for the object with messages that support the use of keys.

The capability flags are enumerated in the definition of OBJ_CAPABILITY in CLSMGR.H. The default capabilities for objects and classes are different.

Table 2-2 lists and briefly describes the capability flags. The sections following the table describe the capabilities in detail.

<table>
<thead>
<tr>
<th>Capability</th>
<th>Object Default</th>
<th>Class Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objCapOwner</td>
<td>true</td>
<td>false</td>
<td>Enables the object to receive msgSetOwner.</td>
</tr>
<tr>
<td>objCapFree</td>
<td>true</td>
<td>false</td>
<td>Enables the object to receive msgDestroy.</td>
</tr>
<tr>
<td>objCapSend</td>
<td>true</td>
<td>false</td>
<td>Enables the object to receive messages sent with ObjectSend().</td>
</tr>
<tr>
<td>objCapCall</td>
<td>false</td>
<td>true</td>
<td>Enables the object to receive messages sent with ObjectCall().</td>
</tr>
<tr>
<td>objCapObservable</td>
<td>true</td>
<td>true</td>
<td>Enables the object to receive msgAddObserver and msgRemoveObserver.</td>
</tr>
<tr>
<td>objCapInherit</td>
<td>n/a</td>
<td>true</td>
<td>Enables clients to create a new class that inherits from the class.</td>
</tr>
<tr>
<td>objCapScavenge</td>
<td>n/a</td>
<td>true</td>
<td>Enables the class to receive msgScavenge.</td>
</tr>
<tr>
<td>objCapCreate</td>
<td>false</td>
<td>false</td>
<td>Enables the class to receive msgObjectNew.</td>
</tr>
<tr>
<td>objCapCreateNotify</td>
<td>false</td>
<td>false</td>
<td>Directs the class manager to send msgCreated to a new object when it is fully created.</td>
</tr>
<tr>
<td>objCapMutate</td>
<td>true</td>
<td>true</td>
<td>Enables the object to receive msgMutate.</td>
</tr>
<tr>
<td>objCapProp</td>
<td>true</td>
<td>true</td>
<td>Enables the object to receive properties-related messages.</td>
</tr>
<tr>
<td>objCapUnprotected</td>
<td>n/a</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>objCapNonSwappable</td>
<td>false</td>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>
Owner Capability

An object is owned by the task that creates it. Sometimes it is useful to be able to change the owner of an object. For example, an object might maintain information required by many clients. If the client that created the object needs to go away for some reason, the client could change ownership of the object to another client that requires the object.

To change the ownership of an object, you send msgSetOwner to the object. The objCapOwner flag specifies whether the object can receive msgSetOwner.

If objCapOwner is true, anyone can change the ownership of the object; if it is false, the only clients who can change ownership of the object are those that know the key value for the object.

The default value for objCapOwner is true for objects and false for classes.

Freeing Capability

You destroy an object and free its memory by sending msgDestroy to the object. The class manager handles msgDestroy and checks the object's objCapFree capability. If the objCapFree is true, any client can destroy the object; if it is false, only the clients that know the key for the object can destroy it.

The default value for objCapFree is true for objects, and false for classes.

ObjectSend() Capability

The objCapSend capability lets an object receive messages sent with the ObjectSend() function, which executes the message handler in the task of the recipient object. When objCapSend is true, the object can receive messages sent with ObjectSend().

The default value for objCapSend is true for objects, and false for classes.

ObjectCall() Capability

The objCapCall capability lets an object receive messages sent with the ObjectCall() function, which executes the message handler code in the task of the message sender. When objCapCall is true, the object can receive messages sent with ObjectCall() from tasks other than the object's owning task.

The default value for objCapCall is false for objects and true for classes.

When a task sends a message to an object that it owns, objCapSend and objCapCall are both assumed to be true. If both objCapSend and objCapCall are false, no task other than the owner of the object can talk to the object.
**Observable Capability**

For each object, the class manager maintains a list of observers. A client can add itself an object's observer list by sending `msgAddObserver` to the object; a client can remove itself by sending `msgRemoveObserver`. When an object wants to send a message to its observers, it sends `msgNotifyObservers` to itself and the class manager sends the message specified in the arguments to all of the observers on the observer list. Observer messages are described in more detail in Chapter 4, Manipulating Objects.

The object capability `objCapObservable` specifies whether an object can receive `msgAddObserver` and `msgRemoveObserver`. If `objCapObservable` is true, the object can receive both messages. If false, the Class Manager returns `stsProtectionViolation` in response to both messages.

The default `objCapObservable` value is true for both objects and classes.

**Inheritance Capability**

You make a class inherit from another by specifying the superclass when you create the subclass. However, there are times when you do not want your class to be subclassed (this is true for some system classes).

The capability `objCapInherit` specifies whether a class can be subclassed or not. If `objCapInherit` is true for a class, clients can create subclasses that inherit from that class. If `objCapInherit` is false for a class, clients will receive `stsProtectionViolation` when they try to create a subclass of the class.

This capability is meaningful only for classes. The default value for a class is true.

**Scavenging Capability**

When a task terminates, the class manager cleans up all existing classes and their objects in a process called scavenging. The Class Manager sends `msgScavenge` to every object that the terminating task owns. The details of scavenging are covered in Chapter 4, Manipulating Objects.

`objCapScavenge` applies to an entire class, not to individual objects. If a class has `objCapScavenge` set to true, its instances will process `msgScavenge` normally. If a class has `objCapScavenge` set to false, its instances will return `stsProtectionViolation` in response to `msgScavenge`.

This capability is meaningful only for classes. The default value for a class is false.
**Creation Capability**

A client can ask an object in another task to create a new object by sending `msgObjectNew` to the object. If the object has `objCapCreate` set to `true`, it can receive `msgObjectNew`.

The default value for this capability is `false` for both objects and classes.

**Creation Notification**

Sometimes an object needs to know when the class manager has created it, before any clients can send messages to the object, so that it can perform some additional work. If `objCapCreateNotify` is set to `true` before sending `msgNew`, the class manager will send `msgCreated` to the object when it is fully created, giving the object an opportunity to respond by performing its additional work.

If you want a specific object to receive notification, you should set `objCapCreateNotify` after sending `msgNewDefaults` and before sending `msgNew`. If you specify `objCapCreateNotify` for a class, the class will receive `msgCreated` when it is fully initialized.

If you want all objects created by a class to receive notification, the class must set `objCapCreateNotify` to `true` when it handles `msgNewDefaults`.

`msgCreated` is described in Chapter 4, Manipulating Objects.

**Mutation Capability**

A client can change the class from which an object inherits by sending it `msgMutate`. `msgMutate` is described in full in Chapter 4, Manipulating Objects.

The capability `objCapMutate` specifies whether a class can be mutated or not. If `objCapMutate` is `true` for a class or for an object, clients can send `msgMutate` to the object. If `objCapMutate` is `false`, clients will receive `stsProtectionViolation` when they try to mutate the object.

**Properties Capability**

Every object has an associated list of properties, data elements attached to the object. Clients set an object’s properties with `msgSetProp`, and read them with `msgProp`. If an object’s `objCapProp` is set to `true`, any object can send `msgProp` and `msgSetProp` to the object. If `objCapProp` is set to `false`, clients cannot access the object’s properties without the object’s key.

The default value for `objCapProp` is `true` for both classes and objects.
**Checking Capabilities**

To check specific capabilities of a particular object, you send `msgCan` to the object. `msgCan` takes an `OBJ_CAPABILITY` value that specifies a group of capability flags that you want to check. If the specified combination of capability flags is enabled for the object, `msgCan` returns `stsOK`; otherwise it returns `stsProtectionViolation`. For example, the following code fragment checks an object to determine whether it has both the `objCapObservable` and `objCapSend` capability flags set.

```c
OBJ_CAPABILITY objCaps;
STATUS s;
OBJECT someObject;
objCaps = objCapObservable | objCapSend;
s = ObjectSend(msgCan, someObject, objCaps);
if (s = stsOK)
    ...
else
    ...
```

**Changing Capabilities**

To enable or disable an object's capabilities, you send the message `msgEnable` or `msgDisable` to the object. The messages take a pointer to an `OBJ_CAPABILITY_SET` structure that contains:

- `key` the key for the object that you want to change.
- `cap` an `OBJ_CAPABILITY` value that contains the capabilities that you want to enable or disable.

If `key` does not match the key value for the object, the message fails with `stsProtectionViolation`. 
Chapter 3 / Creating a New Class

There are two main reasons why you might need to implement a new class:

- You are building a new application. All applications are classes that inherit from clsApp, the application class. Application classes are described in detail in Part Two: Application Framework.

- You need a class that provides functionality that is not available in any existing classes. Usually you can find a class that has most of the behavior that you require. You can then create a new class that is a subclass of that class, reducing the amount of code you must write and test yourself.

This section describes how to write the skeleton for a new class. It is primarily concerned with the organization of the fundamental parts of a class, how the Class Manager routes messages to a particular method, the parameters involved in invoking a method, and the data maintained for each instance of a class. This section does not describe how to actually implement the methods; that aspect of classes is described in the PenPoint Application Writing Guide.

Overview

The next sections describe the different parts that you must create to implement a new class and how the parts interact when you install the class.

The Parts of a Class

When you create a new class, you must create:

- A header file (.H file) that defines the new messages that your class implements.

- A method table. When the Class Manager sends a message to your class, it uses the method table to locate the method for that message. You compile the method table with the PenPoint method table compiler, shipped with the Software Developer's Kit (SDK) as \PENPOINT\SDK\UTIL\CLSMGR\MT.EXE.

- A C code file that contains a method for each message that your class and its instances will handle. A method is a C function that defines the class's behavior for each message. Because methods are the functions with which classes handle incoming messages, you'll sometimes hear methods referred to as message handlers.

A message identifier is defined only once (by a single header file), but there can be any number of handlers for that message, written by any class that wants to handle the message. msgNewDefaults is the classic example of such a message.
Figure 3-1 shows the parts of a class and how they relate to one another. You can find many source code examples for classes in the SDK directory `\PENPOINT\SDK\SAMPLE`.

### Method Table and Class Implementation

#### yourclass.h

```c
#define clsYourClass MakeWKN(...)    
#define msgDoSomething MakeMsg(...)    
```

#### yourmeth.tbl

```c
#include "yourclass.h">

MSG_INFO yourClassMethods[] = {
    ...  
    msgDoSomething, "DoSomething", 0, 0
};

CLASS_INFO classInfo[] = {
    "clsYourClassTable", yourClassMethods, 0, 0
};
```

#### yourclass.c

```c
#include "yourclass.h">
#include "yourmeth.h"> // Created by MT

MSG_HANDLER DoSomething (params)
{
    // method for DoSomething
}

STATUS FAR clsyourClassInit (void)
{
    ...  
    new.object.uid = 'clsYourClass';
    new.class.pMsg = clsYourClassTable;
    ObjectCall(msgNew, clsClass, &new);
    ...  
}
Installation Summary

There are two ways to link a class: you can link a class with an application to create an application executable file, or you can link a class into a DLL file. You use DLL files when multiple clients may want to use your class.

In an application, you create your class by calling a class initialization routine from your application's main() routine. In a DLL, you create the class by calling a class initialization routine from the DLLMain() routine.

When you have compiled a class into an executable or a DLL, you use the Installed Software section of the PenPoint Settings notebook to copy it to the PenPoint computer. The Settings notebook invokes main() or DLLMain(), which then calls the class initialization routine.

In your class initialization routine you install your new class by sending msgNew to clsClass and specifying the new class in the message arguments.

Design Considerations

When you determine that you need to create a new class, you must also ask yourself several questions about the design of your class. As you read the rest of this chapter, keep these questions in mind:

• Do you want to maintain instance data in the object itself, or do you want to maintain pointers to external instance data?
• Should the class be global? Should it be well known?
• Should you handle messages defined by your ancestors, or should you pass them up to your ancestor?
• When you find that several methods require the same subroutine, do you implement it as a function, or do you implement it as a new message and a new method?

The advantages and disadvantages of these techniques are described throughout the rest of this chapter.

To keep a high level of performance, you should avoid structuring your class so that each message substitutes for a single function call from a procedural model of your class. Passing a message using the Class Manager takes approximately two to three times the amount of time as an equivalent function call.

From an object-oriented point of view, it is a good idea to send messages between objects, but you should structure your messages and objects so that they operate on large chunks of application data.

For example, you might want to create a raster object that controls a screen region, but a raster line should not be an object to which you send messages like msgDraw.
Creating Objects

A client creates an instance of your class by sending msgNewDefaults and msgNew to your class. However, in passing msgNew, the Class Manager handles the message and passes msgInit to your class instead. In short, when your class receives msgInit, it is being asked to initialize a new instance of itself (the Class Manager has already created the object).

Usually you handle msgInit by initializing your instance data (described below) creating objects that an instance of your class requires to do its work. For instance, a graphic editor class maintaining a list of shapes could create an instance of clsList when it handles msgInit.

Instance Data

A class consists of a group of methods; the code for these methods is stored in a shared area of memory. When an instance of a class receives a message, the corresponding method handles the message.

However, each instance has its own, separate instance data. This instance data is actually owned by the Class Manager and is available only to the instance. Each ancestor of an object's class provides part of the instance data for the object. When a class is created, it tells the Class Manager how much space needs to be added to the object for the class to keep its part of the instance data.

Memory Protection

The Class Manager maintains the instance data in memory that is protected from writes by unauthorized clients. Essentially only the Class Manager can write instance data (although you can request the Class Manager to write a specific piece of data for you). This restriction protects clients and subclasses from one another—no program can overwrite another program's instance data.

Allocating Instance Data

When the Class Manager creates a new instance of a class, it allocates a specific amount of memory for the instance data. You specify the size of the instance data when you install your class (by sending msgNew to clsClass).

Usually you create a structure called something like XXX_INSTANCE_DATA (where XXX is the class name), and use the sizeof() function to determine its size. The maximum size for all of the instance data for all of the subclasses of an object is slightly less than 64K.
CHAPTER 3 / CREATING A NEW CLASS

Instance Data 35

Accessing Your Instance Data

When the Class Manager invokes a method (a function that you define to handle a message), it passes a pointer to your instance’s instance data (pData). You can read the instance data and copy it to your local storage, but you cannot write the instance data directly.

To store instance data, you must call the ObjectWrite() function. The prototype for the ObjectWrite() function is:

```c
STATUS EXPORTED ObjectWrite(
    OBJECT self, 
    CONTEXT ctx, 
    P UNKNOWN pData 
); 
```

In brief, self is the UID of the object whose instance data you wish to access, ctx maintains information about the ancestor hierarchy, and pData is a pointer to the local copy of the instance data.

Maintaining Dynamic Instance Data

If you need variable-sized instance data, you can use OSHeapBlockAlloc() to allocate data from your own process heap. You can store a pointer to the heap in your instance data, instead of storing the entire block. This also allows faster write access to your instance data, at the risk that another object in your process might overwrite your data.

On the other hand, the window system cannot risk data corruption—it stores all of its data in write-protected memory.

Saving and Restoring Instance Data

clsObject defines messages that tell objects to save and restore their instance data. When an object receives msgSave, it must file its instance data; when an object receives msgRestore, it must restore its instance data.

When an object receives msgSave, it uses the pointer to its instance data (pData) when it writes its instance data.

Part Two: PenPoint Application Framework describes in detail how objects save and restore their instance data when they receive the clsObject messages msgSave and msgRestore.
Creating a Header File

The header file for your class should contain `#defines` for all your well-known UIDs, messages, tags, and status values.

You can use the header files (.H files) in `\PENPOINT\SDK\INC` as an example of how you should write and organize your header files.

For documentation purposes your header files can also list (but shouldn't define) messages that other classes define, but for which your class provides an alternate or enhanced message handler. The actual definitions for those messages will come from the header files that you include at the beginning of your header file.

Ancestor Calls

When an object receives a message, it has the option of sending the message to its ancestor. As described earlier, the task handling the message remains the same and the object is still the same. The only difference is that the method belongs to the ancestor.

The ancestor call is simply a function call to the method defined by the ancestor. You use the ancestor call functions, because the Class Manager knows your ancestor already. This makes it possible to execute code defined by another class without having to rely on knowledge of the other class's structure.

ObjectCallAncestor()

You use `ObjectCallAncestor()` in a method to send a message to the ancestor of your class. `ObjectCallAncestor()` uses `ctx` to determine the class of the object that called it, finds the ancestor's method for the message, and then calls the method with the `pArgs` value. The function prototype is:

```c
STATUS GLOBAL ObjectCallAncestor(
    MESSAGE   msg,
    OBJECT     self,
    P_ARGS     pArgs,
    CONTEXT    ctx
);
```

The arguments to the function are identical to those of `ObjectCall()`, except for the addition of a `CONTEXT` value.

The Class Manager uses `CONTEXT` value `ctx` to locate the correct ancestor. Because `ObjectCallAncestor()` is called from a method, you will have received a `ctx` value in your method parameters.

ObjectCallAncestorCtx()

`ObjectCallAncestorCtx()` is a shorter version of `ObjectCallAncestor()`. It does exactly the same thing as `ObjectCallAncestor()`, but the Class Manager uses the same message and `pArgs` used to invoke the method. You have to specify only your `ctx`. The function prototype is:

```c
STATUS GLOBAL ObjectCallAncestorCtx(
    CONTEXT    ctx
);
```
Creating the Methods

This section describes how to write a method. While you read through this section, refer to the example code for Tic-Tac-Toe in \PEN\SAMPLE\TTT in the PenPoint Software Development Kit.

Declaring Entry Points for Methods

When the Class Manager sends a message to an instance, the Class Manager uses the method table to locate the entry point for the procedure that corresponds to the message. The procedure is called a method or message handler (the terms are synonyms). A single method can handle multiple messages; this allows one method to respond to all messages defined by a class and also avoids unnecessary replication of common code across the handling of several messages.

You must explicitly declare all methods. The Class Manager defines the symbol MSG_HANDLER to identify message handling procedures. MSG_HANDLER and all the macros described in this section are defined in CLSMGR.H. MSG_HANDLER expands to STATUS CDECL. Using the symbol MSG_HANDLER makes your code much easier to read and search.
Message Parameters
When the Class Manager sends a message to your method, it always passes along the same five parameters:

- **msg** the message.
- **self** the UID of the object to which the message was sent.
- **pArgs** the 32-bit value that indicates the client's arguments.
- **ctx** the context, required for calls by the method to ObjectWrite() and ObjectCallAncestor().
- **pData** a pointer to your object's instance data.

These parameters are described in the next five sections.

The msg Parameter
The msg parameter contains the value that identifies the message that you received. This is one of the three parameters specified by the client that sent the message to your object.

You use the msg parameter most frequently when you want to send the message to your ancestor from within your method.

You can also use this parameter to do specific processing for messages, especially if you have a wildcard method that needs to do special work for selected messages.

The self Parameter
The self parameter contains the UID of the object that was originally sent the message. This is the second of the three parameters specified by the client that sent the message to your object.

If an object needs to send a message to itself, the method uses self to identify the object executing the method. To forward the message to the object's ancestor class, use ObjectCallAncestor(); using ObjectCall() results in an infinite loop. Otherwise, use ObjectCall() or ObjectSend() to provide subclasses a chance to modify the message behavior or to simply notice that the message was sent.

The pArgs Parameter
The pArgs parameter contains the 32-bit argument value specified by the client. This is the third of the three parameters specified by the client that sent the message to your object.

The pArgs parameter is typically a pointer to data, but in some cases is the argument data (for example, when the only data required is a 32-bit value). When you design a method, you must decide what argument data it requires. You use the header file to document what data is required by each message.

The ctx Parameter
The ctx parameter contains a CONTEXT structure that maintains information about where the current class is in relation to the class hierarchy. The ctx
parameter is created and maintained by the Class Manager. ctx is used primarily by ObjectCallAncestor(), ObjectRead() and ObjectWrite().

**The pData Parameter**

The pData parameter contains a pointer to the instance data for the object.

As described already in the section on instance data, the Class Manager maintains the instance data for each instance separately in write-protected memory.

If you will need to update your instance data, you must copy the instance data to your local memory, update the instance data, then write the instance data back with ObjectWrite().

**Method Declaration Macros**

Many times when you define a method, you use the same declarations for all or most of the parameters. To free you from declaring all these parameters, and to make your code more readable, the Class Manager defines a number of macros that declare the procedure and its parameters.

Additionally, there are macros to manipulate the instance data pointer pData in your methods.

Table 3-1 lists the method macros.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MsgBoxHandler()</td>
<td>Declares all parameters.</td>
</tr>
<tr>
<td>MsgBoxHandlerArgType()</td>
<td>Allows you to specify a different type for pArgs.</td>
</tr>
<tr>
<td>MsgBoxHandlerWithTypes()</td>
<td>Allows you to specify different types for pArgs and pData.</td>
</tr>
<tr>
<td>MsgBoxHandlerParametersNoWarning()</td>
<td>Suppresses warning messages about unused parameters.</td>
</tr>
</tbody>
</table>

**Using the MsgBoxHandler() Macro**

You use the MsgBoxHandler() macro when you do not need to explicitly define types for the pArgs or pData parameters when you use them. You might do this when your method simply issues a debug statement and returns (you might have specified objCallAncestorBefore or objCallAncestorAfter in your method table).

The syntax for the MsgBoxHandler() macro is:

```
MsgHandler(fn)
```

fn is the name of the message handler. For example:

```
MsgHandler(XferListMsgNewDefaults) // Abbrev. parameters
{
    STATUS s;
    ((P_XFER_LIST_NEW)pArgs)->object.cap |= objCapCall;
    return stsOK;
    // Prevent compiler warnings for unused parameters
    MsgBoxHandlerParametersNoWarning;
} /* XferListMsgNewDefaults */
Using MsgHandlerArgType()

You use the `MsgHandlerArgType()` macro when you want to explicitly cast the type of the pArgs parameter in the method declaration.

The syntax for `MsgHandlerArgType` is:

```c
MsgHandlerArgType(fn, pArgsType)
```

`fn` is the name of the message handler and `pArgsType` is the type to which to cast pArgs. `pArgsType` must be a pointer type. In the following code example, the type for pArgs is `P_XFER_LIST_NEW`:

```c
MsgHandlerArgType(XferListMsgNewDefaults, P_XFER_LIST_NEW)
{
    STATUS s;
    pArgs->object.cap |= objCapCall;
    return stssOK;
    // Prevent compiler warnings for unused parameters
    MsgHandlerParametersNoWarning;
} /* XferListMsgNewDefaults */
```

Using MsgHandlerWithTypes()

You use the `MsgHandlerWithTypes()` macro when you want to explicitly cast the types of both the pArgs and pData parameters in the method declaration. The syntax for `MsgHandlerWithTypes()` is:

```c
MsgHandlerWithTypes(fn, pArgsType, pDataType)
```

`fn` is the name of the message handler, `pArgsType` is the type for pArgs, and `pDataType` is the type for pData. Both `pArgsType` and `pDataType` must be pointer types. In this example, the type of pArgs is `P_OBJ_SAVE` and the type of pData is `PP_TTT_APP_INST`:

```c
MsgHandlerWithTypes(TttAppSave, P_OBJ_SAVE, PP_TTT_APP_INST)
{
    TTT_APP_FILED_0 filed;
    STATUS s;
    DbgTttAppSave(""");
    ...
```

Using MsgHandlerParametersNoWarning()

Some C compilers generate warnings when you declare identifiers but don’t use them in the function where they are declared. The method macros described in this section declare all five parameters passed by the Class Manager. If you do not reference one of these parameters, the compiler may generate a warning message.

To suppress these messages, include the macro `MsgHandlerParametersNoWarning()` in your method. The macro simply adds references to all of a method’s parameters that will satisfy the compiler without generating any unnecessary code.

```c
msg, self, pArgs, ctx, pData;
```

By mentioning all the parameters, the macro effectively suppresses the warning messages.
**Operations of a Method**

The actual processing performed by each method is up to you. You can create objects, send messages to objects, process data, or whatever to accomplish the method's purpose.

Essentially the rest of this volume (the *PenPoint Architectural Reference*) is dedicated to describing the things that you can do from a method.

The *PenPoint Application Writing Guide* explains how to write methods. For examples of methods, see the example code in the \PENPOINT\SDK\SAMPLE directory.

**Creating a Method Table**

A method has three options when it receives a message:
- Pass it up the ancestor hierarchy to be processed.
- Perform the requested service.
- Reject the message with no action.

Each class contains a method table responsible for handling all messages sent to instances of that class. Method tables allow you to specify the procedure in your C code that handles each message and whether the Class Manager should send the message up the class hierarchy either before or after your procedure handles the message.

**Method Table Overview**

When you design a class, you must define the messages that the class will process and what methods will be used during the processing. These definitions are implemented in a method table that is compiled as a separate object file and attached at link time. When the class is installed (at cold boot, or by the Settings Notebook) the method table is installed as part of the class into the Class Manager database.

A method table definition file (with the extension .TBL) contains the tabular definition for your class's messages and methods. There is an entry in the table for each message that your class will handle. For each message you specify:
- The message token (defined in a header file)
- The method to be executed when an instance of your class receives this message
- A flag field that specifies options for memory or speed optimization, and whether the ancestor of the class should process the message before or after the class does, or not at all.
The table contains entries only for those messages that your class intercepts and
processes. If the Class Manager attempts to send a message to an instance of your class
and the message is not defined in your method table, the Class Manager automatically
sends the message to your class's ancestors. If no class in the ancestry chain is able to
handle the message, the message eventually gets to clsObject. If clsObject cannot
handle the message, it returns stsNotUnderstood to the sender of the message.

Your method table is not limited to the messages defined by one of your classes. You
can create method table entries for any message that an instance of your class might
receive (no matter who defines the message). This is known as overriding a message.

If you override a message, you must be sure to include in your method table
definition file the header file that contains the #define for the message. Otherwise,
the reference to the message will fail.

You compile the method table with the PenPoint method table compiler
([PENPOINT\SDK\UTIL\CLSMGR\MT.EXE]), then you link the resulting method
table object file with the object file that defines the methods for the class.

There are five steps to setting up your method tables:
1. Create a method table definition file (a .TBL file).
2. Compile the table with MT.EXE.
3. Define the methods in your class's C code file with the MSG_HANDLER
   identifier.
4. Add the method table's identifier to your class's msgNew initialization
   method.
5. Compile your class's C code file and link it with the method table.

Figure 3-2 shows the build cycle and file dependencies for an application that
includes method tables.

Creating a Method Table Definition File
A method table definition file contains the definitions of at least two arrays:
- A MSG_INFO array that maps message names to method names. Each class
  has exactly one associated MSG_INFO array.
- A CLASS_INFO array that contains one entry for each class. The CLASS_INFO
  array associates each MSG_INFO array with a symbol that you use to identify
  the method table when you install your class (part of the msgNew arguments
to clsClass).

Figure 3-1 shows the relationship between the principle elements in a method
table definition file and the source file in which you implement your class. In the
figure you can see how the method table defines the message name and the entry
point of the method called by the Class Manager. The figure also shows where the
method is defined in the C file that implements your class and also shows the
method that installs your class.
The MSG_INFO Array

The MSG_INFO array identifies the messages handled by a particular class and specifies for each message the method that handles the message. For each class that you implement in an executable file, you must define one MSG_INFO array. Thus, if you defined three classes in a single executable file, the method table definition file must have three MSG_INFO arrays.
You can see a number of examples of method tables in the sample application directories under \PENPOINT\SDK\SAMPLE. Here is an abbreviated example of the MSG_INFO array for clsTttData from TTTTBL.TBL:

```c
MSG_INFO clsTttDataMethods[] = {
    msgNewDefaults,         "TttDataNewDefaults",  objCallAncestorBefore,         
    msgInit,                "TttDataInit",          objCallAncestorBefore,         
    msgFree,                "TttDataFree",          objCallAncestorAfter,          
    msgTttDataGetMetrics,   "TttDataGetMetrics",  0,                     
    msgTttDataSetMetrics,   "TttDataSetMetrics",  0,                     
    msgTttDataSetSquare,    "TttDataSetSquare",  0,                     
    msgTttDataRead,         "TttDataRead",         0,                     
    msgUndoItem,            "TttDataUndoItem",     0,                     
};
```

The identifiers MSG_INFO and objCallAncestorBefore are defined in CLSMGR.H.

Each entry in the array has 3 fields:

- The message name.
- The name of the method for the message. The method name identifies the entry point for the method. The method name must be in quotes.
- One or more options flags.

Table 3-2 lists the option flags and their meanings. The options flags can be OR’d together. Some options may not make much sense when combined. For example, it is impossible to optimize memory usage with the save space option and at the same time optimize performance with the save time option.

<table>
<thead>
<tr>
<th>Option Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>objCallAncestorBefore</td>
<td>Call the object’s ancestor before calling the method.</td>
</tr>
<tr>
<td>objCallAncestorAfter</td>
<td>Call the object’s ancestor after calling the method.</td>
</tr>
<tr>
<td>objClassMessage</td>
<td>Handle the message when sent to the class.</td>
</tr>
<tr>
<td>objSaveSpace</td>
<td>Optimize this entry for space. This option is not implemented.</td>
</tr>
<tr>
<td>objSaveTime</td>
<td>Optimize this entry for time.</td>
</tr>
</tbody>
</table>

If you do not use any of these options, you must specify 0 as a placeholder in the array.

### CLASS_INFO Array

The CLASS_INFO array specifies which MSG_INFO array belongs to which class. In a method table definition file there is only one CLASS_INFO array.

This example shows the CLASS_INFO array from TTTTBL.TBL:

```c
CLASS_INFO classInfo[] = {
    "clsTttViewTable", clsTttViewMethods, 0, 
    "clsTttDataTable", clsTttDataMethods, 0, 
    "clsTttAppTable", clsTttAppMethods, 0, 
};
```
Each entry in the CLASS_INFO array has 3 fields:

- a method table name that you will use when you install the class
- The name of a MSG_INFO struct defined in this method table definition file
- An options flag.

The options flag is reserved for future use; put a 0 in this field as a placeholder.

Method Table Wildcards

The MSG_INFO array can contain wildcard message names. Wildcards match any message within a given set of messages and call the associated method.

There are two types of wildcards:

- those that match all messages defined for a given class
- those that match all messages sent to instances of a given class (essentially you use this as a match-all-others in your MSG_INFO array)

A single MSG_INFO array can contain either type of wildcard, or both types. For example, a MSG_INFO array might contain both a wildcard that matched all messages for one of the classes that it handles and, at the end of the MSG_INFO array, a wildcard for all messages sent to instances of the class for which the MSG_INFO array is defined.

To create a wildcard that matches all messages for a class, you must use the MakeMsg() macro to create a message that uses your class (as usual) and has the special message number objWildCard. For example, to create a wildcard (fooWildCard) that matches all messages of clsFoo, you would use:

```c
#define fooWildCard MakeMsg(clsFoo, objWildCard)
```

To use the class wildcard, insert the message in the MSG_INFO array:

```c
#define fooWildCard MakeMsg(clsFoo, objWildCard)
MSG_INFO foo = {
  fooWildCard, "HandleAllFooMsgs", 0,
  0
}
```

To match messages sent to objects created by a class, add a line to the end of the MSG_INFO array that uses objWildCard as the message name and specifies a method as usual. For example:

```c
MSG_INFO foo = {
  msgNew, "HandleMsgNew", 0,
  ...
  objWildCard, "HandleAllOtherMsgs", 0,
  0
}
```

Compiling a Method Table

To compile a method table you need

- The C compiler (including DOS4GW.EXE)
- The method table compiler, MT.EXE.
The method table compiler is delivered with the PenPoint SDK in the directory \\PENPOINT\SDK\UTIL\MT.

### Running MT

Usually you compile your method table simply by running the method table compiler, MT.EXE. Before you run the method table compiler, you must make sure that MT.EXE is in your local directory or your path includes the directory that contains MT.EXE.

The run command for the method table compiler has the following syntax:

```
MT infile
```

If the input file has a .C or .TBL extension, the method table compiler:

- Compiles your method table with the C compiler, which results in an intermediate object file
- Processes the resulting object file to produce the method table object file and corresponding header file.

### Compiling Method Tables With Wildcards

When you compile method tables that use wildcards, the method table compiler uses the assembler.

You must specify a path that includes the assembler during the compile phase for method tables if you use wildcards. The simplest strategy is to put the assembler in the same directory as the compiler executable files.

### MT Output Files

The method table compiler produces a header file and an object file. These files have the same name as the .TBL file, but with the extensions .H and .OBJ, respectively. If you already have a file with the same name and the extension .H, the method table compiler will overwrite it. For example, FOO.TBL will generate FOO.H and overwrite any previous version of FOO.H.

The header file contains the declarations for each of the methods. You must #include this file in the sources that define the methods for your class.

The object file contains the method table code used by the Class Manager when it routes a message to an instance of your class.

### Compiling in Two Steps

Although the method table compiler will run the C compiler for you, if you run MT from MAKE, you may run out of memory. If this happens, you should compile your method table in two steps:

1. Use the C compiler to compile your method table.
2. Run MT on the resulting intermediate object file to create the method table object and header files. If MT does not detect a .C or .TBL extension, it does not run the C compiler.
Installing a Class in PenPoint

The way that you compile and link your class affects the way in which PenPoint finds the routine used to install your class. You can compile and link your classes one of two ways:

- If your class is part of an application and will not be used by any other clients, you can link your class with your application to create an application executable file.
- If your class might be used by a number of clients (including your application), you can link your class separately to create a DLL file.

Your class must contain some code that sends msgNew to clsClass to install your class. If the class is part of your application executable file, you install your class when you install your application (that is, in main() when process equals 0).

If the class is in a DLL file, you install the class from the DLLMain() routine. If your application requires a class in a DLL file, it can name the DLL file in the application installation .DLC file. For more information on DLL and .DLC files, see PenPoint Application Writing Guide.

The Class Initialization Routine

The purpose of the class initialization routine is to send msgNew to clsClass. clsClass is a special class that exists solely to install new classes.

Before you send msgNew to clsClass, you must create, initialize, and modify a CLASS_NEW structure that describes the new class. You initialize the CLASS_NEW structure by sending msgNewDefaults to clsClass.

The CLASS_NEW structure contains an OBJECT_NEW_ONLY structure (object) and a CLASS_NEW_ONLY structure (cls). The OBJECT_NEW_ONLY structure contains:

- uid the well-known UID that you will use for this class.
- key a key value for the class.
- cap the capabilities for the class.

The CLASS_NEW_ONLY structure contains:

- pMsg a pointer to the method table for your class. The method table name is the first field in the CLASS_INFO entry that you created in your method table definition file. If this is an abstract class (one that defines messages but has no methods), you can specify pNull.
- ancestor the UID of the class that is the immediate ancestor of this class.
- size The size of the instance data for instances of this class. The size of instance data must be less than 64K bytes. If the size is not 0, you should usually have methods for msgSave and msgRestore to handling filing of the instance data.
The instance data can be a pointer to a heap where the data is stored. Pointers are useful when you do not know what the exact size of the data for a given instance will be.

**newArgsSize**  The size of the _NEW structure used to create instances of this class. It is usually set by SizeOf(..._NEW).

When you send **msgNew** to **clsClass**, the message returns a status value. If the Class Manager creates the class successfully, **msgNew** returns **stsOK** and the **object.uid** field contains the UID of the new class.

### Example 3-2

**Creating a New Class**

This example from \PENPOINT\SDK\SAMPLE\TTT\TTT\DATA.C shows how the routine **ClsTttDataInit()** creates the class **clsTttData**. **TTTPRIY.H** defines the token **clsTttData**, and **TTTBL.TBL** establishes **clsTttDataTable** as the identifier for **clsTttData**'s method table.

```pascal
STATUS PASCAL
ClsTttDataInit (void)
{
    CLASS NEW new;
    STATUS s;
    ObjCallJmp(msgNewDefaults, clsClass, &new, s, Error);
    new.object.uid = clsTttData;
    new.object.key = 0;
    new.cls.pMsg = clsTttDataTable;
    new.cls.ancestor = clsObject;
    new.cls.size = SizeOf(P_TTT_DATA_INST);
    new.cls.newArgsSize = SizeOf(TTT_DATA_NEW);
    ObjCallJmp(msgNew, clsClass, &new, s, Error);
    return stsOK;
}
```

**Error:**

```pascal
return s;
} /* ClsTttDataInit */
```
Chapter 4 / Manipulating Objects

When you have created an object you tell it to do things by passing messages to it. Most of these messages are defined by the class that created the object.

Additionally, the PenPoint™ Class Manager provides a number of messages that provide class manager support for those objects. These messages fall into four categories:

- Copying objects.
- Observing objects and object notification.
- Getting object status information.
- Destroying objects.

Copying Objects

Perhaps the most important object manipulation is the ability to copy objects. The PenPoint Class Manager provides a low-level facility to copy objects. When copying an object, all objects owned by the object are copied at the same time.

Copying objects can be done by either the client that owns the original object or the client that will receive the copied object. The original owner and the receiver do not have to be in the same process.

The object copy facility is used by many parts of the PenPoint operating system. For instance, the embedded window move/copy protocol uses object copy to copy objects from one window to another.

While there are two messages used in copying objects, msgCopy and msgCopyRestore, clients only need to send msgCopy to the object to be copied. When clsObject handles msgCopy it sends msgCopyRestore to the destination object. Clients should almost never have to send msgCopyRestore.

Using msgCopy

To copy an object, send msgCopy to the object. The object being copied must be able to file itself, that is, it must respond to msgSave and msgRestore.

msgCopy takes a pointer to an OBJ_COPY structure. The requestor field specifies the UID of the object that will receive the copied object. If requestor is in a different process from the client sending msgCopy, the requestor must have objCapCall set on.

When you send msgCopy to an object, clsObject creates a temporary resource file and sends msgSave to the object. After the object saves itself, clsObject sends msgCopyRestore to the object that you specified in requestor.
When an object receives **msgCopyRestore**, it passes the message to its ancestor. When clsObject receives **msgCopyRestore**, it creates the new object that is the copy and sends it **msgRestore**.

After the requestor restores the object, **msgCopy** closes and destroys the temporary file and passes back the UID of the new object in the **object** field of the **OBJ_COPY** structure.

### Observing Objects

All objects can maintain a list of observers. An observer is an object that wants to be notified when the observed object changes state.

The class manager creates and maintains the list of observers for each object.

Table 4-1 lists the observer messages that are described in the sections following the table. All of these messages are defined in CLSMGR.H.

#### Table 4-1
**Observer Messages**

<table>
<thead>
<tr>
<th>Message</th>
<th>pArgs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAddObserver</td>
<td>OBJECT</td>
<td>Adds an observer to the end of the object’s observer list.</td>
</tr>
<tr>
<td>msgAddObserverAt</td>
<td>P_OBJ_OBSERVER_POS</td>
<td>Adds an observer at the specified position in the observer list.</td>
</tr>
<tr>
<td>msgRemoveObserver</td>
<td>OBJECT</td>
<td>Removes an observer from the object’s observer list.</td>
</tr>
<tr>
<td>msgNotifyObservers</td>
<td>P_OBJ_NOTIFY_OBSERVERS</td>
<td>Sends a message to the observers.</td>
</tr>
<tr>
<td>msgPostObservers</td>
<td>P_OBJ_NOTIFY_OBSERVERS</td>
<td>Posts a message to the observers.</td>
</tr>
<tr>
<td>msgEnumObservers</td>
<td>P_OBJ_ENUM_OBSERVERS</td>
<td>Passes back the observer list.</td>
</tr>
<tr>
<td>msgGetObserver</td>
<td>P_OBJ_OBSERVER_POS</td>
<td>Passes back the observer at the specified position in the observer list.</td>
</tr>
<tr>
<td>msgNumObservers</td>
<td>P_U16</td>
<td>Passes back the number of observers for this object.</td>
</tr>
</tbody>
</table>

#### Observer Notification

<table>
<thead>
<tr>
<th>Message</th>
<th>pArgs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAdded</td>
<td>OBJECT</td>
<td>Sent to the observer when it is added to an object’s observer list.</td>
</tr>
<tr>
<td>msgRemoved</td>
<td>OBJECT</td>
<td>Sent to the observer when it is removed from an object’s observer list.</td>
</tr>
<tr>
<td>msgChanged</td>
<td>OBJECT</td>
<td>Generic message that can be used to notify observers that a change has occurred.</td>
</tr>
</tbody>
</table>

### Adding an Observer

If you want to observe an object, you send **msgAddObserver** to the object that you are interested in. The only argument for the message is the UID of the object that will receive notifications; usually this is **self** (your object). To add an observer to a specific location in an observer list, use **msgAddObserverAt** (described below).

For example, to receive notification messages when the list of volumes in the file system changes, send **msgAddObserver** to **theFileSystem**.

ObjCallRet(msgAddObserver, theFileSystem, self, s);
When the message completes successfully, it returns \textit{stsOK}. The class manager adds the observer to the end of the object's observer list and sends \texttt{msgAdded} to the object that sent \texttt{msgAddObserver}.

If the object's capability \texttt{objCapObservable} is false, the object cannot be observed; the class manager returns \texttt{stsProtectionViolation}.

\section*{Adding an Observer with Position} \hfill 4.2.2

Normally the class manager adds an observer to the end of an object's observer list. If you want your object to be the first one that receives notification, you can add yourself to the beginning of the observer list.

To add an observer to a specific position within the object's observer list, send \texttt{msgAddObserverAt} to the object. The message takes an \texttt{OBJ_OBSERVER_POS} structure that contains:

- \texttt{observer} the \texttt{UID} of the object that will receive notifications. Usually this is \texttt{self}.
- \texttt{position} an integer that specifies the absolute position within the observer list for the object. The value 0 adds the observer to the beginning of the list. The value \texttt{maxU16} adds the observer to the end of the list.

When the message completes successfully, it returns \texttt{stsOK}. When the class manager adds the observer to the object's observer list, it sends \texttt{msgAdded} to the object that sent \texttt{msgAddObserver} or \texttt{msgAddObserverAt}.

\section*{Removing an Observer} \hfill 4.2.3

If you no longer wish to observe an object, you can send \texttt{msgRemoveObserver} to that object. Like \texttt{msgAddObserver}, the only argument to the message is the \texttt{UID} of the object that was to receive notifications.

\begin{verbatim}
ObjCallRet(msgRemoveObserver, theFileSystem, self, s);
\end{verbatim}

When the message completes successfully, it returns \texttt{stsOK}. When the class manager removes the observer from the object's observer list, it sends \texttt{msgRemoved} to the object that sent \texttt{msgRemoveObserver}.

\section*{Getting Observers from a List} \hfill 4.2.4

The class manager defines three messages that you can use to get information about an object's observer list. You can send these messages to another object to get information about that object's observer list, or you can send these messages to \texttt{self} to get information about your own observer list.

\subsection*{Getting the Observer List} \hfill 4.2.4.1

To get the entire observer list for an object, send \texttt{msgEnumObservers} to that object. The message takes a pointer to an \texttt{OBJ_ENUM_OBSERVERS} structure that contains:

- \texttt{max} a \texttt{U16} value that specifies the size of your \texttt{pObservers} array.
count: a U16 value that specifies the number of observers that you expect to receive. If this number is greater than \texttt{max}, the class manager will allocate more memory for the observer list. You can tell the class manager to give you all of the observers in its list (if necessary) by specifying \texttt{maxU16} for \texttt{count}.

\texttt{pObservers}: a pointer to the array of \texttt{OBJECTs} that will receive the list of observers.

\texttt{next}: a U16 value that specifies the position of the first observer to get. If this value is 0, start at the beginning of the list.

When the message completes successfully, it returns \texttt{stsOK} and the \texttt{OBJ_ENUM_OBSERVERS} structure contains:

\texttt{count}: the number of valid entries in \texttt{pObservers}.

\texttt{pObservers}: a pointer to the array of observers. If the class manager had to allocate memory for the list, this pointer may not be the same value that was passed in. If the pointer value is changed, you must free the new \texttt{pObservers} array when you are done with it.

\texttt{next}: if \texttt{count} was less than \texttt{max}, the position of the next entry in the observer list. You can use this value in a subsequent \texttt{msgEnumObservers} to continue enumerating where you left off. To get all observers, continue the enumeration until \texttt{msgEnumObservers} returns \texttt{stsEndOfData}.

\subsection*{Getting an Observer by Position}

To get the UID of a specific observer for an object, send \texttt{msgGetObserver} to that object. The message takes a pointer to an \texttt{OBJ_OBSERVER_POS} structure that contains a U16 value that specifies the position in the observer list (\texttt{position}).

If the message returns successfully, it returns \texttt{stsOK} and passes back the \texttt{OBJ_OBSERVER_POS} structure in which the observer field contains the UID of the object at \texttt{position}.

\subsection*{Getting the Number of Observers}

To get the number of observers for an object, send \texttt{msgNumObservers} to that object. The only argument for the message is a pointer to a U16 value.

When the message completes successfully, it returns \texttt{stsOK} and copies the number of objects in the observer list to the U16 value specified in the message arguments.

\subsection*{Notifying Observers}

When an object needs to send a notification message to its observers it uses \texttt{ObjectCall()} to send itself \texttt{msgNotifyObservers}. The message takes a pointer to an \texttt{OBJ_NOTIFY_OBSERVERS} structure that contains:

\begin{itemize}
  \item \texttt{msg}: the message to send to observers.
  \item \texttt{pArgs}: the arguments for the message.
  \item \texttt{lenSend}: the length of the arguments, in bytes.
\end{itemize}
The class manager handles msgNotifyObservers. It extracts the message and pArgs from the OBJ_NOTIFY_OBSERVERS structure and sends the message to all objects in the observer list.

If you don't want to define your own message to send to observers, you can use the generic message, msgChanged. The only argument for the message is an OBJECT value, which you usually use to pass the UID of the object that sent the notification.

**Posting to Observers**

If you want to defer delivery of your notification message so that the receiver handles it when its input queue is empty, you can use msgPostObservers rather than msgNotifyObservers.

The message takes a pointer to an OBJ_NOTIFY_OBSERVERS structure, just like msgNotifyObservers.

The status code returned from msgNotifyObservers merely indicates that the message was posted successfully, it does not indicate the completion status of the message.

**Example of Observer Notification**

The following sample from TTTDATA.C shows a utility routine used by the Tic-Tac-Toe application when it needs to notify observers.

```c
Example 4-1
Notifying Observers

TttDataNotifyObservers
Sends notifications.
******************************************************************************/
#define DbgTttDataNotifyObservers(x)  
   TttDbgHelper("TttDataNotifyObservers",tttDataDbgSet,0x1,x)

STATIC STATUS PASCAL
TttDataNotifyObservers(
    OBJECT self,
    P_ARGS pArgs)
{
    OBJ_NOTIFY_OBSERVERS nobs;
    STATUS s;
    DbgTttDataNotifyObservers(""
    nobs.msg = msgTttDataChanged;
    nobs.pArgs = pArgs;
    nobs.lenSend = SizeOf(TTT_DATA_CHANGED);
    ObjCallJmp(msgNotifyObservers, self, &nobs, s, Error);
    DbgTttDataNotifyObservers("return stsOK")
    return stsOK;

Error:
    DbgTttDataNotifyObservers("Error; return 0x%lx",s)
    return s;
} /* TttDataNotifyObservers */
```
Getting Object and Class Information

These messages allow you to:

- Get information about objects and classes
- Confirm an object’s inheritance
- Check the validity of an object or class.

These messages are useful in critical-path areas where a failed message transmission will crash the application. Table 4-2 lists the object and class information messages.

The table is divided into two sections: the first section contains messages that you pass directly to the object or class in question; these messages are only useful for information about objects and classes within your task. The second section contains messages that you pass to clsObject; the pArgs for these messages specify the object or class you are interested in. You use these messages to find information out about any object or class in the system.

<table>
<thead>
<tr>
<th>Message</th>
<th>pArgs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgIsA</td>
<td>CLASS</td>
<td>Tests if the object’s class inherits from the class.</td>
</tr>
<tr>
<td>msgAncestorIsA</td>
<td>CLASS</td>
<td>Tests if self inherits from the class.</td>
</tr>
<tr>
<td>msgClass</td>
<td>P_CLASS</td>
<td>Passes back the class of the object.</td>
</tr>
<tr>
<td>msgOwner</td>
<td>P_OS_TASK_ID</td>
<td>Passes back the task that owns this object.</td>
</tr>
<tr>
<td>msgAncestor</td>
<td>P_CLASS</td>
<td>Passes back the ancestor of the class.</td>
</tr>
<tr>
<td>msgVersion</td>
<td>pNull</td>
<td>Returns the version of the object.</td>
</tr>
<tr>
<td>msgObjectIsA</td>
<td>P_OBJ_IS_A</td>
<td>Using the object and the class in the pArgs. Tests if the object’s class inherits from the class.</td>
</tr>
<tr>
<td>msgObjectAncestorIsA</td>
<td>P_OBJ_ANCESTOR_IS_A</td>
<td>Tests if the descendant class inherits from the ancestor.</td>
</tr>
<tr>
<td>msgObjectClass</td>
<td>P_OBJ_CLASS</td>
<td>Passes back the class for the object in pArgs.</td>
</tr>
<tr>
<td>msgObjectOwner</td>
<td>P_OBJ_OWNER</td>
<td>Passes back the owning task for the object in pArgs.</td>
</tr>
<tr>
<td>msgObjectValid</td>
<td>OBJECT</td>
<td>Tests that the object in pArgs exists.</td>
</tr>
<tr>
<td>msgObjectVersion</td>
<td>OBJECT</td>
<td>Returns the version of the object in pArgs.</td>
</tr>
</tbody>
</table>

Confirming an Object’s Class

To test that an object is an instance of a particular class, send msgIsA to the object or send msgObjectIsA to clsObject. The only argument to msgIsA is a class UID. msgObjectIsA takes a pointer to a OBJ_IsA structure that contains:

object  the UID of the object that you want to check.
objClass the UID of the class to which you think object belongs.

If the object is an instance of the specified class, either message returns stsOK.
Confounding an Object's Ancestor

To test that a class inherits from a particular class, send msgAncestorIsA to the class or send msgObjectAncestorIsA to clsObject. The only argument to msgAncestorIsA is a class UID. msgObjectAncestorIsA takes a pointer to an OBJ_ANCESTOR_IS_A structure that contains:

- **descendant** the UID of the class that you want to check.
- **ancestor** the UID of the class that you think is the ancestor of class.

If the class inherits from the specified class, either message returns stsOK.

Getting an Object's Class

To get the class that created an object, send msgClass to the object or send msgObjectClass to clsObject. The only argument to msgClass is a pointer to a CLASS value. msgObjectClass takes a pointer to an OBJ_CLASS structure that contains the UID of the object that you want to check (object).

If either message completes successfully, it returns stsOK. msgClass copies the UID of the class to the CLASS value indicated by pArgs; msgObjectClass passes back the UID of the class in the objClass field of the OBJ_CLASS structure.

Getting the Owner of an Object

To get the owner of an object, send msgOwner to an object or send msgObjectOwner to clsObject. The only argument to msgOwner is a pointer to an OS_TASK_ID value. msgObjectOwner takes a pointer to an OBJ_OWNER structure that contains the UID of the object about which you need information (object).

If either message completes successfully, it returns stsOK and copies the owner identification to the OS_TASK_ID value indicated by pArgs.

Getting a Class's Class

To get the immediate ancestor of a particular class, send msgAncestor to the class. The only argument to the message is a pointer to a CLASS value.

If the message completes successfully, it returns stsOK and copies the UID of the class to the CLASS value indicated by pArgs.

You can only send this message to a class.

Checking an Object for Validity

To check whether an object is valid (that is, whether it still is known to the class manager), send msgObjectValid to clsClass. The only argument for the message is the UID of the object that you are checking. There is no local form of this message.
If the object exists, the message returns stsOK. If the object does not exist, the message returns stsBadObject. If the object has an invalid ancestor, the message returns stsBadAncestor.

### Checking an Object's Version Number

The UID for a well-known object contains a version number. You can check this version number before you send a message to a class, that way you can be sure that you are using the correct version of the class.

To get an object’s version number, send msgVersion or msgObjectVersion to clsObject. The only argument for either message is the UID of the object for which you want the version number.

If the object exists and has a version number, the message returns the version number (instead of stsOK).

Version numbers are only meaningful for well-known objects; if you send this message to a dynamic object, it returns stsScopeViolation.

If the argument value did not specify an existing object, the message returns stsBadObject.

### Getting Notification of Object Creation

If you set the capability objCapCreateNotify to true before passing msgNew, the class manager will send msgCreated to the object when it is fully initialized. The message passes a _NEW structure for the object being created.

### Properties

Properties allow objects to use additional storage space administered by the class manager. Properties are useful if your objects need transient additional storage. By using properties, you don’t have to waste four bytes of instance data for a pointer that is only occasionally used; you just create the property when you need it. Properties also allow other clients to attach information to an object.

Each object maintains its own property list. Since all clients of an object share the object’s property list, you need a unique name for each property you add to an object. You generate unique property names with the MakeTag() macro, using an administered value that belongs to you and your own choice of tag ID.

### Creating a Property

To create a property for an object, you send msgSetProp to that object. The message takes a pointer to an OBJ_PROP structure that contains:

- **key**: a key value.
- **propId**: a tag to identify the property.
- **pData**: a pointer to the data to be copied to the property.
- **length**: the number of bytes to be copied to the property.
To clear a property, you send `msgSetProp` to the object with `pData` set to NULL, and `length` set to 0.

**Retrieving Properties**

To get a copy of the data in a property, send `msgProp` to `self`. The message takes a pointer to an `OBJ_PROP` structure that contains:

- `propId` the tag for the property.
- `pData` a pointer to a local buffer that will receive the property data.
- `length` the size of the buffer indicated by `pData`.

If the property does not exist, the `length` value passed back is 0.

**Object Destruction**

Because an object may be depend on objects that depend on it, the Class Manager defines a protocol that involves several steps to destroy an object. This provides plenty of opportunity for objects to veto their destruction or to notify all related objects that they are going away.

There are two aspects to destroying an object:

- The client that wants to destroy the object sends `msgDestroy` to the object.
- The class manager then sends a series of freeing messages to the object to be destroyed.

**Destroying an Object**

To destroy an object, you send it `msgDestroy`. The message takes an `OBJ_KEY` value that specifies the key used to create the object.

Do not send `msgFree` to an object; only the Class Manager should send `msgFree`.

The class manager handles `msgDestroy` and sends a series of freeing messages to the object being destroyed. If the object eventually destroys itself, `msgDestroy` returns `stsOK`.

It is often necessary to use `ObjectPost()` instead of `ObjectCall()` with `msgDestroy`. For example, if an instance of `clsButton` sends your object `msgButtonNotify`, and your method uses `ObjectCall()` to send `msgDestroy` to that button, the button will fault after your method returns to it because its instance data will have been freed. In general, you cannot use `ObjectCall()` to send `msgDestroy` to any object (or any window subtree containing an object) that is notifying you of some interesting event.

**Handling Object Free Protocol**

When you send `msgDestroy` to an object, the class manager sends these messages to the object being destroyed:

1. `msgFreeOK`, which gives the object the opportunity to veto its destruction.
2. **msgFreeing**, which tells the object that it is time to clean up and remove itself from observer lists.

3. **msgFree**, which tells the object to destroy its data structures.

After the object handles **msgFreeing**, the class manager also sends **msgFreePending** to all observers of the object.

In each of these steps, the object should pass the message to its ancestor with **ObjectCallAncestor()**. This gives all the ancestor classes the chance to veto, to clean up, and destroy data structures.

The object being destroyed should not handle **msgDestroy** directly.

The following sections describe the freeing messages in more detail.

### Handling msgFreeOK

When you receive **msgFreeOK**, the message passes the OBJ_KEY value used by the client that sent **msgDestroy**. You should verify that it is OK to destroy the object. If it is not OK, return **stsVetoed**.

If it is OK to destroy the object, pass **msgFreeOK** to your ancestor; often sending the message to your ancestor is all you have to do. When the message reaches **clsObject**, it checks the capability **objCapFree**. If no key is necessary, it returns **stsOK**. If the capability is false and the object's key matches the **pArgs**, **clsObject** returns **stsOK**. Otherwise **clsObject** returns **stsProtectionViolation**.

If your ancestor returns **stsOK**, return **stsOK**. If your ancestor returns an error status value, return the error value.

You must not assume that just because you returned **stsOK** to **msgFreeOK** that the object will, in fact, be destroyed—some other sub-class could veto the destruction of the object.

If **msgFreeOK** returns to the class manager with **stsOK**, the class manager continues with the protocol. Otherwise, the class manager returns from **msgDestroy** to its caller with the status value that you returned.

### Handling msgFreeing

When you receive **msgFreeing**, it means that all ancestor classes agreed that the object can be destroyed. Most classes ignore **msgFreeing**.

Your handler for **msgFreeing** should clean up its data in preparation for destruction of the object (such as removing itself from notification lists). When you have finished your work, pass **msgFreeing** to your ancestor.

When your ancestor returns, return its status value to your caller.

### Handling msgFreePending

Before the class manager sends **msgFree** to the object, it sends **msgFreePending** to the object's observers. In effect, **msgFreePending** is the class manager's notification to observers that the object is about to go away.
When an observer receives `msgFreePending`, the observed object is still valid and will respond to messages. An observer might take this opportunity to query the state of the object one last time.

Most classes that observe objects will respond to `msgFreePending`, if only to note that they had better not rely on the existence of the observed object.

As with all observer notifications in PenPoint, the class manager ignores the status returned from `msgFreePending`.

### Handling `msgFree`

`msgFree` is the last step in the destruction of an object. Most classes should handle `msgFree`. When you receive `msgFree`, you must destroy objects and destroy and deallocate data structures.

The last thing that you do with `msgFree` is pass it to your ancestor. You usually call ancestor before you do anything else, but when you pass `msgFree` to your ancestor, you won’t see another message again.

When `msgFree` reaches `clsObject`, `clsObject` finishes the destruction by invalidating the object and releasing resources allocated to that object.

### Handling Failures During `msgInit` and `msgRestore`

`msgInit` and `msgRestore` are `clsObject` messages used to initialize instances of objects and to restore instances after they have been saved.

An object must be in prepared to handle `msgDestroy` after `msgInit` or `msgRestore` returns. If the Class Manager receives anything other than `stsOK` from `msgInit` or `msgRestore`, it will turn around and send `msgDestroy` to the object.

When you leave your handler for `msgInit` or `msgRestore`, the object must be prepared so that `msgDestroy` can destroy the object. This must be true even if the allocations of this class or one of the ancestor classes failed.

To give `msgDestroy` something to destroy, the handler for `msgInit` and `msgRestore` should do something like the following, in order. Obviously there are several different coding styles that achieve the same effect.

```c
MyHandler(...) {
    StsJmp(ObjectCallAncestor(msgInit or msgRestore, ...), s, ErrorExit);
    StsJmp(AllocateMyInstanceData(...), s, ErrorExit);
    ObjectWrite(my instance data);
    return stsOK;

    ErrorExit:
        ObjectWrite(some default instance data, perhaps NULL)
        return s;
}
```
Scavenging

When a task is terminated, the Class Manager destroys all objects owned by that task on a class-by-class basis. This process of destroying objects is called scavenging. Clients cannot scavenge; only the Class Manager can start scavenging.

Mutating Objects

Occasionally you might need to override the behavior of an existing object by changing the object's ancestor. This is a risky thing to do and should be avoided if possible; it is not intended for general use. However, there are times when overriding behavior is unavoidable. At these times, you can send msgMutate to the object.

The message takes a pointer to an OBJ_MUTATE structure, which contains:

- newClass: the CLASS of the new ancestor for the object.
- key: a key, if objCapMutate is false.

The size of the instance data for the original ancestor and the new ancestor must be equal. If the size of the new and old instance data is not equal, the message fails with stsSizeLimit.

If the message completes successfully, it returns stsOK. If the newAncestor class is not valid, the message returns stsBadAncestor.
Part 2 / PenPoint Application Framework
<table>
<thead>
<tr>
<th>Chapter 5 / Introduction</th>
<th>67</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>5.1</td>
</tr>
<tr>
<td>Application Framework Classes</td>
<td>5.2</td>
</tr>
<tr>
<td>Application Class (clsApp)</td>
<td>5.2.1</td>
</tr>
<tr>
<td>Application Manager (clsAppMgr)</td>
<td>5.2.2</td>
</tr>
<tr>
<td>Application Monitor (clsAppMonitor)</td>
<td>5.2.3</td>
</tr>
<tr>
<td>Embedded Window (clsEmbeddedWin)</td>
<td>5.2.4</td>
</tr>
<tr>
<td>Marks (clsMark)</td>
<td>5.2.5</td>
</tr>
<tr>
<td>View (clsView)</td>
<td>5.2.6</td>
</tr>
<tr>
<td>Observed Object (Any Class Manager Class)</td>
<td>5.2.7</td>
</tr>
<tr>
<td>Resource File Handle (clsResFile)</td>
<td>5.2.8</td>
</tr>
<tr>
<td>Application Directory Handle (clsAppDirHandle)</td>
<td>5.2.9</td>
</tr>
<tr>
<td>Application Writer's Overview</td>
<td>5.3</td>
</tr>
<tr>
<td>main</td>
<td>5.3.1</td>
</tr>
<tr>
<td>msgInit</td>
<td>5.3.2</td>
</tr>
<tr>
<td>msgAppInit</td>
<td>5.3.3</td>
</tr>
<tr>
<td>msgAppOpen</td>
<td>5.3.4</td>
</tr>
<tr>
<td>msgAppClose</td>
<td>5.3.5</td>
</tr>
<tr>
<td>msgSave</td>
<td>5.3.6</td>
</tr>
<tr>
<td>msgRestore</td>
<td>5.3.7</td>
</tr>
<tr>
<td>msgFree</td>
<td>5.3.8</td>
</tr>
<tr>
<td>Organization of Part 2</td>
<td>5.4</td>
</tr>
<tr>
<td>Other Sources of Information</td>
<td>5.5</td>
</tr>
<tr>
<td>Chapter 6 / Application Environment Concepts</td>
<td>75</td>
</tr>
<tr>
<td>Applications and Components</td>
<td>6.1</td>
</tr>
<tr>
<td>Embedding Documents</td>
<td>6.2</td>
</tr>
<tr>
<td>Application Data</td>
<td>6.3</td>
</tr>
<tr>
<td>Saving and Restoring Data</td>
<td>6.3.1</td>
</tr>
<tr>
<td>Observing Objects</td>
<td>6.3.2</td>
</tr>
<tr>
<td>Displaying Data</td>
<td>6.3.3</td>
</tr>
<tr>
<td>Saved Documents</td>
<td>6.4</td>
</tr>
<tr>
<td>Chapter 7 / Application Concepts</td>
<td>81</td>
</tr>
<tr>
<td>What is a PenPoint Application?</td>
<td>7.1</td>
</tr>
<tr>
<td>Simple Classes and Application Classes</td>
<td>7.2</td>
</tr>
<tr>
<td>A Look at Simple Classes</td>
<td>7.2.1</td>
</tr>
<tr>
<td>Metaclasses</td>
<td>7.2.2</td>
</tr>
<tr>
<td>Application Classes</td>
<td>7.2.3</td>
</tr>
<tr>
<td>The Document</td>
<td>7.3</td>
</tr>
<tr>
<td>Information on Screen</td>
<td>7.3.1</td>
</tr>
<tr>
<td>The File System and Documents</td>
<td>7.3.2</td>
</tr>
<tr>
<td>The Document as a Process</td>
<td>7.3.3</td>
</tr>
<tr>
<td>The Document as an Object</td>
<td>7.3.4</td>
</tr>
<tr>
<td>The Parts of a Document</td>
<td>7.4</td>
</tr>
<tr>
<td>Chapter 8 / Life Cycles</td>
<td>95</td>
</tr>
<tr>
<td>Application Class Life Cycle</td>
<td>8.1</td>
</tr>
<tr>
<td>Installing an Application</td>
<td>8.1.1</td>
</tr>
<tr>
<td>Deinstalling an Application</td>
<td>8.1.2</td>
</tr>
<tr>
<td>Document Life Cycle</td>
<td>8.2</td>
</tr>
<tr>
<td>Creating a Document</td>
<td>8.2.1</td>
</tr>
<tr>
<td>Activating a Document</td>
<td>8.2.2</td>
</tr>
<tr>
<td>Opening a Document</td>
<td>8.2.3</td>
</tr>
<tr>
<td>Closing a Document</td>
<td>8.2.4</td>
</tr>
<tr>
<td>Terminating a Document</td>
<td>8.2.5</td>
</tr>
<tr>
<td>Reactivating a Document</td>
<td>8.2.6</td>
</tr>
<tr>
<td>Deleting a Document</td>
<td>8.2.7</td>
</tr>
<tr>
<td>Chapter 9 / Embedded Documents</td>
<td>117</td>
</tr>
<tr>
<td>Embedded Window Concepts</td>
<td>9.1</td>
</tr>
<tr>
<td>Marking Support</td>
<td>9.1.1</td>
</tr>
<tr>
<td>Descendants of clsEmbeddedWin</td>
<td>9.1.2</td>
</tr>
<tr>
<td>Moving or Copying Between Embedded Windows</td>
<td>9.2</td>
</tr>
<tr>
<td>Separate Messages</td>
<td>9.2.1</td>
</tr>
<tr>
<td>Why Use the Move and Copy Protocol?</td>
<td>9.2.2</td>
</tr>
<tr>
<td>The Move and Copy Protocol</td>
<td>9.3</td>
</tr>
<tr>
<td>Requesting a Move or Copy</td>
<td>9.3.1</td>
</tr>
<tr>
<td>Identify the Selection</td>
<td>9.3.2</td>
</tr>
<tr>
<td>Let the User Indicate the Destination</td>
<td>9.3.3</td>
</tr>
<tr>
<td>Tell the Destination to Move or Copy</td>
<td>9.3.4</td>
</tr>
<tr>
<td>Determining the Data Type</td>
<td>9.3.5</td>
</tr>
<tr>
<td>OK the Move or Copy</td>
<td>9.3.6</td>
</tr>
<tr>
<td>Getting the Destination in the File System</td>
<td>9.3.7</td>
</tr>
<tr>
<td>Moving or Copying the Data</td>
<td>9.3.8</td>
</tr>
<tr>
<td>Getting the Exact Pen Location</td>
<td>9.3.9</td>
</tr>
<tr>
<td>Example: Moving in Tic-Tac-Toe</td>
<td>9.4</td>
</tr>
<tr>
<td>The User Requests a Move</td>
<td>9.4.1</td>
</tr>
<tr>
<td>Presenting a Move/Copy Icon</td>
<td>9.4.2</td>
</tr>
<tr>
<td>The Move or Copy Icon</td>
<td>9.4.3</td>
</tr>
<tr>
<td>Determining the Data Type</td>
<td>9.4.4</td>
</tr>
<tr>
<td>Moving the Data</td>
<td>9.4.5</td>
</tr>
<tr>
<td>Intercepted Messages</td>
<td>9.5</td>
</tr>
<tr>
<td>Gestures and Selection</td>
<td>9.6</td>
</tr>
<tr>
<td>Chapter 10 / Mark Concepts</td>
<td>129</td>
</tr>
<tr>
<td>Mark Class</td>
<td>10.1</td>
</tr>
<tr>
<td>Parts of a Mark</td>
<td>10.1.1</td>
</tr>
<tr>
<td>Implementing Tokens</td>
<td>10.2</td>
</tr>
<tr>
<td>Table Mapping</td>
<td>10.2.1</td>
</tr>
<tr>
<td>Stamp Mapping</td>
<td>10.2.2</td>
</tr>
</tbody>
</table>
Chapter 11 / Printing

Concepts
- Printing Embedded Documents
- Making Pagination Decisions
- Option Sheets for Printing
  - Handling msgAppAddCards
- Print Protocol Description
  - Print Layout Driver (PLD)
  - Removing Frame Decorations
  - Pagination
  - Handling Embedded
  - Default Printing Behavior

Chapter 12 / The Application Manager Class

Application Manager Metrics
- Installing a New Class
- Creating a New Document
- Activating an Application Instance
- Moving or Copying an Application Instance
- Getting Metrics for a Class
- Observer Messages

Chapter 13 / The Application Monitor Class

Application Monitor Concepts
- Application Monitor in Installation
- Other Application Monitor Functions
  - Stationery, Accessories, and Help
clsAppMonitor Messages
- Using clsAppMonitor Messages
  - Getting App Monitor Metrics
  - Loading and Unloading Stationery
  - Loading and Unloading Help
  - Loading Other Information
Subclassing clsAppMonitor
  - Superclass Messages
  - Handling clsAppMonitor Messages

Chapter 14 / The Application Class

clsApp Messages
- Document Hierarchy Messages
  - Managing Embedded Documents
  - Document Links
  - Getting Document Information
  - Setting Hot Mode
  - Renaming a Document
  - Getting an Application’s Name
  - Setting an Application’s Title
Document Window Messages
  - Setting the Main Window
Standard Application Menus
  - Document and Edit Menus
  - Options Menu Protocol
  - Check Gesture Handling
Observing System Preferences
- Advanced Messages
  - Setting a Parent Document
  - Setting Priority

Chapter 15 / The View Class

View Concepts
- View Filing
- Creating a New View
- Setting the Data Object
- Getting the Data Object
Subclassing clsView

Chapter 16 / The Application Directory Handle Class

Using clsAppDir
- Creating an Application Directory Handle
- Destroying an Application Directory Handle
- Getting the Application Directory Global Sequence Number
Getting and Setting Application Directory Attributes
  - Getting and Setting All Attributes
  - Getting or Setting Individual Attributes
Getting Attributes for Many Application Directories
- Determining a Document’s Name
- Counting Embedded Documents
- Setting a Tab
Chapter 17 / Container Application Classes

Concepts 17.1 185
Container Application Class 17.2 186
Root Container Application Class 17.3 186
clsRootContainerApp Concepts 17.3.1 187
clsRootContainerApp Messages 17.3.2 187
Changing the Current Document 17.3.3 187
Cancelling the Reference Document 17.3.4 188
Turning to the Contents Page 17.3.5 188

Chapter 18 / Embedded Window Class

Using clsEmbeddedWin 18.1 189
Creating an Embedded Window 18.1.1 190
Destroying an Embedded Window 18.1.2 190
Getting Embedded Window Metrics 18.1.3 190
Getting and Setting the Style of an Embedded Window 18.1.4 190
Beginning a Move or Copy Operation 18.1.5 191
Moving or Copying an Embedded Window 18.1.6 192
Confirming the Move or Copy 18.1.7 192
Getting the Destination of a Move or Copy 18.1.8 192
Handling Child Embedded Windows 18.1.9 193
Getting Pen Offsets 18.1.10 193
Setting an Embedded Window's UUID 18.1.11 193

Chapter 19 / Application Window Class

Using clsAppWin Messages 19.1 195
Creating an Embedded Application 19.1.1 196
Changing the Style of an Application Window 19.1.2 196
Opening an Application Window 19.1.3 196
Closing an Application Window 19.1.4 196
Getting the Metrics of an Application Window 19.1.5 196

Chapter 20 / The Mark Class

Marks Sent to Components 20.1 200
Creating a Token 20.1.1 200
Deleting a Token 20.1.2 201
Comparing Tokens 20.1.3 201
Finding a Component's Ancestor 20.1.4 201
Finding a Component's Parent 20.1.5 201
Setting the UUIDs 20.1.6 201
Validating a Component 20.1.7 201

List of Figures

5-1 PenPoint Application Framework Classes Hierarchy 71
6-1 PenPoint Software Hierarchy 76
7-1 Predefined Classes 83
7-2 A Simple Class 84
7-3 An Instance of a Simple Class 84
7-4 Predefined Classes with clsAppMgr 85
7-5 Inheritance of an Application Class 86
7-6 Application Hierarchy: Screen Perspective 88
7-7 Application Hierarchy: File System Perspective 89
7-8 Application Hierarchy: Process Perspective 91
7-9 Parts of a Document Process 93
8-1 Application Class State Diagram 95
8-2 Document State Diagram 100
8-3 New Document Process 103
8-4 Document with Objects and Frame 107
8-5 Opened Document 109
8-6 Document after Termination 111
10-1 Parts of a Mark 130
17-1 Container Application Class Hierarchy 186

List of Tables

8-1 Application Class States 96
8-2 Document States 100
9-1 Embedded Window Gestures 127
11-1 Printing Protocol Messages 140
12-1 clsAppMgr Messages 146
12-2 Flags Described by APP_MGR_FLAGS 147
13-1 clsAppMonitorMessages 152
14-1 clsApp Messages 157
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14-2</td>
<td>How clsApp Handles Menu Button Tags</td>
<td>164</td>
</tr>
<tr>
<td>14-3</td>
<td>Standard Application Menus</td>
<td>165</td>
</tr>
<tr>
<td>15-1</td>
<td>clsViewMessages</td>
<td>173</td>
</tr>
<tr>
<td>16-1</td>
<td>clsAppDir Messages</td>
<td>178</td>
</tr>
<tr>
<td>16-2</td>
<td>File Mode Flags</td>
<td>180</td>
</tr>
<tr>
<td>17-1</td>
<td>clsRootContainerApp Messages</td>
<td>187</td>
</tr>
<tr>
<td>18-1</td>
<td>clsEmbeddedWin Messages</td>
<td>189</td>
</tr>
<tr>
<td>19-1</td>
<td>clsAppWin Messages</td>
<td>195</td>
</tr>
<tr>
<td>20-1</td>
<td>The Class Mark Messages</td>
<td>199</td>
</tr>
</tbody>
</table>
Chapter 5 / Introduction

Part 2: Application Framework describes the PenPoint Application Framework and its associated classes.

The PenPoint Application Framework encompasses a number of topics, including the file system, file organization, running processes, and screen displays.

Because this part covers some complicated topics, there is bound to be some confusion about the terms used here.

**An application** is the executable file that you (the software designer) create to run on the PenPoint™ operating system. The application specifies how its documents will respond to user actions and other messages.

**The PenPoint Application Framework** is a collection of classes that the PenPoint operating system uses to run applications.

**An application class** is a PenPoint class that contains the code and initialization data used to create documents. When the user installs your application (your executable code), the PenPoint Application Framework creates the application class for your application.

**A Notebook** is an application written by GO that provides the user interface for the application framework and, along with the Section application, maintains the document hierarchy. The Notebook sends PenPoint Application Framework messages to documents to create, open, close, and destroy documents.

**A document** is a set of objects and data created by an application class that is able to interact with the user and the PenPoint Application Framework. When closed, a document consists of a series of files and directories in the file system; when active, a document consists of a system process, an number of objects (including an instance of the application class), in addition to the files and directories in the file system.

**An instance of an application class, or an application object** is an instance of an application class that provides a document with an interface to the PenPoint Application Framework.

**An embedded document** is a document that is contained within another document. Embedded documents are not shown in the Notebook Table of Contents.

**An open document** is a document that is currently displayed on the screen.
Overview

The PenPoint operating system and its Notebook metaphor differ greatly from other operating systems. In other operating systems, the user issues a command or clicks on an icon to run a particular program. The user must also specify which data file to open. When finished with a task, the user must save the data to some location before quitting, or risk loss of information.

In PenPoint, the user turns to the page that contains the desired document. PenPoint knows where the document’s data is stored and which application to run. When the user turns to another page, PenPoint knows exactly where to store the modified data.

In short, each page in the Notebook contains a document; an open document is a running application.

To support this level of user friendliness, applications must interact much more with the operating system than with traditional operating systems. The PenPoint Application Framework provides the tools for this interaction.

The Notebook is actually a specialized application that is started when you boot the PenPoint computer. The Notebook uses the PenPoint Application Framework, just as any other application.

Much of the PenPoint Application Framework consists of messages defined by several classes. When your application receives some of these messages, it must perform specific processing actions; when it receives other messages, your application must pass the messages to its ancestor.

Application Framework Classes

The PenPoint Application Framework consists of seven basic classes:

- Application manager class
- Application class
- Embedded window class
- Mark class
- View class
- Resource file class
- Application directory class

In addition to these classes, the PenPoint Application Framework makes extensive use of the class manager’s utility classes, described in Part 9: Utility Classes.
Application Class (clsApp)

- Manages program startup, shutdown, checkpointing, and backup.
- Manages the application hierarchy.
- Controls application embedding.
- Is a descendant of clsObject.
- Creates views and objects.
- Informs its views and objects when and where to file.

Application Manager (clsAppMgr)

- Instances of clsAppMgr maintain information about each application class.
- Each application class creates an application manager instance that knows about that application.
- Is a descendant of clsClass.

Application Monitor (clsAppMonitor)

- Drives application installation and deinstallation
- Reactivates applications on warm boot.

Embedded Window (clsEmbeddedWin)

- Controls view embedding.
- Provides move and copy behavior for components.
- Is a descendant of clsWin.

Marks (clsMark)

- Provides a mechanism to refer to a data item within a component.
- Provides a mechanism for traversing embedded documents.

View (clsView)

- Manages the user interface.
- Displays data for an object.
- Controls view embedding.
- Is a descendant of clsEmbeddedWin.
- Observes its objects.
Observed Object (Any Class Manager Class)

- Manages a list of observers.
- Communicates state changes and other events to its observers.
- Any object can be observed.
- Is the mechanism used by viewed objects to notify their views of changes.

Resource File Handle (clsResFile)

- Manages the structured reading and writing of objects to a file.
- Can re-create objects and their classes when their data is read in.
- Is a descendant of clsFileHandle.
- Used by many application classes to store or re-create their objects to and from the file system.
- Used by viewed objects during file writing to file their multiple views.

Application Directory Handle (clsAppDirHandle)

- Represents a document in the document hierarchy.
- Manages application directories.
- Provides a working directory in which applications can checkpoint their data.
- Is a descendant of clsDirHandle.
- Used by applications to communicate with child (embedded) documents.

The PenPoint Application Framework classes are related in the class hierarchy shown in Figure 5-1.
The most elementary PenPoint application doesn’t do anything. It lets the PenPoint Application Framework do all the work by allowing clsApp to handle all PenPoint Application Framework messages. You can see this in the two sample programs, Empty Application and Template Application, described in the PenPoint Application Writing Guide. The sources for these applications are in PENPOINT\SDK\SAMPLE.

As your program increases in sophistication, your application has to handle more and more PenPoint Application Framework messages. This quick-start section lists the PenPoint Application Framework messages that most applications have to handle and describes briefly how to handle them. The messages are:

- main()
- msgInit
- msgAppInit
- msgAppOpen
- msgAppClose
- msgFree
- msgSave
- msgRestore

main() is actually a function.
main()

The function main() is the entry point to your application. An application executable file must have a main() function.

main() has two primary purposes:

- Installing your application class.
- Starting the dispatch loop for each document that uses your application.

The processCount parameter to main() indicates how many other processes are currently running this application program. When processCount is 0, the application is being installed, so you call a routine to install your application class and then call AppMonitorMain(), a PenPoint function that starts the application monitor for your application class.

When processCount is greater than 0, the user is opening a document that uses your application, so you call AppMain(), a PenPoint function that creates an instance of your application class, passes it messages to initialize its data, and enters a dispatch loop to receive messages.

msgInit

Your application's method table should call ancestor before handling msgInit. In response to msgInit, clsApp allocates storage for your application's instance data in protected memory.

When your application receives msgInit, it should initialize its instance data and use the function ObjectWrite to save the initialized instance data to protected memory.

msgAppInit

Your application's method table should call the application's ancestor before handling msgAppInit. In response to msgAppInit, clsApp creates an object resource file and main window for the application.

When your application receives msgAppInit, it creates objects that it will use to store permanent data for the document (as opposed to window and control objects that it displays on screen).

msgAppOpen

Your application's method table should call the application's ancestor after handling msgAppOpen.

When your application receives msgAppOpen, it creates windows to display data and any other control objects and returns.

In response to msgAppOpen, clsApp then inserts the document's frame into the main window, which displays the document on screen.
**msgAppClose**

Your application’s method table should call the application’s ancestor before handling `msgAppClose`. In response to `msgAppClose`, `clsApp` extracts the frame from the main window.

When your application receives `msgAppClose`, it destroys its windows and control objects.

**msgSave**

Your application’s method table should call the application’s ancestor before handling `msgSave`. In response to `msgSave`, your application’s ancestors save data and objects that they created. `clsApp` saves your main window, any data objects observed by views, and, optionally, its client window.

When your application receives `msgSave`, it should:

- Write data that isn’t maintained in objects to the resource file.
- Save any permanent objects.

**msgRestore**

Your application’s method table should call the application’s ancestor before handling `msgRestore`. In response to `msgRestore`, your application’s ancestors restore data and objects that they saved earlier. `clsApp` restores your main window and its client window.

When your application receives `msgRestore`, it should restore any data that it saved in `msgSave`.

**msgFree**

Your application’s method table should call the application’s ancestor after handling `msgFree`.

When your application receives `msgFree`, it should destroy any permanent objects that it created and frees any allocated memory. (The application receives `msgSave` before it receives `msgFree`.)

In response to `msgFree`, your application’s ancestors destroy objects and free memory that they allocated.

**Organization of Part 2**

Part 2 is divided into two sub-parts. Chapters 5 through 11 present the PenPoint Application Framework concepts. Chapters 12 through 20 describe each of the classes used by the PenPoint Application Framework and their significant messages.

Chapter 5, Introduction, provides an introduction to the PenPoint Application Framework and describes the organization of Part 2.
Chapter 6, Application Environment Concepts, describes the PenPoint environment that supports applications and explains why the PenPoint Application Framework is necessary.

Chapter 7, Application Concepts, describes the applications and their parts. This chapter also provides a brief description of the classes that make up the PenPoint Application Framework.

Chapter 8, Life Cycles, describes the life cycles of application classes and documents. The life cycles are presented from both the user perspective and the application perspective.

Chapter 9, Embedded Documents, describes the concepts related to embedded windows and embedded applications.

Chapter 10, Mark Concepts, describes the concepts used when examining applications with embedded documents.

Chapter 11, Printing, describes what you have to do to print.

The remaining chapters discuss the individual classes used by the PenPoint Application Framework.

Chapter 12, The Application Manager Class, describes the messages defined by clsAppMgr.

Chapter 13, The Application Monitor Class, describes the messages defined by clsAppMonitor.

Chapter 14, The Application Class, describes the messages defined by clsApp.

Chapter 15, The View Class, describes the messages defined by clsView.

Chapter 16, The Application Directory Handle Class, describes the messages defined by clsAppDir.

Chapter 17, Container Application Classes, describes the two classes that implement containers (sections) and root containers (the Notebook).

Chapter 18, Embedded Windows Class, describes the messages defined by clsEmbeddedWin.

Chapter 19, Application Windows Class, describes the messages defined by clsAppWin.

Chapter 20, The Mark Class, describes the messages defined by clsMark.

Other Sources of Information

Part 2 should be read in conjunction with the PenPoint Application Writing Guide. While Part 2 describes what the PenPoint Application Framework does for you, the PenPoint Application Writing Guide describes how you use the PenPoint Application Framework when you write an application.
Chapter 6 / Application Environment Concepts

The PenPoint Application Framework provides the environment in which all PenPoint applications run. It is not a user interface shell or window system that applications choose to use. If you want your application to fit into the Notebook metaphor and interact correctly with other applications, it must use the PenPoint Application Framework. Programs that run under the PenPoint™ operating system must use the PenPoint Application Framework; there is no other way to start a program in PenPoint.

The PenPoint operating system uses the PenPoint Application Framework extensively to implement its Notebook user interface. The PenPoint Application Writing Guide explains how your applications fit into an application hierarchy, which includes the pages and section tabs of a Notebook. The PenPoint Application Writing Guide also provides a good introduction to the ideas behind the PenPoint Application Framework.

The PenPoint Application Framework provides a foundation on which you build PenPoint applications. It is implemented as a collection of PenPoint classes that you must use in order to write an application.

The following sections discuss aspects of applications and how they relate to the PenPoint Application Framework:

- Relationships between applications and components.
- Embedded applications.
- Saving and restoring application data.
- Saved documents and application directories.
- Missing application classes.
Applications and Components

Figure 6-1 shows the hierarchy of PenPoint software.

An application is a PenPoint class that defines how its active documents will interact with the user or other devices to perform some useful work. To fit in the PenPoint Notebook environment, an application must also define how its active documents perform housekeeping operations, such as startup, shutdown, filing, and message dispatching provided by the PenPoint Application Framework.

Components are classes that perform more specialized tasks: the file system, windows, buttons, graphics, tables, text, lists, and many other classes. Your application is responsible for orchestrating component behavior, including creation, visual positioning, filing, and destruction. A well-designed PenPoint application will take advantage of components wherever possible.

Components save you from having to write your own routines to perform these tasks. However, if a component doesn’t provide the functionality that you need, you can either subclass an existing component class or create an entirely new component class. You can document and publish your new component class so that other application writers can take advantage of it.

In contrast with applications and components, services are PenPoint programs that do not have a user interface, but do useful work in response to requests from applications and components. Typical examples of services are MIL devices (device drivers), database engines, or E-Mail backends. If you are contemplating a
PenPoint program that performs this type of task, you should read Part 13: Writing PenPoint Services of the PenPoint Architectural Reference.

## Embedding Documents

Just as someone might fill a page of a paper notebook with text, a diagram, and some notes and figures, PenPoint users should be able to intermix different applications. The Notebook user interface makes this easy: the user just creates a new document inside an existing document, or moves or copies one document into another.

To support this style of interaction, the PenPoint Application Framework provides facilities for applications and components to embed inside of other applications and components without detailed knowledge about what is being embedded. Write your applications so that the user can embed another well-behaved application within it.

When you create a new application class, you specify whether the application allows embedded documents. Embedded documents allow an embedded window and its parent application to negotiate how much space the embedded document can use. For more information on embedded windows, see Chapter 9, Embedded Documents.

The PenPoint Application Framework classes allow applications and components to accept or reject embedding requests, determine the size and position of the embedded objects, and tell the embedded objects when to file their data.

## Application Data

Your application is responsible for maintaining, displaying, and filing its application data.

## Saving and Restoring Data

clsObject and clsResFile define messages that tell objects when to save and restore their data. Your application must be prepared to receive these messages and act on them. Your application can maintain its own data, in which case it is responsible for performing the necessary read and write operations. However, if your application stores its data in objects, it simply tells the handle for the resource file to send, save, or restore messages to the objects.

Component classes (such as lists, tables, and text) file their own data. For specialized types of data, you must write your own component class that can save and restore its data. You need only write the component class once and you can use it many times.
Observing Objects

Most applications go beyond creating descendants of clsObject and structure their data into observable objects, which they then observe. The Class Manager allows an object to keep track of other objects that are interested in that object, and to send them messages when it changes. Such a style of interaction makes it easier for applications to support multiple views of the same data: the data is maintained by an observable object that notifies multiple views when the data changes.

Displaying Data

Usually, your application maintains its data as separate objects. Each object is responsible for filing its own data. Your application can use views to display the data.

As mentioned in the PenPoint Application Writing Guide, a view is a specialized descendent of clsWin that displays a data object in its window. The data object sends a message to the view when its data changes; the view takes care of redisplaying the data. An application can associate more than one view with the same data object.

Using views to display data objects automatically sends filing messages to its data objects. Your application doesn’t have to bother sending the filing messages to the viewed objects. (However, the objects must be able to handle the filing messages.)

With careful design, your application data should know nothing about the form of your application’s user interface, nor whether your application is on screen, nor even if your application has a user interface at all. The data objects simply respond to messages on behalf of external agents.

Saved Documents

Each document has its own directory in the file system. The PenPoint Application Framework helps to coordinate the location of each application directory.

The Notebook Table of Contents displays the list of documents. To the user, it looks like each of the documents is ready and available. When the user turns to a page, the document is visible; when the user turns to another page, another document is visible. The user doesn’t realize that when a document is visible on screen, its application is running; when the document is not on screen, usually its application is not running and its data is saved in memory.

Actually, the Notebook Table of Contents displays a list of document directories in the file system. Each directory contains a resource file for the document’s instance data and an attribute that indicates the application that created the document.

When the user turns to a particular document, the PenPoint Application Framework creates a process running an instance of the application and tells that process to read in the document’s data from the resource file. When the user turns away from the page, the PenPoint Application Framework tells the document to save its instance data and terminates the process.
This is how documents survive while the power is off; documents are active only while they are on the screen. When a document is not visible on the screen, its data is safely stored in the file system.

When a document runs in **hot mode** it closes its windows on a page turn, but the process does not terminate.
Chapter 7 / Application Concepts

This chapter describes PenPoint applications and how they relate to the PenPoint™ Application Framework and the Notebook. It does not discuss how to send or respond to Penpoint Application Framework messages (that is covered in Chapter 8).

The following topics are covered in this chapter:

- How application classes differ from other classes.
- How application classes relate to application instances.
- How a document is represented on the screen, in the file system, and as a process in the PenPoint operating system.
- The parts that make up an application.

The chapter concludes with a table that summarizes the purposes and relationships of the PenPoint Application Framework classes.

What is a PenPoint Application?

In its simplest form, a PenPoint application is an executable file that can create and maintain documents in the PenPoint computer.

An application contains a routine to create an application class, and methods that handle messages sent to instances of an application class. The application class is an instance of clsAppMgr.

When the user installs the application, the application invokes the routine to create the application class. An installed application class is an active process in the PenPoint computer.

When the user creates a document, the Notebook responds by creating an instance of the appropriate application class. When active, a document is also a process in the PenPoint computer. In most cases when a document is active, it is displayed on screen. There are times, such as printing, when a document is active but not on screen.

When the user taps or writes on the document, the methods defined in the application class perform the work for the document.
Simple Classes and Application Classes

In Part 1: The Class Manager, we saw how classes generate instances of themselves. Clients use instances of a class to describe particular sets of data. For example, a client can request clsList to create two list objects; in each of those lists, the client maintains separate sets of data.

Similarly, an application class generates application instances. Each application instance describes the set of data that makes up a document. When the user turns to a document in the Notebook, the PenPoint Application Framework creates an application instance that manages that document. In fact, when we use the term document, we are referring to both the application instance and the data that make up that document.

When you write an application, you are actually defining a new application class. The PenPoint Application Framework uses your application class to create new documents. When the user installs an application, the PenPoint Application Framework works with your application to create an application class. When the PenPoint Application Framework sends msgNew to your application class, it creates a document.

A Look at Simple Classes

Before we continue our discussion of application classes, let’s take a look at classes and their relationship to clsClass.

All classes and instances in the PenPoint operating system are objects, represented by a 32-bit UID (unique identifier). An object is an instance of a particular class. When you send msgNew to a class, the class creates a new object. When you send msgNew to clsClass, the new object created by clsClass is a new class.

clsClass is a special class whose sole purpose is to create and destroy classes. Because clsClass cannot create itself, PenPoint creates clsClass at boot time. clsClass inherits from clsObject.

One of the arguments in msgNew to clsClass specifies the ancestor of the new class. The new class inherits the behavior of its ancestor class.

When an instance of a class receives a message, it can handle the message or it can pass the message to its ancestor. The ancestor can handle the message or pass the message to its ancestor, and so on, up to clsObject. clsObject is another special class, which is the great ancestor of all classes, including clsClass. Because clsClass inherits from clsObject, clsClass cannot create clsObject. Therefore PenPoint creates clsObject at boot time.

Figure 7-1 shows clsClass and clsObject in the context of the system process. In this and following drawings, the objects are depicted as lozenges. The label above the object identifies the specific object. The bottom-most class listed inside an object tells what class created the object (and therefore tells you what the object is). Thus, in the figure, both clsObject and clsClass are instances of clsObject.

This paragraph provides a key to the next series of figures.
Labeling the box that contains clsObject and clsClass a “system process” is a gross simplification; the PenPoint operating system uses several processes.

This example from the Tic-Tac-Toe sample application shows a client creating a new class called clsTttData.

```
CLASS NEW new;
STATUS s;
ObjCallJrnp(rnsgNewDefaults, clsClass, &new, s, Error);
new.object.uid = clsTttData;
new.object.key = 0;
new.cls.pMsg = clsTttDataTable;
new.cls.ancestor = clsObject;
new.cls.size = SizeOf(P_TTT_DATA_INST);
new.cls.newArgsSize = SizeOf(TTT_DATA_NEW);
ObjCallJrnp(rnsgNew, clsClass, &new, s, Error);
```

When you send msgNew to a class, the class allocates space for the instance that it is about to create, and sends msgNew to its ancestor. Eventually, clsObject receives msgNew, creates a new object, assigns it a UID, and returns.

In this example, the client sends msgNew to a class, creating an instance of that class.

```
TTT_DATA NEW tttDataNew;
// Create the TttData object
ObjCallJrnp(msgNewDefaults, clsTttData, &tttDataNew, s, Error);
// no defaults to override
ObjCallJrnp(msgNew, clsTttData, &tttDataNew, s, Error);

// get the UID of the new TttData object
pArgs->view.dataObject = tttDataNew.object.uid;
```

Figure 7-2 shows a simple example class (clsList). As you can see, the object labeled “clsList” is an instance of clsClass and inherits from clsObject. When a client sends a message to clsList, the messages are handled by methods defined by clsClass. If clsClass cannot handle a message, it can pass the message to its ancestor, clsObject.
When a client sends `msgNew` to `clsList`, the method defined by `clsClass` creates a new object that is an instance of `clsList`. The new object responds to messages defined by `clsList` and those defined by its ancestor, `clsObject`.

Figure 7-3 shows an instance of `clsList`.

---

**Metaclasses**

An application class is simply an extension of what we have discussed so far. At this point, we know that:

- `clsClass` creates new classes.
- Classes create instances of themselves (objects).
- Objects handle messages by performing methods defined by their class.
- If the class cannot handle a message, it can allow its ancestor to handle the message.
Most classes are instances of \texttt{clsClass}. Each class has a small amount of instance data (describing the location of the method table for the class, how much room to allocate for instances of the class, and so on). This common information is used by all instances of the class, but because it is stored in a single location, it is extremely space efficient.

Other classes can benefit from being able to store common information in a single location. Unfortunately, the structure used by \texttt{clsClass} is specific only to \texttt{clsClass}. It isn't possible for classes to add to the structure.

However, it is possible to define additional instance data by creating a new class that inherits from \texttt{clsClass}. A class that inherits from \texttt{clsClass} is called a \textit{metaclass}. Because the class inherits from \texttt{clsClass}, it can create new classes. These new classes can all use common instance data that is defined specifically for the metaclass.

Space efficiency is important to application classes. Each application class contains a large amount of information that is used by each instance of the application class, yet would be wasteful to copy over and over again. This information includes the application name, icon, and initial size and position.

For application classes, PenPoint defines the metaclass \texttt{clsAppMgr}. An application class is installed by creating an instance of \texttt{clsAppMgr}. An instance of \texttt{clsAppMgr} contains instance data that includes the name, icon, and initial size and position for instances of the application class.

Figure 7-4 shows the system classes, as described above, with the addition of \texttt{clsAppMgr}.

We can add to our list above:

- \texttt{clsAppMgr} creates new application classes.
- Application classes create instances of themselves (documents).

\textit{Metaclasses are an extremely powerful tool for extending the abilities of existing classes.}
Application Classes

While the metaclass clsAppMgr enables an application class to maintain instance data used by all of its instances, the only thing that clsAppMgr actually does is to create application classes.

Applications have to perform many, many actions in response to the PenPoint Application Framework messages. All of the standard behavior for applications is defined by the application class, clsApp.

All application classes are descendents of clsApp.

Because applications inherit from clsApp, they can allow clsApp to handle most PenPoint Application Framework messages, rather than handle the messages themselves. This frees you, the application developer, from coding mundane tasks and allows you to spend time writing the actual behavior of your application.

This example shows a client sending msgNew to clsAppMgr to create the Tic-Tac-Toe application class.

```
APP_MGR_NEW new;
STATUS s;
ObjCallImp(msgNewDefaults, clsAppMgr, &new, s, Error);
new.object.uid = clsTttApp;
new.object.key = 0;
new.cls.pMsg = clsTttAppTable;
new.cls.size = SizeOf(P_TTT_APP_INST);
new.cls.newArgsSize = SizeOf(APP_NEW);
new.appMgr.flags.stationary = true;
new.appMgr.flags.accessory = false;
strcpy(new.appMgr.company, "GO Corporation");
```

Figure 7-5 shows an application class (clsCalcApp). In the figure, you can see the process that contains the application class (CalcApp Process 0), and the process that contains the active document (CalcApp Document). Process 0 contains clsCalcApp, which is an instance of clsAppMgr, and inherits from clsApp. The document process contains aCalcApp, which is an instance of clsCalcApp.
In summary:

- An instance is an object.
- clsObject is an object; it has no ancestors.
- clsClass is an instance of clsObject that inherits behavior from clsObject.
- clsAppMgr is an instance of clsClass that inherits behavior from clsClass and clsObject.
- clsApp is an instance of clsAppMgr that also inherits behavior from clsObject.
- A class is an instance of clsClass that inherits behavior from the class specified by its creator (and all of that class’s ancestors).
- An application class is an instance of clsAppMgr that also inherits behavior from clsApp and clsObject.
- A document is an instance of an application class; it inherits behavior from its application class, clsApp, and clsObject.

The concepts presented here apply to active documents (usually an active document is visible on the screen). Later, we will discuss what happens to documents that are inactive.

An active document has four aspects:

- It can display its information on screen.
- It has its own directory in the file system.
- It is a process.
- It is an object that receives messages from the PenPoint Application Framework.

These aspects are related. However, because they are difficult to address simultaneously, the next four sections describe them individually.

Information on Screen

Figure 7-6 shows a typical screen with four documents visible. The documents are:

- The Notebook.
- The Notebook Table of Contents.
- A MiniText document called “Package Design Letter.”
- A MiniNote document called “Suggestion.”
The PenPoint file system has a hierarchical organization, much like MS-DOS. Each volume has a root directory; directory entries can point to files or other directories.

Figure 7-7 illustrates part of the file system organization for the screen shown in Figure 7-6.
The Notebook uses the file system to organize its documents in parallel with the organization of the table of contents.

- Each document in the Notebook has its own directory. This directory contains the object resource file (DOCSTATE.RES) that holds the document's instance data and a document resource file (DOC.RES) that holds resources used by the document.

- Sections are a type of document; thus each section in the Notebook has its own directory.

- If a document is contained in a section, the document's directory is a subdirectory of that section's directory.

- If a document has an embedded document, the embedded document's directory is a subdirectory of its enclosing document's directory.

- The Notebook Table of Contents is a big section that contains all the other documents and sections.

### The Document as a Process

An open document is a running application. An active document is also a process in a PenPoint computer.

When a document is displayed on the screen, it has an active process. When a document is not displayed on screen, it usually does not have a process. Much of the work performed by the PenPoint Application Framework involves creating the...
process, making it an application object capable of receiving messages, and—when the user turns away—destroying the process and saving its data.

The class manager maintains a database that associates the application process with a UID, which identifies the process as an application object that can receive messages.

Figure 7-8 illustrates the processes running for the screen shown in Figure 7-7. Each of the boxes represent a running process. These processes are described in greater detail in "The Parts of a Document," later in this chapter. Note that the figure also shows the application class processes. Each application class is a process that runs to create instances of itself.
CHAPTER 7 / APPLICATION CONCEPTS

The Document

Figure 7-8

Application Hierarchy: Process Perspective

Document Process

Process 0

NB Process

NB Application Class

Notebook Contents

Section Application Class

Samples

MiniText Application Class

Package Design Letter

MiniNote Application Class

Suggestion
The Document as an Object

An open document is also an instance of a particular application class, and a descendent of clsApp. The document object is shown in Figure 7-8.

The document responds to messages defined by its application class. Because the document object inherits from clsApp, it also responds to clsApp messages.

The Parts of a Document

The preceding discussion of applications and documents presented a number of concepts and described how the document related to the rest of the PenPoint operating system, but it didn't tell you much about what a document is (and what you must provide when you write an application).

A document consists of these components:

- The application code. The application code is the executable file and DLLs that you create. When loaded in PenPoint, it is loaded into a portion of memory allocated specifically for applications (it isn't stored in the RAM file system).

- A process. As described earlier, an active document is a process created by the PenPoint Application Framework. The process has a queue for messages, an entry point (main()), and an AppMain() routine.

- An application object, or an instance of an application class. The document object has a UID that can receive messages from the PenPoint Application Framework. The document object is the core of the document. It performs work for the document, creates any child objects that the document uses, and assigns work to the child objects.

- Component objects. Your application creates these objects from component classes and uses them to perform tasks and store data. You can write classes for your own component objects or you can use component classes that exist already (such as those provided by GO).

- A main window. A main window is a special window created by the PenPoint Application Framework for a document that contains a document's main window, a title bar, a page number, a menu bar, horizontal and vertical scroll bars, and other controls. Windows are described in Part 3: Windows and Graphics; frames are described in Part 4: UI Toolkit.

- A client window. A client window displays a document's data. In PenPoint, rather than have each object display its own data, you create window objects that display the data object (called views). This separation frees the data objects from having to repaint themselves, and allows the UI Toolkit to perform most of the work.

- Other window objects.
A directory. When the user first taps on the Create menu to create a document, the Notebook creates a directory in the file system. A directory attribute indicates the application class that the document uses. Later, when the user turns to the document, the Notebook uses the PenPoint Application Framework to start an instance of that application class.

Resource files. A document uses a resource file to store its instance data. Other resource files can contain replaceable objects and data used by the document. When the user turns away from a document, the only things that remain are the directory and the resource files (and the application code and DLLs).

Figure 7-9 illustrates the parts of a document process. On the left side of the figure are the document directory and the files in that directory (the object file, the resource file, and other data files). As in previous figures in this chapter, the box on the right represents the document process and the lozenges represent the individual objects within the process.
Chapter 8 / Life Cycles

The previous chapters discussed the concepts of where applications exist and what they consist of. This chapter describes the life cycles for an application class and a document.

At this stage, the discussion will begin to concentrate more on what you (the application developer) must do and what the PenPoint™ Application Framework does for you.

The following topics are covered in this chapter:

- Life cycle of an application class.
- The purpose of main in application installation.
- The purpose of clsAppMgr messages in application installation and deinstallation.
- Life cycle of a document.
- The purpose of main in document activation.
- The purpose of clsAppMgr and clsApp messages in document activation and termination.

Application Class Life Cycle

This section describes the life cycle for an application class. As described in Chapter 7, an application class is an instance of clsAppMgr that creates documents.

Figure 8-1 shows the states of an application class; Table 8-1 describes the states shown in the figure.
The following sections describe application installation in broad strokes. Application installation is described in more detail in Part 13: Installation API.

The transitions in Figure 8-1 are labeled. The user actions that cause these transitions begin when the user turns to the Connections notebook in applications view and performs the following actions:

1. Checks the Install check box in the applications view of the Connections notebook. The PenPoint operating system loads the application .EXE file from disk into the loader database, and creates the application class.

2. Clears the Install check box to deinstall the application. The PenPoint operating system deletes the application class and removes the .EXE file from memory.

## Installing an Application

The most important part of installing an application is creating the application class. Although most of your application contains methods (functions) that handle messages that documents can receive, your application is also responsible for sending msgNew to clsAppMgr to create a new application class.

When the user installs your application, the application installer loads the application executable file from a user-specified volume to the PenPoint computer. The object code isn't stored in the file system; rather, PenPoint stores the object code in an area of protected memory known as the loader database.

### main in Application Installation

The installer calls the kernel function OSProgramInstantiate() to create a process using your application and transfers control to your application's main routine, which is the process's entry point. main calls an application class initialization routine, which establishes space for the class instance data and declares the ancestor for instances of the class (usually clsApp).

main is the entry point for all application processes. OSProgramInstantiate() passes three parameters to main, argc, argv, and appProcess, which is a U16 that contains a process number. OSProgramInstantiate() assigns the process a process number of 0, because there are no other processes running this application code (at this time).
If `main` finds that the process number equals 0, it calls initialization routines for the application class and any other classes used by the application class. Finally, it calls `AppMonitorMain()`, which starts the application monitor for the application class.

If `main` finds that the process number is greater than 0, the application calls `AppMain`, a routine provided by PenPoint, which creates a document for that application.

This example shows the `main` routine used by the Tic-Tac-Toe application:

```c
void CDECL
main(
    int    argc,
    char * argv[],
    U16    processCount)
{
    if (processCount == 0) {
        TttSymbolsInit();
        StsWarn(ClsTttAppInit());
        StsWarn(ClsTttViewInit());
        StsWarn(ClsTttDataInit());
        AppMonitorMain(clsTttApp, objNull);
    } else {
        AppMain();
    }
    // Suppress compiler's "unused parameter" warnings
    Unused(argc); Unused(argv);
} /* main */
```

### Application Class Initialization Routine

When `main` finds that the process number is 0, it calls the initialization routine for that application class (and initializes any other classes required by the application).

The initialization routine for the application class creates an instance of `clsAppMgr`, which is the application class object.

The routine declares an APP_MGR_NEW structure, initializes it (by sending `msgNewDefaults` to `clsAppMgr`), modifies some of the arguments, and then sends `msgNew` to `clsAppMgr`.  

You must write the application class initialization routine.
This example shows the applications class initialization routine for the Tic-Tac-Toe application.

```pascal
STATUS PASCAL
ClsTttAppInit (void)
{
    APP_MGR_NEW new;
    STATUS s;
    ObjCallJump(msgNewDefaults, clsAppMgr, &new, s, Error);
    new.object.uid = clsTttApp;
    new.object.key = 0;
    new.cls.pMsg = clsTttAppTable;
    new.cls.ancestor = clsApp;
    new.cls.size = SizeOf(P_TTT_APP_INST);
    new.cls.newArgsSize = SizeOf(APP_NEW);
    new.appMgr.flags.stationery = true;
    new.appMgr.flags.accessory = false;
    strcpy(new.appMgr.company, "GO Corporation");
    // 213 (octal) is the "circle-c" copyright symbol
    new.appMgr.copyright = "\213 1992 GO Corporation, All Rights Reserved."
    ObjCallJump(msgNew, clsAppMgr, &new, s, Error);
    return stsOK;
}
```

The message arguments specify the application class's well-known UID and ancestor (which for applications is almost always clsApp), the name of the application, its initial state (whether it is a floating document, whether it runs in hot mode, and so on), the size of its instance data, and the ID of its method table.

If `new.appMgr.defaultDocName` contains pNull, the PenPoint Application Framework will look for the default document name in `tagAppMgrDefaultDocName` in the resource file APP.RES. This allows you to localize your default document names by having different resource files (and thus a different `tagAppMgrDefaultDocName`) for different languages or locales.

Because the application class uses a process, it can have a full-scale environment, just like an application instance. This environment includes a floating window list and initialized local copies of theProcessResList and theUndoManager. Using this environment, your process can actually provide its own user interface, if necessary for altering application-global settings. However, most of the time this overhead is unnecessary. When you don't need a full environment for process 0, set the fullEnvironment flag to false.

At this point, the process can send messages, but it has no object that can receive messages.
clsAppMgr creates the new object by sending msgInit to its ancestor, clsClass.

clsClass creates an application class object (an instance of clsAppMgr) for process 0. The new application class can now receive messages as well as send them.

clsClass then sends msgInit to the new instance of clsAppMgr (the application class object). In response to msgInit, the application class sends msgInit to its ancestor (clsClass) and then allocates and initializes its instance data.

When msgInit returns from clsClass, clsAppMgr initializes the instance data that your application class will use to create instances of itself (documents). The instance data includes your application name, your company’s name, your icon, the initial size and location for floating documents (if floating), whether it is a hot mode application or not, and whether the user can create instances of it or not. (Some applications, such as device drivers, should not be user-creatable.)

**AppMonitorMain() in Installation**

The last step that main executes in installing an application class is to create an application monitor in your application class.

The application monitor is an object in your application class that helps to maintain installation information about the application, including the UID of the application manager object and the location of the application’s home.

You create the application monitor by calling AppMonitorMain(). The application monitor is described in Chapter 13, The Application Monitor Class.

**Deinstalling an Application**

To remove an application from the PenPoint computer, the user deinstalls the application.

Deinstalling an application removes the program from the loader database and removes the application directory and the application’s attributes from the file system. To use the application again, the user must reinstall the program, its resources, and all other files.

**msgFree in Deinstallation**

When the user deinstalls your application, the installer program sends msgFree to your application class, which destroys the application class object.

clsAppMgr’s routine to handle msgFree first calls ancestor to free itself, then it removes itself from the list of installed applications, and broadcasts its deinstallation to all observers of clsApp.
Document Life Cycle

This section describes the life cycle for a document.

Figure 8-2 shows the states of a document; Table 8-2 describes the states shown in Figure 8-2.

Table 8-2

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Existent</td>
<td>The document does not exist, nor does it have a directory. There is no resource file for the document.</td>
</tr>
<tr>
<td>Created</td>
<td>A directory exists for the document.</td>
</tr>
<tr>
<td>Activated</td>
<td>A process exists, the process contains a document object, and the application data has been either initialized or restored.</td>
</tr>
<tr>
<td>Opened</td>
<td>The document is displayed on the screen.</td>
</tr>
<tr>
<td>Dormant</td>
<td>A directory exists for the document, the application data is stored in a resource file, but there is no process and no document object.</td>
</tr>
</tbody>
</table>
Created and dormant states are similar, because in those states there is a directory, but there is no process. However, in the created state there is no resource file; in dormant, there is.

The transitions shown in Figure 8-2 are labeled. The user actions that cause these transitions are:

1. The user taps on the Create menu and chooses this application. The PenPoint Application Framework creates a directory for the document.

2. The user deletes the document before turning to the document. The PenPoint Application Framework deletes the document directory.

3. The user turns to the page containing the document. The PenPoint Application Framework starts the document process, creates a document object in the process, and initializes the document’s data. If the document displays itself on screen, processing continues with transition 4.

Normally, transition 4 occurs immediately after transition 3 or transition 7 without additional user action. This transition can also occur when the user turns to a document running in hot mode or returns to a document that has an active selection.

4. The PenPoint Application Framework creates the document’s main window and its views, and displays the document on screen.

5. The user turns to another page. The PenPoint Application Framework removes the document from the screen and destroys its view objects. If the user didn’t have anything selected and the document is not in hot mode, processing continues with transition 6.

6. If the user didn’t have anything selected, this step continues from transition 5. This transition is also made if the user had something selected and then tapped anywhere on another page. The PenPoint Application Framework saves the document’s data and shuts down the process.

7. The user turned back to the document. The PenPoint Application Framework starts a new application process and reads the document data from the resource file. Usually when the document reaches the activated state, it continues on transition 4 to the opened state.

8. The user deleted the document. The PenPoint Application Framework deletes the resource file and removes the document directory.

The following sections describe how the PenPoint Application Framework works to create, save, and destroy instances of your document.
Creating a Document

When the user taps on the Create menu or chooses an item on the Stationery menu, the Notebook sends msgAppMgrCopy to your application class. msgAppMgrCopy creates a new subdirectory in the directory specified in the message. The message also stamps the directory with an attribute that indicates the application class that will use this directory.

At this point, the page exists in the Notebook Table of Contents, but does not contain any data, nor does any process exist for the document. The user might turn to the document immediately or might continue working on the current document.

Activating a Document

When the user turns the page to a document, the Notebook locates the directory that corresponds to that document and finds the directory attribute that identifies the document’s application class. The Notebook then sends msgAppMgrActivate to the application class.

Although this discussion centers on the Notebook, any application can start any other application in just the same way, that is, by sending msgAppMgrActivate to an application class. msgAppMgrActivate spawns a new process by calling OSProgramInstantiate().

Because there is at least one other process running the same application code (the application class), OSProgramInstantiate() increments the last process number that it generated and passes the new process number to that new process.

The process number merely distinguishes this process from other processes running the same application code. It does not indicate the number of times that the document has been reactivated, nor does it indicate the number of processes simultaneously running the application code.

main in Activation

The entry point for the new process is main(). msgAppMgrActivate transfers control to the new process at main(). Because the process number is greater than zero, main transfers control to AppMain(). In AppMain(), the process waits to receive something.

If the application has process-private classes, it creates and initializes the classes with separate initialization routines before calling AppMain().
This `main()` comes from the Tic-Tac-Toe application (TTTAPP).

```c
void CDECL main()
  
  int          argc,
  char *       argv[],
  UI6          processCount)
{    
  
    else {
        AppMain();
    }
    // Suppress compiler's "unused parameter" warnings
    Unused(argc); Unused(argv);
  } /* main */

At this point, the document consists of the items shown in Figure 8-3. The document directory exists and there is a process for the document.

**Figure 8-3**

**New Document Process**

SoundAppManagerActivate in Activation

In the meantime, the `SoundAppManagerActivate` that was sent by the Notebook builds a `pArgs` structure for `msgNew` and sends it to the new document process, using one of the kernel's ITC (inter-task communication) messages. (The process does not yet contain a document object to receive messages. That is why `msgAppManagerActivate` uses an ITC message.) The new process (which we last left waiting in `AppMain`), receives the `pArgs` and uses them to send a `msgNew` to your application class, using the arguments received in the ITC message.

Your application class handles `msgNew` by calling its ancestor, `clsClass`, which creates a document object (an instance of your application class) in the new process, and sends `msgInit` to the new object.
msgInit in Activation

Your application's method table for msgInit directs the class manager to pass the message to your ancestor (clsApp) before passing it to your method.

When clsApp receives msgInit, it opens the document's directory in the file system, allocates memory for the document's instance data, zeros the allocated memory, performs some housekeeping, updates its own instance data, and returns.

When your document receives msgInit, it initializes its instance data and saves the instance data in protected memory by calling ObjectWrite(), and then returns.

The instance data and application directory handle are both examples of activate-to-terminate objects that do not have state. That is, they do not need to be saved when the document terminates.

When your application initializes the instance data, the data is local to the method. However the instance data maintained by the class manager is in protected memory. To update the protected instance data with the local copy, call the function ObjectWrite(). ObjectWrite() stores the instance data in protected memory, where it will stay until the process is terminated.

This example shows the method for msgInit used by the Tic-Tac-Toe application.

```c
TttAppInit

Initialize instance data of new object.
Note: clsmgr has already initialized instance data to zeros.
*******************************************************************************/
#define DbgTttAppInit(x) \
  TttDbgHelper("TttAppInit",tttAppDbgSet,Ox2,x)
MsgHandler(TttAppInit)
{
  P_TTT_APP_INST pInst;
  STATUS s;
  DbgTttAppInit(""");
  // Initialize for error recovery.
  // pInst = pNull;
  // Allocate, initialize, and record instance data.
  // StsJmp(OSHeapBlockAlloc(osProcessHeapId, SizeOf(*pInst), &pInst), \n  //   s, Error);
  pInst->placeHolder = -1L;
  ObjectWrite(self, ctx, &pInst);
  DbgTttAppInit("returns stsOK")
  return stsOK;
  MsgHandlerParametersNoWarning;

  Error:
  if (pInst) {
      OSHeapBlockFree(pInst);
  }
  DbgTttAppInit("Error; returns Ox%lx",s))
  return s;
} /* TttAppInit */
```
**msgAppActivate and msgAppInit**

When the msgNew sent by AppMain() returns, AppMain() sends msgAppActivate to the document. You shouldn’t define a method for msgAppActivate; let clsApp handle it.

If this is the first time the document has been opened, msgAppActivate sends msgAppInit and msgAppSave to the document; if the document has been opened before, msgAppActivate sends msgAppRestore to the document.

- Your application’s method table should call ancestor (clsApp) before handling msgAppInit. clsApp creates the resource file and creates the main window. When the message returns from clsApp, your method should create and initialize any objects that both have instance data and that you need to save. For example, if your application uses an instance of clsText, it should create it and load any initial text at this time.

- Your application’s method table should call ancestor before handling msgAppSave. This will result in the document receiving msgSave. The method for msgSave should save the document’s instance data and any stateful objects to the resource file. msgSave is described later.

- Your application’s method table should call ancestor before handling msgAppRestore. This will result in the document receiving msgRestore. The method for msgRestore should restore the document’s instance data and any stateful objects from the resource file. msgAppRestore and msgRestore are described later.

This example shows the method for msgAppInit for the Tic-Tac-Toe application.

```c
#include "TttAppAppInit.x" \nTttAppAppInit(x) \nApp metrics am; \nTtt View new tttViewNew; \nBOOLEAN responsibleForView; \nBOOLEAN responsibleForScrollWin; \nOBJECT dataObject; \nOBJECT scrollWin; \nSTATUS s; \nTttAppAppInit("")
// // Initialize for error recovery. // tttViewNew.object.uid = objNull; scrollWin = objNull; responsibleForView = false; responsibleForScrollWin = false;
```
// Create and initialize view. This creates and initializes
// data object as well.
// ObjCallJmp(msgNewDefaults, clsTttView, &tttViewNew, s, Error);
ObjCallJmp(msgNew, clsTttView, &tttViewNew, s, Error);
responsibleForView = true;

// Check for stationery.
// ObjCallJmp(msgViewGetDataObject, tttViewNew.object.uid, 
// &dataObject, s, Error);
StsJmp(TttAppCheckStationery(dataObject), s, Error);

// Create and initialize scrollWin.
// StsJmp(TttUtilCreateScrollWin(tttViewNew.object.uid, 
// &scrollWin), s, Error);
responsibleForScrollWin = true;
responsibleForView = false;

// Make the scrollWin be the frame’s client win.
// ObjCallJmp(msgAppGetMetrics, self, &am, s, Error);
ObjCallJmp(msgFrameSetClientWin, am.mainWin, (P_ARGS)scrollWin, s, Error);
responsibleForScrollWin = false;

/* */
return stsOK;
MsgHandlerParametersNoWarning;

Error:
if (responsibleForView AND tttViewNew.object.uid) {
    ObjCallWarn(msgDestroy, tttViewNew.object.uid, pNull);
}
if (responsibleForScrollWin AND scrollWin) {
    ObjCallWarn(msgDestroy, scrollWin, pNull);
}
DbgTttAppAppInit(("Error; returns 0x%lx",s))
return s;
} /* TttAppAppInit */

clsApp also activates any embedded documents in the document being activated,
sets its priority, and returns.

Figure 8-4 shows the document at this point. The document process now
contains a number of the objects necessary for the document to be active in the
PenPoint Application Framework, however it doesn’t have any instances of
non-stateful components (such as views on data and user interface control objects).
When `msgAppActivate` returns, `AppMain()` opens the document by sending `msgAppOpen` to the new document.

**Opening a Document**

Usually, when the user turns to your document, processing continues immediately from transition 3 or transition 7, as shown in Figure 8-2.

However, as noted before, documents that run in hot mode or have an active selection might remain in the activated state and make transition 4 when the user turns back to the document.

**msgAppOpen**

When your document receives `msgAppOpen` from `AppMain()`, it creates windows to display data and other control objects. Most of the windows that your document creates here are described in either *Part 3: Windows and Graphics* or *Part 4: UI Toolkit* of the *PenPoint Architectural Reference*.

Your method table entry for `msgAppOpen` should call ancestor after your method handles the message. In response to `msgAppOpen`, `clsApp` displays the document on screen by sending `msgWinInsert` to the document’s main window.

At this point, the user can interact with the document.
This example shows the method for msgAppOpen used by the Tic-Tac-Toe application.

```c
#define DbgTttAppOpen(x) \
    TttDbgHelper("TttAppOpen", tttAppDbgSet, 0x40, x)
```

```c
MsgHandlerWithTypes(TttAppOpen, P_ARGS, PP_TTT_APP_INST)
{
    APP_METRICS am;
    OBJECT menu;
    BOOLEAN menuAdded;
    STATUS s;

    DbgTttAppOpen(""

    // Initialize for error recovery.
    menu = objNull;
    menuAdded = false;

    // Get app and frame metrics.
    ObjCallJmp(msgAppGetMetrics, self, &am, s, Error);

    // Create and add menu bar.
    StsJmp(TttUtilCreateMenu (am.mainWin, self, tttMenuBar, &menu), s, Error);
    DbgTttAppOpen("menu=0x%lx", menu);
    ObjCallJmp(msgAppCreateMenuBar, self, &menu, s, Error);
    StsJmp(TttUtilAdjustMenu(menu), s, Error);
    ObjCallJmp(msgFrameSetMenuBar, am.mainWin, (P_ARGS)menu, s, Error);
    menuAdded = true;

    DbgTttAppOpen("returns stsOK")
    return stsOK;
    MsgHandlerParametersNoWarning;

    Error:
    if (menuAdded) {
        ObjCallWarn(msgFrameDestroyMenuBar, am.mainWin, pNull);
    } else if (menu) {
        ObjCallWarn(msgDestroy, menu, pNull);
    }
    DbgTttAppOpen("Error; return 0x%lx", s)
    return s;
} /* TttAppOpen */
```
When `msgAppOpen` returns, `AppMain()` sends `msgAppDispatch` to the document. This makes the document eligible to receive other messages.

Figure 8-5 shows the fully opened document process, complete with non-stateful objects.

![Diagram of Document Life Cycle](image)

**Closing a Document**

When the user turns away from the document, the Notebook sends `msgAppClose` to the document, which causes it to remove itself from the screen.

If the document wasn't running in hot mode or didn't have a selection, the Notebook also sends `msgFree` to the document, which causes the document to save its instance data and destroy its objects.

**msgAppClose**

Your method for `msgAppClose` must remove your application from the screen by setting the client window pointer to nil and freeing the window. The method for `msgAppClose` should also destroy any non-stateful window objects.

If the document has an active selection in one of the windows, that window object must be preserved.
This code example shows the method for \texttt{msgAppClose} used by the Tic-Tac-Toe application.

```c
#define DbgTttAppClose(x) \ 
    TttDbgHelper("TttAppClose", tttAppDbgSet, 0x80, x)

MsgHandlerWithTypes(TttAppClose, P_ARGS, PP_TTT_APP_INST)
{
    APP_METRICS am;
    STATUS s;
    DbgTttAppClose ("")
    // Get the frame. Extract the menu bar from the frame. Then free the menu bar.
    // ObjCallJmp(msgAppGetMetrics, self, &am, s, Error);
    ObjCallJmp(msgFrameDestroyMenuBar, am.mainWin, pNull, s, Error);
    DbgTttAppClose("returns stsOK")
    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

**Terminating a Document**

Usually, when the Notebook closes a document, it also terminates the document process (unless there is a window with an active selection or the document is running in hot mode).

If there was a window with an active selection and the user taps elsewhere in the Notebook, the selection is lost and termination continues.

The Notebook does not terminate a hot-mode document until the user deletes it.

The Notebook terminates a document by sending it \texttt{msgFree}.

**msgFree**

When your application receives \texttt{msgFree}, it should free any objects that it has created and release any data allocated from heap.

Your application's method table should call ancestor after handling \texttt{msgFree}.

When \texttt{clsApp} receives \texttt{msgFree}, it sends \texttt{msgAppSave} to self (the document). Your
application should not handle \texttt{msgAppSave}. If you omit \texttt{msgAppSave} from the method table, the class manager will send it to \texttt{clsApp}.

When \texttt{clsApp} receives \texttt{msgAppSave}, it performs the following tasks:

1. Tells the embedded documents to save themselves (by self-sending \texttt{msgAppSaveChildren} to each one).

2. Sends \texttt{msgNew} to \texttt{clsResFile} to create a handle on the document state resource file in the document’s directory. Note that when \texttt{clsApp} sends \texttt{msgNew} to \texttt{clsResFile}, it specifies “truncate-on-open,” that is, delete all information in the file. From this point, until all objects have saved their information in the file, a drastic failure (such as loss of power or a crash) in the computer could cause a loss of data.

If data integrity is important, your application could copy its resource file when it receives \texttt{msgAppSave} (provided it knows the name of the resource file).

When \texttt{clsResFile} receives \texttt{msgNew}, it writes a resource file header to the file.

3. Sends \texttt{msgResWriteObject} to the resource file handle and specifies self as the object to file. When the resource file handle receives \texttt{msgResWriteObject}, it:
   - Writes a resource file header that identifies the object.
   - Sends \texttt{msgSave} to the document. Your application’s responsibilities for \texttt{msgSave} are described below.

   When \texttt{msgSave} completes, the resource manager updates the number of bytes written to the resource file and returns \texttt{stsOK} for \texttt{msgResWriteObject}.

4. When \texttt{msgResWriteObject} returns, \texttt{clsApp} closes the resource file.

5. Finally, \texttt{clsApp} sends \texttt{msgAppTerminateProcess} to your document. Allow your ancestor to handle this message, which terminates the document process.

When the document process terminates, the only thing that remains is the directory in the file system, the resource file in the directory, and the resource files for the document (plus any other files created by your document). Figure 8-6 shows what remains of the document.
This example shows the method used by the Tic-Tac-Toe application to handle msgFree.

```c
TttAppFree

Respond to msgFree.
Note: Always return stsOK, even if a problem occurs. This is
(1) because there's nothing useful to do if a problem occurs anyhow
and (2) because the ancestor is called after this function if and
only if stsOK is returned, and it's important that the ancestor
get called.
```

```c
#define DbgTttAppFree(x) 
    TttDbgHelper("TttAppFree", tttAppDbgSet, 0x4, x)
```

```c
MsgHandlerWithTypes(TttAppFree, P_ARGS, PP_TTT_APP_INST)
{ 
    DbgTttAppFree("")
    OSHeapBlockFree(*pData);
    DbgTttAppFree("returns stsOK")
    return stsOK;
    MsgBoxHandlerParametersNoWarning;
} /* TttAppFree */
```

```c
msgSave
```

msgSave passes an OBJ_SAVE structure, which contains a handle on the file where you must store your data.

Your application's method table should call ancestor before handling msgSave. msgSave initiates the following process:

1. When clsApp receives msgSave, it does a ObjectCallAncestor to file its instance data.
2. When ObjectCallAncestor returns, clsApp sends msgResWriteObject to the resource file handle with the document's main window as the object.
3. clsWin files any of its child objects that have wsSendFile set.
4. clsFrame files its client view, if the view has wsSendFile set.
5. The view receives msgSave and saves its instance data (the UID of its data object).
6. The view sends msgResPutObject or msgResWriteObject to its data object.

This files your window data and its client window. Because views file their corresponding data objects, your document doesn't have to save these objects.

When your application receives msgSave, it should save its instance data by sending msgStreamWrite to the resource file handle; the pArgs indicate the data to save.

Finally, your application should save its stateful objects by sending msgResWriteObject or msgResPutObject to the resource file handle; the pArgs indicate the object to save. These resource file messages then send msgSave to the specified object.
This example shows the method for msgSave used by the Tic-Tac-Toe application.

```c
#define DbgTttAppSave(x) \
   TttDbgHelper("TttAppSave", tttAppDbgSet, 0x8, x)
MsgHandlerWithTypes(TttAppSave, P_OBJ_SAVE, PP_TTT_APP_INST)
{
   TTT_APP_FILED_0 filed;
   STATUS s;
   DbgTttAppSave(""");
   StsJmp(TttUtilWriteVersion(pArgs->file, CURRENT_VERSION), s, Error);
   TttAppFiledDataOFromInstData(*pData, &filed);
   StsJmp(TttUtilWrite(pArgs->file, SizeOf(filed) , &filed), s, Error);
   DbgTttAppSave("returns stsOK")
   return stsOK;
   MsgBoxHandlerParametersNoWarning;
   Error:
   DbgTttAppSave("Error; return Ox%lx",s)
   return s;
} /* TttAppSave */
```

### Reactivating a Document

If the user turns back to the document, the notebook sends msgAppMgrActivate to your application class. Processing is just the same as in document activation. However, instead of sending msgAppInit to self, clsApp responds to msgAppActivate by self-sending msgAppRestore.

Your application shouldn't handle msgAppRestore. Omit the message from your method table so that the class manager will pass it to your application's ancestor, clsApp.

When clsApp receives msgAppRestore, it starts the cascade of events that causes all objects in a document to load themselves:

1. One of the msgAppRestore arguments is a directory handle. When clsApp receives msgAppRestore, it looks in the directory for a resource file. When it finds the resource file, it opens it by sending msgNew to clsResFile.

2. When clsResFile receives msgNew, it opens the file. If the resource file index is saved in the file, clsResFile reads it in. If the index does not exist, clsResFile creates one.

3. When the msgNew returns, clsApp sends msgResReadObject to the resource file handle, specifying self as the object to restore.

4. As described below, your application's method table passes msgRestore to its ancestor first, which causes windows and client views with wsSendFile set to receive msgRestore. The object then reads its state information and uses msgResGetObject to re-create its objects.
clsApp determines the name of the directory and resource file, creates a file handle on the resource file, and sends `msgResReadObject` to the resource file handle. The `pArgs` for `msgResReadObject` indicates `self` is the object to read.

msgRestore

`msgResReadObject` sends `msgRestore` to `self` (the document); the `pArgs` includes the handle on the resource file.

Your application's method table should pass `msgRestore` to its ancestor before handling it, which causes the main window to read its instance data. The main window also tells all its subwindows to read their objects back in. (This is similar to how the main window uses `msgSave` to save all of its subwindows.)

When your application receives `msgRestore`, it should read its instance data from the resource file (by sending `msgStreamRead` to the resource file handle) and should restore its stateful objects (by sending `msgResReadObject` to the resource file handle).

When your application reads the instance data, the data is local to the method. To write that local copy to protected memory, you must call `ObjectWriteO`.

This example shows the method for `msgRestore` used by Tic-Tac-Toe:

```c
int TttAppRestore(P_OBJ_RESTORE pArgs) {
    P_TTT_APP_INST pInst;
    TTT_APP_FIELDED_0 filed;
    STATUS s;
    TTT_VERSION version;
    DbgTttAppRestore(""); // Initialize for error recovery.
    pInst = pNull; // Read version, then read filed data. (Currently there's only
    // only one legitimate file format, so no checking of the version
    // need be done.)
    // The allocate instance data and convert filed data.
    StsJmp(TttUtilReadVersion(pArgs->file, MIN_VERSION, MAX_VERSION,
        &version), s, Error);
    StsJmp(TttUtilRead(pArgs->file, SizeOf(filed), &filed), s, Error);
    StsJmp(OSHeapBlockAlloc(osProcessHeapId, SizeOf(*pInst), &pInst),
        s, Error);
    TttAppInstDataFromFiledDataO(&filed, pInst);
    return s;
}
```

You must write the method to handle `msgRestore`.

---

# 5

**PENPOINT ARCHITECTURAL REFERENCE**

Part 2 / PenPoint Application Framework

---

**8.2.6.1**
ObjectWrite(self, ctx, &pInst);
DbgTttAppRestore("returns stsOK")
return stsOK;
MsgHandlerParametersNoWarning;

Error:
    if (pInst) {
        OSHepBlockFree(pInst);
    }  
DbgTttAppRestore("Error; returns Ox%lx",s))
return s;
} /* TttAppRestore */

# Deleting a Document

When the user deletes a document from the Notebook, the Notebook sends msgAppMgrDelete to the application class, indicating the directory of the document to be deleted.

msgAppMgrDelete searches the document’s directory for embedded documents. For each embedded document, msgAppMgrDelete sends msgFree to each embedded document (if it exists), terminates the document processes (if it exists), and deletes the document’s directory.
Chapter 9 / Embedded Documents

One of the features that differentiates the PenPoint™ operating system from many other operating systems is that components can contain other components. Both the containing component and the contained component can have views on their data. The contained component with a view on its data is called an embedded window.

Chapter 9 covers these topics:
- Embedded window marks.
- Descendants of clsEmbeddedWin.
- Moving and copying between embedded windows.
- Creating and destroying embedded windows.
- Moving or copying to an embedded window.
- Handling child embedded windows.
- Selecting an embedded window.

**Embedded Window Concepts**

Embedded windows are described by clsEmbeddedWin, which is a descendant of clsGWin.

As clsApp provides the default behavior for applications (save, restore, open, close), clsEmbeddedWin provides default behavior for views (move, copy, reference).

There are many examples of embedded windows in the PenPoint operating system:
- Any view is actually an embedded window. Thus, text views are embedded windows.
- Each line in the table of contents is a view.
- Reference (goto) buttons are embedded windows that contain an additional mark on another document.

The user can move or copy information directly from any other view to an embedded window (because all views descend from the embedded window class).

In move or copy operations, the source and destination components must agree on the data exchange method and the destination component must be able to present the data in a reasonable way. Is it easy to imagine a user moving data from a text view into a spreadsheet view (perhaps as a label), but it is harder to conceive of a user copying a spreadsheet formula into a drawing view.

*Note* Reference buttons are called “goto” buttons in the PenPoint header files.
To enable the components to agree on a data exchange method and to provide a context for moving or copying the data, clsEmbeddedWin provides default behavior for the move, copy, insert reference button, and other gestures.

**Marking Support**

The embedded window marks also provide support for application marking. When a marking operation finds an embedded window, it must report its find to the mark driver and then return to that location. clsMark uses the embedded window marks to save its locations.

**Descendants of clsEmbeddedWin**

Among the descendants of clsEmbeddedWin are clsGotoButton, clsView, and clsAppWin.

- clsGotoButton implements the reference (goto) buttons.
- clsView is an abstract class that defines messages used to display an object’s data. The descendants of clsView include clsTextView and clsBrowView. This is how text views or the browser responds to move and copy gestures.
- clsAppWin provides the interface for embedded applications. If you embed an application, you must create a clsAppWin object.

The difference between clsEmbeddedWin and clsAppWin is that an embedded window allows you to embed a component in a document (such as a reference button), whereas an application window allows you to embed an application in a document (such as a graph application within a text document). When you embed an application, it affects the file system directory tree because an embedded application maintains its data in a subdirectory of the containing application.

**Moving or Copying Between Embedded Windows**

One of the most important features of clsEmbeddedWin is its implementation of the move and copy protocol.

The user can select data in a document (or an entire document) and move or copy that data to another document that allows embedding. This section describes the messages that are exchanged when copying or moving data from one object to another.

Throughout this discussion, remember that both the source and destination inherit from clsEmbeddedWin. When your class inherits from clsEmbeddedWin, instances of your class can be either a source or a destination.

**Separate Messages**

The messages used by clsEmbeddedWin are usually sent to self. If, when the message is sent to self, the object that received the message cannot (or chooses not) to handle the message, it can allow its ancestor to handle the message. Eventually
the message is handled by clsEmbeddedWin, which provides default behavior for move and copy operations.

Usually, a class specifies that a message can be sent to its ancestor in the method table for the class. However, classes can also call ancestor from their methods.

At first impression, it seems that the messages defined by clsEmbeddedWin could be combined to make them more efficient. However, many of the tasks are separated so that the messages can be sent to self, again allowing classes that inherit from clsEmbeddedWin to implement their own behavior for each message.

By defining a separate message for each of the tasks, clsEmbeddedWin also separates two types of translations that occur when the user makes a gesture:

- The translation from gestures to semantics.
- The translation from semantics to implementation.

Future PenPoint or third-party products can then replace either or both of these translations with their own sets of gestures, semantics, or implementations.

**Why Use the Move and Copy Protocol?**

The move and copy protocol enables clsEmbeddedWin to provide default behavior for moving or copying many types of data. The source could be: a character, a word, a value, a sentence, a spreadsheet column, a file, a reference button, or even an entire document. The destination can be a location in: a text document, a spreadsheet, a directory, or a table of contents.

The advantage to always using the move and copy protocol is that you can intercept messages where you want your application to do the work, and you allow your ancestor to handle the messages when you want clsEmbeddedWin to do the work. When moving a reference button, you allow clsEmbeddedWin to do all the work. However, when moving your own data, you might let clsEmbeddedWin handle the move or copy icon, but you might want to do the actual move or copy of data.

Usually, objects use the transfer protocol defined by clsTransfer, which is documented in *Part 9: Utility Classes*.

**The Move and Copy Protocol**

There are nine essential steps for moving or copying data:

1. The user selects the data and requests a move or copy operation.
2. The source determines what was selected.
3. The source allows the user to indicate the destination.
4. The source tells the destination object to move or copy the selection.
5. The destination determines the type of data that the source has selected and tells the source to send the data.
6. The source determines whether it can move or copy data to the destination.
The source determines the file system location of the destination.
The source moves or copies the data.
The destination determines exactly where to put the data.

The following discussion of the move and copy protocol simply describes the messages used to implement the protocol. The next section describes a move operation that actually uses the messages.

**Requesting a Move or Copy**

After making a selection, the user can start a move or copy in two ways:
- By making a gesture (press or tap and press) on the selection.
- By choosing Move or Copy on the Edit menu.

If the user holds the pen to the selection, the owner of the selection receives a msgPenHoldTimeout input event. If the tap count is 0, the user wants to move the selection; the selection owner sends msgSelBeginMove to self. If the tap count is 1, the user wants to copy it; the selection owner sends msgSelBeginCopy to self.

If the user chooses Move or Copy from the Edit menu:
- clsApp sends msgAppMoveSel or msgAppCopySel to self.
- Your application allows its ancestor (clsApp) to handle the message.
- clsApp receives the message and sends msgSelBeginMove or msgSelBeginCopy to the owner of the selection.

**Identify the Selection**

When your application receives msgSelBeginMove or msgSelBeginCopy (either from self or from the standard application menu), it is in move or copy mode.

The class that handles the message should identify the current selection by:
- Locating the x-y position of the selection and the bounds of the selection.
- Storing the x-y and bounds in an EMBEDDED_WIN_BEGIN_MOVE_COPY structure.
- Sending msgEmbeddedWinBeginMove or msgEmbeddedWinBeginCopy to self.

clsEmbeddedWin uses the bounds of the selection to determine the type of move or copy icon to use. If the bounds are non-zero, the icon is a marquee around the selected area; otherwise clsEmbeddedWin uses a default move or copy icon.

**Let the User Indicate the Destination**

When the source receives msgEmbeddedWinBeginMove or msgEmbeddedWinBeginCopy, it must provide a way for the user to indicate the destination (UID of the object that will receive the data and the x-y coordinates on screen).

clsEmbeddedWin does this by providing the move or copy icon.
Tell the Destination to Move or Copy

When the user indicates the destination, the class that prompted the user sends `msgSelMoveSelection` or `msgSelCopySelection` to the destination.

The source does not necessarily have to be the object that sends `msgSelMoveSelection` or `msgSelCopySelection`. If a destination object is able to determine where the user wants to place the data, it can send one of these messages to self.

For example, if your application maintains an insertion point and supports a Paste command, the Paste command could send `msgSelMoveSelection` or `msgSelCopySelection` to self, specifying the location of the insertion point as the x-y location.

Determining the Data Type

When the destination receives `msgSelMoveSelection` or `msgSelCopySelection`, it should send `msgXferList` to the source to determine the transfer type. At this point, the embedded window move and copy protocol uses the transfer protocol documented in Chapter 5, Class Transfer, in Part 9: The Utility Classes.

The object that handles `msgXferList` must add its transfer types to the list in the messages arguments and send the message to its ancestor (so that the ancestor classes can add their transfer types).

When the destination receives the completed list, it looks for compatible transfer types. (If the data transfer types sent back by the source are unknown by the destination, the object at the destination should call ancestor. Its ancestor can send `msgXferList` to the source again and examine the returned list for compatible transfer types.)

When the destination finds an acceptable transfer type, it determines what protocol it will use for the move or copy operation.

If the transfer type was defined by `clsXfer` (for instance, `xferString`) the destination sends `msgXferGet` to the source. `msgXferGet` is the next step in the `clsXfer` protocol.

If the transfer type was `clsEmbeddedWin`, the destination sends `msgEmbeddedWinMove` or `msgEmbeddedWinCopy` to the source. `msgEmbeddedWinCopy` is the next step in the `clsEmbeddedWin` protocol.

OK the Move or Copy

Continuing with the description of the `clsEmbeddedWin` protocol, when the owner of the source receives `msgEmbeddedWinMove` or `msgEmbeddedWinCopy`, it determines whether it can perform the move or copy by sending `msgEmbeddedWinMoveCopyOK` to self.

The owner of the source receives `msgEmbeddedWinMoveCopyOK` and uses the destination’s UID to determine whether it can move or copy to the destination.
object. For example, a graphic component might reject a request to move to a text object.

The class that handles msgEmbeddedWinMoveCopyOK sends back values that indicate whether it can allow move and copy operations.

The source examines the values and, depending on the type of operation and the value, continues or returns an error.

**Getting the Destination in the File System**

If the move or copy is allowed, source sends msgEmbeddedWinGetDest to the destination to get more specific information about the destination (rather than just the location of the hot spot).

The destination receives the message and sends back a file system locator for its instance directory, a path to the embedded window (if any), and the sequence number in that directory.

When moving documents in the table of contents, the directory and sequence number information are important. They indicate specifically which directory the document will be moved to and the position of the moved document within that directory.

The selection owner needs the destination information to update its references to UUIDs (universal unique identifiers) in the global index table. When moving or copying components that contain or are targets for reference buttons, the PenPoint operating system must be able to redirect the reference button to the new location of the component.

The directory and sequence number information is also important in case the selection includes an embedded application. When the selection is moved, an embedded application’s directory structure must also be moved.

**Moving or Copying the Data**

The source sends its information to the destination using a protocol specific to the transfer type:

- For one-shot and stream transfers, the source uses clsXfer messages.
- For transfers that use special protocols (other than the embedded window protocol), the source and destination exchange data using that protocol.

For transfers that use the embedded window protocol, the message sent by the source depends on the type of move or copy:

- If the move or copy is within the same object, the source sends msgEmbeddedWinMoveChild to the destination.
- If the move or copy is between different objects in the same process, the source sends msgEmbeddedWinInsertChild to the destination.
CHAPTER 9 / EMBEDDED DOCUMENTS

Example: Moving in Tic-Tac-Toe

- If the move or copy is between different processes, the source files the item, sends msgEmbeddedWinRestoreChild to the destination, and then sends msgEmbeddedWinInsertChild to the destination.

Getting the Exact Pen Location

One way to allow the user to indicate the destination is to create a marquee or label that the user can move with the pen and drop at the location. However, for two reasons, this creates a problem when determining the exact location:

- The user can put the pen down anywhere in the label to begin the move.
- When your application gets the x-y position of the label, the returned value is the lower left corner of the label, not the pen position.

To compensate for this imprecision, the destination sends msgEmbeddedWinGetPenOffset to the source. When clsEmbeddedWin created the label and the user put the pen down on the label, clsEmbeddedWin saved the x and y offsets from the pen position to the lower left corner of the label (as part of the embedded window instance data).

When the source receives msgEmbeddedWinGetPenOffset, it allows its ancestor to handle the message. clsEmbeddedWin sends back the offset values.

The destination can then use those offset values to calculate the exact location to insert the moved or copied item.

Example: Moving in Tic-Tac-Toe

The sample Tic-Tac-Toe application implements the move and copy protocol (by allowing clsEmbeddedWin to handle most of the protocol). The user can use the move or copy menu items or make a move or copy gesture on the Tic-Tac-Toe board. With the help of clsEmbeddedWin, the Tic-Tac-Toe application creates a default move or copy icon, which the user can use to move the data in the document to a new document. If the destination document is a MiniText document, the data becomes a string of nine characters (Xs, Os, and spaces) representing the characters on the board.

The User Requests a Move

To begin the example, the user either:

- Makes a hold gesture.
- Selects Move from the Edit menu.

Either way, the gesture translation system sends msgSelBeginMove to the Tic-Tac-Toe view.
Presenting a Move/Copy Icon

When the Tic-Tac-Toe view receives msgSelBeginMove, it handles the gesture with in the function TttViewSelBeginMoveAndCopy(). If Tic-Tac-Toe allowed clsEmbeddedWin to handle the message, it would create a marquee around the entire board (the entire view), which is too large to drag easily to another document. Instead, TttViewSelBeginMoveAndCopy():

- Creates an EMBEDDED_WIN_BEGIN_MOVE_COPY structure.
- Sets the x-y coordinates in the structure using its pArgs.
- Sets the bounds in the structure to zero.
- Sends msgEmbeddedWinBeginMove to self.

```c
#define DbgTttViewSelBeginMoveAndCopy(x) 
   TttDbgHelper("TttViewSelBeginMoveAndCopy",tttViewXferDbgSet,Oxl,x)
```

```
MsgHandlerWithTypes(TttViewSelBeginMoveAndCopy, P_XY32, PP_TTT_VIEW_INST)
{
    EMBEDDED_WIN_BEGIN_MOVE_COPY bmc;
    STATUS s;

    DbgTttViewSelBeginMoveAndCopy("self=0x%lx",self)
    //
    // If we don't handle this message, the default behavior is to
    // draw a marquee around the entire selection. For ttt, the marquee
    // would stretch around the entire board, which is too large to be
    // easily dragged into another document. So, we handle this message,
    // and set the bounds of the move/copy area to zero.
    // msgEmbeddedWinBeginMove/Copy will know to display a move/copy icon
    // instead of drawing the marquee.
    //
    if (pArgs) {
        bmc.xy = *pArgs;
    } else {
        bmc.xy.x =
        bmc.xy.y = 0;
    }
    bmc.bounds.origin.x =
    bmc.bounds.origin.y =
    bmc.bounds.size.w =
    bmc.bounds.size.h = 0;
    ObjCallJmp(MsgEqual(msg, msgSelBeginMove) ?
       msgEmbeddedWinBeginMove : msgEmbeddedWinBeginCopy,
       self, &bmc, s, Error);
    DbgTttViewSelBeginMoveAndCopy("returns stsOK")
    return stsOK;
    MsgHandlerParametersNoWarning;

    Error:
    DbgTttViewSelBeginMoveAndCopy("Error; return 0x%lx",s)
    return s;
} /* TttViewSelBeginMoveAndCopy */
The Tic-Tac-Toe view does not handle msgEmbeddedWinBeginMove, but allows its ancestor to handle the message. When clsEmbeddedWin handles msgEmbeddedWinBeginMove it:

- Detects that the bounds are zero, so it creates a move or copy icon.
- Inserts the icon into the Tic-Tac-Toe view.
- Returns.

At this point, the current process is the icon instance.

**The Move or Copy Icon**

The move or copy icon is an instance of clsMoveCopyIcon. It is the responsibility of this class to handle pen input to move the icon on the screen, and to detect the pen-up event when the user has chosen the exact destination for the move or copy.

The move or copy icon is the only place in moving a reference button where the move and copy protocol is not handled by clsEmbeddedWin.

When the user lifts up on the pen, clsMoveCopyIcon sends msgMoveCopyIconDone to self. The instance of clsMoveCopyIcon handles the message by determining the UID of the destination and sending msgSelMoveSelection to the destination.

When the destination receives msgSelMoveSelection, it sends msgXferList to the source.

**Determining the Data Type**

The Tic-Tac-Toe view handles msgXferList by calling the function TttViewXferList(). The function adds the string type to the list and returns. clsTttView doesn't allow its ancestors to handle the message, to prevent them from adding other data types to the transfer list.

```c
/* ***********************************************
   TttViewXferList
 *********************************************** *

static TAG
sourceFormats[] = {xferString};

#define N_SOURCE_FORMATS (SizeOf(sourceFormats) / SizeOf(sourceFormats[0]))

#define DbgTttViewXferList(x) 
   TttDbgHelper("TttViewXferList", tttViewXferDbgSet, 0x4, x)

MsgHandlerWithTypes(TttViewXferList, OBJECT, PP_TTT_VIEW_INST) {
   STATUS s;

   DbgTttViewXferList("self=0x%lx", self) 
   // Don't let ancestor add types. We aren't interested in
   // moving/copying the window, which is the only type the
   // ancestor supports.
   //
   StsJmp(XferAddIds(pArgs, sourceFormats, N_SOURCE_FORMATS), s, Error);
```
Moving the Data

When the destination receives the list, it selects the transfer type that it can handle. In the case of Tic-Tac-Toe, the only transfer type on the list should be xferString. Because xferString is a type defined by clsXfer, the destination uses the clsXfer protocol. Therefore, the destination sends msgXferGet to the Tic-Tac-Toe view.

The Tic-Tac-Toe view responds to msgXferGet by packing its data into a string and passing the buffer back when it returns.

```c
#define DbgTttViewXferGet(x) 
   TttDbgHelper("TttViewXferGet", tttViewXferDbgSet, 0x2, x)

MsgHandlerWithTypes(TttViewXferGet, P_XFER_FIXED_BUF, P_TTT_VIEW_INST)
{
   STATUS s;

   DbgTttViewXferGet("self=0x%lx",self)

   if (pArgs->id == xferString) {
      OBJECT dataObj;
      TTT_DATA_METRICS dm;
      U16 row;
      U16 col;
      P_XFER_FIXED_BUF p = (P_XFER_FIXED_BUF)pArgs;
      ObjCallJump(msgViewGetDataObject, self, &dataObj, s, Error);
      ObjCallJump(msgTttDataGetMetrics, dataObj, &dm, s, Error);

      // initialize the length to the number of squares (9) plus 1
      // to allow for a string termination character (just in case
      // the user copies/moves the string into a text processor.
      //
      p->len = 10;
      p->data = 0L;
      for (row=0; row<3; row++) {
         for (col=0; col<3; col++) {
            p->buf[(row*3)+col] = dm.squares[row][col];
         }
      }
      p->buf[9] = '\0';
      s = stsOK;
   } else {
      s = ObjectCallAncestorCtx(ctx);
   }
}
DbgTttViewXferGet(("returns 0x%lx",s))
return s;
MsgHandlerParametersNoWarning;

Error:
DbgTttViewXferGet(("Error; return 0x%lx",s))
return s;
} /* TttViewXferGet */

**Intercepted Messages**

clsEmbeddedWin intercepts msgWinSend and msgGWinGesture.

It responds to msgWinSend so that it can modify UUID data in msgEmbeddedWinGotoChild messages. Essentially, it modifies the UUID data so that “go to” operations to a nested embedded window perform the window location operations internally, before displaying the reference location on the screen.

As mentioned in the description of a move operation above, clsEmbeddedWin responds to msgGWinGesture. The class does not respond to all gestures, only those that have meaning for embedded windows.

<table>
<thead>
<tr>
<th>Gesture</th>
<th>Gesture Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td>xgsDbiCircle</td>
<td>Create reference button</td>
</tr>
<tr>
<td>⊃</td>
<td>xgsUpCaret</td>
<td>Show stationery menu</td>
</tr>
<tr>
<td>⊃</td>
<td>xgsUpCaretDot</td>
<td>Show stationery menu</td>
</tr>
<tr>
<td>⊂</td>
<td>xgsDbiUpCaret</td>
<td>Show stationery menu</td>
</tr>
<tr>
<td>⚔</td>
<td>xgsCircleCrossOut</td>
<td>Undo last saved action</td>
</tr>
<tr>
<td>⌚</td>
<td>xgsVertCounterFlick</td>
<td>Toggle borders and controls</td>
</tr>
<tr>
<td>✅</td>
<td>xgsCheck</td>
<td>Show option sheet</td>
</tr>
<tr>
<td>🎁</td>
<td>xgsUGesture</td>
<td>Show option sheet</td>
</tr>
<tr>
<td>🎁</td>
<td>xgsFGesture</td>
<td>Start Find operation</td>
</tr>
<tr>
<td>⌚</td>
<td>xgsCircleFlickUp</td>
<td>Start Find operation</td>
</tr>
<tr>
<td>🎁</td>
<td>xgsCircleFlickDown</td>
<td>Start Find operation</td>
</tr>
<tr>
<td>⚔</td>
<td>xgsSGesture</td>
<td>Start spell check</td>
</tr>
<tr>
<td>⚔</td>
<td>xgsPGesture</td>
<td>Start spell check</td>
</tr>
</tbody>
</table>

clsEmbeddedWin sends all other gestures to its ancestor.
Some abstract messages defined in \texttt{clsSelection} are actually part of the embedded window protocol. These messages are not sent to the owner of the selection, nor are they sent to the selection manager. Rather, they are self-sent by the target of a move, copy, or create reference button operation.

These messages are defined in \texttt{clsSelection} because they indirectly relate to the selection; by defining these messages in \texttt{clsSelection}, others do not have to define their own messages to do the same thing.

If you intend to handle \texttt{msgSelMoveSelection}, \texttt{msgSelCopySelection}, or \texttt{msgSelMarkSelection}, you must perform these tasks:

\begin{itemize}
  \item Establish the owner of the selection.
  \item Use \texttt{clsXfer} messages to obtain the selected data from the owner of the selection.
  \item If the operation was a move, ensure that the owner of the selection knows to delete its copy of the data.
  \item Insert the data at the x-y location specified by the move, copy, or mark message.
\end{itemize}
Chapter 10 / Mark Concepts

In the PenPoint™ operating system, the data for most documents is stored in one or more components (which usually inherit from clsEmbeddedWin). For example, individual words can be stored in a text component, or shapes can be stored in a drawing component. However, because components contain the data, only the component knows how the data is stored.

Of course, there are times when other objects will want to manipulate the data within a component. Reference buttons, for example, allow the user to link to a specific point within a data item in a component; later, the user can use the reference button to turn back to that very same point—even if the user has added data before the point.

Another reason to want to manipulate data within a component is in traversal operations, such as checking spelling, or searching and replacing. In these operations, the traverser needs a mechanism for keeping track of its location within a component that it knows nothing about.

To mark data within a component, the PenPoint operating system provides the mark class (clsMark). Traversal drivers can use the mark class to traverse applications and embedded windows.

All applications or components that allow themselves to be searched or spell-checked must handle clsMark messages, which implement the component half of the protocol. If an application or component implements new functionality similar to search and replace or spell-checking, it uses clsMark.

Mark Class

The mark class (clsMark) inherits from clsObject. Its messages, #defines, and data structures are defined in MARK.H.

A mark (which is an instance of clsMark) provides a way to refer to data items within components. The data item in a component referenced by the mark is called the target, because it is this data that the creator of the mark is really interested in. Notice that the target is not an object.

The object that uses the mark is called the holder of the mark.

For example, in the case of a reference button, the piece of information identified by the reference button is the target. The reference button itself is the holder of the mark.
Parts of a Mark

There are two parts to a mark (as shown in Figure 10-1):

- A component part. This identifies the component that contains the target. The component part is managed by dsMark.
- A token part. This is used by the component to identify the specific location of the target within instances of itself. The token part is managed by the component.

The component part of a mark contains:

- The UUID of the application containing the component.
- The UUID of the component that contains the target.
- The UID of the component that contains the target.

The token part of a mark contains:

- The UID of the class that contains methods to locate the target.
- Two 32-bit index values that are manipulated by the class.

Implementing Tokens

The token part of a mark is managed by the component. When you implement a component that handles marks, you must establish a relationship between the index values in the token and the actual location of the target within the component. In a simple relationship, a database might use the token to hold a record number. The mark then points to that specific record. However, a mark might persist beyond a single operation. Storing a record number in a token wouldn't work if records could be inserted or deleted from the database positionally before the target record.

To support marks that persist, your component must establish a mapping between the token and the target. If the target moves, the token must still be able to find the target.
There are two common forms of mapping: table mapping and stamp mapping. Regardless of how you implement your token, your program must manage and store the token.

**Table Mapping**

In table mapping, your component creates and maintains a table that contains pairs of token values and data that identifies what the token refers to. If the position of the data changes (for example, if the user adds or deletes characters), your component updates the positions in the token table so that the mark points to the target at the correct location.

When a token is requested of a table-mapping component, the component creates a new table entry with a new, unused token value. Once a number is used as a token, it cannot be reused; this ensures the integrity of the token.

MiniText implements tokens using table mapping. Table mapping is recommended if the data the program handles is densely stored.

**Stamp Mapping**

In stamp mapping, your component stores the token as a part of its target data. When your component receive a request for a mark, it creates a token and inserts it directly into the data. Your program must have a way to locate tokens within the data and a way to identify each token.

Stamp mapping is a viable way to implement tokens when the overhead of adding a token to a data item is small. For example, the effort of storing a token for individual characters is high; for a text-based application, it's unlikely that you'll want to implement tokens using stamp mapping. However, with a scribble or some other graphical object, the effort of storing a token may be small enough to justify using stamp mapping.

The difficulty in implementing stamp mapping arises when you consider how to handle multiple marks pointing to the same object. There are two possible solutions:

- If your component can afford the storage space, it can store a token for each mark that points to your data.
- If your component only has room for one token at a time, and it receives a request for another mark, it can pass the current token to the new mark. (Thus, there are two marks with the same token.) When one of the marks moves, your component can assign a new token to the mark that moved. With this scheme, you have to decide how to hand out new token values. There are two alternatives:
  - Create a token value for the data item and leave it there forever. Although this is easy to implement, there is the possibility of running out of token values.
- Keep track of the number of marks that refer to a data item. When a mark that targets the data item is repositioned and the reference count is one, move the token value to the next data item and clear the token on the old data item. MiniNote implements this variant of stamp mapping.

How to Support Marks

At the very least, your component must handle these basic messages:

- `msgMarkCreateToken` requests that your component create a new token.
- `msgMarkDeleteToken` requests that your component delete a token.
- `msgMarkGetDataAncestor` requests that your component return the next higher superclass that can traverse the component’s data.

For a component that can be traversed, you must handle the following messages:

- `msgMarkPositionAtEdge` requests that your component reposition a token to one end or the other of its data.
- `msgMarkPositionAtToken` requests that your component reposition a token to the same position as another token in the same component.
- `msgMarkCompareTokens` requests that your component compare the ordering of two tokens.

For components that have a graphical view of data, you must handle:

- `msgMarkShowTarget` requests that your component identify the window that contains the view on the target.
- `msgMarkSelectTarget` requests that your component select the target data.
- `msgMarkPositionAtSelection` requests that your component reposition the token to the current selection.
- `msgMarkPositionAtGesture` requests that your component reposition the token to the location of a gesture.

For components that manage their own embedded documents, you must handle:

- `msgMarkPositionAtChild` requests that your component position the token to an embedded component.
- `msgMarkNextChild` requests that your component position the token to the next embedded component.
- `msgMarkGetChild` requests that your component identify the component at the token.

For components that are not descendants of `clsEmbeddedWin` or `clsApp`, you must handle:

- `msgMarkGetParent` requests that your component identify its parent component.
- `msgMarkGetUUIDs` requests that your component identify its own application and component UUIDs.
Creating and Holding Marks

When a user selects an operation like Search/Replace or Spell, the PenPoint Application Framework creates a driver for that operation. The driver creates a mark, using some of the information it got from the PenPoint Application Framework. The driver communicates with the application and its embedded components (including embedded applications executing in different processes) by sending messages to the mark. The mark figures out what component to send them to.

The clsMark protocol describes how drivers communicate with the mark and how the mark communicates with client components. What it does not describe are the details of the messages that drivers use to communicate with components (because these messages vary depending on the job to be done).

Drivers communicate with components through a mark with three messages: msgMarkDeliver, msgMarkDeliverPos, and msgMarkDeliverNext. Each of these three messages "piggy-backs" a message and associated arguments that are sent from a driver to a component.

The messages that the Search/Replace and Spell drivers use to communicate with text-containing components are defined in SR.H.

A holder follows these steps to create and use a mark:

- Sends msgNewDefaults and msgNew to clsMark. This creates the mark and sets up the component for the mark and, optionally, positions the mark for you.
- Sends the appropriate msgMarkPosition message to the component using the mark. This sets the mark where you want it.
- Sends a message using msgMarkDeliver, msgMarkDeliverPos, and msgMarkDeliverNext to the component, using the mark, if you want to manipulate the mark.
- Sends msgDestroy to the mark when you're finished with it.

With a persistent mark, the target of the mark will never receive a delete token message, even if the holder is deleted. When a mark is copied or filed, it becomes persistent.
When a mark is saved and restored, clsMark uses three low-level messages to create a link handle and link file that are used to keep track of the target data. These messages are described in API.H:

msgAppCreateLink creates a link to another document, passing back the UUID of the document to be linked to. The linked-to document’s UUID is stored in DOC.LNK.

msgAppDeleteLink deletes a link to another document. You must specify a link handle.

msgAppGetLink passes back the linked-to document’s UUID.
Chapter 11 / Printing

Printing in the PenPoint™ operating system is different from printing in most other operating environments because PenPoint can defer output until the user connects the appropriate printer. When the application is requested to prepare its output for the printer, the application uses the same layout messages that it would for the screen, only the image is displayed on a piece of paper by a printer. This approach frees the application developer from having to understand the command set for a particular printer. All the application has to do is lay itself out for printing (if that is different from laying itself out on screen).

When printing in PenPoint, your application does not explicitly print to a device. It opens the document (or a copy of the document) in the background using a window device that is bound to a printer, rather than the screen.

This chapter describes the printing messages that your application receives and how to respond to them.

The Out box mechanism, which is used to defer documents until the printer is ready, is described in Part 10: Remote Interfaces.

This chapter covers these topics:

- The concepts of printing, which include an overview of the flow of control in printing.
- How to participate in the option card protocol to add your own option cards for printing.
- The print protocol and how an application should respond to printing messages.

Concepts

Printing involves several aspects of the PenPoint operating system.

- When the user prints a document, the PenPoint operating system uses the Out box mechanism to queue the document for printing.
- When the user connects the printer, the Out box service section that contains the document creates, activates, and opens a **printing wrapper** for the document.
- When the service section opens the wrapper, the wrapper uses PenPoint Application Framework messages to open the document.
- The wrapper uses printing messages to tell the document when and what to print. The messages allow the document to behave correctly if it contains embedded documents.
Finally, the document uses the UI Toolkit to lay out its pages for the printer.

If you write an application that prints, you should determine if your application is responsible for:

- Adding option cards to the Print Setup dialog sheets (the option card protocol is discussed later in this chapter).
- Handling open messages so that they remove unnecessary decorations from your frame (such as scroll bars, borders, and menus).
- Laying out pages in response to the printing messages.
- Handling embedded documents so that they print correctly.

All other aspects of printing (finding the device driver, queuing, and sending data to the printer) are handled by the PenPoint operating system.

**Printing Embedded Documents**

The PenPoint operating system's embedded document architecture presents certain challenges when printing documents that have embedded documents. This is particularly true when the embedded document is a full-sized document, of which the user only sees a portion through a scrolling window.

The Embedded Printing card of the Print Setup option sheet determines how to print documents embedded within the document being printed (the parent document). When the user prints a document, the Print option specifies whether to print embeddors at all. The Location option specifies either:

- **Print In Place.** That is, print the visible portion of each embeddee in place in the parent document where it is embedded.
- **Print At End.** That is, print each embedded document in full at the end of the parent document.

Thus, there are three possible printing situations for embedded documents.

- Do not print the embedded document (Print is No).
- Print the portion of the embedded document that you see on screen just as it appears (Print is Yes, Location is In Place).
- Print the entire embedded document after printing its embeddor (Print is Yes, Location is At End).

The PenPoint Application Framework provides default implementations for not printing and for printing At End.

If the user selects print at end and the PenPoint Application Framework encounters an embedded document, it closes the embedded document and queues it to the end of the print job. When the embedded document reaches the top of the queue, the PenPoint Application Framework opens the document as a new, top-level document, which can then receive printing messages. The embedded document begins printing on a new page. This expansion can be recursive. A deferred embeddee becomes top-level when it is printed. If the embedded
document has embeddees of its own, they are queued for printing immediately after their parent.

Embedded documents printed In Place have no control of the page. You do not have to do anything special for these documents when printing. They are opened in place and are confined to the clipping window within their parent, just as they are displayed on screen. However, the embedded documents do know that they’re printing, because they receive msgAppOpen. The message arguments for msgAppOpen and their application metrics indicate that they are printing.

Making Pagination Decisions

The printing protocol handles two forms of pagination:

- **flow** for documents that can display their information in an arbitrarily-sized visual space. For example, MiniText documents can flow.
- **nonflow** for documents that must display their information exactly as described by the user. For example, MiniNote documents cannot flow.

The printing protocols will ask your application whether it is a flow or nonflow document by sending msgPrintGetProtocols to your active document.

Paginating Nonflowing Documents

There are two methods for paginating a nonflowing document: tiling or scaling. Tiling is well defined, invariant across documents, and easy to implement. For these reasons, and to avoid duplication of tiling code in many applications, printing in the PenPoint operating system provides tiling by default. You do not have to do anything to paginate a tiling document.

The user interface for printing does not offer the tiling option to the user. An application that wants to use tiling as a pagination method must present a tile option to the user (on a Print Setup option card) or simply use it as its default pagination method.

Scaling is much more difficult to implement and depends greatly on the type of document being printed. For instance, scaling a spreadsheet that uses PenPoint fonts is possible, because ImagePoint is built to scale its elements. However, scaling a bitmap image, while possible, suffers from rounding errors and other problems. You must design and write your own scaling pagination routines.

Designing a PenPoint Application in the PenPoint UI Design Reference provides user interface guidelines on option presentation.

Paginating Flow Documents

Paginating flow documents is relatively simple. When your application receives a next page message, it flows its data into the space available and adds any other elements to the page. Your application must keep track of what it has printed and what it hasn’t printed. When it receives the next next page message, it can resume flowing data where it left off.
Option Sheets for Printing

This section provides a brief summary of the option sheet protocol. *Part 4: UI Toolkit* describes option sheets and the option sheet protocol in detail.

Before printing, the user can change document printing options by tapping on the **Print Setup** button in the Document menu. Typically, the Print Setup option sheet allows the user to change the margins and the headers and footers for the document.

If your application has other printing options that you want the user to be able to change, you can add your own option cards to the Print Setup sheet. For example, if you are writing a drawing program, you might want to present the user with some options for tiling pages when printing a drawing larger than a single sheet.

When the user taps on **Print Setup**, which sends **msgAppPrintSetup** to the application, the following events occur:

1. clsApp handles **msgAppPrintSetup** by sending **msgAppShowOptionSheet** to self, specifying **tagAppPrintSetupOptSheet**.
2. clsApp handles **msgAppShowOptionSheet** by creating an option sheet by sending **msgAppGetOptionSheet** to self.
3. clsApp handles **msgAppGetOptionSheet** by creating the option sheet for the printer setup dialog. Eventually it sends **msgAppAddCards** to self.
4. Your application should handle **msgAppAddCards** by adding its option cards to the option sheet (as described below).
5. After handling **msgAppAddCards**, your application should allow its ancestor to handle the message, so that its ancestors can add their own cards to the option sheet.
6. clsApp adds its own option sheets and returns, which unwinds back to clsApp handling **msgAppShowOptionSheet**.
7. clsApp then sends **msgOptionShowCardAndSheet** to insert the option sheet into your document's frame.

**Handling msgAppAddCards**

**msgAppAddCards** passes a pointer to an **OPTION_TAG** structure that contains:

- **option** The UID of the option sheet.
- **tag** The tag for that option sheet.

When your application handles **msgAppAddCards**, it should first check which option sheet is being built. You can identify the option sheet by checking the **tag** value. When creating the Print Setup option sheet, **tag** contains the value **tagAppPrintSetupOptSheet**.

To add cards to the option sheet, send **msgOptionAddCard** to the option sheet specified in **option**. When your application adds the last of its cards, use
msgOptionAddLastCard. If your application has only one card, it should use msgOptionAddLastCard exclusively.

After adding option cards, your application sends msgAppAddCards to its ancestor so that it can add its own cards.

### Print Protocol Description

When the user taps on Print in the Document menu to print a document, the button sends msgAppPrint to the document. Usually, your application allows its ancestor (clsApp) to handle msgAppPrint. clsApp provides default print behavior and communicates with thePrintManager. thePrintManager responds by performing two actions:

- Displaying a dialog that asks the user to choose a printer.
- Moving a copy of the document to an appropriate service section in the Out box.

The document remains in the Out box until the user connects the appropriate printer. At that time, the manager of that service section creates a wrapper (a special embeddor application that is an instance of clsPrint). When the service section activates and opens the wrapper, the wrapper activates and opens your document. However, the window device used by your document is not the screen; it is a window device created by the wrapper.

### Print Layout Driver (PLD)

When the wrapper activates and opens your document, it also creates an instance of clsPrLayout called a print layout driver (PLD). The printer layout driver essentially guides the pagination process by sending paginate page messages until the document returns stsEndOfData (or an error).
The printing protocol messages listed in Table are defined by clsPrint and are described in PRINT.H.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgPrintStartPage</td>
<td>P_PRINT_PAGE</td>
<td>Advance document to its next logical page.</td>
</tr>
<tr>
<td>msgPrintLayoutPage</td>
<td>P_PRINT_PAGE</td>
<td>Tells a document to lay out its logical page.</td>
</tr>
<tr>
<td>msgPrintGetMetrics</td>
<td>P_PRINT_METRICS</td>
<td>Gets the print metrics.</td>
</tr>
<tr>
<td>msgPrintSetMetrics</td>
<td>P_PRINT_METRICS</td>
<td>Sets the print metrics.</td>
</tr>
<tr>
<td>msgPrintApp</td>
<td>P_PRINT_DATA</td>
<td>Prints a document.</td>
</tr>
<tr>
<td>msgPrintPaperArea</td>
<td>P_PRINT_AREA</td>
<td>Passes back the width and height of the printing area on the paper.</td>
</tr>
<tr>
<td>msgPrintGetProtocols</td>
<td>P_PRINT_PROTOCOLS</td>
<td>Gets the pagination and embeddee protocols.</td>
</tr>
<tr>
<td>msgPrintEmbeddeeAction</td>
<td>P_PRINT_EMBEDDEE_ACTION</td>
<td>Requests document's permission to perform an action on an embeddee.</td>
</tr>
<tr>
<td>msgPrintExamineEmbeddee</td>
<td>P_PRINT_EMBEDDEE_ACTION</td>
<td>Request from document to interpret embedded window's print properties.</td>
</tr>
<tr>
<td>msgPrintSetPrintableArea</td>
<td>PRINTABLE_AREA</td>
<td>Requests margin adjustment for unprintable area.</td>
</tr>
<tr>
<td>msgPrintGetPrintableArea</td>
<td>PRINTABLE_AREA</td>
<td>Requests margin information.</td>
</tr>
</tbody>
</table>

Each time it is ready for the application to layout a page, the PLD sends msgPrintStartPage to the top-level document being printed. This is a signal to the application to advance its private context to the next page of display (depending on its layout type: flow or nonflow). This could mean simply advancing a data pointer or perhaps inserting a new window in the parent of the application's mainWin.

If an application chooses to account for the unprintable area on a page, it sends msgPrintGetPrintableArea and msgPrintSetPrintableArea to the PLD to make the adjustment. The PLD's default tiling does this by default.

After sending msgPrintStartPage, the PLD sends msgPrintGetProtocols to the top-level document. The message arguments passed back tell the PLD how the application wants to paginate its logical page and how embeddees are to be processed.

**Removing Frame Decorations**

When the printer wrapper is active, it activates and opens your document using msgAppMgrActivate. The arguments for the activation message use the wrapper's window device (the printer) as the window device for your document.

Eventually, the PenPoint Application Framework sends msgAppOpen to your document.

clsApp self-sends msgAppSetPrintControls in its handler for msgAppOpen.

clsApp responds by:
1 Turning off all frame decorations when the application is top level (pArgs is true).

2 Turning off all frame decorations except alternate (user-set) borders when the application is embedded (pArgs is false).

The frame decorations include:
- Title, menu, and command lines.
- Tab bar.
- Scroll bars.
- Borders.

If you want non-standard behavior (for example, if you choose to leave some of your frame decorations on for printing), you should handle msgAppSetPrintControls.

When your application receives msgAppOpen, it should check the printing flag in the message's pArgs. If printing is true, your application should remove anything from its frame that you don't want to print (particularly things that are specific to the user interface rather than presentation).

Your method table entry for msgAppOpen must specify call-ancestor-after. When clsApp receives msgAppOpen, it self-sends msgAppSetPrintControls to remove the title, menu bar, tab bar, border, command lines, and scroll bars from the frame. clsApp does not remove decorations from custom frames that inherit from clsFrame, nor can it remove custom decorations that you added to the frame. The wrapper then sends messages to the document, telling it to paginate.

### Pagination

The PLD can specify two forms of pagination: flow and tiling.

If paginationMethod is prPaginationFlow, pagination is left totally up to the application.

If paginationMethod is prPaginationTile, the PLD will paginate the document's main window by tiling it. All the document needs to do in preparation is to make its main display window lay out as full size. Then, when the print protocols indicate that tiling is requested, the PLD performs these steps:

1 Set the shrink-wrap bits on the document's frame, if it has one.
2 Send msgWinLayout to the print window (prframe) and relax its parent's constraints on the document's main window during the layout.
3 Send msgPrintLayoutPage to the document.
4 Calculate and process each tile.

Thus, the document can size its main display window when the document is opened, in response to msgPrintStartPage, during winLayoutSelf, or in response to msgPrintLayoutPage after the print protocols have been set.
If the document's main display window (usually the frame `clientWin`) is sensitive to shrinking (in other words, it won't expand back to full size when its parent relaxes constraints), the document should protect it with a scroll window to make sure the PLD's clipper window never forces it to shrink. Note that the PLD clipper window always forces the document's outermost window to match its own size before the PLD knows it is tiling.

`msgWinLayout` always follows `msgPrintGetProtocols`. `msgPrintLayoutPage` follows `msgWinLayout`. `msgPrintLayoutPage` is similar to `msgWinLayout`, but is intended for two types of clients:

- Clients wanting to be sure they only get one layout message per page.
- Clients having the page layout logic in the document, as opposed to in a view.

In response to `msgPrintLayoutPage`, a typical flow-type application begins calculations to layout its contents. If the application encounters embedded documents while laying its page out, it calls the print layout driver for the embeddees.

**Handling Embeddees**

There are two ways to handle embeddees:

- The PLD can enumerate the embeddees after laying out a page (`prEmbeddeeSearchByPrintJob`).
- The application finds the embeddees on its own (`prEmbeddeeSearchByApp`).

**Using `prEmbeddeeSearchByPrintJob`**

In `prEmbeddeeSearchByPrintJob`, the PLD uses `winEnum` to find all embedded windows in the document's `childAppParentWin` after the application lays out a logical page. Each time the PLD finds an embedded document, it sends `msgPrintEmbeddeeAction` to the document (document's layout UID, specified in `pArgs` to `msgPrintStartPage`) to inform it of the appropriate action. If the response is `stsOK` or `stsNotUnderstood`, the PLD takes the action, if there is one, and then sends `msgWinLayout` again. The current actions that PenPoint implements are `prEmbedActionAsIs` and `prEmbedActionExtract`.

The PLD proposes `prEmbedActionExtract` when the embeddee's print properties are set to invisible or defer to end. When the document approves the action, the PLD extracts the embedded window from the window tree (if it is inserted) and either forgets about it (invisibility) or puts in on a queue to deal with it after the current document is finished printing.
Handling prEmbeddeeSearchByApp

In prEmbeddeeSearchByApp, the document finds the embedded windows on its own (usually in response to msgPrintLayoutPage) and calls the PLD on each one (msgPrintExamineEmbeddee). The PLD obtains the embeddee's print properties and proposes an action by msgPrintEmbeddeeAction, as above. The document can process its embedded windows in response to either msgPrintStartPage or msgPrintLayoutPage. However, in the former case, msgPrintEmbeddeeAction will always be sent to the document, because the document has not yet returned an appLayoutUID from msgPrintStartPage.

Default Printing Behavior

The printing protocol is designed so that even if an application doesn't handle the printing messages, the PLD makes certain assumptions that enable it to print a document (or at least a part of it).

If the PLD sends printing messages to a document but the application does not define methods for the printing messages, the messages return stsNotUnderstood.

msgPrintStartPage Not Understood

If msgPrintStartPage returns stsNotUnderstood to the PLD, it indicates that the document doesn't know how to paginate or only has one logical page to print (such as a tiled drawing program or the Tic-Tac-Toe sample program).

When the PLD receives stsNotUnderstood in response to msgPrintStartPage, it tells the document to lay out and print the current page. If the document is not tiled, it prints the upper left portion of the document (the portion of the document's display window that is not clipped by the PLD clipper).

msgPrintGetProtocols Not Understood

If msgPrintGetProtocols returns stsNotUnderstood to the PLD, the PLD assumes that:

- paginationMethod is prPaginationFlow.
- embeddeeSearch is prEmbeddeeSearchByPrintJob.

msgPrintEmbeddeeAction Not Understood

If msgPrintEmbeddeeAction returns stsNotUnderstood to the PLD, the PLD acts as if the document returned stsOK. The assumption here is that if the document didn't know enough not to return stsRequestDenied, the PLD should be able to enter an embedded document anyway.

msgPrintLayoutPage Not Understood

If msgPrintLayoutPage returns stsNotUnderstood to the PLD, the PLD acts as if the document returned stsOK.
Chapter 12 / The Application Manager Class

The application manager class (clsAppMgr) defines messages to install application classes, to keep track of installed application classes, and to create, activate, and destroy instances of an application.

Your application uses these messages to create a new application class and to get information about installed classes. The clsAppMgr messages also include two observer messages that inform when applications have been installed or deinstalled.

This chapter covers the following topics:
- Installing a new application class.
- Creating a new document.
- Activating an application instance.
- Moving or copying an application instance.
- Deleting application instances.

Application Manager Metrics

Each application class has metrics that pertain to the class in general (not to each instance):
- The application name.
- The name of the company that wrote the application.
- How the application name appears in the Create menu.
- Whether the application allows embedding.
- Whether instances run in **hot mode** by default.
- Whether the user can create instances of the application.

This information allows the Notebook to let the user know about your application and provide your application with the right environment.

Because clsAppMgr stores this general information once for each application class, clsApp doesn’t have to maintain it for each instance. Nor does clsApp have to respond to messages requesting this information.

The clsAppMgr messages are defined in APPMGR.H. Table 12-1 lists the messages defined by clsAppMgr.
**Installing a New Class**

When your application’s main routine detects that processCount is 0, it should call its application class initialization routine. This routine declares an APPMGR_NEW structure, initializes it (by sending msgNewDefaults to clsAppMgr), modifies some of the arguments, and then sends msgNew to clsAppMgr. In addition to an OBJECT_NEW_ONLY structure, APPMGR_NEW contains two structures: a CLASS_NEW_ONLY structure and an APP_MGR_METRICS structure.

The CLASS_NEW_ONLY structure contains:

- **pMsg** the address of the application’s msgProc.
- **ancestor** the application’s ancestor class.
- **size** a 20-bit value that specifies the size of the application class’s instance data, in bytes.
- **newArgsSize** a 20-bit value that specifies the size of the application class’s NEW structure.

The APP_MGR_METRICS structure contains:

---

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_APP_MGR_NEW</td>
<td>Install a new class.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_APP_MGR_NEW</td>
<td>Initialize the defaults for a new class.</td>
</tr>
<tr>
<td>msgAppMgrGetMetrics</td>
<td>P_APP_MGR_METRICS</td>
<td>Get the class metrics.</td>
</tr>
<tr>
<td>msgAppMgrCreate</td>
<td>P_APP_MGR_CREATE</td>
<td>Create an instance of an application.</td>
</tr>
<tr>
<td>msgAppMgrActivate</td>
<td>P_APP_MGR_ACTIVATE</td>
<td>Activate an instance of an application.</td>
</tr>
<tr>
<td>msgAppMgrMove</td>
<td>P_APP_MGR_MOVE_COPY</td>
<td>Move a doc to a new location.</td>
</tr>
<tr>
<td>msgAppMgrCopy</td>
<td>P_APP_MGR_MOVE_COPY</td>
<td>Copy a doc to a new location.</td>
</tr>
<tr>
<td>msgAppMgrFSCopy</td>
<td>P_APP_MGR_FS_MOVE_COPY</td>
<td>Low-level copy message. Used by msgAppMgrCopy.</td>
</tr>
<tr>
<td>msgAppMgrDelete</td>
<td>P_APP_MGR_DELETE</td>
<td>Delete a doc.</td>
</tr>
<tr>
<td>msgAppMgrRename</td>
<td>P_APP_MGR_RENAME</td>
<td>Rename a doc.</td>
</tr>
<tr>
<td>msgAppMgrShutdown</td>
<td>P_FS_LOCATOR</td>
<td>Unconditionally shutdown an app instance and all children.</td>
</tr>
<tr>
<td>msgAppMgrGetRoot</td>
<td>P_APP_MGR_GET_ROOT</td>
<td>Get the root application (clsRootContainerApp) of a tree of applications.</td>
</tr>
<tr>
<td>msgAppMgrSetIconBitmap</td>
<td>OBJECT</td>
<td>Set the icon bitmap.</td>
</tr>
<tr>
<td>msgAppMgrSetSmallIconBitmap</td>
<td>OBJECT</td>
<td>Set the small icon bitmap.</td>
</tr>
<tr>
<td>msgAppMgrRevert</td>
<td>P_FS_LOCATOR</td>
<td>Revert to filed copy.</td>
</tr>
<tr>
<td>msgAppMgrRenumber</td>
<td>P_FS_LOCATOR</td>
<td>Renumber an application heirarchy.</td>
</tr>
<tr>
<td>msgAppMgrDumpSubtree</td>
<td>P_FS_LOCATOR</td>
<td>Display the attributes of a subtree of apps.</td>
</tr>
<tr>
<td>msgAppMgrGetResList</td>
<td>P_APP_MGR_GET_RES_LIST</td>
<td>Creates a resource list, given an application UUID.</td>
</tr>
</tbody>
</table>
the handle on the application directory that contains the installed application.

appMonitor the UID of the application monitor object. The application monitor is responsible for configuring an application the first time it is loaded. If the application is already loaded, the application monitor loads the application's resource file and other portions of the application.

resFile the handle on the resource file used by the application.

iconBitmap the resource ID of the icon bitmap for the application.

smallIconBitmap the resource ID of the small icon bitmap for the application.

appWinClass the UID of the application's window class.

defaultRect a RECT32 structure that specifies the application's default rectangle in points.

name a string that specifies the name of the application.

version a string that specifies the version of the application.

company a string that specifies your company's name.

defaultDocName a string that specifies the default name for a document created by the application. If you specify pNull, the PenPoint Application Framework will look for the default document name in tagAppMgrDefaultDocName in the resource file APP.RES.

copyright a pointer to a string that contains the copyright notice for the application.

programHandle an OS_PROG_HANDLE structure that specifies the location to receive the handle on the application's executable file in the loader database.

flags an APP_MGR_FLAGS structure that specifies other information about the application. Table 9-2 describes the application manager flags.

---

Table 9-2

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>stationery</td>
<td>Put in Stationery notebook.</td>
</tr>
<tr>
<td>accessory</td>
<td>Put in Accessories menu.</td>
</tr>
<tr>
<td>hotMode</td>
<td>Create instances in hot mode.</td>
</tr>
<tr>
<td>allowEmbedding</td>
<td>Allow child embedded applications.</td>
</tr>
<tr>
<td>confirmDelete</td>
<td>Ask the user for confirmation before deleting.</td>
</tr>
<tr>
<td>deinstallable</td>
<td>The user can deinstall the application.</td>
</tr>
<tr>
<td>systemApp</td>
<td>System application.</td>
</tr>
<tr>
<td>lowMemoryApp</td>
<td>Allow activation of this application under low memory conditions.</td>
</tr>
<tr>
<td>fullEnvironment</td>
<td>Provide instance 0 with a full environment.</td>
</tr>
</tbody>
</table>
Creating a New Document

To create a new document, send `msgAppMgrCreate` to the application class. `msgAppMgrCreate` creates a new directory and gives the directory an attribute that specifies the application class. The actual application instance does not exist until you send `msgAppMgrActivate` to `clsAppMgr`.

`msgAppMgrCreate` takes a pointer to an `APP_MGR_CREATE` structure that contains:

- `locator` a locator for the directory of the document's parent. Usually, this is the Notebook Table of Contents or a section; for embedded documents, this can be any document.
- `pName` a pointer to a string that contains the name of the new application instance.
- `sequence` a `U32` value that specifies the sequence number of the new application instance in its parent application.
- `renumber` a Boolean value that indicates whether the global sequence numbers should be updated when creating this application instance. If the value is true, `clsAppMgr` updates the global sequence numbers.

Activating an Application Instance

To activate a document and all its embedded documents, send `msgAppMgrActivate` to the application class. The message takes a pointer to an `APP_MGR_ACTIVATE` structure that specifies:

- `winDev` the UID of the window device for this application.
- `locator` a locator for the directory that contains the application to activate.
- `parent` the UID of the parent application.
- `uid` a UID value that will receive the UID of the new application instance.

`msgAppMgrActivate` creates a new process and sends it the information it needs to create a `msgNew`. In `AppMain`, your application creates a `msgNew` and sends it to `clsClass`. This creates an application object in the process. From this point on, the application instance is on its own, and responds to `clsApp` messages.

If the application monitor was unable to activate a document because there wasn't enough memory, it returns `stsAppMgrLowMemNoActivate`.

Moving or Copying an Application Instance

To move or copy a document to a new location, send `msgAppMgrMove` or `msgAppMgrCopy` to the application class for that document. Both messages take a pointer to an `APP_MGR_MOVE_COPY` structure, which contains:

- `locator` a file system locator for the source document.
- `source` the UID of the source object.
- `dest` the UID of the destination object that will contain the moved or copied document.
Getting Metrics for a Class

To get the metrics for a particular class, send `msgAppMgrGetMetrics` to the application class. The message takes a pointer to an APP_MGR_METRICS structure, which is the same structure used in `msgNew`.

Deleting Application Instances

You use `msgAppMgrDelete` to destroy a document. That is, you use it to free the application object, destroy the process, delete the resource file, and delete the directory. If the application instance doesn't go away when you send it `msgAppDelete`, you can send `msgAppMgrDelete` to its class. `msgAppMgrDelete` does the same thing as `msgAppDelete`, but it does not rely on the application instance being able to respond to messages.

You send `msgAppMgrDelete` to the application's class, not to an instance of the application. The message takes a pointer to an APP_MGR_DELETE structure that contains:

- **locator** a locator that specifies the document's directory.
- **renumber** a Boolean value that indicates whether the global sequence numbers should be updated. If the value is `true`, `clsAppMgr` updates the global sequence numbers.

Getting Metrics for a Class

To get the metrics for a particular class, send `msgAppMgrGetMetrics` to the application class. The message takes a pointer to an APP_MGR_METRICS structure, which is the same structure used in `msgNew`. 

| xy     | an XY32 structure that specifies the x-y coordinates of the location within the destination object where the document will be moved or copied. |
| name   | a string buffer that contains the name of the document being moved or copied. If the destination already contains a document with the same name, the application manager creates a new name for the document and stores the new name in this buffer. |
| renumber | a Boolean value that indicates whether the global sequence numbers should be updated. If the value is `true`, `clsAppMgr` updates the global sequence numbers. |
| style  | An APP_MGR_MOVE_COPY_STYLE enumerated value that specifies the style for the move or copy. The possible values are: |
| showConfirm | show the confirmation sheet to the user before moving or copying. |
| showProgress | show the progress of the move or copy to the user. |
| appWin | the UID of the application window that was moved or copied. |
Observer Messages

When applications are installed or deinstalled, these messages are sent to observers of clsApp:

- `msgAppInstalled` indicates that an application has been installed.
- `msgAppDeInstalled` indicates that an application has been deinstalled.

Both messages have a single argument, the UID of the class that was installed or deinstalled.
Chapter 13 / The Application Monitor Class

The application monitor drives application installation, performs application global functions, and helps deinstall an application. An application monitor instance runs in process 0 of an application, along with an application manager instance.

The application monitor functions are defined by clsAppMonitor. Applications can subclass clsAppMonitor to provide additional functions, such as performing specialized installations, providing file import, setting and saving global application configurations, and providing file converters.

Application Monitor Concepts

The application monitor assists in application installation and deinstallation.

For the application monitor to perform its work properly, your file organization must conform to the organization described in the File Organization chapter of Part 12: Installation API.

Application Monitor in Installation

When the user installs your application:

1. The installer calls OSProcessCreate, specifying your application executable file. The entry point for that file is main. Because the user is installing your application, this is the first process running your application executable; thus, it receives the process number 0.

Your main routine examines the process number and, if it is 0, calls some class initialization routines and then calls AppMonitorMain. AppMonitorMain is a PenPoint-provided routine that creates an instance of clsAppMonitor, running in the context of your process 0. This instance of clsAppMonitor is your application's application monitor.

3. AppMonitorMain sends msgApplInit to your application monitor.

4. If this is the first time the application has ever been installed, the application monitor puts up the installation option sheet so the user can specify the configuration. The results from the option sheet are stored in the application's resource file (APP.RES). The PenPoint™ operating system defines a default property sheet; your application can provide its own property sheet to override the defaults.

5. The application monitor uses the application's resource file to drive the rest of the installation.
Using the application resource file, the application monitor loads all other portions of the application, such as Stationery, Help, Accessories, and any miscellaneous resources.

To override the default behavior provided by clsAppMonitor, you may have to subclass it. The considerations for subclassing clsAppMonitor are described in this chapter in “Subclassing clsAppMonitor.”

**Other Application Monitor Functions**

After installing an application, the application monitor does not go away. When requested, the application monitor can deinstall the application.

**Stationery, Accessories, and Help**

The application monitor is also responsible for loading stationery, accessories, and help templates from the application’s distribution disk to their appropriate notebooks.

The application monitor creates a section in the Stationery notebook for your application and copies the stationery templates into that section. If you do not provide stationery, the application monitor creates a blank document for your application and stores it in the Stationery notebook.

If the application’s distribution disk contains accessory templates, the application monitor copies the accessories templates into Accessories. If there are no accessories, the application monitor skips this stage.

The application monitor creates a section in the Help notebook for your application and copies the help templates into that section.

**clsAppMonitor Messages**

The clsAppMonitor messages are defined in APPMON.H. Table 13-1 lists the application monitor’s messages.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAMGetMetrics</td>
<td>P_AM_METRICS</td>
<td>Gets the application monitor’s metrics.</td>
</tr>
<tr>
<td>msgAMLoadInitDll</td>
<td>OBJECT</td>
<td>Loads, runs, and unloads an optional initialization .dll.</td>
</tr>
<tr>
<td>msgAMLoadMisc</td>
<td>void</td>
<td>Loads the application’s miscellaneous files.</td>
</tr>
<tr>
<td>msgAMLoadStationery</td>
<td>void</td>
<td>Loads stationery and accessories templates from application’s distribution disk.</td>
</tr>
<tr>
<td>msgAMRemoveStationery</td>
<td>void</td>
<td>Removes all the stationery and accessory templates for this application.</td>
</tr>
<tr>
<td>msgAMLoadHelp</td>
<td>void</td>
<td>Load help into the Help notebook.</td>
</tr>
<tr>
<td>msgAMRemoveHelp</td>
<td>void</td>
<td>Removes all Help notebook items for this application.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance Messages</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAMGetMetrics</td>
<td>Gets the application monitor’s metrics.</td>
</tr>
<tr>
<td>msgAMLoadInitDll</td>
<td>Loads, runs, and unloads an optional initialization .dll.</td>
</tr>
<tr>
<td>msgAMLoadMisc</td>
<td>Loads the application’s miscellaneous files.</td>
</tr>
<tr>
<td>msgAMLoadStationery</td>
<td>Loads stationery and accessories templates from application’s distribution disk.</td>
</tr>
<tr>
<td>msgAMRemoveStationery</td>
<td>Removes all the stationery and accessory templates for this application.</td>
</tr>
<tr>
<td>msgAMLoadHelp</td>
<td>Load help into the Help notebook.</td>
</tr>
<tr>
<td>msgAMRemoveHelp</td>
<td>Removes all Help notebook items for this application.</td>
</tr>
</tbody>
</table>
Using clsAppMonitor Messages

You usually use the clsAppMonitor messages to get information about an application class (msgAMGetMetrics). From the application monitor's metrics, you can find the UID of the application class.

Most of the other messages are used by the installer application to load and unload an application's stationery, help, and related components.

Getting App Monitor Metrics

To get the application monitor's metrics, send msgAMGetMetrics to an application monitor. The message takes a pointer to an AM_METRICS structure that is used to return the metrics. The structure contains a single element, the UID of the application class (appClass).

Loading and Unloading Stationery

To load stationery templates from the application's distribution disk to the Stationery notebook, send msgAMLoadStationery to the application monitor. The message takes no arguments.

clsAppMonitor looks for stationery in a directory named statnry (in the application directory). The stationery can be either a complete saved instance of a document, or simply a directory. A piece of stationery is loaded into the Stationery notebook only if its file has the attribute smAttrStationeryMenu.
To remove all stationery that belongs to an application from the Stationery notebook, send \texttt{msgAMRemoveStationery} to the application monitor. The message takes no arguments.

This message will remove:

- Stationery defined by the application that was loaded when the application was installed.
- Stationery that the user defined and added to the Stationery notebook after the application was installed.

\textbf{Loading and Unloading Help}

To load help applications from the application's distribution disk to the Help notebook, send \texttt{msgAMLoadHelp} to the application monitor. The message takes no arguments.

\texttt{clsAppMonitor} looks for subdirectories in a directory named \texttt{help} (in the application's directory). Each help subdirectory can contain a help application and its files or a single file that contains ASCII text or Microsoft RTF text.

If the application monitor finds a help directory in your application, it creates a section for your application in the Help notebook and loads the contents of each subdirectory as documents in that section. The name of each subdirectory in the \texttt{help} directory becomes the name of a help topic in the Help notebook.

When the application monitor encounters a help directory that contains a single file, the application monitor creates an instance of the text editor and imports the file as a text editor document.

To remove help for an application from the Help notebook, send \texttt{msgAMRemoveHelp} to the application monitor. The message takes no arguments.

\texttt{clsAppMonitor} locates the help files that belong to the application and removes them from the Help notebook.

\textbf{Loading Other Information}

Some applications require other information, such as additional resource files or data files. When distributing an application, these files are stored in a directory named \texttt{misc} (in the application directory).

To load these files into an installed application directory, you send \texttt{msgAMLoadMisc} to the application monitor. The message takes no arguments.

If the \texttt{misc} directory exists, \texttt{clsAppMonitor} copies the directory into the installed application directory.

When you remove an application, it deletes everything in the application directory, including anything loaded with \texttt{msgAMLoadMisc}.
Subclassing clsAppMonitor

When you subclass clsAppMonitor, your subclass will receive the messages described in this section. Although you might want to allow clsAppMonitor to handle these messages, receiving these messages gives you the opportunity to handle them yourself.

Because these messages and their arguments are already described above, this section deals with special considerations and return values you that you may have to use when you receive these messages.

Superclass Messages

The first set of messages you will receive are clsApp messages that are intercepted and altered by clsAppMonitor.

Handling msgApplnit

If you subclass clsAppMonitor, you will receive msgApplnit at installation time.

Your method table for msgApplnit should send the message to ancestor (clsAppMonitor) first. When you receive msgApplnit, you can perform any first-time processing that you might need.

Handling msgAppTerminate

When the installer deinstalls your application class, it sends msgAppTerminate to your application monitor.

Your method table for msgAppTerminate should call ancestor after you handle the message. When you handle msgAppTerminate, you should clean up and save whatever objects you need to save. When your ancestor (clsAppMonitor) receives the message, it destroys your task, so you cannot expect to do any further processing after the call ancestor.

Handling clsAppMonitor Messages

You should not have to handle most clsAppMonitor messages, your ancestor will handle them for you. Those messages that must be handled by your ancestor are labeled in the header file appmon.h.

Handling msgAMTerminateOK

Your method table entry for msgAMTerminateOK should call ancestor before handling the message. clsAppMonitor then sends msgAppMgrShutdown to your application class for each of its active documents (by default, clsAppMonitor unconditionally terminates all of your application's documents).
Chapter 14 / The Application Class

While clsAppMgr provides methods to create and activate a document (up to the point where the process contains an application object), clsApp provides methods to control the document once it has an application object. Most of these messages are sent by clsApp or the PenPoint Application Framework; applications should not need to send them.

Messages defined by clsApp perform tasks such as initializing, saving, or re-creating the application instance data, opening and controlling the application instance’s windows, starting message dispatching, setting hot mode, floating, zooming, and deleting application instances. All these topics are covered in this chapter.

Application instances must be clsApp objects. When you define your application class (in the application class initialization routine called by main()), you must specify that its ancestor is clsApp.

clsApp Messages

Table 14-1 lists the clsApp messages.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_APP_NEW</td>
<td>Creates and initializes a new document.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_APP_NEW</td>
<td>Initializes the APP_NEW structure to default values.</td>
</tr>
<tr>
<td>msgAppDispatch</td>
<td>nothing</td>
<td>Starts message dispatching.</td>
</tr>
<tr>
<td>msgAppInit</td>
<td>DIR_HANDLE</td>
<td>Creates the default document data file and main window.</td>
</tr>
<tr>
<td>msgAppActivate</td>
<td>nothing</td>
<td>Activates a document and its children.</td>
</tr>
<tr>
<td>msgAppOpen</td>
<td>P_APP_OPEN</td>
<td>Opens a document’s main window.</td>
</tr>
<tr>
<td>msgAppOpenTo</td>
<td>U32</td>
<td>Opens a document to a specific state.</td>
</tr>
<tr>
<td>msgAppClose</td>
<td>nothing</td>
<td>Closes a document’s main window.</td>
</tr>
<tr>
<td>msgAppCloseTo</td>
<td>U32</td>
<td>Closes a document to a specific state.</td>
</tr>
<tr>
<td>msgAppTerminateOK</td>
<td>nothing</td>
<td>Checks if a document is willing to terminate.</td>
</tr>
<tr>
<td>msgAppTerminate</td>
<td>BOOLEAN</td>
<td>Terminates a document.</td>
</tr>
<tr>
<td>msgAppSave</td>
<td>nothing</td>
<td>Saves a document to its working directory.</td>
</tr>
<tr>
<td>msgAppSaveTo</td>
<td>DIR_HANDLE</td>
<td>Saves a document to a specified directory.</td>
</tr>
<tr>
<td>msgAppRestore</td>
<td>nothing</td>
<td>Restores an document from its saved instance data.</td>
</tr>
</tbody>
</table>
### Document Hierarchy Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAppRestoreFrom</td>
<td>DIR_HANDLE</td>
<td>Restores a document from a specified directory.</td>
</tr>
<tr>
<td>msgAppDelete</td>
<td>nothing</td>
<td>Deletes a document from the system.</td>
</tr>
<tr>
<td>msgFree</td>
<td>nothing</td>
<td>Destroys a document.</td>
</tr>
<tr>
<td>msgFreeOK</td>
<td>nothing</td>
<td>Checks to see if a document and its children are</td>
</tr>
<tr>
<td></td>
<td></td>
<td>willing to be freed.</td>
</tr>
<tr>
<td>msgAppGetRoot</td>
<td>P_APP</td>
<td>Gets the document's root document (which is typically the Notebook).</td>
</tr>
<tr>
<td>msgAppSetParent</td>
<td>APP</td>
<td>Sets the parent document.</td>
</tr>
<tr>
<td>msgAppActivateChildren</td>
<td>nothing</td>
<td>Activates the document's embedded documents.</td>
</tr>
<tr>
<td>msgAppActivateChild</td>
<td>P_APP_ACTIVATE_CHILD</td>
<td>Instantiates and activates an embedded document.</td>
</tr>
<tr>
<td>msgAppSaveChild</td>
<td>APP</td>
<td>Saves the specified child document.</td>
</tr>
<tr>
<td>msgAppSaveChildren</td>
<td>nothing</td>
<td>Saves a document's children.</td>
</tr>
<tr>
<td>msgAppOpenChildren</td>
<td>BOOLEAN</td>
<td>Opens all of the documents on the metrics.children list.</td>
</tr>
<tr>
<td>msgAppOpenChild</td>
<td>APP_OPEN_CHILD</td>
<td>Opens a specific child.</td>
</tr>
<tr>
<td>msgAppCloseChildren</td>
<td>nothing</td>
<td>Closes a document's children.</td>
</tr>
<tr>
<td>msgAppCloseChild</td>
<td>APP</td>
<td>Closes a specific child.</td>
</tr>
<tr>
<td>msgAppGetEmbeddor</td>
<td>P_APP</td>
<td>Gets a document's direct parent in the filesystem hierarchy.</td>
</tr>
<tr>
<td>msgAppCreateLink</td>
<td>P_APP_LINK</td>
<td>Creates a link to another document.</td>
</tr>
<tr>
<td>msgAppDeleteLink</td>
<td>P_APP_LINK</td>
<td>Deletes the specified link handle.</td>
</tr>
<tr>
<td>msgAppGetLink</td>
<td>P_APP_LINK</td>
<td>Gets the document's UUID for the specified link handle.</td>
</tr>
<tr>
<td>msgAppActivateCorkMarginChildren</td>
<td>nothing</td>
<td>Activates the embedded documents in the cork margin.</td>
</tr>
</tbody>
</table>

### Document Attributes

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAppSetHotMode</td>
<td>BOOLEAN</td>
<td>Turns hot mode on or off.</td>
</tr>
<tr>
<td>msgAppSetMainWin</td>
<td>WIN</td>
<td>Sets the document's main window.</td>
</tr>
<tr>
<td>msgAppGetMetrics</td>
<td>P_APP_METRICS</td>
<td>Gets a copy of the application metrics.</td>
</tr>
<tr>
<td>msgAppSetName</td>
<td>P_STRING</td>
<td>Sets a document's displayed name (in its main window title).</td>
</tr>
<tr>
<td>msgAppGetName</td>
<td>P_STRING</td>
<td>Gets a document's name.</td>
</tr>
<tr>
<td>msgAppRename</td>
<td>P_STRING</td>
<td>Renames a document.</td>
</tr>
<tr>
<td>msgAppSetReadOnly</td>
<td>BOOLEAN</td>
<td>Sets the read only flag.</td>
</tr>
<tr>
<td>msgAppSetDeletable</td>
<td>BOOLEAN</td>
<td>Sets the deletable flag.</td>
</tr>
<tr>
<td>msgAppSetMovable</td>
<td>BOOLEAN</td>
<td>Sets the movable flag.</td>
</tr>
<tr>
<td>msgAppSetCopyable</td>
<td>BOOLEAN</td>
<td>Sets the copyable flag.</td>
</tr>
<tr>
<td>msgAppOwnsSelection</td>
<td>P_APP_OWNS_SELECTION</td>
<td>Tests if any object in a document owns the selection.</td>
</tr>
<tr>
<td>msgAppIsPageLevel</td>
<td>nothing</td>
<td>Asks a document if it shows up as a page in the Notebook (as opposed to being embedded).</td>
</tr>
<tr>
<td>msgAppSetSaveOnTerminate</td>
<td>BOOLEAN</td>
<td>Tells the document to save itself before terminating.</td>
</tr>
</tbody>
</table>

(continued)
### clsApp Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAppSetTitleLine</td>
<td>U32</td>
<td>Turns the document's title line on or off.</td>
</tr>
<tr>
<td>msgAppSetMenuLine</td>
<td>U32</td>
<td>Turns the document's menu bar on or off.</td>
</tr>
<tr>
<td>msgAppSetScrollBars</td>
<td>U32</td>
<td>Turns the document's scroll bars on or off.</td>
</tr>
<tr>
<td>msgAppSetBorderStyle</td>
<td>U32</td>
<td>Sets the border style.</td>
</tr>
<tr>
<td>msgAppSetCorkMargin</td>
<td>U32</td>
<td>Turns the document's cork margin on or off.</td>
</tr>
<tr>
<td>msgAppProvideMainWin</td>
<td>P_OBJECT</td>
<td>Asks a document to provide its main window.</td>
</tr>
<tr>
<td>msgAppCreateMenuBar</td>
<td>P_OBJECT</td>
<td>Creates the standard application menu bar.</td>
</tr>
<tr>
<td>msgAppCreateClientWin</td>
<td>P_OBJECT</td>
<td>Creates the document's client window.</td>
</tr>
<tr>
<td>msgAppGetEmbeddedWin</td>
<td>P_APP_GET_EMBEDDED_WIN</td>
<td>Finds the specified clsEmbeddedWin object within a document.</td>
</tr>
<tr>
<td>msgAppGetAppWin</td>
<td>P_APP_GET_APP_WIN</td>
<td>Finds a clsAppWin object within a document.</td>
</tr>
<tr>
<td>msgAppAddFloatingWin</td>
<td>WIN</td>
<td>Adds a window to the document's list of floating windows.</td>
</tr>
<tr>
<td>msgAppRemoveFloatingWin</td>
<td>WIN</td>
<td>Removes a window from the document's list of floating windows.</td>
</tr>
<tr>
<td>msgAppFindFloatingWin</td>
<td>P_APP_FIND_FLOATING_WIN</td>
<td>Finds the floating window that matches the given tag.</td>
</tr>
<tr>
<td>msgAppSetChildAppParentWin</td>
<td>WIN</td>
<td>Sets the window that is used as the parent window for child documents.</td>
</tr>
<tr>
<td>msgAppHide</td>
<td>nothing</td>
<td>Hides an open document.</td>
</tr>
<tr>
<td>msgAppSetFloatingRect</td>
<td>P_RECT32</td>
<td>Sets a document's floating size and position.</td>
</tr>
<tr>
<td>msgAppSetOpenRect</td>
<td>P_RECT32</td>
<td>Sets a document's open size and position.</td>
</tr>
</tbody>
</table>

### Document Window Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAppRevert</td>
<td>BOOLEAN</td>
<td>Reverts to the filed copy of the document.</td>
</tr>
<tr>
<td>msgAppSend</td>
<td>OBJECT</td>
<td>Sends a document.</td>
</tr>
<tr>
<td>msgAppPrint</td>
<td>OBJECT</td>
<td>Prints a document.</td>
</tr>
<tr>
<td>msgAppPrintSetup</td>
<td>nothing</td>
<td>Displays the print setup option sheet.</td>
</tr>
<tr>
<td>msgAppImport</td>
<td>nothing</td>
<td>Obsolete message. Not implemented.</td>
</tr>
<tr>
<td>msgAppExport</td>
<td>OBJECT</td>
<td>Prepares to export a document as a file.</td>
</tr>
<tr>
<td>msgAppAbout</td>
<td>nothing</td>
<td>Displays the document's &quot;About&quot; option sheet.</td>
</tr>
<tr>
<td>msgAppHelp</td>
<td>nothing</td>
<td>Shows help for the application. Not implemented reserved.</td>
</tr>
<tr>
<td>msgAppUndo</td>
<td>nothing</td>
<td>Undoes the previous operation.</td>
</tr>
<tr>
<td>msgAppMoveSel</td>
<td>nothing</td>
<td>Prepares to move the document's selection.</td>
</tr>
<tr>
<td>msgAppCopySel</td>
<td>nothing</td>
<td>Prepares to copy the document's selection.</td>
</tr>
<tr>
<td>msgAppDeleteSel</td>
<td>nothing</td>
<td>Deletes the document's selection.</td>
</tr>
<tr>
<td>msgAppSelectAll</td>
<td>nothing</td>
<td>Selects all of the objects in the document.</td>
</tr>
<tr>
<td>msgAppSearch</td>
<td>OBJECT</td>
<td>Searches a document for a string.</td>
</tr>
<tr>
<td>msgAppSpell</td>
<td>OBJECT</td>
<td>Prepares to check the document's spelling.</td>
</tr>
<tr>
<td>msgAppInvokeManager</td>
<td>OBJECT</td>
<td>Routes a message to a manager.</td>
</tr>
</tbody>
</table>

### Standard Application Menu Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAppSetMenuLine</td>
<td>U32</td>
<td>Turns the document's menu bar on or off.</td>
</tr>
<tr>
<td>msgAppSetScrollBars</td>
<td>U32</td>
<td>Turns the document's scroll bars on or off.</td>
</tr>
<tr>
<td>msgAppSet_BORDER_STYLE</td>
<td>U32</td>
<td>Sets the border style.</td>
</tr>
<tr>
<td>msgAppSetCORK_MARGIN</td>
<td>U32</td>
<td>Turns the document's cork margin on or off.</td>
</tr>
<tr>
<td>msgAppProvide_MAIN_WIN</td>
<td>P_OBJECT</td>
<td>Asks a document to provide its main window.</td>
</tr>
<tr>
<td>msgAppCreate_MENU_BAR</td>
<td>P_OBJECT</td>
<td>Creates the standard application menu bar.</td>
</tr>
<tr>
<td>msgAppCreate_CLIENT_WIN</td>
<td>P_OBJECT</td>
<td>Creates the document's client window.</td>
</tr>
<tr>
<td>msgAppGet_EMBEDDED_WIN</td>
<td>P_APP_GET_EMBEDDED_WIN</td>
<td>Finds the specified clsEmbeddedWin object within a document.</td>
</tr>
<tr>
<td>msgAppGet_APP_WIN</td>
<td>P_APP_GET_APP_WIN</td>
<td>Finds a clsAppWin object within a document.</td>
</tr>
<tr>
<td>msgAppAdd_FLOATING_WIN</td>
<td>WIN</td>
<td>Adds a window to the document's list of floating windows.</td>
</tr>
<tr>
<td>msgAppRemove_FLOATING_WIN</td>
<td>WIN</td>
<td>Removes a window from the document's list of floating windows.</td>
</tr>
<tr>
<td>msgAppFind_FLOATING_WIN</td>
<td>P_APP_FIND_FLOATING_WIN</td>
<td>Finds the floating window that matches the given tag.</td>
</tr>
<tr>
<td>msgAppSet_CHILD_APP_PARENT_WIN</td>
<td>WIN</td>
<td>Sets the window that is used as the parent window for child documents.</td>
</tr>
<tr>
<td>msgApp_HIDE</td>
<td>nothing</td>
<td>Hides an open document.</td>
</tr>
<tr>
<td>msgAppSet_FLOATING_RECT</td>
<td>P_RECT32</td>
<td>Sets a document's floating size and position.</td>
</tr>
<tr>
<td>msgAppSet_OPEN_RECT</td>
<td>P_RECT32</td>
<td>Sets a document's open size and position.</td>
</tr>
</tbody>
</table>

---

Table 14-1 (continued)
### Table 14-1 (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAppExecute</td>
<td>P_APP_EXECUTE</td>
<td>Sent to the manager to execute the manager's behavior on a document.</td>
</tr>
<tr>
<td>msgAppExecuteGesture</td>
<td>P_GWIN_GESTURE</td>
<td>Invokes the default gesture behavior for a document's title line.</td>
</tr>
<tr>
<td>msgAppAddCards</td>
<td>P_OPTION_TAG</td>
<td>Adds cards to the specified option sheet.</td>
</tr>
<tr>
<td>msgAppShowOptionSheet</td>
<td>P_APP_SHOW_OPTION_SHEET</td>
<td>Shows or hides an option sheet.</td>
</tr>
<tr>
<td>msgAppGetOptionSheet</td>
<td>P_APP_GET_OPTION_SHEET</td>
<td>Gets the requested option sheet.</td>
</tr>
<tr>
<td>msgAppGetDocOptionSheetClient</td>
<td>P_OBJECT</td>
<td>Gets the client for the document's option sheets.</td>
</tr>
</tbody>
</table>

#### Printing Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAppApplyEmbeddeeProps</td>
<td>OBJECT</td>
<td>Applies Embedded Printing option card values to first level embeddees.</td>
</tr>
<tr>
<td>msgAppGetBorderMetrics</td>
<td>P_APP_BORDER_METRICS</td>
<td>Gets the document's border metrics.</td>
</tr>
<tr>
<td>msgAppSetControls</td>
<td>U32</td>
<td>Turns controls on or off.</td>
</tr>
<tr>
<td>msgAppSetPrintControls</td>
<td>BOOLEAN</td>
<td>Turns screen decorations off for printing.</td>
</tr>
</tbody>
</table>

#### Observer Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAppDeleted</td>
<td>P_APP_DELETED</td>
<td>Sent to observers of clsApp when a document is deleted.</td>
</tr>
<tr>
<td>msgAppFloated</td>
<td>P_APP_FLOATED</td>
<td>Sent to observers when a document is floated or unfloated.</td>
</tr>
<tr>
<td>msgAppTerminateConditionChanged</td>
<td>nothing</td>
<td>Try to terminate the document; sent when a terminate condition changed.</td>
</tr>
<tr>
<td>msgAppSelChanged</td>
<td>BOOLEAN</td>
<td>Sent to a document when something in it becomes selected or deselected.</td>
</tr>
<tr>
<td>msgAppOpened</td>
<td>APP_OPENED</td>
<td>Sent to observers when a document is opened.</td>
</tr>
<tr>
<td>msgAppClosed</td>
<td>APP_CLOSED</td>
<td>Sent to observers when a document is closed.</td>
</tr>
<tr>
<td>msgAppCreated</td>
<td>P_APP_CREATED</td>
<td>Sent to observers of clsApp when a document is created.</td>
</tr>
<tr>
<td>msgAppMoved</td>
<td>P_APP_MOVED_COPIED</td>
<td>Sent to observers of clsApp when a document is moved.</td>
</tr>
<tr>
<td>msgAppCopied</td>
<td>P_APP_MOVED_COPIED</td>
<td>Sent to observers of clsApp when a document is copied.</td>
</tr>
<tr>
<td>msgAppChanged</td>
<td>P_APP_CHANGED</td>
<td>Sent to observers of clsApp when a document has changed.</td>
</tr>
<tr>
<td>msgAppChildChanged</td>
<td>P_APP_CHILD_CHANGED</td>
<td>Sent to observers of a document when a child document is opened or closed.</td>
</tr>
<tr>
<td>msgAppInstalled</td>
<td>CLASS</td>
<td>Sent to observers of clsApp when an application is installed.</td>
</tr>
<tr>
<td>msgAppDeInstalled</td>
<td>CLASS</td>
<td>Sent to observers of clsApp when an application is deinstalled.</td>
</tr>
</tbody>
</table>

Although the header file for clsApp (APP.H) describes msgNew in its comments, clsApp does not define a method for msgNew. If you subclass clsApp, you should provide a method for msgNew that initializes a new document life cycle messages.
Several clsApp messages are intended only to be sent by the PenPoint Application Framework, especially those used in the document lifecycle. If you need to use any of these messages, refer to the PenPoint API Reference or the header file, APP.H.

- msgAppInit
- msgAppSave
- msgAppRestore
- msgAppOpen
- msgAppClose
- msgAppDelete

### Document Hierarchy Messages

If your application cannot use the default PenPoint Application Framework behavior for dealing with embedded documents, it can use these messages.

#### Managing Embedded Documents

Embedded documents are also known as child application instances. Usually the Notebook and the create menu take care of creating embedded documents for the user. However, if your application has a special need, you can use these messages to create embedded documents.

#### Activating All Embedded Documents

To activate all embedded documents, send `msgAppActivateChildren` to the parent application object. The message has no arguments.

#### Activating a Single Embedded Document

To activate a single embedded document, send `msgAppActivateChild` to the parent application object. The message requires a pointer to an `APP_ACTIVATE_CHILD` structure that contains two arguments:

- `pPath` the path to the child’s directory, relative to self.
- `uid` a location to receive the new child application’s UID.

### Document Links

The document links allow a document with embedded documents to be moved to other locations, yet keep track of its embeddees.

### Getting Document Information

There are two messages that can get information about a specific document: `msgAppGetMetrics` and `msgAppGetRoot`.

When a client sends `msgAppGetMetrics` to a document, it obtains the metrics for that document. The message requires a pointer to an `APP_METRICS` structure that receives these arguments:

- `uuid` the document’s UUID.
dir the document's directory.
parent the document's parent document.
children a list of document activated by this document.
mainWin the main window UID.
floatingWins the floating windows, if any.
childAppParentWin the parent window of a child application.
resList the resource file list for the document. The default resource list contains
- A document resource file.
- An application resource file.
- A preference resource file.
- The system resource file.
resFile The document's resource file.
flags An APP.Flags value that contains flags for the document. The flags values are:
state the document state. Possible values are: appTerminated, appActivated, appOpened.
hotMode whether the document is in hot mode.
floating whether the document is floating.
printing whether the document is printing.
topLevel whether the document is printing as top level.

When a client sends msgAppGetRoot to an application object, it obtains the root application object for a tree of applications. The only argument for the message is a pointer to an application UID to receive the UID for the root application object.

Setting Hot Mode

To change an application's hot mode metric, send msgAppSetHotMode to the application object. The only argument is a Boolean value that indicates whether hot mode should be on or off.

Most applications do not handle this message. They allow clsApp to handle it for them.

Renaming a Document

To rename a document, send msgAppRename to the application object. The only argument for the message is a pointer to a string that contains the new name for the document.

Applications should not process this message. They should allow clsApp to handle it for them. When clsApp has changed the application's metrics, it sends msgAppSetName to self (see below).
Getting an Application’s Name
To get an application’s name, send msgAppGetName to the application object. The only argument is a pointer to the string that receives the document’s name.

Setting an Application’s Title
To set an application’s main window title, send msgAppNameSet to the application object. The only argument is a pointer to the string that receives the document’s name.

Document Window Messages

Setting the Main Window
After creating a main window (usually in msgAppInit), applications need to set the main window. To do this, they send should send msgAppSetMainWin to self. The only argument is a UID that identifies the main window.

The application should not process msgAppSetMainWin, but should pass it to clsApp.

To add a window to the list of floating windows, send msgAppAddFloatingWin to the application object. The only argument is the UID of the window to add to the list.

To remove a window from the list of floating windows, send msgAppRemoveFloatingWin to the application object. The only argument is the UID of the window to remove from the list.

Standard Application Menus
To provide continuity among all applications running under the PenPoint operating system, three standard application menus (or SAMs) should appear on the menu bar of all applications: Document, Edit, and Options. These menus contain a number of predefined buttons. The system resource file contains a menu bar with these standard application menus. When you create an application’s user interface, you can use this resource to add your own buttons to the menu bar. (Part 4: UI ToolKit describes how to add the standard application menus to your interface.)

When your application is opened, it should create a menu bar and then send msgAppCreateMenuBar to self. clsApp handles msgAppCreateMenuBar by adding the Document, Edit, and Option menus in front of any other menus specified by your application. When msgAppCreateMenuBar returns, your application can further adjust the menu by removing buttons from the menus that your application does not support or adding additional buttons to the menus.

When the user taps on a menu (but before the menu is displayed), the menu sends msgControlProvideEnable to the document for each of the buttons in the menu. Each time msgControlProvideEnable is sent, it contains a tag for a menu button. The protocol for modifying the Options menu is discussed in “Options Menu Protocol,” later in this chapter.
It is up to your application and its ancestors to determine whether the particular button should be enabled or disabled. When a button is disabled, it still appears on the menu, but it is grayed out.

Your application should first allow its ancestor to handle `msgControlProvideEnable`. `clsApp` will enable or disable the buttons as described in Table 14-2. If your application owns the selection, `clsApp` enables or disables `tagAppMenuMove`, `tagAppMenuCopy`, or `tagAppMenuDelete`, depending on what is selected. If there is no data selected, `clsApp` disables all three buttons. If the application data is read-only, `clsApp` disables Move and Delete.

Apart from the selection-dependent buttons, `clsApp` simply enables or disables buttons based on its own defaults. Your application does not necessarily have to take `clsApp`'s decision as final; `clsApp` doesn't know enough about an individual application's data. For example, `clsApp` always disables `tagAppMenuSelectAll`. However, if your application supports the Select All button, it should re-enable `tagAppMenuSelectAll`.

When `clsApp` returns, your application should handle `msgControlProvideEnable` by:

- Examining the tag that identifies the button in question.
- Determining whether you should disable or enable the button.
- Setting the `enable` bit in the `pArgs` for the message.

### Table 14-2: How `clsApp` Handles Menu Button Tags

<table>
<thead>
<tr>
<th>Tag</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>tagAppMenuPrintSetup</td>
<td>Always enabled.</td>
</tr>
<tr>
<td>tagAppMenuAbout</td>
<td>Always enabled.</td>
</tr>
<tr>
<td>tagAppMenuCheckpoint</td>
<td>Always enabled.</td>
</tr>
<tr>
<td>tagAppMenuRevert</td>
<td>Always enabled.</td>
</tr>
<tr>
<td>tagAppMenuSearch</td>
<td>Always enabled.</td>
</tr>
<tr>
<td>tagAppMenuSpell</td>
<td>Always enabled.</td>
</tr>
<tr>
<td>tagAppMenuUndo</td>
<td>Enabled when undo transactions exist.</td>
</tr>
<tr>
<td>tagAppMenuPrint</td>
<td>Asks thePrintManager.</td>
</tr>
<tr>
<td>tagAppMenuSend</td>
<td>Asks theSendManager.</td>
</tr>
<tr>
<td>tagAppMenuMove</td>
<td>Asks theSelectionManager.</td>
</tr>
<tr>
<td>tagAppMenuCopy</td>
<td>Asks theSelectionManager.</td>
</tr>
<tr>
<td>tagAppMenuDelete</td>
<td>Asks theSelectionManager.</td>
</tr>
<tr>
<td>tagAppMenuSelectAll</td>
<td>Always disabled.</td>
</tr>
<tr>
<td>Any unrecognized tag</td>
<td>Always disabled.</td>
</tr>
</tbody>
</table>
## Document and Edit Menus

When the user taps a menu button, the menu sends a specific message to self (the document that owns the current selection). Table 14-3 lists the buttons in the Document and Edit menus and the messages that result from tapping that button. Some buttons in the table list the name of a well-known manager object, which is sent as one of the message arguments. (The reason for this is described after the table.)

<table>
<thead>
<tr>
<th>Menu Button</th>
<th>Message</th>
<th>Manager (if exists)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Revert</td>
<td>msgAppRevert</td>
<td></td>
</tr>
<tr>
<td>Send</td>
<td>msgAppSend</td>
<td>theSendManager</td>
</tr>
<tr>
<td>Print</td>
<td>msgAppPrint</td>
<td>thePrintManager</td>
</tr>
<tr>
<td>Print Setup</td>
<td>msgAppPrintSetup</td>
<td></td>
</tr>
<tr>
<td>Export</td>
<td>msgAppExport</td>
<td></td>
</tr>
<tr>
<td>About</td>
<td>msgAppAbout</td>
<td></td>
</tr>
<tr>
<td>Undo</td>
<td>msgAppUndo</td>
<td></td>
</tr>
<tr>
<td>Move</td>
<td>msgAppMoveSel</td>
<td></td>
</tr>
<tr>
<td>Copy</td>
<td>msgAppCopySel</td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>msgAppDeleteSel</td>
<td></td>
</tr>
<tr>
<td>Select All</td>
<td>msgAppSelectAll</td>
<td></td>
</tr>
<tr>
<td>Find</td>
<td>msgAppSearch</td>
<td>theSearchManager</td>
</tr>
<tr>
<td>Spell</td>
<td>msgAppSpell</td>
<td>theSpellManager</td>
</tr>
</tbody>
</table>

For application-specific behavior, applications can intercept these messages and perform the operations themselves. For example, the browser must intercept `msgAppPrint`, `msgAppSend`, `msgAppSpell`, and `msgAppSearch` so that it can act on the selected documents, not the Browser itself.

If your application does not intercept these messages, `clsApp` does one of two things:

- It provides the generic behavior.
- It routes the message to an object that provides the behavior.

To route the message to an object that provides the behavior, `clsApp` sends `msgAppInvokeManager` to self. The argument to the message is the well-known UID of a manager that performs the operation. When `clsApp` receives `msgAppInvokeManager`, it sends `msgAppExecute` to the manager object indicated by the argument to `msgAppInvokeManager`.

`msgAppExecute` takes a pointer to an APP_EXECUTE structure that contains four arguments:
app  the UID of the application requesting the action.

sel  the UID of the object that is currently selected. This argument is
     optional, depending on the manager's implementation.

count the number of UUIDs in the uuid array.

uuid an array of UUIDs that the action will affect. Each UUID indicates an
     application. This allows standard application menu messages to affect
     more than one application. The minimum content of the uuid array is
     usually this application.

When the manager object receives msgAppExecute, it uses this information to
perform the operation.

### Adding Menu Buttons

Usually when you add a button to a menu, the message sent by the button is
defined by your application. However, if the button activates a system-wide
facility that you expect many applications to use (such as a grammar checker), the
button should send msgAppInvokeManager.

Because msgAppInvokeManager is handled by clsApp and reroutes the button
information, you can use it to add additional buttons to the standard application
menus without actually modifying clsApp.

If you are adding a system wide button to a standard application menu, specify
that the button sends msgAppInvokeManager, and that its argument is the
manager that handles the operation. When the user taps the button, the button
will send msgAppInvokeManager to self. clsApp then sends msgAppExecute to
your manager object.

### Managers for Menu Buttons

A manager that is referenced by a standard application menu must respond to
msgAppExecute.

As described above, the arguments to msgAppExecute include:

- The UID of the application that is making the request. This allows the
  application to implement behavior (pop-up dialogs go away when the
  application goes away).
- An optional selection UID to indicate where functions such as spell and find
  should start.
- An array of UUIDs that indicate the applications to operate upon. The
default behavior of clsApp is to send only the UUID of the application itself.

### Default Behavior Provided by clsApp

The following sections describe the default behavior when clsApp handles the
messages generated by the standard application menus.
**msgAppPrintSetup**

When clsApp receives msgAppPrintSetup, it sends msgAppShowOptionSheet to self. This allows your application to add its own option cards to the Print Setup option sheet.

Your application should allow its ancestor to handle msgAppShowOptionSheet. When clsApp receives msgAppShowOptionSheet, it creates an option sheet by sending msgAppGetOptionSheet to self. Again, your application should allow clsApp to handle msgAppGetOptionSheet. Eventually it sends msgAppAddCards to self.

When your application receives msgAppAddCards, it can add its own option cards to the option sheet (as described below).

**msgAppAddCards** passes an OPTION_TAG structure that contains two arguments:

- **option** UID of the option sheet.
- **tag** tag for that option sheet.

To add your own option cards to the Print Setup option sheet, make sure that tag contains tagAppPrintSetupOptSheet. If it does, you add cards by sending msgOptionAddCard to the option sheet specified in option.

*Part 4: UI Toolkit* describes option sheets and the option sheet protocol in detail.

After adding your option cards, allow your ancestor to handle msgAppAddCards, so that it can add its own cards.

**msgAppPrint**

When clsApp receives msgAppPrint, it prints the application that was sent msgAppPrint. If your application allows other applications to be selected, you must handle msgAppPrint yourself to print the selected applications.

clsApp will forward the msgAppExecute message, the application UID, the selection UID (if the application owns the selection), and the application UUID to a well-known object, thePrintManager, for processing.

**msgAppSend**

When clsApp receives msgAppSend, it sends the application that was sent msgAppSend to the Out box. If your application allows other applications to be selected, you must handle msgAppSend yourself to send the selected applications.

clsApp will forward the msgAppExecute message, the application UID, the selection UID (if the application owns the selection), and the application UUID to a well-known object, theSendManager, for processing.

**msgAppSpell**

When clsApp receives msgAppSpell, it starts a spelling check on the current application. The spelling manager must use the traverse protocol to determine the range to be checked.
clsApp will forward the `msgAppExecute` message, the application UID, the selection UID (if the application owns the selection), and the application UUID to a well-known object, `theSpellManager`, for processing.

**msgAppSearch**

When clsApp receives `msgAppSearch`, it starts a search of the current application. The search manager must use the traverse protocol to determine the range to be searched.

clsApp forwards the `msgAppExecute` message, the application UID, the selection UID (if the application owns the selection), and the application UUID to a well-known object, `theSearchManager`, for processing.

**msgAppRevert**

When clsApp receives `msgAppRevert`, it terminates the current application without saving the data and creates a new instance from the application files. Applications that modify their application files (or other files that affect their state directly) must provide their own support for `msgAppRevert`.

**msgAppMoveSel**

When clsApp receives `msgAppMoveSel`, it sends `msgSelBeginMove` to the owner of the selection. This causes the move icon to pop up.

**msgAppCopySel**

When clsApp receives `msgAppCopySel`, it sends `msgSelBeginCopy` to the owner of the selection. This causes the copy icon to pop up.

**msgAppAbout**

When clsApp receives `msgAppAbout`, it brings up the Comments option sheet for the application and displays the About card. clsApp constructs the About card from the information available to it when the class is installed.

**msgAppSelectAll**

There is no default clsApp behavior for `msgAppSelectAll`.

**Options Menu Protocol**

The Options menu is dynamic; it contains options for both the document and the selection. When the user taps on the Options menu, clsApp sends `msgAppAddCards` to the document that contains the current selection. The document can add cards to the option sheet before or after sending the message to its ancestor. Each of its ancestors handle the message in the same way until clsApp receives the message.

clsApp then sends `msgAppAddCards` to the selected object. Again, the object and its ancestors can add cards to the option sheet. Finally, when clsApp receives the message again, it presents the Options menu on screen. The buttons in the menu
reflect the titles of all the option cards in the option sheets. The position of your
card titles in the Options menu depends on whether you call ancestor before or
after you send msgOptionAddCard.

Responding to msgAppAddCards
When your application or object receives msgAppAddCards, you can add cards by
sending msgOptionAddCard or msgOptionAddLastCard to the option sheet.
Send msgOptionAddLastCard to add your last (or only) card to the option sheet;
msgOptionAddLastCard creates the separator line in the option menu. At some
point in handling msgAppAddCards, you should also send msgAppAddCards to
your ancestor so that it can add its cards to the option sheet. You can send
msgAppAddCards to your ancestor before or after sending msgOptionAddCard
and msgOptionAddLastCard.

msgAppAddCards passes a pointer to an OPTION_TAG structure that contains:

- **option**  the UID of the option sheet to which you can add cards.
- **tag**  a tag that identifies the option sheet. The tag is provided as a
  convenience so that your application can quickly determine the option
  sheet for which it is creating a card. For the Options menu protocol, tag
  will always contain tagAppMenuOptions.

If you want to add cards to the option sheet, you send msgOptionAddCard to the
option sheet specified in option; to add the last (or only card) send
msgOptionAddLastCard to the option sheet. msgOptionAddCard and
msgOptionAddLastCard are described in Part 4: UI Toolkit. In brief, these
messages take a pointer to an OPTION_CARD structure that contains:

- **tag**  a tag that identifies the card to add.
- **pName**  pointer to a buffer holding the card’s name.
- **win**  this field must contain pNull.
- **client**  the UID of the client that manages the card.

You do not actually create the card until the option sheet sends you
msgOptionProvideCardWin. This prevents a card from being created until the
user turns to it.

Check Gesture Handling
When the user makes a check ✓ gesture on an object in your application, you
must be prepared to begin an option sheet creation. You respond to the check
gesture by sending msgAppShowOptionSheet to OSThisApp().
This code fragment shows how an application can respond to `msgGWinGesture`:

```c
APP_SHOW_OPTION_SHEET show;

case Msg(xgsCheck):
  case Msg(xgsUGesture): // Allow the user to be sloppy
    // Make self the selection.
    ObjCallRet(msgSelSelect, self, pNull, s);
    // Show the doc option sheet turned to the correct card.
    show.sheetTag = tagAppDocOptSheet;
    show.cardTag = tagAppOptGotoButtonCard; // your tag here
    show.show = true;
    ObjCallRet(msgAppShowOptionSheet, OSThisApp(), &show, s);
    break;
```

## Observing System Preferences

The system preferences are resources stored in the preferences resource file (in `\PENPOINT\SYS\PREFS\PREFS`). The system preferences describe system settings, such as the system font, left- or right-handed operation, screen orientation, and so on. Clients get and set the system preferences through the well-known object `theSystemPreferences`.

Applications observe `theSystemPreferences` by default. Thus, when the user changes the system preferences (such as the system font, user font, orientation, and hand preference), `theSystemPreferences` sends `msgPrefsPreferenceChanged` to observers.

Your application should not handle `msgPrefsPreferenceChanged`, but should allow its ancestor to handle the message so that `clsApp` can dirty all windows that are visible and lay them out again. This means that all your visible windows will acquire the attributes specified by the new preference.

The message passes a pointer to a `PREF_CHANGED` structure that contains:

- `manager` the UID of the object that sent the notification (usually `theSystemPreferences`).
- `prefId` a `RES_ID` value that contains the id of the preference that changed.

If you receive `msgPrefsPreferenceChanged` while you have windows that are not visible, you must dirty these windows and lay them out again yourself (after you send the message to `clsApp`). This code fragment suggests how to do this.

```c
WIN_METRICS wm;
OBJECT someWin; // a window that needs re-laying out
ObjCallRet(msgWinSetLayoutDirtyRecursive, someWin, (P_ARGS)true, s);
wm.options = wsLayoutDefault;
ObjCallRet(msgWinLayout, someWin, &wm, s);
```
Advanced Messages

The messages described in this section are useful only if you are writing a systems
application such as the Notebook. Unless your application is performing systems
functions, it should not send or respond to these messages.

Setting a Parent Document

You can establish the parent of a document by sending msgAppSetParent to the
document. The message takes the UID of the parent.

Setting Priority

To change the priority for an application task, send msgAppSetPriority to the
application object. msgAppSetPriority uses the same enums as the
OSTaskPrioritySet system service defined in OS.H. The message requires
an APP_SET_PRIORITY structure that contains:

- priorityMode  the task or group of tasks to be altered. Possible enum values
  are: osThisTaskOnly and osTaskFamily. You cannot use osAllTasks with
  this message.
- priorityClass  the priority class. The priorities are grouped into five classes:
  system (which is for use by PenPoint only, and has no enum), high
  (osHighPriority), medium high (osMedHighPriority), medium low
  (osMedLowPriority), and low (osLowPriority). These enum values are
  defined in OS.H.
- priority  the new priority value. The priority value must be between 1 and
  50, where 50 is the lowest priority. If you specify 0 for this argument, the
  priority value will stay the same; this allows you to change the class
  without changing the priority.
Chapter 15 / The View Class

The view class (clsView) is an abstract class that defines messages used to display an object's data. clsView is a descendent of clsEmbeddedWin. A view is a window that observes a single data object. When inserted in a window, the view displays the contents of the data object. A component is typically composed of a view-object pair.

- Chapter 15 covers these topics:
  - Creating new views
  - Subclassing clsView

View Concepts

Because any PenPoint object can be observed and can send messages to its observers, any object can be a data object. The observed object is responsible for maintaining its data, including storing and manipulating it.

The messages defined by clsView allow clients to create new views, and change the viewed object. clsView also responds to the object filing messages (msgSave and msgRestore) and passes these messages to its viewed objects.

Because clsView is an abstract class, you must either define your own subclass of clsView or use an existing subclass. Subclasses of clsView include clsTextView and clsGrafpaper.

You don't have to use clsView to display information on screen. Any descendant of clsWin is free to maintain its own data without having an associated data object.

clsView uses the window hierarchy; it does not maintain a separate view hierarchy. clsView does not filter out or respond to any messages defined by clsWin, so a subclass of clsView is free to manipulate the view with window messages if it wishes.

Table 15-1 lists the clsView messages.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_VIEW_NEW</td>
<td>Initializes arguments for a new view.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_VIEW_NEW</td>
<td>Creates a new view.</td>
</tr>
<tr>
<td>msgViewSetDataObject</td>
<td>OBJECT</td>
<td>Sets a view's data object.</td>
</tr>
<tr>
<td>msgViewGetDataObject</td>
<td>P_OBJECT</td>
<td>Gets a view's data object.</td>
</tr>
</tbody>
</table>
View Filing

Views file themselves like other objects. clsView responds to the common filing messages defined by clsObject (msgSave and msgRestore). An advantage to using views is that they then pass the filing messages to their viewed objects. The viewed objects must then respond to the filing messages.

Creating a New View

To create a new view, send msgNewDefaults and msgNew to the subclass of clsView. The message requires a VIEW_NEW structure that contains:

- object an OBJECT_NEW_ONLY structure.
- win a WIN_NEW_ONLY structure.
- view a VIEW_NEW_ONLY structure that contains:
  - dataObject the UID of the object the view is to observe.
  - createDataObject a Boolean value that indicates whether the view should automatically create the data object.

The msgNew defined by clsView cannot create the observed object (the createDataObject argument is meaningless); however, if you subclass clsView, your class can redefine msgNew to create the observed object.

When your subclass of clsView receives this message, it should check the createDataObject argument. If the argument is true and the object does not exist, your view should create the object. However, the arguments for VIEW_NEW_ONLY do not tell your view which object to create. Either your view must know which object to create, or it must define its own arguments to msgNew that include the object to create.

Setting the Data Object

To set or change the data object observed by a particular view, send msgViewSetDataObject to the view. The only argument to the message is the UID of the object to view.

Getting the Data Object

To find out what data object is observed by a particular view, send msgViewGetDataObject to the view. The only argument to the message is a pointer to the location that receives the UID of the object.
Subclassing clsView

The messages defined by clsView provide object filing, allow clients to create new views, and allow clients to change the viewed object. By themselves, these messages do not perform any of the tasks that clients require from views. It is up to your subclass of clsView to perform these tasks.

Your view class should:

- Make itself a subclass of clsView.
- Define a message that an observed object can send to your view when its data changes.
- Repaint its window when it receives the notification message.

Your view can also:

- Enable the user to interact with the data.
- Notify its observed object when the user makes a change (if the observed object defines such a message).
Chapter 16 / The Application Directory Handle Class

The application directory handle class, clsAppDirHandle, provides access methods to manipulate documents in the application hierarchy. clsAppDirHandle inherits from clsDirHandle.

clsAppDirHandle provides methods to access and modify application directory attributes. Because clsAppDirHandle is a descendant of clsDirHandle, clients can use clsAppDirHandle to create, destroy, and modify application directories.

Most application designers should not need to create an application directory handle. It is created for you during AppInit by the PenPoint Application Framework and stored in the dir element of the APP_METRICS structure. Therefore, normal applications should never need to use the messages defined by clsAppDirHandle.

This chapter covers the following topics:

- Creating application directories.
- Getting and setting application directory attributes.
- Getting attributes for many application directories.
- Determining a document's name.
- Counting embedded documents.
- Setting a document's tab.

Using clsAppDir

An application directory node represents a document in the document hierarchy. The PenPoint Application Framework uses the application directory structure to keep its documents in order and to communicate with the documents. Each document uses its application directory to store its resource files and any other files it uses. Because each document is contained in a separate directory in the file system, clsAppDir defines messages that augment the messages defined by its ancestor—clsDirHandle.

Application directory attributes contain information that pertains to several components:

- The document.
- The document's application class.
- The document's position within a section (or within another document).
The document's position within a section is determined by the document's **sequence number**. A document's page number within the Notebook is determined by its **global sequence number**. Embedded documents do not have global sequence numbers.

<table>
<thead>
<tr>
<th>Message</th>
<th>Tokens</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgAppDirGetAttrs</td>
<td>P_APP_DIR_GET_SET_ATTRS</td>
<td>Get the application directory attributes for a document.</td>
</tr>
<tr>
<td>msgAppDirSetAttrs</td>
<td>P_APP_DIR_GET_SET_ATTRS</td>
<td>Set the application directory attributes for a document.</td>
</tr>
<tr>
<td>msgAppDirGetFlags</td>
<td>P_APP_DIR_GET_SET_FLAGS</td>
<td>Get the application directory flags for a document.</td>
</tr>
<tr>
<td>msgAppDirSetFlags</td>
<td>P_APP_DIR_GET_SET_FLAGS</td>
<td>Set the application directory flags for a document.</td>
</tr>
<tr>
<td>msgAppDirGetClass</td>
<td>P_APP_DIR_UPDATE_CLASS</td>
<td>Get the application directory class.</td>
</tr>
<tr>
<td>msgAppDirSetClass</td>
<td>P_APP_DIR_UPDATE_CLASS</td>
<td>Set the application directory class.</td>
</tr>
<tr>
<td>msgAppDirGetUUID</td>
<td>P_APP_DIR_UPDATE_UUID</td>
<td>Get the application directory uuid.</td>
</tr>
<tr>
<td>msgAppDirSetUUID</td>
<td>P_APP_DIR_UPDATE_UUID</td>
<td>Set the application directory uuid.</td>
</tr>
<tr>
<td>msgAppDirGetSequence</td>
<td>P_APP_DIR_UPDATE_SEQ</td>
<td>Get the application directory sequence.</td>
</tr>
<tr>
<td>msgAppDirSetSequence</td>
<td>P_APP_DIR_UPDATE_SEQUENCE</td>
<td>Set the application directory sequence.</td>
</tr>
<tr>
<td>msgAppDirGetNumChildren</td>
<td>P_APP_DIR_UPDATE_NUM_CHILDREN</td>
<td>Get the numChildren attribute.</td>
</tr>
<tr>
<td>msgAppDirSetNumChildren</td>
<td>P_APP_DIR_UPDATE_NUM_CHILDREN</td>
<td>Set the numChildren attribute.</td>
</tr>
<tr>
<td>msgAppDirGetGlobalSequence</td>
<td>P_APP_DIR_GET_GLOBAL_SEQUENCE</td>
<td>Get the application directory global sequence number.</td>
</tr>
<tr>
<td>msgAppDirGetBookmark</td>
<td>P_APP_DIR_GET_BOOKMARK</td>
<td>Get the application tab.</td>
</tr>
<tr>
<td>msgAppDirSetBookmark</td>
<td>P_APP_DIR_SET_BOOKMARK</td>
<td>Set the application tab.</td>
</tr>
<tr>
<td>msgAppDirGetNextInit</td>
<td>P_APP_DIR_NEXT</td>
<td>Initialize the get next structure.</td>
</tr>
<tr>
<td>msgAppDirGetNext</td>
<td>P_APP_DIR_NEXT</td>
<td>Get the next application directory attributes.</td>
</tr>
<tr>
<td>msgAppDirReset</td>
<td>P_APP_DIR_NEXT</td>
<td>Free resources after a series of msgAppDirGetNext.</td>
</tr>
<tr>
<td>msgAppDirSeqToName</td>
<td>P_APP_DIR_SEQ_TO_NAME</td>
<td>Return an embedded document's name given its sequence number.</td>
</tr>
<tr>
<td>msgAppDirGetDirectNumChildren</td>
<td>P_U32</td>
<td>Get the number of direct embedded documents (not recursive).</td>
</tr>
<tr>
<td>msgAppDirGetTotalNumChildren</td>
<td>P_U32</td>
<td>Get the number of embedded documents (recursive).</td>
</tr>
</tbody>
</table>
Creating an Application Directory Handle

To create an application directory handle, send msgNewDefaults and msgNew to clsAppDir. Both messages require a FS_NEW structure that contains:

- **object** an OBJECT_NEW structure.
- **fs** an FS_NEW_ONLY structure that contains:
  - **locator** a locator for the node, which includes a directory handle and a path. If you do not specify the directory handle, the default is the WorkingDir; if you do not specify a path, the default is nil.
  - **dirIndex** an optional directory index. If you use a directory index, the locator indicates the volume to use.
  - **mode** flags that indicate handle characteristics.
  - **exist** what to do if the directory does or doesn’t exist.

Part 7: *The File System* describes in great detail how to create directory handles. Use msgNewDefaults to initialize the fields to their default values, then modify any of the fields.

Destroying an Application Directory Handle

To destroy an application directory handle, send msgFree to the application directory handle. The message takes a lock value.

Getting the Application Directory Global Sequence Number

To get the global sequence number for an application directory, send msgAppDirGetGlobalSequence to the application directory. The message takes a pointer to an APP_DIR_GET_GLOBAL_SEQUENCE structure that contains:

- **pPath** a path, relative to the target directory.
- **globalSequence** a U32 that will receive the directory global sequence value.

msgAppDirGetGlobalSequence calculates the global sequence number each time you call it.

If you need to calculate many global sequence numbers or need to make the calculation often, you should use the file system directly and not rely on msgAppDirGetGlobalSequence because it traverses the File System for each request.

For example, because the browser caches directories as it displays them, it makes the global sequence calculation based on its own data structures. You can calculate the globalSequence number of a document based on its local sequence number (with msgAppDirGetSequence) and number of children (with msgAppDirGetNumChildren).
Getting and Setting Application Directory Attributes

There are two ways to get or set the application directory attributes: you can get or set all of the attributes at once, or you can get or set specific attributes with specific messages.

Getting and Setting All Attributes

To get all system attributes for a single application directory, send msgAppDirGetAttrs to an application directory handle; to set all the attributes, send msgAppDirSetAttrs to the application directory handle. Both messages take an APP_DIR_GET_SET_ATTRS structure that contains:

- pPath a path to the target application directory.
- attrs an APP_DIR_ATTRS structure that contains:
  - appClass the application class.
  - uuid the application uuid.
  - sequence the local sequence number.
  - numChildren the total number of embedded documents within a document, including nested embedded documents.
  - flags the application directory flags. The flags are described in Table 16-2.

16.5.1

As a rule, it is faster to get or set single attributes.

Getting or Setting Individual Attributes

These messages get or set individual attributes for application directories.

Getting and Setting Flags

To get or set the application directory flags, send msgAppDirGetFlags or msgAppDirSetFlags to the application directory handle. Both messages take a pointer to an APP_DIR_GET_SET_FLAGS that contains:

- pPath a path, relative to the target directory.
- flags an APP_DIR_FLAGS structure, as described in Table 16-2.

Table 16-2

<table>
<thead>
<tr>
<th>Flag</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>application</td>
<td>This is an application.</td>
</tr>
<tr>
<td>newInstance</td>
<td>This is a new application instance.</td>
</tr>
<tr>
<td>disabled</td>
<td>The application is disabled. Do not activate it.</td>
</tr>
<tr>
<td>bookmark</td>
<td>The application has a tab.</td>
</tr>
<tr>
<td>readOnly</td>
<td>The application is read only.</td>
</tr>
<tr>
<td>deletable</td>
<td>The application can be deleted.</td>
</tr>
<tr>
<td>moveable</td>
<td>The application can be moved.</td>
</tr>
<tr>
<td>copyable</td>
<td>The application can be copied.</td>
</tr>
</tbody>
</table>

16.5.2.1
Getting and Setting the Class
To get or set the application class, send `msgAppDirGetClass` or `msgAppDirSetClass` to the application directory handle. Both messages take a pointer to an `APP_DIR_UPDATE_CLASS` structure that contains:

- `pPath` a path, relative to the target directory.
- `appClass` a CLASS specifier. On `msgAppDirSetClass`, this field specifies the application class; on `msgAppDirGetClass`, the application class is passed back in this field.

Getting and Setting the UUID
To get or set the application directory UUID, send `msgAppDirGetUUID` or `msgAppDirSetUUID` to the application directory handle. Both messages take a pointer to an `APP_DIR_UPDATE_UUID` structure that contains:

- `pPath` a path, relative to the target directory.
- `uuid` a UUID. On `msgAppDirSetUUID`, this field specifies the UUID; on `msgAppDirGetUUID`, the UUID is passed back in this field.

Getting and Setting the UID
To get or set the application directory UID, send `msgAppDirGetUID` or `msgAppDirSetUID` to the application directory handle. Both messages take a pointer to an `APP_DIR_UPDATE_UID` structure that contains:

- `pPath` a path, relative to the target directory.
- `uid` an application directory UID. On `msgAppDirSetUID`, this field specifies the UID; on `msgAppDirGetUID`, the UID is passed back in this field.

Getting and Setting the Sequence Number
To get or set the application directory sequence number, send `msgAppDirGetSequence` or `msgAppDirSetSequence` to the application directory handle. Both messages take a pointer to an `APP_DIR_UPDATE_SEQUENCE` structure that contains:

- `pPath` a path, relative to the target directory.
- `sequence` a U32 for the sequence number. On `msgAppDirSetSequence`, this field specifies the sequence number; on `msgAppDirGetSequence`, the sequence number is passed back in this field.

Getting and Setting the Number of Embedded Docs
To get or set the number of embedded documents in an application directory, send `msgAppDirGetNumChildren` or `msgAppDirSetNumChildren` to the application directory handle. Both messages take a pointer to an `APP_DIR_UPDATE_NUM_CHILDREN` structure that contains:

- `pPath` a path, relative to the target directory.
numChildren a U32 that specifies or passes back the number of embedded documents.

**Getting and Setting the Application Tab**

To get or set the application tab, send `msgAppDirGetBookmark` or `msgAppDirSetBookmark` to the application directory handle. `msgAppDirGetBookmark` takes a pointer to an `APP_DIR_GET_BOOKMARK` structure that contains:

- **pPath** a path, relative to the target directory.
- **label** a buffer of `nameBufLength` characters that will contain the tab label.

`msgAppDirSetBookmark` takes a pointer to an `APP_DIR_SET_BOOKMARK` structure that contains:

- **on** a Boolean value that specifies whether to turn the tab on or off.
- **pPath** a path, relative to the target directory.
- **label** a buffer of `nameBufLength` characters that specifies the tab label. If `label[0]` is null, use the default label (the name of the document).

**Getting Attributes for Many Application Directories**

`clsAppDir` provides messages to recursively get the attributes for a number of application directories, starting at a particular application directory. To get attributes in this way, you must first send `msgAppDirGetNextInit` to `clsAppDir`. The message takes an `APP_DIR_NEXT` structure that contains:

- **attrs** an `APP_DIR_ATTRS` structure. This is the same structure that was described above in `msgAppDirGetAttrs`.
- **pName** a pointer to a name string.
- **fsFlags** a set of file system flags.
- **pFirst** a pointer to the first directory handle. When you initialize the structure, you use this argument to indicate where to start. On subsequent reads, this argument serves as a reminder for where you started.
- **pNext** a pointer to the next directory handle.
- **handle** a pointer to the current directory handle.

When you have initialized the structure, you can get the attributes for the first directory in the search. Send `msgAppDirGetNext` to the directory handle, specifying the `APP_DIR_NEXT` structure that you initialized with `msgAppDirGetNextInit`.

To get attributes for the succeeding application directories, send `msgAppDirGetNext` to the directory handle.
When you have examined all application directories (the pNext attribute is Nil), free the resources used by msgAppDirGetNext. To free the resources, send msgAppDirReset to the directory handle.

### Determining a Document’s Name

If you have a document's sequence number or UUID, you can determine its name.

To determine the document's name from its sequence number, send msgAppDirSeqToName to the application directory file handle. The message takes a APP_DIR_SEQ_TO_NAME structure that contains:

- **sequence**: the sequence number of the document.
- **pName**: a pointer to the string that receives the document name.

To determine an embedded document's name from its UUID, send msgAppDirUUIDToName to the application directory file handle of the document in which it is embedded. The message takes a APP_DIR_UUID_TO_NAME structure that contains:

- **uuid**: the UUID number of the embedded document.
- **pName**: a pointer to the string that receives the document name.

### Counting Embedded Documents

To get the number of embedded documents in a document, send msgAppDirGetNumChildren to the application directory handle of the document you are interested in. The only argument for the message is a pointer to the U32 value that receives the number of children.

The number of children is determined recursively. It includes not only the number of directly embedded documents, but the embedded documents within an embedded document, and so on.

### Setting a Tab

Users can identify a document by creating a tab for that document (these tabs appear as the tabs on the edge of the Notebook). Because each application directory contains a specific document, it is natural for clsAppDir to provide a method for a client to create a tab associated with an application directory.

To set a tab, send msgAppDirSetBookmark to the root container application. The message takes a pointer to an APP_DIR_SET_BOOKMARK structure that contains:

- **on**: a Boolean value that indicates whether the tab is on or off. When this value is true, the tab is turned on.
- **pPath**: the path to the file being marked, relative to the target directory.
  
  Usually this field is null, because the tab identifies the application directory. However, if you want to identify an embedded document, pPath contains the path to that document.

**Note**: Tabs are called bookmarks in the clsAppDir API.
label  a string containing the tab label. The label can be up to
nameBuffLength bytes. If the first byte of label is null, clsAppDir uses
the default label (the application directory name).

When the message completes successfully, it returns.

You turn to a tab by sending msgRCAppGotoDoc and specifying the application UUID of the tab for the application UUID.
Chapter 17 / Container Application Classes

The PenPoint™ operating system uses container application classes to identify applications whose primary purpose is to contain (embed) other documents. You might use the container application classes if you write an alternative to the Notebook or if you create your own section application.

Concepts

The browser and other such components rely on the container application classes to distinguish between documents embedded in a section and documents embedded in any other document. Documents embedded in a section or Notebook can be listed by the browser and must be page numbered. Documents embedded in any other document should not be numbered by the browser and should not have page numbers. Embedded documents are described in Chapter 9 and Chapter 18.

The PenPoint operating system defines two container application classes: clsContainerApp, which defines a container application (such as a section) and clsRootContainerApp, which defines a root container application (such as a Notebook). Both container application classes are descendents of clsApp; both are abstract superclasses.

Figure 17-1 shows the hierarchy of the container application classes.
**Container Application Class**

*clsContainerApp* describes an abstract superclass used to embed documents. Creating an application that descends from *clsContainerApp* implies that the application's embeddees can be page numbered. Note that other application classes can contain embedded documents (such as a text editor document containing a database application), but that those embedded documents are not assigned page numbers.

Currently, the only descendent of *clsContainerApp* is *clsSectApp*, the section application.

*clsContainerApp* doesn't define any messages.

To test that a particular class descends from *clsContainerApp*, send *msgAncestorIsA* to the class, specifying *clsContainerApp*.

**Root Container Application Class**

*clsRootContainerApp* is an abstract superclass used to embed documents and containers. Creating a class that descends from *clsRootContainerApp* implies that the class is the highest level in the file system that should be searched for other containers.

Currently, the only descendent of *clsRootContainerApp* is *clsNBAppl*, the Notebook application. The desktop manager creates an instance of *clsNBAppl* when the PenPoint operating system is booted. Other instances of *clsNBAppl* include the Help notebook and the Stationery notebook.
clsRootContainerApp Concepts

Because an application cannot cause itself to be displayed on the screen, it must send a request to an application manager of some form. In PenPoint, clsNBApp receives the messages indicating which application to display. clsNBApp acts on the page turn messages by activating and deactivating instances of applications.

The messages defined by clsRootContainerApp are concerned with turning to different pages. This is the generalized set of messages that any implementation of a root container would have to handle in order to replace the clsNBApp.

A descendent of clsRootContainerApp manages its contents. It is responsible for displaying the current page on the screen and handling the page turn messages.

To test that a particular class descends from clsRootContainerApp, send msgAncestorIsA to the class, specifying clsRootContainerApp.

The root container application instance is created at boot time. If you need to get the UID of an application's root container application, send msgAppGetRoot to the application; to get the UID of your own root container application, send msgAppGetRoot to self.

clsRootContainerApp Messages

The messages for clsRootContainerApp are defined in RCAW.H.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgRCAppNextDoc</td>
<td>nothing</td>
<td>Increments pointer to the next document.</td>
</tr>
<tr>
<td>msgRCAppPrevDoc</td>
<td>nothing</td>
<td>Decrements pointer to the previous document.</td>
</tr>
<tr>
<td>msgRCAppExecuteGotoDoc</td>
<td>nothing</td>
<td>Moves to the goto document.</td>
</tr>
<tr>
<td>msgRCAppCancelGotoDoc</td>
<td>P_UUID</td>
<td>Resets the goto document to the current document.</td>
</tr>
<tr>
<td>msgRCAppGotoContents</td>
<td>nothing</td>
<td>Turns to the contents of the target root container.</td>
</tr>
<tr>
<td>msgRCAppGotoDoc</td>
<td>P_RCA_APP_GOTO_DOC</td>
<td>Turns to or brings to a document.</td>
</tr>
</tbody>
</table>

Changing the Current Document

To tell the root container application to go to the next or previous document, you must send two messages to the root container application. First, to establish the document to turn to (the reference document, you must send msgRCAppNextDoc or msgRCAppPrevDoc to the root container application. Then, to turn to the reference document, send msgRCAppGotoDoc to the root container application.

All three messages take no arguments. msgRCAppNextDoc and msgRCAppPrevDoc return the UUID of the new reference document. When msgRCAppGotoDoc returns stsOK, the reference document is the current document.
Cancelling the Reference Document

If you have set the reference document and want to reset the reference document to be the current document, send msgRDAppCancelGotoDoc to the root container application instance. The message takes no arguments. If the message returns stsOK, it has successfully cancelled the reference document.

Turning to the Contents Page

To go to the contents page, send msgRCAppReferenceContents to the root container application instance. The message takes no arguments. If the message returns stsOK, the current document is the contents page.
Chapter 18 / Embedded Window Class

Using clsEmbeddedWin

The following table lists the messages defined by clsEmbeddedWin in EMBEDWIN.H.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_EMBEDDED_WIN_NEW</td>
<td>Initializes embedded window data.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_EMBEDDED_WIN_NEW</td>
<td>Creates a new embedded window.</td>
</tr>
<tr>
<td>msgEmbeddedWinGetMetrics</td>
<td>P_EMBEDDED_WIN_METRICS</td>
<td>Passes back the embedded window metrics.</td>
</tr>
<tr>
<td>msgEmbeddedWinGetStyle</td>
<td>P_EMBEDDED_WIN_STYLE</td>
<td>Passes back the style of the embedded window.</td>
</tr>
<tr>
<td>msgEmbeddedWinSetStyle</td>
<td>P_EMBEDDED_WIN_STYLE</td>
<td>Sets the style of the embedded window.</td>
</tr>
<tr>
<td>msgEmbeddedWinBeginMove</td>
<td>P_EMBEDDED_WIN_BEGIN_MOVE_COPY</td>
<td>Places the embedded window in move mode.</td>
</tr>
<tr>
<td>msgEmbeddedWinBeginCopy</td>
<td>P_EMBEDDED_WIN_BEGIN_MOVE_COPY</td>
<td>Places the embedded window in copy mode.</td>
</tr>
<tr>
<td>msgEmbeddedWinMove</td>
<td>P_EMBEDDED_WIN_MOVE_COPY</td>
<td>Moves an embedded window to the destination.</td>
</tr>
<tr>
<td>msgEmbeddedWinProvideIcon</td>
<td>P_EMBEDDED_WIN_PROVIDE_ICON</td>
<td>Provides the icon for move/copy.</td>
</tr>
<tr>
<td>msgEmbeddedWinCopy</td>
<td>P_EMBEDDED_WIN_MOVE_COPY</td>
<td>Copies an embedded window to the destination.</td>
</tr>
<tr>
<td>msgEmbeddedWinMoveCopyOK</td>
<td>P_EMBEDDED_WIN_MOVE_COPY_OK</td>
<td>Checks that it is OK to move or copy the window to a destination.</td>
</tr>
<tr>
<td>msgEmbeddedWinGetPenOffset</td>
<td>P_XY</td>
<td>Gets pen offset during move or copy.</td>
</tr>
<tr>
<td>msgEmbeddedWinGetDest</td>
<td>P_EMBEDDED_WIN_GET_DEST</td>
<td>Gets the destination for embedded window move or copy.</td>
</tr>
<tr>
<td>msgEmbeddedWinForwardedGetDest</td>
<td>P_EMBEDDED_WIN_GET_DEST</td>
<td>Gets the destination for embedded window move or copy.</td>
</tr>
<tr>
<td>msgEmbeddedWinInsertChild</td>
<td>P_EMBEDDED_WIN_INSERT_CHILD</td>
<td>Inserts a child window into an embedded window.</td>
</tr>
<tr>
<td>msgEmbeddedWinExtractChild</td>
<td>P_EMBEDDED_WIN_EXTRACT_CHILD</td>
<td>Extracts a child window into an embedded window.</td>
</tr>
<tr>
<td>msgEmbeddedWinPositionChild</td>
<td>P_EMBEDDED_WIN_POSITION_CHILD</td>
<td>Repositions a child window within the same embedded window.</td>
</tr>
<tr>
<td>msgEmbeddedWinShowChild</td>
<td>P_EMBEDDED_WIN_SHOW_CHILD</td>
<td>Shows a given area to the user.</td>
</tr>
<tr>
<td>msgEmbeddedWinSetUUID</td>
<td>P_UUID</td>
<td>Sets the embedded window's UUID.</td>
</tr>
<tr>
<td>msgEmbeddedWinDestroy</td>
<td>OBJ_KEY</td>
<td>Destroys an embedded window.</td>
</tr>
<tr>
<td>msgEmbeddedWinGetPrintInfo</td>
<td>P_EMBEDDED_PRINT_INFO</td>
<td>Gets print information from an embedded window.</td>
</tr>
</tbody>
</table>
Creating an Embedded Window

Most clients do not send `msgNew` to `clsEmbeddedWin`. Rather, they send `msgNew` to one of the descendents of `clsEmbeddedWin`. If you are writing a descendent of `clsEmbeddedWin`, you must be prepared to define your own procedure for `msgNew` and create the arguments required by `clsEmbeddedWin`.

`msgNew` takes an `EMBEDDED_WIN_NEW` structure that contains structures for `clsEmbeddedWin`'s ancestors (`clsGWin`, `clsWin`, and `clsObject`) and an `EMBEDDED_WIN_NEW_ONLY` structure (`embeddedWin`) that contains:

- **uuid**: a UUID that identifies the embedded window.
- **style**: an `EMBEDDED_WIN_STYLE` style structure that specifies the attributes of the embedded window. The style attributes specify whether the embedded window itself can be an embeddor or an embeddee. The actual attributes are described later in this chapter in “Getting and Setting the Style of an Embedded Window.”

Destroying an Embedded Window

To destroy an embedded window, send `msgEmbeddedWinDestroy` to the embedded window object. The message takes a single argument: the object key used to create the embedded window object.

Before destroying the object, the message cleans up all references to the embedded window (such as reference buttons that point to components in the window).

After sending `msgEmbeddedWinDestroy` to the embedded window object, you should also send `msgFree` to the object.

Getting Embedded Window Metrics

To get the metrics for an embedded window, send `msgEmbeddedWindowGetMetrics` to an embedded window object. The message takes a pointer to an `EMBEDDED_WIN_METRICS` structure that the message uses to send back the metrics of the window. The structure contains:

- **uuid**: the UUID of the embedded window.
- **style**: an `EMBEDDED_WIN_STYLE` structure that contains the style of the window. The contents of the style structure are explained in the following section.

Getting and Setting the Style of an Embedded Window

The `EMBEDDED_WIN_STYLE` structure describes the attributes of an embedded window. This structure is part of the embedded window metrics, and is also used when creating a new embedded window.

To get the style of an embedded window, send `msgEmbeddedWinGetStyle` to the window. To set the style of an embedded window, send `msgEmbeddedWinSetStyle` to the window. Both messages take a pointer to an `EMBEDDED_WIN_STYLE` structure. The structure contains:
**emddor** a Boolean value that indicates whether the embedded window allows embedding.

**embeddee** a Boolean value that indicates whether the embedded window can be embedded.

**selection** a two-bit value that indicates the selection style to take when the embedded window receives `msgSelSelect`. The three possible selection styles are:
- **ewSelectUnknown** the selection style is unknown; use the style of your parent windows.
- **ewSelect** make this window the current selection.
- **ewSelectPreserve** preserve the current selection before making this the current selection.

**moveable** a Boolean value that indicates whether the embedded window can be moved.

**copyable** a Boolean value that indicates whether the embedded window can be copied.

**moveCopyMode** a two-bit value that indicates the current move or copy mode. Clients and subclasses must not alter the value of this field. Possible values are:
- **ewMoveCopyModeOff** not moving or copying.
- **ewMoveMode** moving in progress.
- **ewCopyMode** copying in progress.

**deletable** a Boolean value that indicates whether the embedded window can be destroyed.

**moveCopyContainer** a private Boolean value.

**embedForward** a Boolean value that indicates whether `msgEmbeddedWinGetDest` should be forwarded to the embedded window’s parent window.

**quickMove** a Boolean value that indicates that clsEmbeddedWin should use optimizations when moving windows within a common parent or process.

The selection styles are fully described in the Selection Manager chapter of *Part Nine: Utility Classes*.

---

**Beginning a Move or Copy Operation**

If the source in a move or copy operation uses the clsEmbeddedWin protocol it should indicate that it is ready to go into move or copy mode by sending `msgEmbeddedWinBeginMove` or `msgEmbeddedWinBeginCopy` to self. The message takes a pointer to an `EMBEDDED_WIN_BEGIN_MOVE_COPY` structure that contains:

- **xy** an `xy32` structure that specifies the pen location when you begin the move or copy.
bounds a rect32 structure that specifies the bounding box around the area to move or copy.

If the source allows clsEmbeddedWin to handle msgEmbeddedWinBeginMove or msgEmbeddedWinBeginCopy, it creates a default icon or a marquee around the selection and waits for the user to drop the object somewhere. If the bounds are zero, clsEmbeddedWin uses the default move/copy icon; otherwise, it uses the marquee.

Moving or Copying an Embedded Window

When the destination is ready to move or copy an embedded window from the source, it sends msgEmbeddedWinMove or msgEmbeddedWinCopy to the owner of the embedded window. The message takes a pointer to an EMBEDDED_WIN_MOVE_COPY structure that contains:

- xy an xy32 structure that specifies the x-y location in the destination window.
- dest the UID of the destination.
- format a tag that specifies the data transfer format.

The source should respond to this message by moving or copying itself to the destination. When the message returns, it passes back the UID of the object that was moved or copied (uid).

Confirming the Move or Copy

When the owner of the source in a move or copy operation receives msgEmbeddedWinMove or msgEmbeddedWinCopy, it sends msgEmbeddedWinMoveCopyOK to self to ensure that the destination can handle the thing being moved. The message takes a pointer to an EMBEDDED_WIN_MOVE_COPY_OK structure that the receiver uses to send back permission for a move and copy operation. The structure contains:

- moveOK a Boolean value that the receiver uses to specify that a move is possible.
- copyOK a Boolean value that the receiver uses to specify that a copy is possible.
- target an EMBEDDED_WIN_MOVE_COPY structure that indicates the proposed move or copy operation.

Getting the Destination of a Move or Copy

Before the source actually moves or copies the selection, it needs to determine the actual file system location of the destination object. This is especially true when moving or copying an embedded window. To get the file system location, send msgEmbeddedWinGetDest to the destination. The message takes a pointer to an EMBEDDED_WIN_GET_DEST structure that the destination uses to send back the destination information. The structure contains:
Handling Child Embedded Windows

To move or copy a child window between different objects in the same task, the easiest thing to do is to extract the child window from one parent and insert it in another. To extract the child window, send msgEmbeddedWinExtractChild to the parent. The message takes a pointer to an EMBEDDED_WIN_EXTRACT_CHILD structure that contains:

- xy an xy32 value that indicates the location in the source.
- win the UID of the embedded window to extract.
- source the UID of the object requesting the extraction (usually your UID).

To insert the child in a different parent, send msgEmbeddedWinInsertChild to the new parent. The message takes a pointer to an EMBEDDED_WIN_INSERT_CHILD structure that contains:

- xy an xy32 structure that contains the destination relative to the window.
- win the UID of the embedded window to insert.
- source the UID of the object requesting the insertion (usually your UID).

Getting Pen Offsets

As explained in this chapter, the x-y coordinates used by the embedded window move/copy protocol does not necessarily indicate the pen location when the user indicates the destination. Rather, it usually indicates the lower left corner of the icon or marquee when the user lifts the pen. To retrieve the offset of the pen from the lower left corner, send msgEmbeddedWinGetPenOffset to the source. The message takes a pointer to an xy32 structure that the message uses to send back the offsets.

Setting an Embedded Window’s UUID

To set the UUID for an embedded window, send msgEmbeddedWinSetUUID to an embedded window. The only argument for the message is a pointer to the new UUID. If the message returns stsOK, the embedded window has the new UUID.
Chapter 19 / Application Window Class

clsAppWin is a descendent of clsEmbeddedWin that wraps an embedded application and provides the interface between the application and the PenPoint Application Framework.

When clsAppWin contains an embedded application, the user can tap on the close box to shrink the embedded application down to a button in the document. When the user taps on the button, the embedded application expands to the size of the Notebook. When the user double taps on the button, the embedded application floats above the Notebook. clsAppWin implements this application window behavior.

Using clsAppWin Messages

The messages and #defines for the application window class are defined in APPWIN.H.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_APP_WIN_NEW</td>
<td>Initializes data for a new application window.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_APP_WIN_NEW</td>
<td>Creates a new object.</td>
</tr>
<tr>
<td>msgAppWinGetMetrics</td>
<td>P_APP_WIN_METRICS</td>
<td>Gets the application window metrics.</td>
</tr>
<tr>
<td>msgAppWinGetState</td>
<td>P_U16</td>
<td>Gets the application window state.</td>
</tr>
<tr>
<td>msgAppWinSetState</td>
<td>U16</td>
<td>Sets the application window state.</td>
</tr>
<tr>
<td>msgAppWinGetStyle</td>
<td>P_APP_WIN_STYLE</td>
<td>Gets the application window style.</td>
</tr>
<tr>
<td>msgAppWinSetStyle</td>
<td>APP_WIN_STYLE</td>
<td>Sets the application window style.</td>
</tr>
<tr>
<td>msgAppWinSetLabel</td>
<td>P_STRING</td>
<td>Sets the application window label.</td>
</tr>
<tr>
<td>msgAppWinSetIconBitmap</td>
<td>BITMAP</td>
<td>Sets the application window’s icon bitmap.</td>
</tr>
<tr>
<td>msgAppWinSetSmallIconBitmap</td>
<td>BITMAP</td>
<td>Sets the application window’s icon bitmap.</td>
</tr>
<tr>
<td>msgAppWinOpen</td>
<td>nothing</td>
<td>Opens a document associated with an app win.</td>
</tr>
<tr>
<td>msgAppWinClose</td>
<td>nothing</td>
<td>Closes a document associated with an app win.</td>
</tr>
<tr>
<td>msgAppWinDelete</td>
<td>BOOLEAN</td>
<td>Deletes an application window.</td>
</tr>
<tr>
<td>msgAppWinSetUUID</td>
<td>P_UUID</td>
<td>Sets the UUID to which an application window is linked.</td>
</tr>
<tr>
<td>msgAppWinCreateIcon</td>
<td>P_UUID</td>
<td>Creates the app win’s icon.</td>
</tr>
<tr>
<td>msgAppWinDestroyIcon</td>
<td>P_UUID</td>
<td>Destroys the app win’s icon.</td>
</tr>
<tr>
<td>msgAppWinStyleChanged</td>
<td>OBJECT</td>
<td>Notification that an app win style changed.</td>
</tr>
<tr>
<td>msgAppWinEditName</td>
<td>nothing</td>
<td>Pop-up edit pad to allow user to rename document.</td>
</tr>
</tbody>
</table>

Table 19-1 clsAppWin Messages

<table>
<thead>
<tr>
<th>Class Messages</th>
<th>Instance Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>msgAppWinGetMetrics</td>
</tr>
<tr>
<td>msgNew</td>
<td>msgAppWinGetState</td>
</tr>
<tr>
<td>msgAppWinSetState</td>
<td>msgAppWinGetStyle</td>
</tr>
<tr>
<td>msgAppWinGetStyle</td>
<td>msgAppWinSetState</td>
</tr>
<tr>
<td>msgAppWinSetStyle</td>
<td>msgAppWinSetStyle</td>
</tr>
<tr>
<td>msgAppWinSetLabel</td>
<td>msgAppWinSetLabel</td>
</tr>
<tr>
<td>msgAppWinSetIconBitmap</td>
<td>msgAppWinSetSmallIconBitmap</td>
</tr>
<tr>
<td>msgAppWinOpen</td>
<td>msgAppWinOpen</td>
</tr>
<tr>
<td>msgAppWinClose</td>
<td>msgAppWinClose</td>
</tr>
<tr>
<td>msgAppWinDelete</td>
<td>msgAppWinDelete</td>
</tr>
<tr>
<td>msgAppWinSetUUID</td>
<td>msgAppWinSetUUID</td>
</tr>
<tr>
<td>msgAppWinCreateIcon</td>
<td>msgAppWinCreateIcon</td>
</tr>
<tr>
<td>msgAppWinDestroyIcon</td>
<td>msgAppWinDestroyIcon</td>
</tr>
<tr>
<td>msgAppWinStyleChanged</td>
<td>msgAppWinStyleChanged</td>
</tr>
<tr>
<td>msgAppWinEditName</td>
<td>msgAppWinEditName</td>
</tr>
</tbody>
</table>
Creating an Embedded Application

To create an application window, send `msgNewDefaults` and `msgNew` to `clsAppWin`. Both messages take an `APP_WIN_NEW` structure that contains structures for `clsCustomLayout` and its ancestors and an `APP_WIN_NEW_ONLY` structure, which contains:

- `appUUID` the UUID of the application that will be embedded in the application window.
- `style` an `APP_WIN_STYLE` value that specifies the styles for the application window.
- `open` the open style for the window. The possible values are: `awOpenInPlace` and `awOpenFloating`.
- `type` the icon type. The possible values are: `awPictAndTitle`, `awPictOnly`, `awSmallPictAndTitle`, `awSmallPictOnly`, and `awSmallPictOverTitle`.
- `openStyleLock` a Boolean value that specifies whether the open style can change. If `true`, the open style cannot be changed.
- `state` a `U16` that specifies the state of the application window. The valid states are: `awClosed`, `awOpenedFloating`, `awOpenedInPlace`, and `awOpenedInPlaceFloating`.
- `label` a buffer that contains the icon label for the application window.

Changing the Style of an Application Window

The styles for application windows (described above in `msgNew`) can be changed. To change the style, send `msgAppWinSetStyle` to the application window. The only argument is the new style (`style`).

Opening an Application Window

When you open an application window, you specify whether it should float above the or be in-line in its embeddor. To open a window, send `msgAppWinOpen` to the application window. The message takes no arguments.

Closing an Application Window

To close an application window, send `msgAppWinClose` to the application window object. The message takes no arguments.

Getting the Metrics of an Application Window

To get the current settings of an application window, send `msgAppWinGetMetrics` to the application window object. The message takes an `APP_WIN_METRICS` structure that returns:

- `appUUID` the UUID of the contained application.
- `icon` the application window icon.
- `iconBitmap` the icon used for the open button.
smallIconBitmap  the small icon used for the open button.
appClass    the application's class.
style      the style of the application window. The styles are defined in
            APP_WIN_STYLE, and are listed in the description of msgNew.
state      the application window state.
label      the label used for the button.
Chapter 20 / The Mark Class

This chapter provides a brief description of the messages defined by clsMark in MARK.H. The concepts of using marks are explained in Chapter 10.

<table>
<thead>
<tr>
<th>Message</th>
<th>Tokens</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_MARK_NEW</td>
<td>Initializes the MARK_NEW structure to default values.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_MARK_NEW</td>
<td>Creates a new mark, initialized to the given component.</td>
</tr>
<tr>
<td>msgMarkDeliver</td>
<td>P_MARK_MESSAGE</td>
<td>Delivers a message to the target that does not move the token.</td>
</tr>
<tr>
<td>msgMarkDeliverPos</td>
<td>P_MARK_MESSAGE</td>
<td>Delivers a message to the target that moves the token but does not change the component.</td>
</tr>
<tr>
<td>msgMarkDeliverNext</td>
<td>P_MARK_MESSAGE</td>
<td>Delivers a message to the target that moves the token and sometimes (but not always) changes the component.</td>
</tr>
<tr>
<td>msgMarkSend</td>
<td>P_MARK_SEND</td>
<td>Sends a message not designed to work with marks to the target.</td>
</tr>
<tr>
<td>msgMarkSetComponent</td>
<td>P_MARK_COMPONENT</td>
<td>Sets the mark to refer to the given component.</td>
</tr>
<tr>
<td>msgMarkGetComponent</td>
<td>P_MARK_COMPONENT</td>
<td>Returns the UUID of the application that contains the token, and the UUID and UID of the component that contains the token.</td>
</tr>
<tr>
<td>msgMarkCompareMarks</td>
<td>MARK</td>
<td>Determines if two marks refer to the same component, and if so, what order their targets are in.</td>
</tr>
<tr>
<td>msgMarkCopyMark</td>
<td>P_MARK</td>
<td>Creates a new mark identical to this mark.</td>
</tr>
<tr>
<td>msgMarkGotoMark</td>
<td>P_MARK_GOTO</td>
<td>Causes a mark to be selected and displayed to the user.</td>
</tr>
<tr>
<td>msgMarkSetSaveMode</td>
<td>U32</td>
<td>Sets the mode that the mark will use for future saves.</td>
</tr>
<tr>
<td>msgMarkCreateToken</td>
<td>P_MARK_TOKEN</td>
<td>Instructs a component to create a token for its data items, and starts the token pointing at before all data items.</td>
</tr>
<tr>
<td>msgMarkGetToken</td>
<td>P_MARK_TOKEN</td>
<td>Sent from one mark to another to get the other's token.</td>
</tr>
<tr>
<td>msgMarkDeleteToken</td>
<td>P_MARK_TOKEN</td>
<td>Tells a component that the given token will no longer be in use.</td>
</tr>
<tr>
<td>msgMarkCompareTokens</td>
<td>P_MARKCOMPARE_TOKENS</td>
<td>Asks a component to compare the ordering of two tokens.</td>
</tr>
</tbody>
</table>

Table 20-1
The Class Mark Messages

Mixed with the previous discussion, some text is continued at the bottom of the page.
<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgMarkGetDataAncestor</td>
<td>P_CLASS</td>
<td>Asks for the next higher superclass that contains traversable data.</td>
</tr>
<tr>
<td>msgMarkGetParent</td>
<td>P_MARK_COMPONENT</td>
<td>Asks a component to set the argument to its parent (embedding) component.</td>
</tr>
<tr>
<td>msgMarkGetUUIDs</td>
<td>P_MARK_COMPONENT</td>
<td>Asks a component to set the argument to its own application and component UUIDs, if it can.</td>
</tr>
<tr>
<td>msgMarkValidateComponent</td>
<td>P_MARK_COMPONENT</td>
<td>Asks a component to verify that it is OK to traverse it.</td>
</tr>
<tr>
<td>msgMarkPositionAtEdge</td>
<td>P_MARK_POSITION_EDGE</td>
<td>Asks a component to reposition the token to one end or the other of the data.</td>
</tr>
<tr>
<td>msgMarkPositionAtToken</td>
<td>P_MARK_POSITION_TOKEN</td>
<td>Asks a component to reposition the token to the same position as another token for the same component.</td>
</tr>
<tr>
<td>msgMarkPositionAtChild</td>
<td>P_MARK_POSITION_CHILD</td>
<td>Asks a component to reposition the token to the given child component, which is given as a UUID/UID pair.</td>
</tr>
<tr>
<td>msgMarkPositionAtGesture</td>
<td>P_MARK_POSITION_GESTURE</td>
<td>Asks a component to reposition the token at the given gesture.</td>
</tr>
<tr>
<td>msgMarkNextChild</td>
<td>P_MARK_MESSAGE</td>
<td>Requests the component to move the token to the next child.</td>
</tr>
<tr>
<td>msgMarkGetChild</td>
<td>P_MARK_GET_CHILD</td>
<td>Requests the component to fill in the component at the current token.</td>
</tr>
<tr>
<td>msgMarkSelectTarget</td>
<td>P_MARK_MESSAGE</td>
<td>Requests the component to select the target data item.</td>
</tr>
<tr>
<td>msgMarkEnterChild</td>
<td>P_MARK_MESSAGE</td>
<td>Sent when a component requests the mark to enter a child (usually via returning stsMarkEnterChild to a message sent with msgMarkDeliverNext).</td>
</tr>
<tr>
<td>msgMarkEnterLevel</td>
<td>P_MARK_MESSAGE</td>
<td>Sent when a component requests the mark to bump up a level in its class chain, or when a position or next message fails and the mark tries the next class level.</td>
</tr>
<tr>
<td>msgMarkEnterParent</td>
<td>P_MARK_MESSAGE</td>
<td>Sent when a component runs out of data altogether and the mark needs to move on (and up).</td>
</tr>
</tbody>
</table>

### Messages Sent to Components via msgMarkDeliver

- msgMarkCreateToken instructs a component to create a token for its data items and start the token pointing before all data items.

## Marks Sent to Components

If a component wants to support search and replace, spell checking, reference buttons, and so on, it must respond to clsMark messages.
Deleting a Token

msgMarkDeleteToken informs a component that a token will no longer be in use. It is up to the component and its mapping scheme to determine how to handle this message.

Comparing Tokens

msgMarkCompareTokens asks a component to compare the order of two tokens. Your component should examine the tokens and return one of these status values:

- **stsMarkTokensEqual** the two tokens point to the same place.
- **stsMarkTokenAfter** the first token comes after the second.
- **stsMarkTokenBefore** the first token comes before the second.

Finding a Component’s Ancestor

msgMarkGetDataAncestor asks your component to pass back the first of its ancestors that can handle traversal within the component.

Finding a Component’s Parent

If your component doesn’t inherit from clsEmbeddedWin or clsApp, it should handle msgMarkGetParent by passing back either the UID or UUID of its parent (embedding) component. Components that inherit from clsEmbeddedWin or clsApp can allow their ancestors to handle this message.

Setting the UUIDs

If your component doesn’t inherit from clsEmbeddedWin or clsApp, it should handle msgMarkGetUUID by passing back its UUID and the UUID of the application that contains it. If your component can’t get this information, it should return **stsMarkNoUUIDs**. Components that inherit from clsEmbeddedWin or clsApp can allow their ancestors to handle this message.

Validating a Component

msgMarkValidateComponent is sent to objects before a mark refers to them, allowing the component to redirect the mark to another object.

Messages Sent to Components by msgMarkDeliver

The messages in this section are sent by the holder to marks or components by the mark delivery messages (msgMarkDeliver, msgMarkDeliverPos, or msgMarkDeliverNext). The mark delivery messages are described at the end of this chapter.

These messages can be sent directly to the component without using mark delivery messages, but some special processing performed when the message is sent with mark delivery will not be done.
Positioning Messages

msgMarkPositionAtEdge requests your component to reposition the token to the beginning or the end of the data.

msgMarkPositionAtToken requests your component to reposition the token to the same position as another token in the same component.

msgMarkPositionAtChild requests your component to reposition the token to a specified child component.

msgMarkPositionAtSelection requests your component to reposition the token to the current selection, assuming that your component owns the selection.

msgMarkPositionAtGesture requests your component to reposition the token at a specified gesture.

Messages Sent to Components That Have Children

If your component contains children (embedded components), it might receive these mark messages.

msgMarkNextChild requests your component to move the token to the next child.

msgMarkGetChild requests your component to pass back the UID and UUID of the embedded component at the current token.

Messages Sent by Holders

A holder is an object that creates and uses a mark. Most applications and components will not need to hold marks. However, if your application must traverse documents or otherwise keep a pointer to a location in data, it should create a mark.

Sending Messages to Marks

After creating a mark and telling it to map its token, your application or component uses mark delivery messages to pass your own protocol messages to the mark. For example, a spell checker might ask the component containing the target to search for the next word or embedded document.

There are three delivery messages:

- Deliver a message to the target.
- Deliver a message that might reposition the mark.
- Deliver a message to the component that contains the target.

You use msgMarkDeliver to deliver a message to the target. Usually, you use msgMarkDeliver after the mark fills in the token field.

You use msgMarkDeliverNext to deliver a message that might reposition the mark or that might move the mark to a different component.
You use **msgMarkSend** to deliver an arbitrary message to the component that contains the mark. Use this form when the message you are sending applies to the component as a whole, not the target in particular.

**Setting a Mark to a Component**

To set the mark to refer to a specific component, send **msgMarkSetComponent** to the mark. This message deletes the previous mark, if necessary. To make the mark point to nothing, pass a pointer to an all-zero structure; *do not* pass it a null pointer.

This message sends a **msgMarkCreate** to the component.

**Getting a Component's UUID and UID**

To get the UUID and UID of the component that contains the mark, and the UUID of the application that contains the component, send **msgMarkGetComponent** to the mark.

**Copying a Mark**

**msgMarkCopyMark** creates a new mark that is identical to the mark that receives this message. The new mark is not associated with the mark it was copied from (the receiver of **msgMarkCopyMark**). If the receiver is moved, the mark created by **msgMarkCopyMark** does not move.
Part 3 / Windows and Graphics
Coordinate Systems
  Logical Unit Coordinates 25.2 267
  Coordinate Unit Size 25.2.2 268
  Coordinate Rounding Error 25.2.3 269
Drawing Context Features
  Bound Accumulation 25.6 270
  Figure Drawing Operations 25.7 271
    Open Figures 25.7.1 271
    Closed Figures 25.7.2 271
  Sampled Images 25.8 272
    Drawing Sampled Images 25.8.1 273
    Cached Images 25.8.2 273
  Color 25.9 274
    RGB Color Values 25.9.1 274
    Hardware-Dependent Palette Colors 25.9.2 274
Text
  Fonts 25.10 275
    Opening a Font 25.10.1 275
    Scaling a Font 25.10.3 276
    Drawing Text 25.10.4 276
Character Encoding and Missing Glyphs 25.11 277
Graphic State 25.12 277
Picture Segments 25.13 279

Chapter 26 / The Drawing Context Class

System Drawing Context Messages 26.1 281
Creating a DC 26.2 285
  Default Drawing Context State 26.2.1 285
Should You Create a Drawing Context? 26.3 286
Drawing with a Drawing Context 26.4 286
Drawing Coordinates 26.5 286
  Defaults 26.5.1 287
  Units 26.5.2 287
  Scale 26.5.3 288
  Rotation 26.5.4 288
  Translation 26.5.5 288
  Resetting LUC 26.5.6 288
  Scaling to your Window (World Coordinates) 26.5.7 288
  Transformation Matrices 26.5.8 289
Sending Window Messages to Drawing Contexts 26.6 289
Graphic State 26.7 290
  Filling and Stroking 26.7.1 290
  Line Styles 26.7.2 290
  Line and Fill Patterns 26.7.3 291
Using Picture Segments in Graphics

Applications

- Editing a Picture Segment
- The Current Graphic
- Drawing by Adding Graphics
- Hit Testing a Picture Segment
- Invisible Graphics
- Converting to Other Formats
- What Picture Segments DON’T Do

Moving and Copying Picture Segments

- Editor Responsibilities
- Receiver Responsibilities

Chapter 28 / Bitmaps and TIFFs

Bitmaps

- Creating a New Bitmap
- Using a Bitmap
- Modifying
- Notifications

clsTiff

- Creating a New TIFF Object
- TIFF Image Metrics
- Repaint
- TIFF Images in Picture Segments
- Filing
- Destroying

Chapter 29 / ImagePoint Rendering Details

Logical Unit Coordinate (LUC)
Logical Device Coordinate (LDC)
Line Drawing
  - Rectangular Figures Without Borders
Rectangular Figures With Borders
Polygons
Line Width and Corner Radius Scaling
Rendering Geometric Shapes with Thick Borders
  - Figures Described by a Bounding Rectangle
  - Figures Described by a Point Path
Differences from Earlier Releases

List of Tables

- Window Style Flags
- clsWin Messages
- Default Window Style Flags
- clsSysDrwCtx Messages
- Default clsSysDC State
- Messages to Set LUC Units
- Common PenPoint Fonts
- clsPicSeg Messages
- Drawing Messages Corresponding to Picture Segment Grafics
- clsBitmap Messages
- clsTiff Messages

List of Figures

- A Drawing Context Bound to a Window
- PenPoint Windows and Graphics Classes Hierarchy
- A Simple Window Tree
- A User Interface Built of Windows

22-3 Clipping an Image
22-4 Window Clipping Regions
22-5 Clipping Children
22-6 Clipping Siblings
22-7 Sharing a Parent’s Clipping Region
22-8 Grouping Windows in a Transparent Window
22-9 Window Size and Position
23-1 The Notebook
25-1 A Simple Imaging Model
25-2 Transforming the LUC
25-3 Using a Local Clipping Rectangle
25-4 Open Figures
25-5 Closed Figures
26-1 Line Styles
26-2 Fill and Pattern Styles
26-3 Font Geometry
29-1 A Figure in a Logical Unit Coordinate System
29-2 End Points in One-Pixel Lines
29-3 LDC Rounding to Positive Side of LUC
29-4 Rectangles With and Without Borders
29-5 Left-Hand and Right-Hand Edges
29-6 Line Thickness in a Rectangle

List of Examples

- Creating and Drawing in a Window
- Allocating a Small Bitmap
- More Complex Use of an Image Device
- Drawing a Box
Chapter 21 / Introduction

Every PenPoint™ operating system application with a user interface presents that interface in a rectangular area of the screen called a window. The application draws its user interface elements in that window using an imaging model called ImagePoint™. This part describes windows and ImagePoint, and how you use them together to create images on the screen or other devices (such as printers).

Overview

When your application creates an image on an output device such as the screen or a printer, it uses the features of ImagePoint to draw in a window on the output device. ImagePoint provides a rich selection of drawing operations to facilitate the construction of complex images. The fundamental window class, clsWin, defines basic characteristics of a window such as the area of the output device in which to draw. In practice, however, effective use of windows usually requires subclasses of clsWin that add to these basic characteristics. For example, the PenPoint User Interface (UI) Toolkit, described in Part 4: UI Toolkit, is a collection clsWin subclasses that implements common user interface components such as buttons, menus, and input fields.

Windows

A window defines an area of an output device in which a drawing context can draw. In PenPoint, a window is an instance of clsWin, and an output device that allows the display of windows is an instance of clsWinDev. PenPoint provides a well-known instance of clsWinDev called the Screen, which represents the primary display screen. Devices other than the screen (printers, for example) can also serve as window devices.

ImagePoint

An application uses a drawing context (or DC) to create images in a window. A drawing context is an a set of drawing operations (such as an operation to draw a rectangle) together with a graphic state (a set of attributes such as line thickness). The graphic state determines the way in which the DC executes its drawing operations. For example, with the line thickness attribute set to 0.1 centimeters, the DC draws lines 0.1 centimeters thick. If an application changes the line thickness attribute to 0.25 centimeters, the DC draws subsequent lines 0.25 centimeters thick.
The standard imaging model for the PenPoint operating system is called **ImagePoint**. Because ImagePoint is the system standard, an ImagePoint drawing context is called a **system drawing context**. In PenPoint, a system drawing context is an instance of `clsSysDrwCtx`. Every system drawing context shares the same set of ImagePoint drawing operations, but each drawing context maintains its own graphic state to determine the way in which it renders images.

### Windows and ImagePoint Together

Before a drawing context can render an image, an application must specify the window in which to render it. The process of connecting a drawing context to a window is called **binding**. Once a drawing context and a window are bound, an application sends messages to the drawing context telling it which drawing operations to execute. The corresponding images appear in the area of the output device defined by the bound window.

### Windows and Graphics Classes

This part describes three main class groups: devices, windows, and drawing contexts. These groups form the class hierarchy shown in Figure 21-2.

### Window Devices

In PenPoint, an output device that supports the display of windows in an instance of `clsWinDev`. Each instance of `clsWinDev` corresponds to a physical output device that can display windows. For example, `theScreen`, a well-known instance of `clsWinDev`, represents the primary system display. `clsImgDev` is a subclass of `clsWinDev` which supports windows in memory, without a physical output device. See Chapter 24, Window Device Classes, for more information.
**Windows**

clsWin is the fundamental window class. PenPoint creates the `RootWindow`, a well-known instance of clsWin, to serve as the root window of the `Screen`. The User Interface Toolkit classes, described in Part 4: UI Toolkit, are subclasses of clsWin with specialized user interface behavior. You will probably use the UI Toolkit classes in your applications, or create your own clsWin subclasses, rather than use clsWin directly, so it is important that you understand the basic behavior of clsWin. This chapter and Chapter 23, The Window Class, provide most of the information you need to understand the UI Toolkit classes and clsWin subclasses in general.

**Drawing Contexts**

System drawing contexts, drawing contexts that implement the system-standard ImagePoint imaging model, are instances of clsSysDrwCtx. clsSysDrwCtx is a subclass of the abstract drawing context class clsDrwCtx. It is possible to implement an imaging model other than ImagePoint as another subclass of clsDrwCtx. Chapter 26, The Drawing Context Class, describes clsDrwCtx and clsSysDrwCtx, but does not explain how to implement another imaging model as a subclass of clsDrwCtx.
Application Writer's Overview

The following steps provide an outline of the messages required to create a window and render images in it.

1. Send msgNew and msgNewDefaults to clsSysDrwCtx and clsWin to create a drawing context and a window, respectively.

2. Send msgWinInsert to the window to insert the window into a window tree. The parent window often is an application frame created by the PenPoint Application Framework. Part 2: PenPoint Application Framework and Part 4: UI Toolkit describe frames in more detail.

3. Send msgDcSetWindow to the drawing context to bind it to the window.

4. Send graphic state messages such as msgDcSetLineThickness to the drawing context.

5. To render images in the window, send drawing messages such as sysDcDrawRectangle to the drawing context. The drawing context will render the images in the window to which it is bound.

Usually, your application renders an image not because it has new elements to draw, but because PenPoint informs one of your application's windows that it must repaint itself. Chapter 22, Window System Concepts, describes repainting in more detail.

Example

For almost every application, the PenPoint Application Framework creates a window called the application frame. The frame serves as a container for all of the application's other windows. The primary application window within the frame is the frame's client window (you can learn more about frames in Part 2: PenPoint Application Framework and Part 4: UI Toolkit).

The following code example shows how to create a client window for an application frame. You'll probably need to learn more about windows and ImagePoint before you can practically use the techniques shown here (for example by drawing in a window), the example provides a general idea of common use of windows.
Example 21-1

Creating and Drawing in a Window

This example from the Hello World SDK sample application (on the SDK distribution under \\
\PENPOINT\SDK\SAMPLE\HELLO\HELLO.C). The HelloOpen() message handler shows how to create a window and insert the window as the client of the application's main window, or application frame.

```c
MsgHandler(HelloOpen)
{
    HELLO_WIN_NEW    hwn;
    APP_METRICS      am;
    STATUS           s;
    // Get the app's metrics, including the app main window, or frame.
    ObjCallRet(msgAppGetMetrics, self, &am, s);
    // Create the Hello window.
    // The Hello World app defines clsHelloWin as a subclass of clsWin
    ObjCallWarn(msgNewDefaults, clsHelloWin, &hwn);
    ObjCallRet(msgNew, clsHelloWin, &hwn, s);
    // Insert the HelloWin (hwn.object.uid) as client window of the frame.
    ObjCallJmp(msgFrameSetClientWin, am.mainWin, (P_ARGS)hwn.object.uid,
               s, exit);
    // Ancestor clsApp will display the windows on the screen.
}
```

Organization of This Part

This part is organized into nine chapters. This chapter, Chapter 21, provides an introduction to windows and ImagePoint. Of the remaining chapters, three describe windows, and four describe ImagePoint and related drawing classes.

Chapter 22, Window System Concepts, provides a detailed overview of windows and window management in the PenPoint operating system.

Chapter 23, The Window Class, describes the messages defined by clsWin for creating, displaying, and manipulating windows. It also discusses the use of subclasses of clsWin to implement specialized window behavior.

Chapter 24, Window Device Classes, describes the messages defined by clsWinDev for creating a device that supports the display of windows. The chapter also describes clsImgDev, a subclass of clsWinDev that supports a window tree without an associated display device.

Chapter 25, Graphics Concepts, provides a detailed overview of the features of the ImagePoint imaging model. If you can do your work entirely with the UI Toolkit classes (described in Part 4: UI Toolkit), you may be able to skip Chapter 5 and subsequent chapters in this part. Chances are good, though, that you will need to implement one or more custom subclasses of clsWin in order to build a fully functional application.

You may be able to use the UI Toolkit classes after reading only the first four chapters in this part.
Chapter 26, The Drawing Context Class, describes the messages defined by clsSysDrwCtx for creating ImagePoint drawing contexts, binding them to windows, and drawing figures and text.

Chapter 27, The Picture Segment Class, provides an overview of the features of clsPicSeg, a subclass of clsSysDrwCtx which adds special features to the ImagePoint model, and describes the messages defined by clsPicSeg.

Chapter 28, Bitmaps and TIFFs, describes two classes that support the rendering of sampled bitmap and TIFF (Tagged Image File Format) images.

Chapter 29, ImagePoint rendering details, provides a technical overview of some low-level implementation details of ImagePoint's rendering mechanism.

**Other Sources of Information**

Some aspects of PenPoint windows and graphics are described elsewhere in this document. For example, *Part 4: UI Toolkit* describes frames, the application windows provided by the PenPoint Application Framework.

Outside of this *PenPoint Architectural Reference*, the best source of information on using PenPoint windows and graphics is the *Application Writing Guide*. In particular, its Hello World and Tic-Tac-Toe application development tutorials provide good examples of the topics described in this part of the *PenPoint Architectural Reference*. 
Chapter 22 / Window System Concepts

This chapter describes the PenPoint™ operating system model of windows and window management. It provides a detailed conceptual treatment of features of the window system, preparing you for the chapters describing the classes that implement those features.

Topics covered in this chapter include:

- The hierarchical window tree model
- Graphics support such as clipping
- The interactions between windows and the window system
- Programming support features
- Creating subclasses of clsWin.

What is a Window?

In the simplest sense, a window is an object that represents and defines a rectangular area of an output device. A window acts as a kind of canvas onto which a drawing context can render an image (in fact, some other window systems use the term canvas to describe what PenPoint refers to as a window). clsWin is the fundamental window class in PenPoint.

In practice, PenPoint applications use subclasses of clsWin to implement a wide variety of display elements such as status messages and user interface controls. The User Interface Toolkit (described in Part 4: UI Toolkit) is a standard collection of subclasses of clsWin that implements most of the user interface elements applications require. You need to understand clsWin in order to understand the features the UI Toolkit classes inherit from clsWin.

Much of the information in this chapter is useful in designing subclasses of clsWin. Because the UI Toolkit provides many standard clsWin subclasses, it may be tempting to skip this part and jump ahead to Part 4: UI Toolkit. However, every application requires that you design at least one custom subclass of clsWin (to act as the main window of the application), so it is important that you understand the information in this part.

Window System Concept Overview

The PenPoint window system offers a number of features that enhance the value of windows as a programming tool:

- A hierarchical window tree supports the appearance of multiple, overlapping windows.
Windows are **lightweight**. You can use hundreds of them at once without overburdening the system, you can use subclasses of clsWin for frequently-used objects such as user interface elements.  

- Windows support **clipping**, so that images rendered outside the window bounds do not appear on the output device.  
- PenPoint automatically notifies a window that it needs to **repaInt** itself when its image is damaged.  
- Window position is not tied absolutely to the display device, but is relative its parent window. Applications and the user can change the position and size of most windows.  
- Windows support a comprehensive **layout** system so that the arrangement of elements in the window is not dependent on the size of the window.  
- PenPoint provides methods for enumerating, tagging, sorting, searching for, and filing windows, and for sending a message that will propagate up the window tree.  
- clsWin self-sends a number of messages, providing hooks for custom behavior in your clsWin subclasses.

The remainder of this chapter describes these features in more detail.

### Hierarchical Model

The PenPoint window system can maintain many windows on a single output device, organizing them into a hierarchy called a **window tree**. Every output device that supports the display of windows must have an initial window to serve as the **root window** of its window tree. PenPoint provides a well-known object called theRootWindow to serve as the root window on the primary system display.

### Parents, Children, and Siblings

The window tree is a hierarchy in which every window (except for the root window on the window device) is a **child** of some other window in the window tree. A window that has a child window is the **parent** of that child. Any two or more windows with the same parent share a **sibling** relationship. A tree of windows whose root is a child of the root of another tree is a **subtree** of that other tree.

In figure 22-1, windowA and windowB are siblings, since both are children of the same parent (theRootWindow). windowC is the only child of windowB, and thus has no sibling. windowB is the root of a subtree that includes windowB and windowC. Even though it has no children, windowA is also a subtree (this is a degenerate case). In the same sense, theRootWindow is a window tree even when it has no children.
Window Devices

In almost every case, the system displays a window tree on a physical output device. PenPoint represents each of these physical devices as an instance of clsWinDev. PenPoint associates the root window of a window tree with a particular window device, and renders the window tree on that device. Any device that supports the rendering of output, from display screens to printers, can be a window device. PenPoint provides a well-known object called theScreen to represent the primary system display.

Window Life Cycle

An instance of clsWin typically undergoes four procedures: creation, insertion, extraction, and destruction.

Creation

When an application creates a new window, the window exists in memory just like any other object, but it is not part of any window tree. A window such as this is called an orphan, because it has no parent in the window tree. Orphan windows are not part of any window tree, and therefore cannot display themselves on the corresponding window device.

Insertion

Establishing an orphan window's parent in a window tree is called inserting the window into the window tree. Once inserted into the window tree, the window can display itself on the window device. In most cases, this means the window will display itself on the window device, because the window system instructs windows in the tree to redisplay themselves when there are changes in the window tree.

When PenPoint prints a document, it simply attaches a copy of that document's window as the root window of the printer window device. The document displays itself on the printer just as it would on the screen.
Extraction
The process of removing a window from a window tree is called extracting the window from the window tree. Extracted windows (and their children, if any) are not part of any window tree, and therefore cannot display themselves on any window device. If an application later reinserts an extracted window into a window tree, the window will appear exactly as when it was extracted (unless the application changed some attributes of the window while it was extracted).

Destruction
Destroying a window is like destroying any other PenPoint object: the window ceases to exist and the system frees up its memory for other uses. When you send a message to a window to destroy itself, it recursively sends the same message to its children (so the entire window subtree is destroyed). It is possible to set a flag in a window so that its parent does not destroy it in this way, but the flag is not on by default. Windows with this flag set become orphans, windows without parents, when their parent window is destroyed.

Windows are Lightweight
PenPoint implements windows as lightweight objects. This means that windows are not costly in terms of system resources such as memory or processor time. It is possible for the system to display and maintain hundreds of windows with very little performance degradation.

Figure 22-2 shows a typical user interface built of windows (actually, specialized subclasses of clsWin). Note that many windows, such as buttons and text fields, are not what the user normally considers to be a window. Nevertheless, these items inherit from clsWin. Because windows are so lightweight, even windows as common as user interface elements do not seriously reduce system performance.
Using Window Subclasses

Because windows are lightweight, you can design specialized classes that inherit from clsWin to implement commonly used user interface elements such as message displays and input fields. You can combine dozens of small windows, each responsible for managing a small part of your application's user interface, to create a complete interface for your application. In addition to reducing the time required to implement your application's user interface, the reuse of standard user interface elements makes your application easier to use.

Clipping

If an application attempts to draw an image that extends outside the boundary of the window to which it is bound, the basic behavior of clsWin prevents the part of the image that falls outside the window bounds from appearing. This process of filtering out the parts of images that fall outside the window's bounds is called clipping the image.

The Clipping Region

Each window maintains a clipping region that defines the area in which it will allow drawing. Normally, a window clips any part of a child window that extends past the parent's bounds. In Figure 22-4, windowB clips its child, windowC (the gray line indicates the part of windowC that does not appear on the display). One consequence of this feature is that only siblings, such as windowA and windowB, can overlap one another.

Figure 22-3
Clipping an Image
Clipping allows drawing operations to treat a window as a coherent drawing area. Drawing operations do not need to account for the fact that parts of the rectangle defined by a window object may be obscured on screen, either by the window's children or by its overlapping siblings. The disadvantage of clipping is that it can slightly increase the performance overhead of drawing operations.

**Overriding Automatic Clipping**

Clipping is not always necessary. For example, the UI Toolkit class `clsTabButton` is a descendant of `clsWin` that implements a tab along the side of a PenPoint notebook. An instance of `clsTabButton` displays a label inside a border, and its window is always large enough to contain the entire image. This means that the label image in each tab will never extend past the tab window bounds, and therefore that there is no need to prevent the window from drawing in its parent.

PenPoint provides several levels of control over automatic clipping: **clip children**, **clip siblings**, and share **parent clipping region**. This allows you to optimize drawing operations by turning off the automatic clipping you know is unnecessary.

**Clip Children**

By default, window clipping prevents drawing operations from affecting the child windows contained within the bounds of the target window. When you want to allow drawing operations to affect children of the target (to fill them all with the same color, for example), you can turn off child clipping.
**Clip Siblings**

By default, window clipping prevents drawing operations from affecting siblings of the target window that happen to overlap the target window. When you want to allow drawing operations to affect overlapping siblings of the target window, you can turn off sibling clipping.

**Parent Clipping Region**

A window must reserve some memory to store its clipping region. To conserve memory, a window can use its parent's clipping region instead of maintaining its own. This is appropriate when you know there will never be an attempt to draw images in the child window that extend outside the window bounds. In the clsTabButton example above, for instance, because tabs are always as large as the label they display, a clsTabButton does not require its own clipping region.
Repainting

Painting is the process of drawing an image in a window. An application paints a window by sending it a series of drawing requests, so painting is entirely under the application's control. However, a window's appearance may change because of factors outside of the application's control. For example, the user may resize the window.

When the display no longer accurately reflects the structure of the window tree, the window system notifies modified windows that they must repaint themselves. The default repaint behavior of clsWin, the fundamental window class, is simply to fill its bounding rectangle with white. Applications typically use subclasses of clsWin to implement more useful user interface behavior.

The window system does not keep track of the images displayed in windows. Each window is responsible for sending the appropriate drawing instructions when it receives a repaint request. By default, an instance of clsWin simply fills its bounding rectangle with white. Descendants of clsWin such as the User Interface Toolkit classes, add to or alter this basic behavior to repaint more meaningful imagery.

Why Windows Must Repaint

Each window must be prepared to repaint itself at any time. The window system will instruct a window to repaint itself whenever part of the window's image becomes damaged. A window is considered damaged when a change to the window tree affects one or more pixels contained within the window (the term dirty is a synonym for damaged, especially when referring to individual pixels).

For example, suppose a sibling overlaps an underlying window. If the user moves the sibling, the move may expose some pixels in the underlying window. These newly-exposed pixels, still display whatever was in the overlapping window before the user moved it. The pixels are therefore dirty, so the window system instructs the underlying window to repaint itself.

The Update Region

Before sending a repaint request, the window system calculates the update region, a rectangle containing all damaged pixels. The performance cost of repainting an entire window can be high, especially if the window receives damage frequently or its imagery is complex. In these cases, if you are writing a class that inherits from clsWin, you can improve its performance by repainting only that portion of the window that falls within the update region. However, it is rare that a window will require a level of processing that warrants this kind of optimization.

Printing

Another reason it is important for clsWin subclasses to respond to repaint requests is that PenPoint uses the repaint mechanism to print documents. When the user asks to print a document, the system makes a copy of the document window on a
window device representing the printer, turns off the window borders, then sends a repaint request to the window. The window uses exactly the same code to repaint itself on the printer as it would on any window device, providing a printer image that is virtually identical to the screen image.

**Transparency**

Windows can be transparent. Transparent windows do not fill themselves with white when they repaint, so they don't obscure other windows. When a transparent window needs to repaint, the window system causes windows below it to repaint first. In effect, the transparent window uses whatever is in the window behind it as its own background. In other words, if you don't draw in a transparent window, it is invisible.

You can draw in a transparent window just as in an opaque one. The effect is like painting on a piece of glass. One use for a transparent window is to group its children without obscuring what is behind them.

The UI Toolkit class clsTabBar is one example of the use of a transparent window to group several opaque child windows. clsTabBar implements the tab bar along the edge of the notebook as a transparent window. When the user adds a tab to the notebook, the notebook application creates the tab as an opaque child of the tab bar. Because the tab bar window is transparent, the window system instructs the window *behind* the tab bar to repaint before the tab bar can repaint itself. This ensures that any area not obscured by the opaque children shows through the grouping window.

**Figure 22-8**

*Grouping Windows in a Transparent Window*

---

**Window Size and Position**

A window defines a rectangular region of the display device. Applications measure the x (horizontal) and the y (vertical) size of a window in pixels, and specify the origin (lower-left corner) of a window in pixels relative to the origin of its parent window. For example, in Figure 22-9, windowA is a child of theRootWindow. Relative to theRootWindow's origin, windowA's origin is 20 pixels in the x direction and 50 pixels in the y direction. windowA's size is 100 pixels in the x direction and 75 pixels in the y direction.
Although applications specify each window's initial position and size, several factors can change them. For example, if your application allows it, the user can move and resize the window outside of your application's control. If you are designing your own subclass of clsWin, the repaint code will have to take into account that the window size, or that the positions of child windows, can change. PenPoint provides a mechanism for prohibiting changes to a window's position and size outside of application control, on an instance-by-instance basis.

**Layout**

PenPoint provides a scalable user interface model. That means that your application can make no assumptions about the size of the display device. One user may run it on a clipboard-sized machine, while another may use a pocket-sized machine. An application window needs to lay out its child windows (user interface components) in a reasonable fashion, independent of the size of the display.

Every window class inherits a mechanism called window layout that gives each window varying degrees of control over the position and sizes of its children. PenPoint provides three layout models: unconstrained layout, parent-veto layout, and episodic layout.

**Unconstrained Layout**

Unconstrained window layout is the type of layout most window systems provide. In such window systems, you are responsible for hard-coding layout algorithms into your application. PenPoint allows you to use unconstrained layout in which the parent allows child windows of any position and size. By default, the parent will always clip children that extend past the bounds of the parent. Unconstrained layout may leave the parent completely covered by children, leaving no space of its own on which to draw.
**Parent-Veto Layout**

In the parent-veto layout model, the window notifies the parent window of any pending operation that will insert, remove, move, or resize a child window. The parent must respond by approving, constraining, or vetoing the operation. If the parent approves the operation, it proceeds as requested. If the parent constrains the operation, the operation must select from a range of sizes and positions the parent specifies. Finally, the parent may veto outrageous requests, in which case the insert, remove, move, or resize operation does not take place.

**Episodic Layout**

Episodic layout is the most flexible layout model. When the application determines that it is necessary to layout a window's children (for example when the parent changes its size), it instructs the parent to initiate a *layout episode*. The parent requests size and position proposals from all of its children, the children respond with proposals, and the parent lays out the child windows while trying to preserve the positions and sizes they requested.

Since there may be conflicts between the various children's initial proposals, the parent may need to request additional rounds of proposals, mediating the children's requests until it establishes an acceptable compromise. Because of this, a layout episode can have a high performance cost, especially when there are many children involved. On the other hand, episodic layout provides the greatest degree of flexibility, because it keeps a user interface entirely independent of the size of the display or other variable elements.

Episodic layout provides a great deal of flexibility at the cost of a somewhat complex programming model. Chapter 23, *The Window Class*, describes episodic layout in more technical detail.

**Window Management**

clsWin provides a number of features that facilitate the management of group of windows. These facilities can be particularly useful when you are designing a special-purpose subclass of clsWin:

- A window can *enumerate* its children. In other words, it can generate a list of its children. The window system provides a number of enumeration options, including recursive enumeration.
- An application can associate a unique identifying *tag* with any window, allowing it to later refer to the window by its tag.
- An application can send a message to a window in a way that, if the window cannot respond to the message, the window will *propagate* a message up the window tree until one of its ancestors responds to the message (or the message reaches the root window).
- A window can *sort* its children, changing their placement on top of or below one another, based on a comparison routine your application provides.
A window can file its state, possibly including its child windows, for later reference. For example, a window that files a complex layout state can later restore the layout from the file, without the performance overhead of a layout episode.

**Subclassing clsWin**

clsWin is a fairly abstract class; to get really useful behavior, you must use subclasses of clsWin. For example, the UI Toolkit classes (described in Part 4: UI Toolkit), are clsWin subclasses that implement standard user interface elements such as text fields and buttons. However, you may have to design at least one custom subclass of clsWin to create an application. You can override the inherited response to certain messages to implement custom behavior.

For example, when a window system instructs a window to repaint itself, an instance of clsWin simply fills itself with white. If you were writing a spreadsheet application, you might design a subclass of clsWin that responds to msgWinRepaint by displaying the elements of an array of values and formulas in a grid. Layout is another area that may require custom behavior in clsWin subclasses. See Chapter 23, The Window Class, to learn more about clsWin's default behavior and how you might want to override it.
Chapter 23 / The Window Class

The window class (clsWin) provides basic window behavior upon which descendants such as the User Interface Toolkit classes (described in Part 4: UI Toolkit) are built. The primary purpose of windows is to serve as a canvas for drawing contexts, described in Chapter 26, The Drawing Context Class. This chapter describes the programming interfaces for clsWin, going on the assumption that you have read and understood the preceding chapter on window system concepts.

Topics covered in this chapter:
- Important structures in the implementation of clsWin
- Messages to which clsWin responds.

clsWin Structures

Before using clsWin, it is important to have a general understanding of two structures: window metrics and window flags. Window metrics, encoded in the WIN_METRICS structure, define what makes each instance of clsWin unique. One important element of the WIN_METRICS structure is the flags field, which defines a window’s display style and, for some subclasses of clsWin, its input behavior.

Window Metrics

Each instance of clsWin maintains a set of window metrics that define its appearance and behavior. An application can directly modify the window metrics before sending msgNew to create an instance of clsWin. After creating the window object, however, the application should modify its metrics only by sending messages that clsWin defines.

The messages clsWin defines for reporting and modifying window metrics generally take a P_WIN_METRICS as an argument. P_WIN_METRICS is a pointer to a WIN_METRICS structure, which includes fields for all of the window metrics information. clsWin messages that use a P_WIN_METRICS argument either report particular window metrics in the corresponding fields of the WIN_METRICS argument, or set particular window metrics according to the corresponding fields in the WIN_METRICS argument. These messages ignore the WIN_METRICS fields that are not relevant to the operation at hand.

The WIN_METRICS structure includes parent, child, bounds, flags, tag, and options fields. This section briefly describes how each of these fields is commonly interpreted.

Although messages ignore irrelevant WIN_METRICS fields in the final result, they may use them for intermediate calculations. Do not rely on the state of any WIN_METRICS fields except those the message is intended to report or set.
The parent field normally represents the window’s parent in a window tree. One notable exception is that when used with `msgWinInsertSibling`, the parent field represents the intended sibling of the window.

The child field is used by some window layout messages.

The bounds field represents the origin and size of the window. The origin is a coordinate pair (bounds.origin.x and bounds.origin.y) in parent window coordinates that defines the location of the window’s lower left corner. The size is a coordinate pair (bounds.size.w and bounds.size.h) in logical window coordinates that defines the width and height of the window relative to its origin.

The flags field defines two sets of flags, the window style flags (flags.style) and the window input flags (flags.input). The style flags define the display and layout behavior of the window. `clsWin` does not support input, so this chapter does not describe the input flags. The input flags support a common mechanism for specifying input behavior classes that inherit from `clsWin` (see Part 5: Input and Handwriting Translation for information about input flags).

The tag field holds the window’s tag. A tag is an arbitrary value you can associate with a window. `clsWin` defines a message that searches for a window with a particular tag.

The purpose of the options field is defined by the message that is taking the `P_WIN_METRICS` as an argument. Most of the `clsWin` messages use `P_WIN_METRICS` as an argument. For those that need special information to complete their tasks, `WIN_METRICS` provides the options field to carry that information.

**Window Flags**

The display style and input behavior of each instance of `clsWin` is modified by a set of flags. `clsWin` supports the input flags to support subclasses of `clsWin` that respond to input, but because `clsWin` does not respond to input, it ignores the input flags. Window style flags, on the other hand, affect every instance of `clsWin` and all its subclasses. Table 23-1 lists the window style flags.
Table 23-1
Window Style Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>What happens if flag is set</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsClipChildren</td>
<td>Drawing operations in the window do not affect children.</td>
</tr>
<tr>
<td>wsClipSiblings</td>
<td>Drawing operations in the window do not affect overlapping siblings.</td>
</tr>
<tr>
<td>wsParentClip</td>
<td>The window shares the same clipping region as its parent.</td>
</tr>
<tr>
<td>wsSaveUnder</td>
<td>The window stores for later retrieval the pixels it obscures on insertion.</td>
</tr>
<tr>
<td>wsGrowTop</td>
<td>When the window's size changes, the bottom edge of its image stays with the bottom edge of the window.</td>
</tr>
<tr>
<td>wsGrowBottom</td>
<td>When the window's size changes, the top edge of its image stays with the top edge of the window.</td>
</tr>
<tr>
<td>wsGrowLeft</td>
<td>When the window's size changes, the right edge of its image stays with the right edge of the window.</td>
</tr>
<tr>
<td>wsGrowRight</td>
<td>When the window's size changes, the left edge of its image stays with the left edge of the window.</td>
</tr>
<tr>
<td>wsCaptureGeometry</td>
<td>The window gets a chance to veto any insertion, change in position or size, or extraction of a child window.</td>
</tr>
<tr>
<td>wsSendGeometry</td>
<td>The window system tells the window whenever it has been successfully inserted, moved, resized, or extracted.</td>
</tr>
<tr>
<td>wsSendOrphaned</td>
<td>When the window's parent is freed, the window system sends msgWinOrphaned to the window instead of msgFree.</td>
</tr>
<tr>
<td>wsSynchRepaint</td>
<td>The window receives msgWinRepaint synchronously (via ObjectCall rather than ObjectSend).</td>
</tr>
<tr>
<td>wsTransparent</td>
<td>The window is transparent.</td>
</tr>
<tr>
<td>wsVisible</td>
<td>The window is visible.</td>
</tr>
<tr>
<td>wsPaintable</td>
<td>Painting operations can affect the window.</td>
</tr>
<tr>
<td>wsSendFile</td>
<td>msgFile will store the window in a file.</td>
</tr>
<tr>
<td>wsShrinkWrapWidth</td>
<td>During a layout episode, the width of the window will shrink to the minimum necessary to contain its children.</td>
</tr>
<tr>
<td>wsShrinkWrapHeight</td>
<td>During a layout episode, the height of the window will shrink to the minimum necessary to contain its children.</td>
</tr>
<tr>
<td>wsLayoutDirty</td>
<td>The window will lay out its children during the next layout episode.</td>
</tr>
<tr>
<td>wsCaptureLayout</td>
<td>The window will set wsLayoutDirty whenever a child window is inserted, moved, resized, or extracted.</td>
</tr>
<tr>
<td>wsSendLayout</td>
<td>The window will set wsLayoutDirty whenever it changes parent, position, or size.</td>
</tr>
<tr>
<td>wsFileInLine</td>
<td>The window will not file its object header when responding to msgFile.</td>
</tr>
<tr>
<td>wsFileNoBounds</td>
<td>The window will not file its bounds when responding to msgFile.</td>
</tr>
<tr>
<td>wsFileLayoutDirty</td>
<td>The window will set wsLayoutDirty when restored from a file.</td>
</tr>
</tbody>
</table>
**clsWin Messages**

Table 23-2 lists the messages defined by clsWin. They are organized into the following categories:

- **Creation messages** handle the common tasks of creating and destroying windows.
- **Window attribute messages** set and report flags that determine the appearance of the window.
- **Window display messages** handle the display of window contents.
- **Window layout messages** implement the protocols for a parent to lay out its children.
- **Window management messages** provide special functions for organizing and managing windows.
- **Filing messages** provide a mechanism for storing windows as files for later retrieval.

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>msgNew</code></td>
<td>Creates a window.</td>
</tr>
<tr>
<td><code>msgNewDefaults</code></td>
<td>Initializes the WIN_NEW structure to default values.</td>
</tr>
<tr>
<td><code>msgFree</code></td>
<td>Destroys a window and frees its memory, and recursively destroys and frees any of the window's children that do not have wsSendOrphaned set.</td>
</tr>
<tr>
<td><code>msgWinInsert</code></td>
<td>Sets the window's parent in the window tree.</td>
</tr>
<tr>
<td><code>msgWinInsertSibling</code></td>
<td>Sets the window's parent to the parent of another window.</td>
</tr>
<tr>
<td><code>msgWinExtract</code></td>
<td>Sets the window's parent to objNull, removing the window from the window tree.</td>
</tr>
<tr>
<td><code>msgWinDelta</code></td>
<td>Sets the window's size and position (bounds).</td>
</tr>
<tr>
<td><code>msgWinSetFlags</code></td>
<td>Sets the window's flags.</td>
</tr>
<tr>
<td><code>msgWinSetVisible</code></td>
<td>Sets the window's visibility flag, returns previous value.</td>
</tr>
<tr>
<td><code>msgWinSetPaintable</code></td>
<td>Sets the window's paintability flag, returns previous value.</td>
</tr>
<tr>
<td><code>msgWinGetFlags</code></td>
<td>Reports window style and input flags.</td>
</tr>
<tr>
<td><code>msgWinGetTag</code></td>
<td>Reports the tag (if any) associated with a window.</td>
</tr>
<tr>
<td><code>msgWinSetTag</code></td>
<td>Sets the window tag.</td>
</tr>
<tr>
<td><code>msgWinGetMetrics</code></td>
<td>Reports full window metrics.</td>
</tr>
<tr>
<td><code>msgWinRepaint</code></td>
<td>Tells a window to repaint itself.</td>
</tr>
<tr>
<td><code>msgWinBeginRepaint</code></td>
<td>Sets up window for painting on dirty region.</td>
</tr>
<tr>
<td><code>msgWinEndRepaint</code></td>
<td>Tells window system that repainting has ended for this window.</td>
</tr>
</tbody>
</table>

Table 23-2

**clsWin Messages**

continued
### clsWin Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgWinStartPage</td>
<td>Advises window that it is on a printer, and printing is about to commence.</td>
</tr>
<tr>
<td>msgWinBeginPaint</td>
<td>Sets up window for painting on its visible region.</td>
</tr>
<tr>
<td>msgWinEndPaint</td>
<td>Tells window system that painting has ended for this window.</td>
</tr>
<tr>
<td>msgWinDirtyRect</td>
<td>Marks all or part of a window dirty.</td>
</tr>
<tr>
<td>msgWinUpdate</td>
<td>Forces a window to repaint now, provided that it needs repainting.</td>
</tr>
<tr>
<td>msgWinCleanRect</td>
<td>Marks all or part of a window clean.</td>
</tr>
<tr>
<td>msgWinCopyRect</td>
<td>Copies pixels within a window.</td>
</tr>
<tr>
<td>msgWinTransformBounds</td>
<td>Transforms bounds from receiver's to another window's LWC.</td>
</tr>
</tbody>
</table>

### Window Layout Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgWinLayout</td>
<td>Tells a window sub-tree to layout.</td>
</tr>
<tr>
<td>msgWinLayoutSelf</td>
<td>Tells a window to layout its children.</td>
</tr>
<tr>
<td>msgWinGetDesiredSize</td>
<td>Gets the desired size of a window (used during layout).</td>
</tr>
<tr>
<td>msgWinGetBaseline</td>
<td>Gets the desired x-y alignment of a window.</td>
</tr>
<tr>
<td>msgWinSetLayoutDirty</td>
<td>Turns wsLayoutDirty bit on or off, returns previous value.</td>
</tr>
<tr>
<td>msgWinSetLayoutDirtyRecursive</td>
<td>Turns wsLayoutDirty bit on for every window in subtree.</td>
</tr>
<tr>
<td>msgWinInsertOK</td>
<td>Informs a potential parent of a pending child insertion.</td>
</tr>
<tr>
<td>msgWinExtractOK</td>
<td>Informs parent of a pending child extraction.</td>
</tr>
<tr>
<td>msgWinDeltaOK</td>
<td>Informs parent of a pending change in a child window's size or position.</td>
</tr>
<tr>
<td>msgWinFreeOK</td>
<td>Informs parent of the pending destruction of a child window.</td>
</tr>
<tr>
<td>msgWinInserted</td>
<td>Advises window that it has been inserted.</td>
</tr>
<tr>
<td>msgWinExtracted</td>
<td>Advises window that it has been extracted.</td>
</tr>
<tr>
<td>msgWinMoved</td>
<td>Advises window that it, or an ancestor, has moved.</td>
</tr>
<tr>
<td>msgWinSized</td>
<td>Advises window that it, or an ancestor, has changed size.</td>
</tr>
<tr>
<td>msgWinOrphaned</td>
<td>Tells a window its parent has been freed.</td>
</tr>
</tbody>
</table>

### Window Management Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgWinSend</td>
<td>Sends a message up a window ancestry chain.</td>
</tr>
<tr>
<td>msgWinEnum</td>
<td>Enumerates a window's children.</td>
</tr>
<tr>
<td>msgWinGetEnv</td>
<td>Gets the current window environment.</td>
</tr>
<tr>
<td>msgWinFindTag</td>
<td>Searches for a window tag in a window subtree. Returns match or objNull.</td>
</tr>
<tr>
<td>msgWinFindAncestorTag</td>
<td>Searches for a window tag in ancestor windows. Returns match or objNull.</td>
</tr>
<tr>
<td>msgWinSort</td>
<td>Sorts a window's children into a back to front order determined by a client supplied comparison function.</td>
</tr>
<tr>
<td>msgWinDumpTree</td>
<td>In lieu of msgDump, dumps a dense subset of information for the window and all it's children recursively.</td>
</tr>
</tbody>
</table>

### Filing Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgFile</td>
<td>Stores the window as a file.</td>
</tr>
<tr>
<td>msgRestore</td>
<td>Restores a filed window.</td>
</tr>
</tbody>
</table>
Creating a New Window

The simplest window class is clsWin. To prepare to create an instance of clsWin, an application first sends msgNewDefaults to clsWin, with a WIN_NEW structure as an argument. msgNewDefaults initializes the WIN_NEW structure to contain default values. The application then must specify the window's parent in the window tree. Additionally, it may need to specify the window size and position, and various flags that determine the display and input behavior of the new window. Once the application has initialized the WIN_NEW structure, it creates an instance of clsWin by sending msgNew to clsWin, with the initialized WIN_NEW structure as an argument.

Size and Position

Unless you know that the new window's parent will position the new window (in a layout episode, for example), you should specify the window's initial size and location in bounds. bounds is a RECT32 structure, made up of four signed 32-bit coordinates:

bounds.origin.x
bounds.origin.y
bounds.size.w
bounds.size.h

The origin coordinates specify the position of the new window's lower-left corner in pixels relative to the lower-left corner of the parent window. The size coordinates are in logical window coordinates (LWC), or pixels relative to the origin of the new window.

Window Style Flags

The flags.style attributes control the display behavior of a window. msgNewDefaults initializes flags.style to wsDefault, a combination of window style flags summarized in Table 23-3. It sets all other window style flags false.

<table>
<thead>
<tr>
<th>Flag</th>
<th>What happens when the flag is set</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsPaintable</td>
<td>The window can be painted.</td>
</tr>
<tr>
<td>wsVisible</td>
<td>The window is visible.</td>
</tr>
<tr>
<td>wsClipChildren</td>
<td>Drawing operations in the window do not affect children.</td>
</tr>
<tr>
<td>wsClipSiblings</td>
<td>Drawing operations in the window do not affect overlapping siblings.</td>
</tr>
<tr>
<td>wsLayoutDirty</td>
<td>The window will lay out its children during the next layout episode.</td>
</tr>
<tr>
<td>wsCaptureLayout</td>
<td>The window will set wsLayoutDirty whenever a child window is inserted, moved, resized, or extracted.</td>
</tr>
<tr>
<td>wsSendLayout</td>
<td>The window will set wsLayoutDirty whenever it changes parent, position, or size.</td>
</tr>
</tbody>
</table>
Window Metrics Messages

An application can directly modify the window metrics before sending msgNew to create an instance of clsWin. After creating the window, however, the application should send messages that clsWin defines for changing the window's metrics values. These messages generally take a P_WIN_METRICS (a pointer to a WIN_METRICS structure) as an argument, but most of the messages refer to only those WIN_METRICS fields relevant to the operation at hand.

To determine a given window's complete metrics information, an application sends msgWinGetMetrics to the window with a P_WIN_METRICS as its argument. This sets the WIN_METRICS fields equal to the window's metrics information. The following sections describe how an application sets and retrieves subsets of a window's metrics information.

Setting the Window Device or Parent

To establish a window's position in a window tree, the application must insert it into the tree. To insert a window into a window tree is to establish its parent window in that tree. clsWin supports three ways to insert a window into a window tree. The application may specify the intended parent window, the intended window device, or an intended sibling window.

To remove a window from its window tree, an application must extract it. An extracted window still exists as an object, but it has no parent window. Because an extracted window has no parent window, neither the extracted window nor any of its children are visible on any window device.

Inserting the Window

An application sends msgWinInsert to a window to specify the window's parent or window device. In either case, msgWinInsert takes a P_WIN_METRICS as an argument, either with the parent field set to the intended parent window or with the device field set to the intended window device. If the argument specifies a device, the window will have the device's root window as its parent. If the argument specifies a window, the new window will share the parent window's device.

msgWinInsert also looks at the options field of the passed WIN_METRICS to determine the front-to-back ordering of the new window. If options is set to wsPosTop, the window will insert in front of all of its siblings. If options is set to wsPosBottom, the window will insert behind all of its siblings.

Instead of inserting a window by specifying an intended window device or parent window, an application can specify an intended sibling window. msgWinInsertSibling takes as an argument a P_WIN_METRICS with the parent field set, but the parent field specifies the intended sibling window rather than an intended parent. After receiving msgInsertSibling, the newly-inserted window has the same parent as does the sibling window specified in the parent field.
Neither msgWinInsert nor msgWinInsertSibling refer to argument fields other than parent, device, and options. In particular, the insertion messages ignore the bounds information in the WIN_METRICS argument. Until an application sends msgWinDelta to a window, the window retains the bounds specified when it was created with msgNew.

**Extracting the Window**

To extract a window from the window tree, an application sends msgWinExtract to the window. msgWinExtract takes either a P_WIN_METRICS or pNull (a pointer to the null object) as its argument. If the window does not have a parent when the application sends it msgWinExtract, the message returns the status stsWinParentBad.

If the argument is pNull, msgWinExtract simply extracts the window from the window tree. If the argument is a P_WIN_METRICS, msgWinExtract sets the WIN_METRICS fields to reflect the metrics of the extracted window before extraction.

This is useful, for example, when you want to temporarily extract a window subtree, then reinsert it with the original parent. If your application gives msgWinExtract a P_WIN_METRICS argument, it will set the parent field to the parent of the window before it was extracted. If the application later sends msgWinInsert with the same WIN_METRICS argument, it will insert the window as a child of the same parent.

**Setting the Window Bounds**

A window's bounds are defined by two coordinate pairs: an origin (starting position) measured in pixels relative to the lower left corner of the window's parent, and a size measured in pixels relative to the origin. In a WIN_METRICS structure, these two coordinate pairs are encoded as a RECT32 called bounds.

To change the origin or size of an existing window, an application sends msgWinDelta to the window with a P_WIN_METRICS as an argument. msgWinDelta uses the bounds.origin.x and bounds.origin.y of the WIN_METRICS structure to set the origin of the window, and bounds.size.w and bounds.size.h to set its size.

**Setting the Window Flags**

In a WIN_METRICS structure, the window style flags are stored in the flags.style field. To set a window's flags, an application sets the flags.style fields of a WIN_METRICS structure to the desired values, then sends msgWinSetFlags to the window with a pointer to the WIN_METRICS structure (a P_WIN_METRICS) as its argument. To restore a window's style flags to the default, an application uses msgWinSetFlags twice: once to set the flags to zero, and once to set them to wsDefault.

Although this chapter does not discuss flags.input (described in Part 5: Input and Handwriting Translation), the messages described in this section use flags.input as well as flags.style.
Window Display Messages

Chapter 23 / The Window Class

Setting Individual Flags

If an application sends msgWinGetFlags to a window, with a P_WIN_METRICS as its argument, the flags field of the argument is set equal to the current flags of the window. The application can use bitwise logical operators to set and clear individual flags in the WIN_METRICS structure, then use the modified structure as an argument to msgWinSetFlags to set the window flags to the new values.

Because the window system sets the wsVisible and wsPaintable flags frequently, clsWin defines messages optimized for this purpose. msgWinSetVisible takes a Boolean value as its argument, sets the window's wsVisible flag equal to the argument, and returns the previous value of the wsVisible flag.

msgWinSetPaintable works exactly the same way for the wsPaintable flag.

Setting the Window Tag

clsWin lets an application associate an arbitrary 32-bit number, called a window tag, with any window. Window tags are zero by default. An application can determine the tag of a given window by sending msgWinGetTag to the window, with a P_WIN_METRICS as its argument. This will set the tag field of the argument WIN_METRICS equal to the window's tag.

An application sets a window's tag by sending msgWinSetTag to the window with a P_WIN_METRICS as its argument. This sets the window tag equal to the tag field of the argument WIN_METRICS. Tags allow an application to identify the windows it creates without having to assign them well-known UIDs.

If an application maintains many child windows with meaningful tags, it can search through them for a particular tag value by sending msgWinFindTag to the parent window, with the target tag value as its message argument. This recursively enumerates each child window (not including the parent) in breadth-first order and returns the UID of the first child window with a tag equal to the argument.

Window Display Messages

You display images in a window by sending messages to a drawing context bound to the window. The drawing context alters the window, but it is also subject to constraints the window defines. For example, the window defines the clipping region, which restricts the area in which the drawing context can render images. This section discusses the interactions between objects that result in the display of images in windows.

Clipping

clsSysDrwCtx cannot draw your entire figure if it is larger than your window or pixmap, and if you're drawing in a window it won't draw in hidden portions of your window. This process of selective drawing is called clipping. Repainting Dynamics, covers window clipping in intense detail.
Window Clipping

Normally windows don't get to draw on each other; each window has its own clipping region which does excludes all other windows. However, if you want to draw in child or overlapping sibling windows, you can reset the wsClipChildren and wsClipSiblings window flags.

For example, the UI Toolkit has a button table class which arranges its child windows into tables. The button table paints its background white; because it resets wsClipChildren this also paints the background of each child button.

As a performance optimization, child windows can share the clipping region of their parent if their wsParentClip flag is set. This means that child window painting is not restricted to the boundaries of the child window. For example, buttons don't attempt to draw outside their window boundaries, so it is safe for the button table to turn on wsParentClip in its children.

Unclipped Children

If a parent window does not have wsClipChildren set, then its drawing operations will show up in its child windows. This might be useful if a parent window is drawing a background that is appropriate for its child windows—without wsClipChildren, each child will have to redraw the background in itself.

Similarly, if a child window does not have wsClipSiblings set, then its drawing may show up on overlapping sibling windows. This isn't especially useful.

Maintaining a clipping region for each window takes some memory. If a child window only draws inside itself, it does not need its own clipping. If you set wsParentClip, the child window will try to borrow its parent's clipping region. For example, UI Toolkit labels don't draw outside themselves, so they can share their parents clipping.

Drawing in a Window

Assuming you have a window, and have associated a drawing context with it, you can then proceed to draw in the window.

Repainting is as Important as Painting

The window system does not remember what you have painted in a window. If some or all of your window's pixels are no longer visible on the display, then when they reappear, you have to repaint their contents. If you are going to be painting in a window, you need to concern yourself with how its pixels will be repainted. It turns out that figuring out how you are going to repaint your window is as important as what you will paint in it.
Repainting Dynamics

Dirty Windows

All windows must be able to redraw themselves at any time. The window system tells a window to repaint itself by sending it \texttt{msgWinRepaint}. This takes no arguments. Descendant classes sometimes create a DC and pass it with \texttt{msgWinRepaint} for their ancestors to draw with. You can check the message argument of \texttt{msgWinRepaint} to see if it is \texttt{pNull}.

When You’re Told to Repaint

When part of the screen must be repaired, windows in that part can be sent \texttt{msgWinRepaint} redraw messages. Most programmers will want to have their windows sent redraw messages automatically. If you know that temporarily you don’t want to receive these messages you can send your window \texttt{msgWinSetPaintable} with message argument \texttt{false} to reset its \texttt{wsPaintable} flag. For example, you might be repositioning many windows and want to delay repainting until you have finished.

Once a window is dirtied, it will eventually receive \texttt{msgWinRepaint} if it is paintable. As for when it receives \texttt{msgWinRepaint}:

- A window normally receives \texttt{msgWinRepaint} when its task has no other work to do: the task unwinds to its message dispatching loop, and takes \texttt{msgWinRepaint} off the queue.
- A window may receive \texttt{msgWinRepaint} immediately if it has \texttt{wsSynchRepaint} set. This causes any dirtying of itself by code in the same process to be repaired by an immediate \texttt{ObjectCall} of \texttt{msgWinRepaint}.
- A dirty window also receives \texttt{msgWinRepaint} if you send it \texttt{msgWinUpdate} yourself.
- A window could receive \texttt{msgWinRepaint} at random if someone else sends the message. This should never happen. \texttt{msgWinUpdate} is the correct way to tell a particular window and its children to repaint.

What to Do When Repainting

The first thing to do is to create a DC and bind it to your window, if you have not already.

If your window has changed size, you may want to change the scale of your DC. (Often you set window flags in order to be notified of changes in size and position in advance of repaint.)

Then you issue the appropriate \texttt{msgDc...} commands to repair the damage to the window by repainting part of it. Now, you would like to redraw only those areas of your window that are dirty. The window system makes it easy to do this, because it knows what area of your window is dirty and requires repainting. You
must send `msgWinBeginRepaint` before you start repainting, so that your
drawing operations only appear in the dirty region.

After `msgWinBeginRepaint`, any drawing messages you send to your DC only
affect the dirty portion of the window. (This clipping is independent of the
clipping area which you can set for your DC.)

### Sample Repaint Code

Most programmers will want to have their windows sent redraw messages
automatically (this is the default). Here is a typical code fragment for repaint:

```c
MsgHandler(MyRepaint)
{
    SYSDC myDc; // Created here, or in instance data, or shared.
    SYSDC_NEW sdn;
    STATUS s;
    ObjCallRet(msgWinBeginRepaint, self, pNull);
    // Create DC, if not already existing.
    ObjCallRet(msgNewDefaults, clsSysDrwCtx, &sdn, s);
    ObjCallRet(msgNew, clsSysDrwCtx, &sdn, s);
    myDc = sdn.object.uid;
    // Bind DC to window, if not already done.
    ObjCallJmp(msgDcSetWindow, myDc, (P_ARGS)self, s, Error);
    // Send messages to set up myDc to as necessary (scale, color, line thickness, ...).
    ...
    // Send messages to myDc to repaint entire window.
    ...
    ObjCallJmp(msgWinEndRepaint, self, Nil(P_ARGS), s, Error);
    return stsOK;
}
```

`clsHelloWin` in the sample program Hello World (in \PENPOINT\ SDK\SAMPLE\HELLO\HELLOWIN.C) implements a similar system except that it creates its DC
when it is created instead of during repaint.

### Smart Repainting

Even though the window systems restricts the drawing to the dirty region, the
window object still sent messages to redraw every pixel in the window. This
includes pixels which didn't need repainting. The window system overhead for
drawing that gets clipped away is marginal, so this may not be a problem.
However, if repainting the window requires a lot of computation, or if it involves
dozens of drawing messages, it may be worth being smarter about what to repaint.
GrafPaper and text views both use smart repaint strategies.

When you send `msgWinBeginRepaint`, the window system sets the **update
region** to be the dirty region. If you pass in a pointer to a `RECT32` structure,
`msgWinBeginRepaint` passes back the coordinates of the bounding rectangle of
your window's dirty region. If you write smart repainting code that can repaint
parts of the window on demand, then you can look at the coordinates of this rectangle and only issue the messages to redraw the areas which need repainting.

For example, in an organizational-chart drawing program you could soup up the repainting code to use the rectangle enclosing the dirty region. You determine which boxes fall in the dirty region, and only repaint those boxes and their attached connector lines.

Ordinary Painting by Repainting

We have gone through how to repaint a window, without covering how to get something drawn in it in the first place. It turns out that the easiest and best way to paint something in your window is often to mark the area you want painted as dirty, then wait for the window system to send your window a `msgWinRepaint` message telling it to repaint itself. You send `msgWinDirtyRect` to a window (or to a DC bound to it) to dirty part of it. The message takes a pointer to a `RECT32` structure in which you specify the area of the window to mark as dirty. The window system adds that area to the existing dirty region.

The benefits of this approach are:

- A single piece of code handles both painting and repainting.
- There's a minimum of flashing while you paint, since the window system will often have collapsed several dirty regions into one repaint message by the time you receive it.

However, the corresponding disadvantage of painting by marking dirty is that the image the user sees on the screen will be incorrect for a longer instant than if you painted immediately.

For example, suppose the org-chart application needs to add a new box to its chart at the top right. It could update its internal state to reflect a medium-size box in the upper right corner, and then dirty that region of its window with `msgWinDirtyRect`. At some point, the window gets sent `msgWinRepaint`. When the window starts repainting with `msgWinBeginRepaint`, the right corner will be part of the dirty region.

To mark your entire window as dirty (and hence paint the entire window), pass `pNull` as the message argument to `msgWinDirtyRect`.

Telling a Window to Repaint

If you don't want to wait for the window system to get round to telling your window to repaint, you can use `msgWinUpdate`. This makes the window system send (using `ObjectCall`) `msgWinRepaint` to your window and all its children that need repainting. Because the `msgWinRepaint` is delivered synchronously, the repaint happens before `msgWinUpdate` returns.
The Few Occasions When You Want to Explicitly Paint

There are a few inter-related situations in which you might want to explicitly paint:

- If you need to repaint synchronously (instead you could set wsSynchPaint and mark yourself dirty)
- If you don’t want the window system to collapse painting operations. Normally the window system collapses all pending window damage into a single msgWinRepaint message covering the entire damaged area.
- If you need to paint during a grab, or in a single thread of control.

In all these cases, if you paint by repainting, the interaction is over before your window receives msgWinRepaint and you can start repainting.

Before starting painting, you must send msgWinBeginPaint to your window, and afterwards you must send msgWinEndPaint. This is like msgWinBeginRepaint, except that it sets the update region to the visible part of the window instead of the dirty region.

The Update Region

The update region is created at msgWinBeginPaint/msgWinBeginRepaint time. It belongs to the window and applies to all drawing (including nested calls to msgWinBeginPaint/msgWinBeginRepaint). DCs paint into this region or a subset of it (if they have a clip rectangle set up with msgDcClipRect).

msgWinBeginPaint sets the update region to be the visible area of the window.

msgWinBeginRepaint sets the update region to be the dirty area of the window. The update region will always be the same as, or smaller than, the visible area, since only visible parts of the window can need repainting.

When Updating Ends

The window system doesn’t know when you’ve finished painting or repainting your window. You must always end a string of painting operations initiated with msgWinBeginPaint or msgWinBeginRepaint with a corresponding msgWinEndPaint or msgWinEndRepaint message.

When you send msgWinEndRepaint, if you’ve been repainting the dirty region, the window system believes that all damage that has accumulated has been repaired (new dirt may have slapped on while you were repainting—see the next section).

When you send the window system assumes that the region which was dirty is now clean, so long as you received that area via msgWinBeginRepaint.

If you’ve been painting, the window system makes no such assumption. If a region of your window is dirty, and you start painting (not repainting) in it, the window system can’t tell whether or not your painting operations will repaint that region successfully. You might not repaint that area, or only paint every fifth pixel. So the
window system will still later send you \texttt{msgWinRepaint} to repaint that area properly.

The window system doesn't attempt to figure out which pixels you actually updated, so it's your responsibility to completely repaint the area which you were supposed to repaint. Also, it is important to remember that when you get \texttt{msgWinRepaint}, your window's dirty pixels are not white or anything but pure garbage. You must completely paint the damaged area when responding to \texttt{msgWinRepaint}.

\textbf{Nesting}

Both repainting dirty regions and painting new pixels are designed to nest properly. Every \texttt{msgWinBeginPaint} must be matched with a corresponding \texttt{msgWinEndPaint}, and \texttt{msgWinBeginRepaint} must similarly be matched with \texttt{msgWinEndRepaint}. Only the outer-level \texttt{msgWinBeginPaint/msgWinBeginRepaint} chooses the region (dirty or visible) to update—it doesn't change with nested paints and repaints.

\textbf{Avoiding Repaints}

Transient windows can turn on their \texttt{wsSaveUnder} flag to avoid dirtying the windows they appear over. See the explanation of \texttt{wsSaveUnder} elsewhere.

\textbf{Stages in Optimizing Painting}

In summary, here are the stages you should follow in writing your painting/repainting code.

1. Write code which repaints everything in response to \texttt{msgWinRepaint}. To paint stuff, mark your entire window as dirty by sending \texttt{msgWinDirtyRect} to self with argument \texttt{pNull}.

2. To speed up painting, figure out what area of your window needs to be painted, and only mark that area as dirty with \texttt{msgWinDirtyRect}. If your window is an observer of a separate data object (the data/view model), write code to translate from notification of a change in the viewed object to marking part of the window dirty.

3. To speed up repainting and painting, add code to your repaint routine which tries to reduce repaint operations to the area which is in the dirty rectangle. Remember, even without this optimization, the graphics subsystem will only draw in the area in the update region, but it may increase repaint speed.

4. For areas of your application (if any) which need snappier or sequential painting, add code to do explicit painting (\texttt{msgBeginPaint, msgEndPaint}).

5. Design your data structures so that when repainting you can quickly determine from the dirty rectangle what you need to redraw, and that when you want to paint you can quickly figure out what rectangle to mark as dirty.
Design your application carefully to figure out how best to trade off between code size and speed of repainting. You want your window to draw quickly so that page turns and window manipulations are snappy, but you don’t want to expend a lot of code space or memory on it.

**The wsSynchRepaint Flag**

Repaint messages always appear at the end of the message queue of the window’s task. This means that when a window moves or disappears, other windows will ordinarily not repair themselves even though you may want them to. Consider a scrollwin (a window which scrolls another larger window inside itself) and its client window (the window inside the scrollwin). As the user scrolls the scrollwin by dragging its drag handle or thumb:

- The scrollwin moves its client window.
- The window system marks as dirty the parts of the client window which become exposed.

The client window will find out that it is dirty when it receives msgWinRepaint. But while the user drags the scroll thumb, it receives input, so ordinarily other windows don’t receive their messages. Thus, the window inside the scrollwin may not repaint until the user finishes scrolling.

You might want the client window to repaint while the user drags the scroll thumb. If you set the wsSynchRepaint flag of a window, then if it needs repainting it will be repainted synchronously. The window system sends it msgWinRepaint using ObjectCall instead of ObjectSend. In the scroll thumb example, if you set the wsSynchRepaint flag of the client window, then if it is in the same subtask as the scrollwin, every motion of the scroll thumb will result in a call to the scrollbar to repaint.

Although wsSynchRepaint ensures smooth repainting of other windows in the same process, it is no smarter than an ordinary repaint, since it results in the same msgWinRepaint. In the scrollwin example, it is possible that the underlying window called to repaint because of scrolling has other dirty areas which require repaint; their repainting might delay repainting so much that interactive response suffers. There is a tradeoff between smooth repainting versus jerky tracking. One answer would be to make the area which requires synchronous repainting a separate window from areas which take a long time to repaint.

What Really Goes on When Repainting

The window system and the Class Manager message system conspire to deliver window repaints. The effect is that:

- msgWinRepaint is always the last message in a subtask’s queue of Inter-Task Communication Messages. Thus objects receive all other pending messages before receiving msgWinRepaint.
- There is only ever one msgWinRepaint in a subtask’s message queue. Only windows owned by the current subtask are painted.
• The order in which a subtask's windows receive `msgWinRepaint` is as follows:
  • Windows damaged as a result of resizes, movements, insertions, and extractions in the same subtask which have the `wsSynchRepaint` flag set are painted synchronously, outside the normal message queue.
  • Otherwise, windows in the same subtask receive `msgWinRepaint` in back-to-front order.

• The order in which windows owned by different subtasks receive `msgWinRepaint` is undefined—if moving a window exposes underlying windows in two different subtasks, there's no way of knowing which will receive `msgWinRepaint` first.

### Copying Pixels in Windows

You can use `msgWinCopyRect` to copy pixels within a window. This copies all the pixels from a source rectangle to another area in the same window. You can use this message to reduce the amount of repainting you need to do. If what you want to draw is already in another part of your window, you can just copy it over instead of reissuing drawing commands. The window system uses an equivalent of this message to implement the different window grow styles. You can use it if you do your own scrolling. (It may be easier to implement scrolling using `clsScrollWin`, which scrolls by repositioning your window inside another.)

`msgWinCopyRect` takes a pointer to a `WIN_COPY_RECT` structure, in which you specify:

- `srcRect` the source rectangle in LWC.
- `xy` the lower-left corner of the destination location in LWC.
- `flags` a number of style and input attribute flags (the input flags are described in Part 5: Input and Handwriting Translation).
- `planeMask` an optional plane mask.

Normally the `flags` value is `wsCopyNormal` (a normal copy of the normal painting planes). If you set the `wsPlanePen` flag, the pen plane is copied too. If you set the `wsPlaneMask` flag, the window system copies the planes specified in the `planeMask` message argument.

### Copied Pixels and Window Damage

The other flags determine how the window is dirtied after copying pixels.

Normally when you copy part of the image in a window elsewhere, its original location needs to be filled with something else, so the window system marks the source rectangle dirty. However, this can be turned off by setting the `wsSrcNotDirty` flag.

Similarly, suppose that some of the source rectangle pixels have been damaged or are covered by another window. The window system does not copy these bad source pixels to the destination, and marks the corresponding destination locations dirty. However, this can be turned off by setting the `wsDstNotDirty` flag.
Copied Pixels and Child Windows

Further complications arise when the source rectangle has child windows in it. Normally, the window system moves the child windows along with the source pixels, unless you set the \texttt{wsChildrenStay} flag.

The operation of copying pixels in windows with child windows is complicated, and there are a few things to note about the interaction.

- The child windows are not damaged, they are moved.
- However, the move operation is not a first-class \texttt{msgWinDelta} operation, so capture geometry, layout, \texttt{msgWinDeltaOK}, and \texttt{msgWinMoved} are not involved.
- The sibling order of child windows is unchanged.
- If the child window is not completely enclosed by the \texttt{srcRect}, there may be problems. The window will not be moved, and the dirty region will probably be incorrect. It's best for you to expand the \texttt{srcRect} area to fully enclose any child windows partly within it.

Layout Messages

Applications sometimes require windows to have a particular size and (less often) location. Furthermore, when a window has many child windows, it needs to position each child window. As an example, take the main Notebook itself:
The Notebook is usually full-width, and takes over most of the screen leaving some space for the Bookshelf. It attempts to provide maximum room for a page, but also needs to leave room for the tabs on the right side of the display. Moreover, the user can set a preference to allow the Notebook to float and zoom, which changes the permitted sizing.

As this example shows, often the wants and needs of the parent window and its child windows conflict. This is one reason for the elaborate message passing involved when a window lays out its children.

The overriding principle of window interaction is that child windows must live with whatever their parent does. Parents have (rather, they can elect to have) the ultimate say in the geometry of their children, while children have the ultimate say in what they paint.
Altering Child Windows

When a window receives msgWinInsert or msgWinDelta, clsWin sends msgWinInsertOK or msgWinDeltaOK to the window's parent if the parent has the wsCaptureGeometry flag set. These OK messages include the WIN_METRICS structure passed to the child window. The parent window can return stsOK to allow the insertion/move/resize to proceed, but it can change the relevant WIN_METRICS fields such as bounds and wsPosTop/wsPosBottom too.

For example, a client sends its window msgWinInsert to insert it in the parent window. However, the child's position would be outside the visible area of its parent. The parent has wsCaptureGeometry set, so the window system sends it msgWinInsertOK. The parent realizes that the child window would be outside its boundary, so it changes the child's bounds.origin and bounds.size so that the child fits inside it, then returns stsOK. The msgWinInsert succeeds, but the child window appears at a different place than the client code specified. When msgWinInsert returns to the caller the WIN_METRICS structure has the revised metrics in it.

Finally, the child and its descendant windows receive msgWinInserted if wsSendGeometry is set in their flags. The message argument is the window that was actually inserted.

Resizing and Moving Windows

Your window receives msgWinDelta when someone wants to resize (or move) it. There are flags which control the behavior of a window on a resize.

A visible non-transparent window has an image in its pixels. When it is resized (whether expanded or shrunk), what should happen to the existing pixels? The flags wsGrowTop, wsGrowBottom, wsGrowLeft, and wsGrowRight control how visible pixels in the window are adjusted if the window expands or shrinks. You can think of these flags as controlling which sides of the window grow (or shrink) when the window changes size. The window system moves the window's contents (including repositioning child windows) as necessary.

For example, if a window has wsGrowTop and wsGrowLeft set, then when the window is expanded, the visible pixels remain in the lower right corner and the window is damaged so that new pixels will be painted along its top and left edges. When the window is shrunk, pixels vanish from its top and left sides.

This feature doesn't do much good in a layout episode (described below) because after a layout episode windows are usually made to completely repaint.

If a window has wsSendGeometry set, then after it or its parent or any ancestor window is sent msgWinDelta, the window system sends it msgWinMoved or msgWinSized depending on whether the result moved or resized the window. msgWinMoved is sent, then msgWinSized, if the window was both moved and resized. Note that msgWinMoved is sent if the window moves relative to the root window; this is regardless of whether the window moved relative to the parent, or whether the parent's parent (and so on...) moved.
The message arguments to msgWinMoved and msgWinSized are P_WIN_METRICS, as usual. clsWin sets child to be the window which was moved or sized; it is either self or some ancestor. clsWin also sets the bounds to the previous size and position of the window which actually moved in its parent's LWC. Note that wsWinMoved and wsWinSized flags in options are not set. Don’t “borrow” the WIN_METRICS passed with msgWinMoved/msgWinSized. It is used to send the same message to self’s sibling and child windows. If you write into it you will spoil the message for them.

**Window Layout**

Windows in PenPoint operating system are lightweight; you can afford to have lots of them. With windows inside windows, you must be concerned with how to position and reposition windows. There are three broad styles of window layout possible in PenPoint:

- Please be nice to me.

  The parent allows child windows to position and size themselves wherever they want. The parent may wind up covered by greedy children with little space of its own to draw on.

  This is the style used by some notebook applications, which allow themselves to be covered by child windows.

- I’ll decide what you’ll get away with.

  The parent has its wsCaptureGeometry flag set, so it gets msgWin...OK messages whenever a child window attempts to insert/extract/move/resize itself. The parent can veto outrageous requests, or returnstsWinConstrained along with suggested acceptable positions and sizes.

  This is the style used by applications with embedded applications inside them. The outside application can’t allow an embedded application to obscure all its own views, and it must assume that the embedded application may make mistakes.

- Give me your proposals.

  This is the most flexible. The parent asks the child windows to propose sizes and locations for themselves, then the parent lays the child windows out while trying to preserve the sizes and locations they asked for. This only works well if everyone who wants to adjust windows cooperates over use of msgWinLayout. Windows should not spontaneously resize themselves; instead an outside object should decide when windows need to be re-laid out and send msgWinLayout to a window at that point.

  This is the scheme used by the PenPoint User Interface Toolkit (described in Part 4: UI Toolkit). The client creates the various user interface components and then tells the parent window to lay itself and its children out.
A window subclass may veto or modify the `msgWinDelta` and `msgWinInsert` size/position messages it receives, but will work better with others if it accepts whatever size and position that its parent deltas it to.

**More on msgWinLayout**

The third technique uses `msgWinLayout`. When you want windows to lay out, send `msgWinLayout` to them. This message takes a pointer to a `WIN_METRICS` structure, but only pays attention to two fields, the `bounds` rectangle and `options`. If `wsLayoutResize` is set in `options`, the window receiving `msgWinLayout` is allowed to change its own size. If `wsLayoutResize` is not set, the window receiving `msgWinLayout` must use the `bounds.origin` and `bounds.size` in the message arguments. As an example of the latter, when the user resizes a floating window by dragging its shadow, the frame must lay out exactly in that rectangle. The shadow sends the floating window `msgWinLayout`, with `wsLayoutResize` set false, and the `bounds` are the rectangle indicated by the user.

Another flag in `options` that affects layout is `wsLayoutMinPaint`. If `wsLayoutMinPaint` is not set (the default), then the window system ensures that the window(s) will not repaint during layout. Making windows invisible during layout is faster for complicated layout changes, but means that entire windows must repaint when layout is finished.

The window system handles `msgWinLayout` by figuring out which windows need to lay out, by looking at the window flags of the windows. It sends these windows other layout messages.

The idea is that the window laying out will reposition and resize its child windows as desired. The term lay out is somewhat ambiguous here: laying out yourself involves positioning your child windows, but does not necessarily mean changing your own size or allowing your child windows to lay themselves out.

`clsWin` does not know how the child windows of a given window should be positioned, but many descendant classes do. All the windows in the UI Toolkit know how to lay themselves out, and the `clsCustomLabel` and `clsTableLayout` classes know how to position their child windows using complex relationships and rules.

Since windows may be inserted inside windows inside other windows, re-laying out one window may involve a lot of messages and computation. The Application Framework sends your application's documents `msgWinLayout` when they first appear on-screen. To see an intricate lay out in the system take place, change the system font. The implementation of window layout tries to be very smart about asking windows to lay out, and is quite complex as a result.
The Client Interface to Layout

When inserting, extracting, or moving windows, you alter the structure of a window subtree. When you are finished you should probably send `msgWinLayout` to the window you altered. Set `wsLayoutResize` if that window can change its size or specify the required `bounds` for that window.

If you modify several related windows at once, instead of sending `msgWinLayout` to each one, you can instead `dirty` the layout of each one by sending it `msgWinSetLayoutDirty` with `true` as the message argument. When you finish perturbing the window subtree, send `msgWinLayout` to the parent window. The idea is that you set the layout dirty bit when you know the layout is dirty, and later send `msgWinLayout` when you want it too look right.

Many operations, such as `msgWinDelta`, will mark a window's layout dirty automatically.

If you know that a change makes an entire subtree dirty, you can send `msgWinSetLayoutDirtyRecursive` to the parent of the subtree to dirty the layouts of every window in the subtree.

The rest of this subsection talks about what happens internally during layout, and what descendant classes must do to handle layout correctly. (The UI Toolkit classes do all of this for you.)

What Happens in Layout Processing

First the window system must figure out how much of the window tree will need to be re-laid out. It goes up the window hierarchy from the receiving window looking for a window which is not shrink-to-fit around its contents. A parent window with either of these constraints true will not cause its own parent to have to lay out, hence it becomes the root of the layout subtree.

A Layout Episode

What happens next is the window system goes into a layout episode involving every window in the layout subtree. It looks at each window in the layout subtree and if necessary sends each window either `msgWinLayoutSelf` or `msgWinGetDesiredSize`. As a result of one window figuring out how to lay out or choose its size, other windows may be perturbed and will require laying out. A window may even have to re-lay out several times. The layout episode ends when every window in the layout subtree has settled on a size and position, at which point the window system actually modifies those windows whose desired size and position are different from before the layout episode.

If you implement a descendant of `clsWin`, `msgWinLayoutSelf` and `msgWinGetDesiredSize` are the two messages you should respond to in order to lay out nicely. You need not normally respond to `msgWinLayout` itself.
Laying Out Self

msgWinLayoutSelf is similar to msgWinLayout, but window clients do not send it; instead the window system sends it to windows during layout. msgWinLayoutSelf uses WIN_METRICS the same way: if wsLayoutResize is set, you can change the window's size; if not, you must use the size in bounds.size. Don’t send the window msgWinDelta to change size; instead, pass back the desired size in bounds.size to whomever sent you msgWinLayoutSelf.

If your window has child windows, it should position them. For example, your class might tile windows, or allow them to overlap. To position the child windows, send them msgWinDelta. Because the entire window layout subtree is in a layout episode, clsWin intercepts the msgWinDelta and caches the size and position.

Some window classes ask child windows what size they would like to be. For example, a window class which lays out child windows in a single row might fix the height of the child windows, but be flexible as to the width of each child window according to how wide the child would like to be. If your window class cares about the size of child windows, send them msgWinGetDesiredSize. This too takes a WIN_METRICS structure. The child should pass back its desired size in bounds.size.

Getting Self’s Desired Size

Your class should respond to msgWinGetDesiredSize by figuring out what size the window should be.

Shrink-to-Fit

clsWin responds to msgWinGetDesiredSize by checking self’s wsShrinkWrapWidth and wsShrinkWrapHeight flags. If these aren’t set, then it assumes that the window is not going to return a desired size, so it passes back the window’s current bounds.size. For this reason, it’s important to be accurate about your windows’ shrink-to-fit flags: if your class’ windows ever change size in response to msgWinGetDesiredSize, set one or both of these flags.

If one or the other of the shrink-to-fit flags is set, then clsWin sends self msgWinLayoutSelf, allowing it to change size.

Caching Desired Sizes

The window system usually caches desired window sizes during a layout episode, so that it doesn’t have to repeatedly send msgWinLayoutSelf to windows to get their desired sizes. If your window’s desired size is dynamic, your class should set the wsLayoutNoCache flag in the options field of WIN_METRICS in response to msgWinLayoutSelf to turn off this caching.
Baseline Alignment
The UI Toolkit's layout classes allow clients to ask that child windows be aligned on their baselines. This may not be the same as the bottom of the window. For example, the baseline of a boxed text label in a large font is several pixels away from the bottom. Hence sophisticated parent windows may send msgWinGetBaseline to their child windows. This takes WIN_METRICS; classes should set the bounds.origin to reflect the child's desired position.

The baseline also applies in the vertical direction. For example, in option sheets choices and toggle tables should align vertically on their vertical lines, not on the left-most area where the checkmark appears. Hence bounds.origin.y is also significant.

clsWin sets both bounds.origin.x and bounds.origin.y to 0.

Layout and Geometry Capture
At the end of the layout episode, the window system actually moves those windows which ended up with a different position and location. If a window is capturing geometry and changes bounds in response to msgWinDeltaOK, this may result in altering the window positions. The window system detects this and goes back into the layout episode.

Since layout assumes that windows are cooperating, it rarely makes sense to turn on wsCaptureGeometry or wsSendGeometry on while using msgWinLayout.

Window Management Messages
classWin defines a variety of messages to facilitate the effective use of windows in an application.

Sending Messages to a Window Hierarchy
Sometimes you know only that some ancestor window will respond to a message. You can send a message and its message arguments to a window using msgWinSend. If that window's classes do not respond to the message, the message arrives at classWin. classWin responds to msgWinSend by forwarding the msgWinSend message onto self's parent window using ObjectSendUpdate. If the message reaches the root window, then clsWin returns stsMessageIgnored.

msgWinSend takes a pointer to a WIN_SEND structure. In this you specify:

lenSend the size of the data to send.
flags some options to the way the message is sent (wsSendIntraProcess is currently the only flag used).
msg the message you're packaging inside the msgWinSend.

data an array containing the data for the message.
If the wsSendIntraProcess flag is set, then the message will not be propagated between processes (based on the owner of the window object) and clsWin returns stsMessageIgnored.

lenSend must be at least SizeOf(WIN_SEND), but may be larger to move more data to a window owned by another process. You can move 32 bits of data by storing it in data[0]. If you have more data, declare a larger structure with WIN_SEND as its first field.

Note that msgWinSend goes up the window hierarchy, whereas msgWinEnum and msgWinFindTag go down the hierarchy.

### Sorting Windows

You can sort a window's child windows, thereby reordering their placement in front of or behind each other, by sending msgWinSort. This is used by the UI Toolkit to sort tabs in the tab bar. clsWin doesn't know how to sort the windows: you must supply a callback routine which can compare two windows.

msgWinSort takes a pointer to a WIN_SORT structure for its message arguments. In this you specify:
- The callback routine which compares windows (pSortProc).
- A parameter which clsWin passes to your callback routine (pClientData).

clsWin passes your comparison routine (pSortProc) two windows at a time, together with whatever pClientData you specified. Your comparison routine must match the function prototype type P_WIN_SORT_PROC:

```c
typedef S16 (FAR * PASCAL P_WIN_SORT_PROC)(WIN winA, WIN winB, P_UNKNOWN pClientData);
```

It should return -1 if winA comes before winB, +1 if winB comes before winA, and 0 if winA compares the same as winB.

Typically your comparison routine will look at some feature of the two windows to compare them, such as the window tags, or the strings in label windows. You can pass arbitrary instructions to the routine in pClientData, such as instructions to sort alphabetically, in descending order, etc.

### Debugging Windows

You can send msgDump to a window to get a dump of its metrics. If you send msgWinDumpTree to a window, it dumps a dense subset of the information about the window and all its children, recursively. If you set the /DW0002 debug flag, the input flags for the windows are printed along with the window style flags.

Both of these messages only work if you have loaded the debugging version of WIN.DLL, the PenPoint windowing dynamic link library.
Window Management Messages

Filing Windows

When a window receives msgSave, it files its state. One vital piece of state information is all the child windows inside that window. If a child window has wsSendFile set, then it is filed as part of its parent filing. clsWin sets wsSendFile by default; some descendants of clsWin clear this style flag, but it is still true that filing one window may result in many child windows being filed.

Details of Window Filing

Windows file their position and size (bounds in window metrics), unless wsFileNoBounds is set. When a parent window is restored, if its new size is the same as its old size, it doesn’t need to be sent msgWinLayout, and consequently all its child windows do not have to take part in a layout episode either.

Window layout is usually affected by the orientation and pixel size of the windowing device (windows are restored onto a different windowing device when printed). Window layout is also affected by the default system font and system font scale.

The window system saves this window environment information for the root of a tree of windows so that when the windows are restored it can decide what needs to change. The window system uses the root and resId fields in OBJ_SAVE to figure out whether it’s saving the root or not. During msgRestore processing, the same root and resId fields in OBJ_RESTORE tell the window system whether this window filed the environment information or not. If you’re sending msgSave or msgRestore yourself, set root to objNull, and resId to the resource ID for the object being saved or restored.

(The window system provides a message to get the current window environment, msgWinGetEnv.)

If wsFileInLine is set in a child window’s style flags, the child window is not filed with an object header. Although this conserves storage space, this prevents someone using a resource file to read in the child window independently.

Re-Layout of Filed Windows

If wsFileNoBounds is set in a window’s style flags, then the window’s position and size are not saved. This means that on restore, the window will need re-layout, so clsWin sets wsLayoutDirty for the window upon restore.

You can also force a window to have dirty layout (wsLayoutDirty) when it is restored by setting wsFileLayoutDirty.

However, if neither wsLayoutDirty or wsFileLayoutDirty is true when the window is filed, the window is restored with wsLayoutDirty set to false, so it doesn’t need to be laid out.
Summary

clsWin provides basic window behavior. Though it isn't practically useful by itself, it provides a common foundation for useful subclasses such as the User Interface Toolkit classes (described in Part 4: UI Toolkit). The primary purpose of windows is to serve as a canvas for drawing contexts, described in Chapter 26, The Drawing Context Class.

Window metrics, encoded in the WIN_METRICS structure, define what makes each instance of clsWin unique. The WIN_METRICS structure is the common argument for most messages clsWin defines, although most of the messages pay attention to only a few fields of the structure.

The window style flags, part of the window metrics, define the display style of a window. Windows also include input flags, but only to support clsWin subclasses that handle input events.

clsWin defines a variety of messages for setting and getting window metrics, handling painting in the window, implementing a protocol for laying out child windows, and managing windows within an application.
Chapter 24 / Window Device Classes

Every window is associated with a windowing device. The one you are most familiar with is the Screen. However, there are other kinds of windowing devices, including printers. In particular, there are image devices which you create to draw in off-screen windows.

This chapter describes windowing devices and then image devices. Because the Screen and printer windowing devices are managed by the window system, you rarely interact directly with clsWinDev. Image devices, window devices in memory without an associated display, are more common. clsImgDev inherits from clsWinDev and you use several clsWinDev messages with image devices.

Windowing Devices

When you create a window, the window system must know what windowing device it is intended for, so at window creation you either specify a device for the new window or a parent window from which the window system can figure out the target windowing device. The latter is much more common, since you usually nest windows in other windows.

Creation

Application code should rarely need to create a windowing device. The window system creates a windowing device, the Screen, during initialization, and creates a transient windowing device on the printer during printing.

Root Window

When it creates a windowing device, clsWinDev creates a root window (an instance of clsWin) on that device.

The message msgWinDevGetRootWindow passes back the UID of the root window on a windowing device. The root window on the Screen is another well-known object, called the Root Window.
Image Devices

For drawing to appear on the screen or printer, it must take place in a window. Often this is all that applications need: they draw what they want to appear in their windows, and the window system colors in the visible pixels of these windows on the screen or printer.

Unlike some other window systems, there is no backing store in PenPoint™ operating system for windows. If a screen window is not on-screen and you draw on it, the pixels aren’t saved somewhere.

In some situations it is useful to create an image off-screen
- Drawing a complex image over and over.
- Drawing an image through a mask or stencil.
- Storing window contents to speed repainting.
- Creating bitmap images such as icons in advance.

To speed these operations, you can use an image device. clsImgDev inherits from clsWinDev. An image device is a windowing device where the image memory is under the control of the client. All the things you can do on the screen you can do on an off-screen image device: create trees of windows, bind a DC to a window, draw in a window, etc. The differences with the screen or a printer are that you must create an imaging device yourself, and that you can access the memory in which the window system sets pixel values (the pixmap).

The contents of windows on an image device can be copied to a window on another device using pixel copy operations defined by clsSysDrwCtx. Unlike a windowing device, an image device is not designed to be shared, so no automatic locking takes place when drawing into one of its windows.

Creating an Image Device

Sending msgNew to clsImgDev creates an image device and automatically creates a root window on it.

clsImgDev has no msgNew arguments of its own. In the WIN_DEV_NEW_ONLY structure of its ancestor clsWinDev, you specify the number of windows on the device (winDev.initialWindows). The default is 100. This is a soft limit: if you create more windows on the device, the window system will silently allocate more memory. However, if you know that you will be creating, say, exactly one window on your image device, set this to 1.

Accessing Image Devices

Since you draw using a DC bound to a window, the UID of the image device isn’t very useful. Instead of referring to an image device by its UID, you can send it msgWinDevGetRootWindow to retrieve its root window, and then send windowing device messages to its root window. When this root window receives msgDestroy, it will destroy the image device along with itself.
You can also send windowing device messages to a DC bound to a window on the image device.

**Binding an Image Device to a Pixelmap**

At creation, the image device has no notion of the size or depth of its pixels, or of the size of its pixelmap. Also, it has no memory associated with it to store the pixelmap; attempts to draw on it will fail, because there's nowhere to hold the pixel values. You specify all this information by sending `msgWinDevBindPixelmap`. This takes a `WIN_DEV_PIXELMAP` structure, which includes:

- **device** the device with which the image device should be compatible.
- **size** the width and height to allocate for the image device.
- **planeCount** the number of planes to allocate for the image device.
- **pPlanes** a pointer to memory for the planes in the image device.

As with all windowing device messages, you can send `msgWinDevBindPixelmap` to a window on the device. Moreover, you can send this message to a drawing context bound to a window on the image device, in which case `size` is interpreted in LUC—the units and scale of the DC.

**Pixel and Pixelmap Information**

You tell the image device what kind of pixels it has by specifying in `device` a target device with the same kind of pixels. For example, if you're going to use the image device to store images copied from the `Screen`, or if images drawn into the image device will be copied to the `Screen`, the image device will be trading images with the `Screen`.

Instead of specifying a windowing device to be compatible with in `device`, you can specify a window UID. `clslmgDev` figures out the window's device. Thus, you could specify the `RootWindow` instead of the `Screen` for `device`.

Also, you need to specify the width, height and depth of the pixelmap that is to be allocated or provided by the application. You can specify the size in LWC, or you can specify it in LUC by sending the message to a DC bound to a window on the image device. The DC figures out how many pixels on the target device correspond to the size you want.

When you send `msgWinDevBindPixelmap`, `clslmgDev` does several things:

- It sets up the pixelmap with pixels like those of the specified `device`.
- If you sent `msgWinDevBindPixelmap` to a DC, it figures out the number of pixels needed by translating `size` into the right number of pixels for `device`.
- It resizes the root window of the image device to conform to the dimensions of the pixelmap; it and any other windows will be marked dirty.
- It allocates the number of planes specified in `planeCount`. 
If you set planeCount to zero, clsImgDev creates a number of planes appropriate for the device the image device is to trade with.

Accessing Memory
An image device has no pixelmap at first. You use msgWinDevBindPixelmap to associate the image device with memory for its pixels’ values. There are two ways to manage the memory for the pixels.

The first is to let the device itself allocate and hold the memory needed. With this method you can’t directly access the memory. To get the image device to allocate pixelmap memory, set pPlanes to pNull in the message arguments to msgWinDevBindPixelmap. clsImgDev allocates enough memory for the number of planes and number of pixels.

The second way is for you to allocate the memory yourself and pass a pointer to it to the image device. This method is described in the next section.

Allocating Your Own Pixelmap
If you want to allocate the pixelmap memory yourself, then the first step is to figure out how many bytes each plane requires. Instead of computing this yourself, it’s better to let the window system compute the size. To do so, you send msgWinDevSizePixelmap to the image device. This takes a WIN_DEV_PIXELMAP structure, as does msgWinDevBindPixelmap. You specify the same parameters for target device and size. msgWinDevSizePixelmap passes back in planeSize the number of bytes of memory for each plane, and in planeCount it passes back the number of planes to allocate to match the plane count of the target device.

You should then allocate planeSize bytes of memory for each plane. To bind the image device to the pixelmap, send it msgWinDevBindPixelmap. In this case, you should make pPlanes point to an array of pointers; one for each plane specified in planeCount. You don’t have to allocate a pixelmap for each plane in the target device; if you’re using the image device to create stencils or black and white images, you would allocate one plane, and set planeCount to 1.

Example 24-1
Allocating a Small Bitmap

This example establishes a one-bit deep plane on the stack in the buffer plane0.

```c
U8 plane0[200];
P_UNKNOWN ppp[1];

ppp[0] = plane0;
pm.device = id;
pm.size.w = 72;
pm.size.h = 72;
pm.planeCount = 1;
pm.pPlanes = ppp;
ObjectCall(msgWinDevBindPixelmap,aDc,&pm);
```
Dirty Windows

After binding an image device to a pixmap, the root window of its image device is dirty. Also, windows on image devices can overlap, be resized, inserted, and extracted, all of which cause the windows to be dirtied.

However, windows on image devices do not automatically receive msgWinRepaint from the window system as do windows on other types of devices. To make them receive msgWinRepaint, you must send them msgWinUpdate.

The root window on an image device paints itself white in response to msgWinRepaint.

Drawing on an Image Device

You can just paint directly on the root window; or you can create one or more windows as children of it and build up a window tree in the usual manner.

Accessing Pixels in an Image Window

Having created an image in a window on an image device, you’ll probably want to use it with another window at some point.

Remember that to draw correctly on the destination window, it must be within a msgWinBeginPaint/msgWinEndPaint or msgWinBeginRepaint/msgWinEndRepaint pair.

Getting and Setting Pixel Values

msgDcGetPixel gets the value of a pixel, msgDcSetPixel sets its value. Both take a pointer to a SYSDC_PIXEL structure for their message arguments, which includes:

- rgb whether the value should be an RGB value or a hardware-dependent palette color.
- color the color of the pixel.
- xy the location of the pixel, in LUC.

When you send msgDcSetPixel, you specify color; when you send msgDcGetPixel, the current value of the pixel is passed back in color. If rgb is set, color is interpreted as an RGB value, otherwise it is a SYSDC_COLOR (a hardware color palette index).

Copying

To copy the pixels in one window to another, send msgDcCopyPixels to a DC bound to the destination window. This takes a SYSDC_PIXELS structure for its message arguments in which you specify:

- srcWindow the source window on the image device.
- pBounds a pointer to the bounds of the area to copy in the source window.
- xy the location in the destination window.
- dstDirty whether pixels in the destination window should be marked dirty if the corresponding source pixels are dirty.
msgDcCopyPixels copies the rectangle of pixels in the source window at pBounds to the destination at location xy. If you specify a pBounds of pNull, then the entire image window is copied to the destination.

It's possible for windows in an image window to overlap, or be clipped by their parent. This may result in some or all of the pixels in the source rectangle not being part of the source window. If dstDirty is set, then the pixels in the destination window that correspond to dirty pixels in srcWindow are marked dirty. The destination window will then get a msgWinRepaint and can paint the dirty pixels some other way. This allows an image device to capture an image from theScreen where some of the pixels may be dirty or covered up by other windows. When those pixels are copied, the copies are marked dirty.

Remember that image windows are created in a dirty state. Thus, if you never cleaned up the damage and set dstDirty, the destination window will be marked dirty. You can clear the dirty state of an image window using msgWinUpdate or msgWinCleanRect.

Note that the dirty pixels are copied first and are then marked dirty. The source pixels are copied to the destination regardless, whether dirty or not.

Landscape and portrait mode rotation is handled correctly; however, axis flipping and rotation in the DC LUC space will work less intuitively than one might hope.

When pixels are copied from the image device to a visible window device (such as theScreen) using msgDcCopyPixels, the meaning of the pixels is determined by the visible device. For example, on a two-plane device, black is 00, and white is binary 11. If you draw with RGB colors you should not have any surprises copying images to other devices.

### Stenciling

You can also use a one-plane image device as a stencil through which the DC renders the current foreground and background colors. To do this, send msgDcDrawPixels to the DC of the destination window. This takes the same SYSDC_PIXELS message arguments as msgDcCopyPixels. The difference is that instead of copying the pixels in the source to the destination, instead the pixels in the source determine which pixels in the destination get the foreground and background colors.

When using an image device as a stencil, it is important for the stencil to have a 1 bit where foreground color is to be applied and a 0 bit where background color is applied (the background color can be transparent). To ensure this, it's easiest to set the colors with msgDcSetForegroundColor and msgDcSetBackgroundColor.

Usually you set RGB colors to be device-independent, but here the meaning of the stencil's pixels are independent of its device. You would do it as follows (theDc is the DC bound to the stencil window on the image device):

```c
ObjectCall(msgDcSetForegroundColor, theDc, (P_ARGS)1);
ObjectCall(msgDcSetBackgroundColor, theDc, (P_ARGS)0);
```
msgDcDrawPixels does not pay attention to the dstDirty flag. When drawing images with a stencil, the window system assumes that the stencil is completely constructed by the application and is not a snapshot of the screen.

**Example 24-2**

**More Complex Use of an Image Device**

The following code creates an image device, allocates memory for it, draws in it, then copies its pixels to another window.

```c
IMG_DEV_NEW idn;
OBJECT id;
ObjectCall(msgNewDefaults, clsImgDev, &idn);
ObjectCall(msgNew, clsImgDev, &idn);
id = idn.object.uid;
ObjectCall(msgWinDevGetRootWindow, id, &id);
```

Notice that the `msgWinDevGetRootWindow` line is trading in the UID of the image device for the UID of the image device's root window. From now on your code can forget about the image device itself, and treat it as a window.

Also, it is necessary to specify the width, height and depth of the pixelmap that is to be allocated or provided by the application. As usual we want to be able to send these messages to a DC to obtain coordinate transformation into device units; so the first thing we need to do is bind a DC to the window with which the image device will be trading images.

```c
WIN SYSDC aScreenWindow
aDc;
// create the DC...
ObjectCall(msgDcSetWindow, aDc, (P_ARGS)aScreenWindow);
```

To get the image device to allocate pixel map memory, we fill out the parameters to `msgWinDevBindPixelmap` like this:

```c
WIN_DEV_PIXELMAP pm;
pm.device = id;  // really "image device" root window
pm.size.w = 72; // 72 points
pm.size.h = 72; // 72 points
pm.planeCount = 0; // use default #
pm.pPlanes = pNull; // I'm not providing memory
ObjectCall(msgWinDevBindPixelmap, aDc, &pm);
```

Note that we are sending this message to `aDc`, which is bound to `aScreenWindow`, so the units are in LUC (points, by default), not device units.

If we want to allocate the memory then the first step will be to find out how many bytes each plane must be. The following message:

```c
ObjectCall(msgWinDevSizePixelmap, aDc, &pm);
```

(with the same parameters in `pm` as before) will pass back in `pm.planeSize` the number of bytes of memory each plane should be for a 72x72 image; and it will return in `pm.planeCount` the number of planes to allocate.

Assume we simply want to draw on the root window of our device and then copy the image to `aScreenWindow`. We can paint on the window returned by `msgWinDevGetRootWindow` in the usual manner:

```c
RECT32 r;
WIN win;
ObjectCall(msgDcSetWindow, aDc, (P_ARGS)id);
ObjectCall(msgWinBeginPaint, aDc, &r);
ObjectCall(msgDcFillWindow, aDc, pNull);
ObjectCall(msgDcDrawEllipse, aDc, &r);
ObjectCall(msgWinEndPaint, aDc, pNull);
```
To copy the image in id to aScreenWindow use msgDcCopyPixels as follows:

```c
SYS_DC_PIXELS cp;
cp.srcWindow = id;
// copy from window "id"
cp.pBounds = pNull; // copy entire window
cp.xy.x = 72; // copy to 72,72 in the
cp.xy.y = 72; // destination window (aScreenWindow)
cp.dstDirty = FALSE; // see discussion
ObjectCall(msgDcSetWindow,aDc, (P_ARGS)aScreenWindow);
ObjectCall(msgWinBeginPaint,aDc,pNull);
ObjectCall(msgDcCopyPixels,aDc,&cp);
ObjectCall(msgWinEndPaint,aDc,pNull);
```

The lower left hand corner of id will be copied to 72,72 in aScreenWindow.

---

### Multiple Pixelmaps

After an image device has been created you can send it `msgWinDevBindPixmap` many times to create new pixelmaps, or to bind to old ones. If the window system allocated the memory for the image device, it will free it. Thus, a single image device can be used to handle a larger collection of pixelmap data.

### Destruction

When you are finished with the image device itself, destroy it by sending it `msgDestroy`; this frees it and any pixelmap it was holding at the time (unless you supplied the memory). You can send `msgDestroy` to the root window on the image device and it will destroy the image device as well.

### Landscape and Portrait Mode

Image devices handle landscape and portrait modes correctly; the pixelmaps are inherently device dependent and clsImgDev adjusts their notions of width and height accordingly. However, it is your responsibility to flush and rebuild pixelmaps when the user switches the orientation of the `Screen`. You can receive notifications of changes in screen orientation by observing the well-known object `theSystemPreferences`. 

---

Example 24-2 (continued)
Performance Tips

If you use bitmap images repeatedly in your application, there are a variety of ways to display them:

- Use `msgDcDrawImage` as necessary to draw the bitmaps.
- Create one image device, and draw the various bitmaps in its root window so that they don't overlap, and copy the right pixels by specifying the right `pBounds` in `msgDcCopyPixels`.
- Create one image device, create several child windows on it of the right sizes for the bitmaps, and use `msgDcCopyPixels` with a `pBounds` of `pNull`.
- Create one image device, allocate memory for each bitmap, and bind the image device to each pixmap as needed.

All of these are appropriate in different situations.
Chapter 25 / Graphics Concepts

This chapter provides a detailed description of the PenPoint™ operating system standard imaging model, ImagePoint™, as well as a conceptual overview of its implementation as clsSysDrwCtx (the system drawing context class). You'll find a more detailed description of graphics-related classes in the remaining chapters of this part.

If you plan to use the standard User Interface Toolkit classes (described in Part 4: UI Toolkit) to build your user interface, you may be able to skip the remaining chapters in this part and go directly to Part 4. You may nevertheless find the information in this chapter useful, since it provides a framework for understanding the graphics behavior the UI Toolkit classes provide.

Topics covered in this chapter include:
- The relationship between an imaging model and a drawing context.
- The ImagePoint imaging model as PenPoint’s system drawing context.
- Coordinate systems and coordinate transformations.
- ImagePoint drawing operations.
- ImagePoint support for text and color.
- Special features of clsPicSeg, a subclass of clsSysDrwCtx.

Models and Implementation

Every computing system that displays graphics has some kind of underlying imaging model. Imaging models provide an abstract method for describing images. Computing systems must implement a programming interface based on the imaging model. PenPoint’s system imaging model is called ImagePoint. The programming interface based on ImagePoint is the drawing context.

Imaging Models

An imaging model is a set of concepts and assumptions that supports the description of visual images. Figure 25-1 shows a simple two-dimensional imaging model involving line segments in an x-y coordinate system. The coordinate system is defined by an arbitrary point called the origin and an arbitrary measure of distance called the unit. It is possible to describe any point in the coordinate system by specifying its unit distance from the origin in the x (horizontal) and y (vertical) directions. For example, the coordinate pair (2, 3) specifies a point two units to the right of the origin and three units above the origin. Two such points describe a line segment, and you can combine several such line segments into geometric forms.
PenPoint's system imaging model uses an x-y coordinate system as in the above example, but supports more powerful features such as coordinate transformations, filled and unfilled geometric figures, text, and color.

**Drawing Contexts**

In PenPoint, graphics appear in windows, rectangular areas of an output device. PenPoint represents the output device internally as an instance of `clsWinDev` (window device), which inherits from `clsPixDev` (pixel device). Each window is an instance of a descendant of `clsWin`. Windows and window devices provide the canvas for application graphics operations, but their graphic ability is limited to defining the pixels a drawing operation can affect. The ability to render an image is separately defined in descendants of `clsDrwCtx` (drawing context).

A **drawing context** (DC) is an implementation of an abstract imaging model. A DC includes a set of **drawing operations** (such as an operation to draw a rectangle) together with a **graphic state** (a set of attributes such as line thickness). The graphic state determines the way in which the DC executes its drawing operations. For example, with the line thickness attribute set to 0.1 centimeters, the DC draws lines 0.1 centimeters thick. If an application changes the line thickness attribute to 0.25 centimeters, the DC draws subsequent lines 0.25 centimeters thick.

To draw an image in a window, your application must **bind** a drawing context and a window together. A drawing context renders images in the window to which it is bound. Binding isn't permanent; your application can share a single drawing context between many windows by binding the drawing context to one window after another.
The System Drawing Context

Because a drawing context based on ImagePoint is the standard for PenPoint systems, an ImagePoint drawing context class is called a system drawing context. A system drawing context is an instance of clsSysDrwCtx, which inherits from clsDrwCtx. The drawing context class, clsDrwCtx, is an abstract class. It defines the basic ability of binding to a target window, but does not provide any graphics ability. clsSysDrwCtx adds the ability to draw images, according to the underlying ImagePoint imaging model, in the bound window. Because a system drawing context is the standard type of drawing context in PenPoint systems, it is often referred to simply as a drawing context.

Coordinate Systems

The ImagePoint imaging model includes the concept of an x-y coordinate system for specifying location in the drawing space. There are two coordinate systems which you must understand:

- **Logical unit coordinates** (LUC) are the coordinates of the abstract ImagePoint imaging model. Drawing contexts use LUC for almost all drawing operations.

- **Logical window coordinates** (LWC) are measured in pixels relative to the lower left corner of the window. The window system uses LWC to describe the size of a window.

Drawing contexts use logical unit coordinates (LUC). The window system uses logical window coordinates (LWC). These are the coordinate systems you need to understand to understand the material in this chapter.

Logical Unit Coordinates

The LUC system holds a special place because it is the abstract coordinate system of the ImagePoint imaging model. LUC's abstract nature keeps it free of dependency on pixel size, allowing flexibility and convenience for use with drawing contexts.

The origin of the logical unit coordinate system initially maps to the origin of the logical window coordinates (the lower left corner of the window). The unit measurement of LUC starts out as one typographer's point (1/72 of an inch). Applications use logical unit coordinates when sending messages to drawing contexts. Drawing contexts can receive all clsWin messages, and convert from LUC to LWC as needed.
Coordinate System Transformations

Because logical unit coordinates are the abstract coordinates of the ImagePoint imaging model, you are free to transform them in any way that is convenient for your application. Transformations include scaling the coordinate units (changing their size), translating (moving) the origin of the coordinate system, and rotating the entire coordinate system about its origin. For example, as shown in Figure 25-2, you could change the LUC system to measure x (horizontal) units in inches and y (vertical) units in millimeters, relative to an origin one centimeter to the right of and above the lower left corner of the window, and rotated to a 45-degree angle from horizontal.

Figure 25-2
Transforming the LUC

Drawing contexts support on-the-fly transformation from LUC to LWC. As a convenience, clsSysDrwCtx implements all of the clsWin messages. You can send a clsWin message to a drawing context with coordinates specified in LUC; the DC will transform the coordinates into LWC for its bound window, then send the message to the window. This allows you to implement all windows and graphics code with a single, flexible coordinate system (LUC) that you customize to be as convenient as possible for your application.

Coordinate Unit Size

To improve performance, drawing contexts use integral coordinates. In other words, ImagePoint does not support fractional coordinates. Although this improves graphics performance, it may be difficult to create effective images if your LUC unit is too large. For example, if your LUC unit is one inch (2.54 cm), you cannot specify locations that do not fall on one inch increments from the origin. This can result in unattractive displays, especially when drawing text.

As a result of this implementation, it is important to keep LUC units sufficiently small. Typically, this means keeping the units smaller than the smallest graphic element your window will contain. By default, a new DC has a unit size of one typographer's point (1/72 of an inch), which is small enough for most applications. With this unit size, a rectangle 72 units high by 144 units wide is
one inch by two inches on the display. If points are not convenient for your application, drawing contexts supports a variety of convenient unit sizes in addition to its arbitrary scaling support.

**Coordinate Rounding Error**

The advantage of logical units is device-independent displays. With a unit size of one point, 72 units is one inch, independent of whether the display device offers 72 pixels per inch or 1200 pixels per inch. The trade-off is that, because of rounding error, you cannot specify an exact size or screen location unless you set the unit size equal to the device units. As a rule, the rounding error is never larger than two pixels (usually one pixel or less). Nevertheless, because window sizes and positions are specified in device units, window edges always fall exactly on pixel boundaries. Unless you give up device independence by setting the DC unit to pixels, it is difficult to specify an image that is aligned exactly with the edge of a window.

Rounding error also can cause an image to appear differently depending on whether its location rounds up to the next pixel or down to the previous pixel. One solution to this problem, if it occurs, is use origin translation or some other method to snap the starting coordinate to the pixel nearest the intended location. This guarantees consistent rounding no matter where the image is displayed.

If pixel alignment is absolutely critical to your application, you can use device units as your DC unit. The cost is a complete loss of device independence. For example, on a 72 pixel per inch display, 72 device units is one inch. On a 300 pixel per inch display, the same 72 units spans slightly less than a quarter of an inch. So, unless your application demands exact pixel alignment, you should use an LUC unit size that is not tied to device unit size.

**Drawing Context Features**

The remainder of this chapter describes features of ImagePoint as implemented in clsSysDrwCtx, as well as the additional features of clsPicSeg (picture segment), a subclass of clsSysDrwCtx. Drawing context features include:

- **Local clipping** to clip images outside a specified rectangle within the window.
- **Hit detection** to determine whether a specified rectangle intersects with a series of drawing operations.
- **Bounds accumulation** to determine the bounding rectangle of a series of drawing operations.
- **Figure drawing** operations to create open figures (such as lines, arcs, and Bezier curves) and closed figures (such as rectangles, ellipses, and arbitrary polygons).
- **Sampled image** operations to display pixelmaps.
- RGB and palette **color** support.
Text operations supporting **scalable fonts**.

- A structured **graphic state** that can be stored and retrieved (a single DC can maintain a collection of useful graphic states).

**clsPicSeg**, which inherits from **clsDrwCtx**, adds the ability to record a series of drawing operations. In addition, picture segments support splines (smoothly connected sequences of Bezier curves) and the inclusion of arbitrary objects in images.

### Local Clipping

In addition to the clipping regions the window system maintains for each window, you can define a **local clipping rectangle** for each drawing context. A local clipping rectangle effectively reduces the clipping region to the intersection of the local clipping rectangle and the standard clipping region. As you can see in Figure 25-3, this can only reduce the size of the standard clipping region.

**Figure 25-3**

**Using a Local Clipping Rectangle**

- Filled w/no local clip
- Local clip enclosed in window
- Local clip extends outside window

### Hit Detection

PenPoint provides some low-level functions for determining whether one rectangle contains or intersects another. **clsSysDrwCtx** supports the notion of **hit detection** to determine whether an arbitrary drawing operations intersects with or contains a specified rectangle. Hit detection is a special mode of operation for drawing contexts. When hit detection is on, the drawing context does not render images. Instead, it responds to drawing messages by reporting whether the requested figures intersect with or are contained by the rectangle.

### Bounds Accumulation

**Bounds accumulation** lets you calculate the bounding rectangle of a complex set of figures. Like hit detection, bounds accumulation is a mode of operation during which drawing contexts do not render images. While bounds accumulation is on, the drawing context maintains a rectangle and responds to each drawing message by expanding the rectangle to enclose the requested figures. Your application can get the bounding rectangle while bounds accumulation is on, or get the final rectangle when it turns bounds accumulation off.
For applications whose graphic elements are not static, you can use bounds accumulation to improve the efficiency of window repainting. For example, if the user moves a figure in a drawing application, the application can use bounds accumulation to determine the bounding rectangles of the figure before and after moving it. The application can then use the accumulated bounds to set the dirty region of the window. This precludes the need to repaint the entire window whenever the user moves an object.

**Figure Drawing Operations**

ImagePoint supports a relatively simple set of figure drawing operations. By setting the various attributes of the graphic state (described later), your application can use these drawing operations to create a wide variety of visual effects.

**Open Figures**

Open figures are composed of lines drawn according to the line pattern and other line attributes of the graphic state. ImagePoint supports the following open figures:

- A **polyline** is a series of connected line segments defined by a list of points.
- A **Bezier curve** is a smooth curve defined by four control points.
- An **arc** is a segment of the circumference of an ellipse, defined by an ellipse and two points. The rays from the center of the ellipse through the two points mark the limits of the arc.

Figure 25-4 shows some examples of open figures.

**Closed Figures**

Closed figures are composed of lines that properly encloses a region. The drawing context draws the lines according to the line attributes of the graphic state, just as with open figures, and additionally fills the enclosed region with the fill pattern specified in the current graphic state. A transparent fill pattern to prevents the
drawing context from filling the region, and a line width of zero prevents the
drawing context from drawing the enclosing lines.

ImagePoint supports the following closed figures:

- **A rectangle** is a four-sided figure composed of four lines connected at right
  angles. A rectangle is defined by a point representing the origin of the
  rectangle, and a coordinate pair representing the rectangle's size in the x and y
  directions.

- **An ellipse** is a regular, round figure (such as a circle) defined by an enclosing
  rectangle.

- **A polygon** is a series of line segments defined by a list of points, like the
  polyline described above, but with the first and last points in the list
  connected to close the figure.

- **A sector** is a segment of an ellipse defined by an ellipse and two points, like
  the arc described above, closed with two lines drawn from the endpoints of
  the arc to the center of the ellipse.

- **A chord** is a segment of an ellipse defined by an ellipse and two points, like
  the arc described above, closed with a single line connecting the endpoints of
  the arc.

Figure 25-5 shows some examples of closed figures.

---

**Figure 25-5**

**Closed Figures**

| Rectangle | Polygon | Sector | Chord |

---

**Sampled Images**

A **sampled image** is an ordered collection of rows of image samples such as the
output of an image scanner. Displayed one after the other, the rows recreate the
image from which their values were sampled.

clsSysDrwCtx supports the conversion of sampled images into grayscale pixelmaps
it can render. It also provides a mechanism for rendering the sampled image once
into off-screen memory, and to quickly copy the stored pixelmap into a window.
This can improve performance, especially in situations where the drawing context
must repeatedly display a single sampled image.
Drawing Sampled Images

A drawing context renders the sampled image into a destination rectangle within the window. To prepare the sampled image for rendering, your application must provide a pointer to the sample data in memory. In cases where the sampled image is too large to fit in memory, your application can provide a callback function or object that provides sample data to the drawing context, one row at a time, as the DC requests it.

The drawing context scales the sampled image to fit into the destination rectangle, then renders it. You provide the drawing context with information about the sampled image, such as the maximum sample value. Sample values from zero to the maximum sample value represent shades of grey increasing linearly from black to white. A drawing context can reduce the sampled image to a single bit-deep image mask which it can render in the current foreground and background color.

Cached Images

In rendering a sampled image, a drawing context handles the LUC coordinate transformations, different device resolutions, different pixel aspect ratios, different number of bit planes, dithering, and so on. This is a powerful mechanism, but it can be somewhat slow especially with large images. If your application renders a sampled image repeatedly on the same device at the same scale, you can improve efficiency by performing these transformations once to generate the desired pixelmap, and then copying the pixelmap to the display as necessary.

Rather than directly rendering a sampled image, a drawing context can create a cached image. The DC creates a memory buffer where it stores the pixelmap. It is up to the controlling application to maintain and eventually free the cached image buffer after the drawing context creates it. When the application wishes to display the cached image, it provides the drawing context a pointer to the cached image and an x-y position in LUC. The drawing context quickly copies the preprocessed pixel image to the window, aligning the image's hot spot (see below) with the specified x-y position.

A cached image can have an associated hot spot and a mask. The hot spot is a position offset in LUC. If the image has a nonzero hot spot, the drawing aligns the hot spot with the x-y position the application supplies (rather than aligning the lower left corner of the image with the x-y position). The image mask is a bitmap of the same width and height as the cached pixelmap. If a cached image has an associated mask, the drawing context renders only those pixels in the cached image that correspond to 1 bits in the mask.

Cached images are optimized for the current window device when they are created. They are inherently device-dependent. The application must free and recreate the cached image whenever the window device changes pixel aspect ratio, orientation, plane depth, and so on. Furthermore, you cannot render anything but sampled images into a cached image buffer. If you need to draw synthesized images that a drawing context can quickly copy to an on-screen window, use a
window on an image device. Image devices, which are optimized for use with large windows and window trees, are described in Chapter 24, Window Device Classes.

**Color**

Drawing contexts support a foreground color and background color. Drawing context operations normally draw in the foreground color. Some draw in the foreground and background colors simultaneously, such as when filling a closed figure with a two-color pattern. The standard method for specifying colors is the device-independent RGB (red-green-blue) method. While the use of palette colors can result in slightly better graphics performance, it is highly device-dependent and therefore not recommended for writers of portable applications.

**RGB Color Values**

An RGB (red-green-blue) color value is a combination of three values: red, green, and blue. Each part of an RGB value can range from zero to 255. The higher the value, the brighter the corresponding color component. For example, a red value of 255 and blue and green values of zero yields a bright red. Red and blue values of 75 and a green value of zero yields a muted magenta. If the red, green, and blue values are equal, the resulting color is a shade of gray, from black (all zero) to white (all 255).

For the background color only, drawing contexts support a transparency value in addition to the red, green, and blue values. A transparent background means that the DC will not paint background pixels; whatever is behind the background will show through. Like the RGB values, transparency can range from zero to 255. A transparency value of zero yields a completely opaque background color, while a transparency of 255 makes the background completely transparent. At present, ImagePoint supports only these two levels of transparency, and only the background color can be transparent.

When a drawing context's foreground and background colors are specified using the RGB model, the drawing context automatically finds the best color match independent of the display device to which it is bound. Whether it is drawing on a black and white LCD or on a full-color printer, the drawing context will do its best to render in the specified colors.

**Hardware-Dependent Palette Colors**

Some hardware platforms support a palette, or numbered list, of colors. Each index into the palette corresponds to the color stored at that location in the list. It is possible to specify a drawing context's foreground and background colors as indices into the palette, but this is extremely device-dependent. The only advantages to using this technique for specifying colors is a slight performance improvement, particularly when repeatedly and rapidly changing the foreground and background colors. If you specify either the foreground or background color as a palette color, you must specify both as palette colors. Palette colors and RGB colors are not compatible with one another.
Drawing contexts support a sophisticated model for the creation and display of text. The programming interface, however, is not too complex: the drawing context opens a font, scales the font to the appropriate size, and draws text just as it draws other synthesized images such as rectangles and ellipses. Some features of the ImagePoint text model include:

- Device-independent, user-installable outline fonts.
- On-the-fly generation of font bitmaps (cached for improved performance) from font outlines, for any resolution device.
- Automatic selection of a closest match from the installed fonts when a requested font is not available.
- Automatic transformation of font attributes (for example, transforming a roman font to italic) when requested fonts are not available.
- Automatic substitution of glyphs from other fonts for characters not available in the current font.

This section provides an overview of the technical concepts involved in the ImagePoint font model. See Chapter 26, The Drawing Context Class, for more detailed information.

### Fonts

Every ImagePoint font has a font ID, a set of font metrics, and a set of glyph outlines. The font ID is a unique code which GO Corporation assigns, so a given font ID identifies the same font on all PenPoint systems for all time. Font metric information includes size information such as the x height, em size, space width, ascender height and descender depth, and largest and smallest character sizes for the font. In addition, the font metrics provide a set of font attributes that describe the general appearance of the font. Font attributes include information such as the font group (modern Roman or sans serif, for example), weight or boldness, the aspect or relationship between character width and height, whether the font is italic, and whether it is monospaced.

The glyph outlines define the shape of each character, or glyph, in the font. All ImagePoint fonts scale to any size, so the outlines do not include absolute size information. Fonts can include bitmaps tuned for specific sizes and resolutions. When a drawing context draws text in a font size that has a tuned bitmap for the particular window device's resolution, it will use the tuned bitmap rather than generate a bitmap from the font outline. This improves legibility at low resolution or in very small font sizes.

### Opening a Font

Before a drawing context can draw text in a particular font, it must open the font. Because fonts are user-installable, there is no guarantee that a particular font will be available when a drawing context tries to open it. For this reason, in addition to
specifying a font ID for the font to open, you must also specify the desired attributes of the font (bold, italic, and so on). The drawing context opens the closest-match font and synthesizes any missing attributes.

For example, suppose you try to open a font ID corresponding to the serif font Palatino Bold, with the group attribute set to modern Roman and the weight attribute set to bold. If Palatino Bold is one of the installed fonts, the drawing context will open it. If Palatino Bold is not installed and Palatino Roman is the closest match, the drawing context will open it and alter it to synthesize the bold attribute. If no Palatino font is installed and the closest match is Times Roman, the drawing context will use Times Roman and alter it to synthesize the bold attribute.

Scaling a Font

When a drawing context opens a font, it scales the font so that the point size of the font (its em height) is equivalent to one unit in the DC's current logical unit coordinates. By default, the LUC unit is the point (1/72 of an inch), so the default font size is one point, much too small to be legible. Applications therefore need to scale the font before drawing text with it. The font scaling factors describe a multiple of the LUC unit.

For example, in the default case described above, the font is one point high when the drawing context opens it. If you set the font scale to 12 in both the x and y directions, the font scales up to 12 points (12 times the LUC unit of one point). If you then scale the LUC so that the x unit is 0.1 inch and the y unit is one millimeter, the font size for subsequent drawing becomes 1.2 inches in the x direction and 12 millimeters in the y direction. The LUC scale affects the font scale, but font scale does not affect the LUC scale.

Drawing Text

After a drawing context opens and scales a font, it can draw text in that font. An application instructs the drawing context to draw a specific string of text beginning at a specific LUC origin. The drawing context generates bitmaps for each character of the font, based on the font's scaled glyph outlines and the resolution of the window device. To improve performance, the drawing context generates the bitmaps for a particular font size and device resolution only once, caching the bitmaps for later use.

The drawing context renders the character bitmaps in its bound window, filling them with the current foreground color. The application may modify the way the drawing context draws text with a number of drawing format attributes including underline, strikeout, and modifiers for word spacing, letter spacing, and font alignment. Underline and strikeout line thickness is defined in the font metrics, independent of the current drawing context line thickness, to achieve an attractive line weight. Alignment sets the ascender height, midpoint of x height, baseline, or
descender depth of the font equal to the y coordinate of the specified drawing origin.

**Character Encoding and Missing Glyphs**

Text strings exist in the system as strings of byte-encoded characters. The *encoding* attribute with which the drawing context opens the font determines which glyphs correspond to which character bytes. Most text encodings have the same glyphs for the printable ASCII characters, but different encodings produce different glyphs for control characters and ASCII characters beyond ASCII DEL (decimal 127). If a character in a text string corresponds to a glyph that is not defined by the current font, the drawing context will try to locate the glyph in another installed font. The replacement glyph may not look like part of the opened font (it isn’t), but at least it appears in the rendered text.

**Graphic State**

Every instance of clsSysDrwCtx supports the same methods for drawing figures and text in a bound window. It is information such as logical unit coordinate scaling, translation, and rotation, line thickness, foreground and background color, and so on that differentiates one instance of clsSysDrwCtx from another. This differentiating information, collectively, is a drawing context’s graphic state.

Drawing contexts provide limited support for graphic state storage. clsDrwCtx defines messages for temporarily storing one or more graphic states while creating a new one for a special purpose, and for restoring a stored graphic state. You must implement your own storage mechanism, such as a stack.

The most important elements of a graphic state, for purposes of rendering images, include:

- The LUC unit (default is one point). The logical unit coordinate (LUC) system is the coordinate system by which a drawing context interprets coordinates for drawing operations. By default, a new drawing context provides an LUC system whose units are one point (1/72 of an inch) in both the x (horizontal) and y (vertical) directions. Drawing contexts provide mechanisms for scaling the size of the LUC unit independently in each direction. Logical unit coordinates are also discussed in “Coordinate Systems,” earlier in this chapter.

- Local clipping rectangle (default is entire window). The local clipping rectangle restricts the area of a window which drawing operations affect. For example, if the local clipping rectangle covers the upper left quarter of a window, an operation to draw a line from the upper left corner of the window to the lower right corner will draw only the portion of the line from the upper left corner to the center of the window. The remainder of the line, which falls outside of the local clipping rectangle, is not rendered. The concept of the local clipping rectangle is discussed in “Local Clipping,” earlier in this chapter.
• Line thickness (default is one LUC unit). The drawing context maintains a line thickness value. Initially, the drawing context draws lines that are one unit thick (as mentioned above, the default unit is one point). If you change the line thickness, the drawing context draws subsequent lines with the new line thickness. The line thickness determines the thickness of all lines the drawing context draws, including the outlines of closed figures.

• Line cap and join, which determine how line segments join together (default is butt caps and mitre joins). The line cap determines the shape of the end of a line segment that does not join with another line. A butt line cap simply cuts off the line segment at its endpoint, perpendicular to the line's direction. A square line cap has the effect of drawing a square as wide as the line thickness, centered around the endpoint, with a center line parallel to the line segment. A round line cap has the effect of drawing a circle whose diameter is the line thickness, centered around the endpoint of the line.

• Foreground and background color (default is black foreground, white background). Initially, the drawing context draws lines in the foreground color, and fills closed figures with the background color. While it is possible to display as many colors as the hardware supports on screen at one time, the drawing context draws in only the foreground and background color at one time.

• Line and fill pattern (default is foreground color lines, background color fills). You can change the foreground and background color, as described above, but the lines initially are solid foreground color, while fills are solid background color. You can change the line pattern and fill pattern independently to change the way the drawing context combines the foreground and background color to draw lines and fills. For example, you can change the line pattern so that the drawing context draws lines in a color that is 75% foreground color and 25% background color, while it draws fills as alternating diagonal lines of foreground color and background color.

• Font (default is system font, sized to one LUC unit). Graphic states implement the ability to render text with a scaleable, outline font. In order to do this, they maintain information about font with which to render text, and the size at which to draw the font. It is a simple matter to change the font family, the size, and a variety of other attributes of the font.

Chapter 26, The Drawing Context Class, describes the elements of the graphic state in more detail.
A picture segment is an instance of clsPicSeg, which inherits from clsSysDrwCtx. Picture segments have all of the capabilities and features of drawing contexts, but add the notion of the grafic, a record of a drawing message that the picture segment can redrew at a later time. A picture segment maintains a list of grafics recording the drawing messages the picture segment has received, in the order it received them. You can retrieve, alter, reorder, and delete individual grafics. The picture segment can redrew all of the grafics, or just one, at any time, and it can add and alter grafics with or without affecting the image in the picture segment’s bound window.

Picture segments also support splines (smoothly connected sequences of Bezier curves), adding them to the suite of drawing context primitives such as polygons and text. Furthermore, with minimal assumptions, picture segments support the inclusion of arbitrary objects other than the standard DC primitives and splines into their images. Chapter 27, The Picture Segment Class, describes picture segments in more detail.
Chapter 26 / The Drawing Context Class

**System Drawing Context Messages**

Here are the messages defined by clsSysDrwCtx. Many are described here; some are described in later chapters.

<table>
<thead>
<tr>
<th>Message</th>
<th>pArgs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_SYSDC_NEW</td>
<td>Creates a system drawing context.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_SYSDC_NEW</td>
<td>Initializes the SYSDC_NEW structure to default values.</td>
</tr>
<tr>
<td>msgDcSetWindow</td>
<td>new WIN</td>
<td>Binds a window to the receiver and returns the previously bound window.</td>
</tr>
<tr>
<td>msgDcGetWindow</td>
<td>pNull</td>
<td>Gets the window to which the drawing context is bound.</td>
</tr>
<tr>
<td>msgDcInitialize</td>
<td>pNull</td>
<td>Sets graphics state to initial values.</td>
</tr>
<tr>
<td>msgDcPush</td>
<td>P_SYSDC_STATE</td>
<td>Gets the graphics state and stores it.</td>
</tr>
<tr>
<td>msgDcPop</td>
<td>P_SYSDC_STATE</td>
<td>Sets the graphics state from one saved by msgDcPush.</td>
</tr>
<tr>
<td>msgDcPushFont</td>
<td>P_SYSDC_FONT_STATE</td>
<td>Gets the font state and stores it.</td>
</tr>
<tr>
<td>msgDcPopFont</td>
<td>P_SYSDC_FONT_STATE</td>
<td>Sets the font state from one saved by msgDcPushFont.</td>
</tr>
<tr>
<td>msgDcSetMode</td>
<td>new SYSDC_MODE</td>
<td>Sets the drawing mode and returns the old SYSDC_MODE.</td>
</tr>
<tr>
<td>msgDcGetMode</td>
<td>pNull</td>
<td>Gets the drawing mode.</td>
</tr>
<tr>
<td>msgDcSetPreMultiply</td>
<td>BOOLEAN</td>
<td>Sets the pre-multiply state and returns the old state.</td>
</tr>
<tr>
<td>msgDcSetRop</td>
<td>SYSDC_ROP</td>
<td>Sets the raster op and returns the old rop.</td>
</tr>
<tr>
<td>msgDcPlaneNormal</td>
<td>nothing</td>
<td>Sets the plane mask to the normal plane(s), returning the old mask.</td>
</tr>
<tr>
<td>msgDcPlanePen</td>
<td>nothing</td>
<td>Sets the plane mask to the plane(s) for pen ink, returning the old mask.</td>
</tr>
<tr>
<td>msgDcPlaneMask</td>
<td>SYSDC_PLANE_MASK</td>
<td>Sets an arbitrary plane mask, returning the old mask.</td>
</tr>
<tr>
<td>msgDcGetLine</td>
<td>P_SYSDC_LINE</td>
<td>Gets all line attributes if pArgs is P_SYSDC_LINE.</td>
</tr>
<tr>
<td>msgDcSetLine</td>
<td>P_SYSDC_LINE</td>
<td>Sets all line attributes. Returns old line thickness.</td>
</tr>
<tr>
<td>msgDcSetLineThickness</td>
<td>COORD16</td>
<td>Sets line thickness to new value; returns old line thickness.</td>
</tr>
<tr>
<td>msgDcHoldLine</td>
<td>BOOLEAN</td>
<td>Turns hold line thickness mode on/off; returns old hold mode.</td>
</tr>
<tr>
<td>Message</td>
<td>pArgs</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>----------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>msgDcSetForegroundRGB</td>
<td>U32</td>
<td>Sets foreground color using an RGB specification.</td>
</tr>
<tr>
<td>msgDcSetBackgroundRGB</td>
<td>U32</td>
<td>Sets background color using an RGB specification.</td>
</tr>
<tr>
<td>msgDcInvertColors</td>
<td>pNull</td>
<td>Swaps foreground and background colors.</td>
</tr>
<tr>
<td>msgDcGetForegroundRGB</td>
<td>P_U32 or P_SYSDC_RGB</td>
<td>Returns foreground RGB value.</td>
</tr>
<tr>
<td>msgDcGetBackgroundRGB</td>
<td>P_U32 or P_SYSDC_RGB</td>
<td>Returns background RGB value.</td>
</tr>
<tr>
<td>msgDcMatchRGB</td>
<td>U32</td>
<td>Returns palette entry that best matches an RGB.</td>
</tr>
<tr>
<td>msgDcSetForegroundColor</td>
<td>SYSDC_COLOR</td>
<td>Sets foreground color using a hardware palette index, returning old color.</td>
</tr>
<tr>
<td>msgDcSetBackgroundColor</td>
<td>SYSDC_COLOR</td>
<td>Sets background color using a hardware palette index, returning old color.</td>
</tr>
<tr>
<td>msgDcSetLinePat</td>
<td>SYSDC_PATTERN</td>
<td>Sets the line pattern; returns old value.</td>
</tr>
<tr>
<td>msgDcSetFillPat</td>
<td>SYSDC_PATTERN</td>
<td>Sets the fill pattern; returns old value.</td>
</tr>
<tr>
<td>msgDcGetLinePat</td>
<td>pNull</td>
<td>Gets the line pattern.</td>
</tr>
<tr>
<td>msgDcGetFillPat</td>
<td>pNull</td>
<td>Gets the fill pattern.</td>
</tr>
<tr>
<td>msgDcAlignPattern</td>
<td>P_XY32</td>
<td>Sets the pattern alignment in LUC.</td>
</tr>
<tr>
<td>msgDcGetMatrix</td>
<td>P_MAT</td>
<td>Returns the LWC matrix.</td>
</tr>
<tr>
<td>msgDcGetMatrixLUC</td>
<td>P_MAT</td>
<td>Returns the LUC matrix.</td>
</tr>
<tr>
<td>msgDcSetMatrixLUC</td>
<td>P_MAT</td>
<td>Replaces the LUC matrix.</td>
</tr>
<tr>
<td>msgDcUnitsMetric</td>
<td>pNull</td>
<td>Sets input units to 0.01 mm.</td>
</tr>
<tr>
<td>msgDcUnitsMil</td>
<td>pNull</td>
<td>Sets input units to 0.001 inch.</td>
</tr>
<tr>
<td>msgDcUnitsPoints</td>
<td>pNull</td>
<td>Sets input units to points (1/72 of an inch).</td>
</tr>
<tr>
<td>msgDcUnitsTwips</td>
<td>pNull</td>
<td>Sets input units to 1/20 of a point.</td>
</tr>
<tr>
<td>msgDcUnitsPen</td>
<td>pNull</td>
<td>Sets input units to pen sample units.</td>
</tr>
<tr>
<td>msgDcUnitsLayout</td>
<td>pNull</td>
<td>Sets input units to UI toolkit layout units.</td>
</tr>
<tr>
<td>msgDcUnitsRules</td>
<td>pNull</td>
<td>Sets input units to the rules associated with the system font.</td>
</tr>
<tr>
<td>msgDcUnitsDevice</td>
<td>pNull</td>
<td>Sets input units to device pixels.</td>
</tr>
<tr>
<td>msgDcUnitsWorld</td>
<td>pNull</td>
<td>Sets input units to an arbitrary number of device pixels.</td>
</tr>
<tr>
<td>msgDcUnitsOut</td>
<td>MESSAGE</td>
<td>Sets output units produced by transformation of input units.</td>
</tr>
<tr>
<td>msgDcIdentity</td>
<td>pNull</td>
<td>Sets LUC matrix to identity.</td>
</tr>
<tr>
<td>msgDcRotate</td>
<td>ANGLE</td>
<td>Rotates LUC matrix.</td>
</tr>
<tr>
<td>msgDcScale</td>
<td>P_SCALE</td>
<td>Scales LUC matrix.</td>
</tr>
<tr>
<td>msgDcScaleWorld</td>
<td>P_SIZE32</td>
<td>Creates a world scale of window width/height.</td>
</tr>
<tr>
<td>msgDcTranslate</td>
<td>P_XY32</td>
<td>Translates LUC matrix.</td>
</tr>
</tbody>
</table>
CHAPTER 26 / THE DRAWING CONTEXT CLASS
System Drawing Context Messages

Table 26-1 (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>pArgs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgDcLCtoLUC_XY32</td>
<td>P_XY32</td>
<td>Transforms a point from window (device) space to logical space.</td>
</tr>
<tr>
<td>msgDcLCtoLUC_XY32</td>
<td>P_XY32</td>
<td>Transforms a point from logical space to window (device) space.</td>
</tr>
<tr>
<td>msgDcLCtoLUC_SIZE32</td>
<td>P_SIZE32</td>
<td>Transforms a size from window (device) space to logical space.</td>
</tr>
<tr>
<td>msgDcLCtoLUC_SIZE32</td>
<td>P_SIZE32</td>
<td>Transforms a size from logical space to window (device) space.</td>
</tr>
<tr>
<td>msgDcLCtoLUC_RECT32</td>
<td>P_RECT32</td>
<td>Transforms a rectangle from window (device) space to logical space.</td>
</tr>
<tr>
<td>msgDcLCtoLUC_RECT32</td>
<td>P_RECT32</td>
<td>Transforms a rectangle from logical space to window (device) space.</td>
</tr>
</tbody>
</table>

| msgDcClipboard          | P_RECT32 or pNull   | Sets or clears clip rectangle.                   |
| msgDcClipClear          | pNull               | Returns clipping to entire window.               |
| msgDcClipNull           | pNull               | Suspends all clipping (except to raw device).    |
| msgDcHitTest            | P_RECT32 or pNull   | Turns hit testing on/off.                        |
| msgDcAccumulateBounds   | P_RECT or pNull     | Starts or stops bounds accumulation; retrieve bounds. |
| msgDcDirtyAccumulation  | P_RECT32 or pNull   | Marks accumulation dirty; turns accumulation off; retrieves bounds. |
| msgDcGetBounds          | P_RECT32            | Retrieves current accumulation bounds rectangle. |

| msgDcDrawPolyline       | P_SYSDC_POLYLINE    | Draws a line; needs at least 2 points. Returns either hit test or stsOK. |
| msgDcDrawBezier         | P_XY32 (array of 4) | Draws a Bezier curve; needs exactly 4 points. |
| msgDcDrawArcRays        | P_SYSDC_ARC_RAYS    | Draws an arc using the two rays method. Returns either hit test or stsOK. |

| msgDcSetPixel           | P_SYSDC_PIXEL       | Sets a pixel with a value.                       |
| msgDcGetPixel           | P_SYSDC_PIXEL       | Gets a pixel value.                              |
| msgDcDrawRectangle      | P_RECT32            | Draws a rectangle. Returns either hit test or stsOK. |
| msgDcDrawEllipse        | P_RECT32            | Draws an ellipse. Returns either hit test or stsOK. |
| msgDcDrawPolygon        | P_SYSDC_POLYGON     | Draws a polygon. Returns either hit test or stsOK. |
| msgDcDrawSectorRays     | P_SYSDC_ARC_RAYS    | Draws a sector (pie wedge) using the two rays method. |
| msgDcDrawChordRays      | P_SYSDC_ARC_RAYS    | Draws a chord using the two rays method. Returns either hit test or stsOK. |
| msgDcFillWindow         | pNull               | Frames window with a line and fills the window.  |
| msgDcDrawImage          | P_SYSDC_IMAGE_INFO  | Draws an image from sampled image data. The image will be scaled, rotated, translated, according to the current state. |

continued
### Table 26-1 (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>pArgs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgDcDrawImageMask</td>
<td>P_SYSDC_IMAGE_INFO</td>
<td>Draws a mask from sampled image data. Similar to msgDcDrawImage.</td>
</tr>
<tr>
<td>msgDcCacheImage</td>
<td>P_SYSDC_CACHE_IMAGE</td>
<td>Passes back a cached image in pCache, given a sampled image and an optional mask.</td>
</tr>
<tr>
<td>msgDcCopyImage</td>
<td>P_SYSDC_COPY_IMAGE</td>
<td>Copies a cached image to the bound window.</td>
</tr>
<tr>
<td>SysDcFontId()</td>
<td></td>
<td>Takes a 4 byte string font description and returns a 16-bit font id number.</td>
</tr>
<tr>
<td>SysDcFontString()</td>
<td></td>
<td>Takes a 16-bit font id number and passes back a 4 char string.</td>
</tr>
<tr>
<td>msgDcOpenFont</td>
<td>P_SYSDC_FONT_SPEC or pNull</td>
<td>Opens a font.</td>
</tr>
<tr>
<td>msgDcScaleFont</td>
<td>P_SCALE or pNull</td>
<td>Scales font matrix.</td>
</tr>
<tr>
<td>msgDcIdentityFont</td>
<td>pNull</td>
<td>Sets font matrix scale to default of 1 unit (LUC).</td>
</tr>
<tr>
<td>msgDcDrawText</td>
<td>P_SYSDC_TEXT_OUTPUT</td>
<td>Draws text in the current font.</td>
</tr>
<tr>
<td>msgDcMeasureText</td>
<td>P_SYSDC_TEXT_OUTPUT</td>
<td>Computes size of text and advances pArgs-&gt;cp accordingly.</td>
</tr>
<tr>
<td>msgDcDrawTextRun</td>
<td>P_SYSDC_TEXT_OUTPUT</td>
<td>Like msgDcDrawText, except run spacing applies.</td>
</tr>
<tr>
<td>msgDcMeasureTextRun</td>
<td>P_SYSDC_TEXT_OUTPUT</td>
<td>Like msgDcMeasureText, except run spacing applies.</td>
</tr>
<tr>
<td>msgDcDrawTextDebug</td>
<td>P_SYSDC_TEXT_OUTPUT</td>
<td>Like msgDcDrawText, except text is drawn with debugging lines around each char.</td>
</tr>
<tr>
<td>msgDcPreloadText</td>
<td>P_SYSDC_TEXT_OUTPUT</td>
<td>Preloads pText into cache.</td>
</tr>
<tr>
<td>msgDcGetCharMetrics</td>
<td>P_SYSDC_CHAR_METRICS</td>
<td>Gets char metrics information for a string.</td>
</tr>
<tr>
<td>msgDcGetFontMetrics</td>
<td>P_SYSDC_FONT_METRICS</td>
<td>Gets the font metrics for the current font.</td>
</tr>
<tr>
<td>msgDcGetFontWidths</td>
<td>P_SYSDC_FONT_WIDTHS</td>
<td>Gets the font width table of the current font.</td>
</tr>
<tr>
<td>msgDcDrawPageTurn</td>
<td>P_SYSDC_PAGE_TURN</td>
<td>Draws a page turn effect over the bound window.</td>
</tr>
<tr>
<td>msgDcCopyPixels</td>
<td>P_SYSDC_PIXELS</td>
<td>Copies pixels from srcWindow to the bound window.</td>
</tr>
<tr>
<td>msgDcDrawPixels</td>
<td>P_SYSDC_PIXELS</td>
<td>Draws foreground and background colors in the bound window's pixels using srcWindow's pixel values as a stencil.</td>
</tr>
</tbody>
</table>

In addition, drawing contexts respond to all clsWin messages by passing them on to their bound windows. You often send the following window messages to a DC because the DC transforms the specified coordinates from LUC into LWC before passing it to its window.

- msgWinDirtyRect
- msgWinBeginRepaint
- msgWinGetMetrics
- msgWinDelta
- msgWinCleanRect
- msgWinBeginPaint
- msgWinTransformBounds
Creating a DC

To create a drawing context, send msgNewDefaults and then msgNew to clsSysDrwCtx.

Default Drawing Context State

Here are the defaults for the new drawing context. You can also reset a DC to these values by sending it msgDcInitialize. Most defaults are explained in more detail in following sections.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>units in</td>
<td>msgDcUnitsPoints</td>
<td>Application specifies units to DC in points (1/72 of an inch).</td>
</tr>
<tr>
<td>units out</td>
<td>msgDcUnitsDevice</td>
<td>DC specifies units to windows in pixels.</td>
</tr>
<tr>
<td>matrix</td>
<td>identity</td>
<td>No scaling or rotation of LUC.</td>
</tr>
<tr>
<td>clipping rectangle</td>
<td>none</td>
<td>No local clipping rectangle.</td>
</tr>
<tr>
<td>premultiply</td>
<td>false</td>
<td>Standard matrix concatenation order (post-multiply).</td>
</tr>
<tr>
<td>raster op</td>
<td>sysDcRopCopy</td>
<td>Source copied into destination.</td>
</tr>
<tr>
<td>drawing mode</td>
<td>sysDcDrawNormal</td>
<td>Keep narrow lines visible.</td>
</tr>
<tr>
<td></td>
<td>sysDcHoldDetail</td>
<td></td>
</tr>
<tr>
<td>plane mask</td>
<td>sysDcPlaneNormal</td>
<td>Don’t draw in the pen ink plane.</td>
</tr>
<tr>
<td>line cap</td>
<td>sysDcCapButt</td>
<td>Line ends are squared off.</td>
</tr>
<tr>
<td>line join</td>
<td>sysDcJoinMiter</td>
<td>Line joins are mitered.</td>
</tr>
<tr>
<td>line thickness</td>
<td>1</td>
<td>One unit in (a point, by default).</td>
</tr>
<tr>
<td>line miter limit</td>
<td>10</td>
<td>Lines meeting at angles less than 10 degrees don’t produce long spikes.</td>
</tr>
<tr>
<td>line radius</td>
<td>0</td>
<td>Corners of rectangles are squared off, not rounded.</td>
</tr>
<tr>
<td>foreground color</td>
<td>sysDcRGBBlack</td>
<td>Closest to black on every device.</td>
</tr>
<tr>
<td>background color</td>
<td>sysDcRGBWhite</td>
<td>Closest to white on every device.</td>
</tr>
<tr>
<td>fill pattern</td>
<td>sysDcPatBackground</td>
<td>Fill with background color (white, by default).</td>
</tr>
<tr>
<td>fill mode</td>
<td>even/odd</td>
<td>Method used to compute the area to fill.</td>
</tr>
<tr>
<td>line pattern</td>
<td>sysDcPatForeground</td>
<td>Draw lines in foreground color (black, by default).</td>
</tr>
<tr>
<td>font scale</td>
<td>1</td>
<td>Initial font size is one unit in (one point, by default) tall.</td>
</tr>
<tr>
<td>default font</td>
<td>unspecified</td>
<td>It is best to open a font, though text output will find some font.</td>
</tr>
</tbody>
</table>
Should You Create a Drawing Context?

A system drawing context is about 500 bytes long. You can create dozens of windows without using up a lot of memory, but DC's are much bigger objects. There is no global DC you can borrow. It is reasonable to create a DC per window, if you have a small number of windows; this will also give you maximum performance. Or, you can use one DC for all your windows, and bind it to each window in turn. It is also OK to create a DC for every msgWinRepaint message, use it to paint, and then free it; this will reduce the dynamic memory use of your application.

If you have subtasks in your application, and wish to share a DC throughout your process, you'll need to use semaphores to control access to it. (The same is true for any shared object.)

Drawing with a Drawing Context

Your drawing context is not bound to any window or device when you create it. Use msgDrwCtxSetWindow to bind it to a window. Or, use msgDcSetWindow; it's the same message with the common SysDC Dc prefix. Once bound, drawing via the DC occurs on the bound window.

Drawing Coordinates

When you issue drawing commands to your drawing context, you do so in LUC (Logical Unit Coordinates). These go through several transformations before finally ending up on the device. Unlike the other coordinate systems at work in PenPoint™ operating system, you can set LUC to suit your drawing:

- You can set the size of LUC units (so that one unit is one typographer's point, for example).
- You can scale LUC units (so that one unit equals one inch, for example).
- You can change translate the origin (0,0) to some point other than the lower left hand corner of the window.
- You can rotate the coordinate system.

Other levels of scaling/translation/rotation are going on before your drawing finally fills in the screen pixels. Your output window is probably not at the screen origin, the hardware pixels may not be square, the hardware coordinate system may start in a different quadrant, etc. If you need to use screen coordinates or device units, send messages to your DC to convert back to these.
By default, when you create a new DC, one unit in LUC is one **point**. One point is 1/72 of an inch. If you draw a box 72 units on a side, it will be one inch square on the physical device with which your DC is bound. Also, by default the origin is in the lower-left corner of the window to which your DC is bound. So if you draw the square at (72, 144), it will be one inch to the right and two inches above the lower-left corner.

```
SYSDC_NEW  dcn;
SYSDC       dc;
WIN         myWin;
RECT32      rect;
STATUS      s;

ObjCallRet(msgNewDefaults, clsSysDrwCtx, &dcn, s);
ObjCallRet(msgNew, clsSysDrwCtx, &dcn, s);

myRect.origin.x = 72;
myRect.origin.y = 144;
myRect.size.w = myRect.size.h = 72;

ObjectCall(msgDcDrawRectangle, dc, &myRect);
```

### Units

You can initialize the mapping of LUC units to physical dimensions, by sending your DC one of:

<table>
<thead>
<tr>
<th>Message</th>
<th>Sets One LUC Unit to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgDcUnitsPoints</td>
<td>one point, or 1/72 of an inch (this is the default)</td>
</tr>
<tr>
<td>msgDcUnitsMetric</td>
<td>0.01 millimeters, or 10 microns</td>
</tr>
<tr>
<td>msgDcUnitsMil</td>
<td>one mil, or 0.001 inch</td>
</tr>
<tr>
<td>msgDcUnitsTwips</td>
<td>one twip, or 1/20 of a point</td>
</tr>
<tr>
<td>msgDcUnitsDevice</td>
<td>one device pixel</td>
</tr>
<tr>
<td>msgDcUnitsLayout</td>
<td>1/8 of an em in the system font (not the open font);</td>
</tr>
<tr>
<td></td>
<td>this is the unit the UI Toolkit classes use</td>
</tr>
<tr>
<td>msgDcUnitsWorld</td>
<td>a division of the bound window's width and height</td>
</tr>
<tr>
<td></td>
<td>(msgDcScaleWorld specifies the number of divisions)</td>
</tr>
<tr>
<td>msgDcUnitsPen</td>
<td>(obsolete) one pixel of the pen tracking sensor</td>
</tr>
</tbody>
</table>
Scale

After the unit system is initialized you can send msgDcScale to further scale your coordinate system. You specify scale using 32 bit fixed-point numbers. Fixed-point numbers are special numbers which encode 16 bits of integer and 16 bits of fraction in 32 bits. You create and manipulate fixed-point numbers using special routines as explained in Part 8: System Services.

For instance, if you send msgDcScale to a new DC as follows:

```c
SCALE scale;
SYSDC dc;
scale.x = FxMakeFixed(2,0);
scale.y = FxMakeFixed(3,0);
ObjectCall(msgDcScale, dc, &scale);
```

Then a 72-point square will appear two inches wide and three inches high.

Rotation

You can send msgDcRotate to rotate LUC. This takes an ANGLE in degrees. The angle must be an integral number of degrees from zero to 359.

Translation

You can send msgDcTranslate to move the origin in LUC to any point. Thus instead of drawing the square at (72, 144), you could achieve the same effect by translating the origin (0,0) to (72, 144) and then drawing a 72-point square at (0,0).

Resetting LUC

You can send msgDcIdentity to the DC to set the LUC matrix to identity; this will set the scale to 1:1, the rotation to 0, and the origin to (0,0). The LUC units remain the same.

Scaling to your Window (World Coordinates)

Often you want to match your units to your window. For example, suppose you want to draw a symbol which fills your window, and you have code to draw it 100 units across by 200 units tall. You can do this in a two-part operation. First you send msgDcUnitsWorld to your DC. Then, you must send msgDcScaleWorld to scale (100,200) up to match the current size of the window. Note that each unit in LUC will not necessarily be square. msgDcScaleWorld takes a SIZE32 as its message arguments, so in this example if you pass it (100,200), then (50,100) in LUC will be the center of your window (assuming you haven’t sent other coordinate transformation messages.)

To keep the world-scaled coordinates accurate, you must recalculate them whenever your window changes size. Because msgDcUnitsWorld does not reset the scaling matrix, you should normally send msgDcIdentity to the DC before sending msgDcScaleWorld a second time. Also, you should not set any of the
wsGrow window flags, or some pixels from the previous image will be preserved when the window changes size.

Transformation Matrices

You can get and set matrices corresponding to the transformations of scaling, rotating, translating, etc. Working with these matrices is more complicated than using transformation messages defined by clsSysDrwCtx, but is potentially faster than sending messages to your DC. Discussions of transformation matrices is available in most textbooks on computer graphics.

msgDcGetMatrix passes back a matrix which transforms LUC space into LWC space. Since LWC space uses pixel units, this transformation includes information about the device, its orientation, and the units (Points, TWIPS, etc.) You can use this matrix to generate coordinates for positioning windows instead of sending msgDcLUCtoLWC_RECT32, msgDcLUCtoLWC_XY32, and msgDcLUCtoLWC_SIZE32. There is no message to set this matrix—you must use msgDcScale, msgDcRotate, msgDcUnitsTwips, etc.

msgDcGetMatrixLUC returns a matrix which indicates transformations to LUC space as a result of translations, scaling, and rotation, including msgDcScaleWorld, but not msgDcUnitsPoints, msgDcUnitsMils, and other unit-specific scalings. The matrix is a concatenation of these transformations. For a default DC, or one which has been sent msgDcIdentity, the matrix is identity.

The LUC matrix represents transformations to an ideal, device-independent, first-quadrant coordinate system with device independent units established separately by the various msgDcUnits messages. You can set this matrix using msgDcSetMatrixLUC.

If you need to control a complex pipeline of transformations, you can save and restore the LUC matrix instead of sending the various scale/rotate/transform messages. You can also use the matrix manipulation functions in \PENPOINT\SDK\INC\GEO.H to modify this matrix directly.

Sending Window Messages to Drawing Contexts

Drawing contexts respond to all of the messages defined by clsWin. The drawing context passes the message on to the window to which it is bound. This lets you use the same device-independent coordinate system of your DC for window operations. For instance, clsSysDrwCtx responds to msgWinDelta, the clsWin message to position and/or resize a window. Since clsSysDrwCtx gives you an arbitrary coordinate system, it is inconvenient to have to convert DC coordinates into device coordinates in order to size and position windows. Instead, the DC behaves as if it were a window. You send it window messages but specify positions and sizes in DC coordinates (LUC); it translates the coordinates to logical window
coordinates (LWC), and sends the same messages with the transformed positions and sizes to the window to which it is bound.

Another example of the tight coupling between DCs and windows is `msgDcFillWindow`. The DC determines the size of the window to which it is bound, fills that window then strokes its border.

**Graphic State**

Many elements in the drawing context state affect drawing operations. These elements, collectively called the graphic state, and the messages you send to set them are described below. Color and text, because they are complex topics, are covered separately.

**Filling and Stroking**

Some drawing operations create a closed geometric figure which is stroked with a line and is also filled. The line is drawn with the line characteristics specified in the DC, and the enclosed region is filled with the current fill pattern in the DC. (The stroke and the fill do not overlap—a pixel is never touched twice by one drawing primitive.) If the line pattern is null but the line width is non-zero, the fill pattern still avoids where the surrounding line would be. If you want an area without an edge that is completely filled, you must set the line width to zero.

**Line Styles**

The DC specifies several characteristics for lines drawn:

- Thickness.
- End style.
- Miter limit.
- Rectangle radius.

You can also specify patterns for lines. You can tinker with the DrawingPaper™ application (on the SDK Goodies disk) to get a sense for the handling of these characteristics. Figure 26-1 shows some combinations.

**Figure 26-1**

**Line Styles**
If you want to get the current line state, send msgDcGetLine, specifying a pointer to a SYSDC_LINE structure as the message argument. msgDcGetLine returns the current line thickness; if all you want is the current line thickness, send msgDcGetLine with a message argument of pNull, and cast the return value from ObjectCall to a U16.

To change the line state, send msgDcSetLine. If you only want to set the line thickness you can send msgDcSetLineThickness. Both messages return the old line thickness.

Thickness and Modes

By default, line thickness is affected by the units and scale chosen. Thus the stroked outlines of figures and other lines get thicker as you scale up and thinner as you scale down. If the scaling is different in the x and y directions, the line thickness will be averaged. This is usually what you want, but not always:

- When you decrease the scale, at some point lines become invisible as their thickness drops below half a pixel (thickness rounds down to zero).
- As you change the scale, lines get fatter or thinner.

The former is bad when you want to keep lines from disappearing. The latter is bad when the lines should not change size, for example when drawing a grid or drawing annotation lines over a figure.

You can control both of these using the SysDC mode flags sysDcHoldDetail and sysDcHoldLine.

The sysDcHoldDetail mode flag stops lines from vanishing. You send msgDcHoldLine with message argument of true or false. Setting sysDcHoldLine using msgDcSetMode causes the current line thickness to be held so that it won’t be affected by subsequent scaling and units changes. You can use this to set a particular line thickness that won’t change as you scale your picture.

Sending msgDcHoldLine with an argument of False causes the window system to recompute the correct line thickness relative to the current scale and units.

Note that line thickness is independent of device pixel size, shape, and resolution. Note also that underline and strikeout lines drawn by the clsSysDrwCtxt text messages are unaffected by these line modes.

Line and Fill Patterns

The DC specifies a pattern for both lines and fills. The messages msgDcSetLinePat, msgDcGetLinePat, msgDcSetFillPat, and msgDcGetFillPat all take a SYSDC_PATTERN. This is a U16 which specifies a pattern mixing the foreground and background colors. The pattern determines which pixels will be set to the foreground color and which will be set to the background color. There are many predefined patterns of foreground and background in sysgraf.h, some set on the diagonal. The following screen shot shows some combinations.
**Fill Pattern Alignment**

ImagePoint uses *tiling* to fill closed figures. In other words, it maintains a small rectangle, or *tile*, of the fill pattern and copies it again and again to fill an area. Each tile lines up with the tiles adjacent to it to create a seamless, repeating fill pattern. To calculate the origin of a tile in the fill, ImagePoint measures from the LUC origin in multiples of the tile width and height. That way, if your application scrolls by translating the LUC origin (as most applications do), the fill pattern alignment will move along with all the other rendered objects.

If you determine for some reason that your application needs to scroll by actually moving rendered figures relative to the LUC origin, then the fill pattern alignment will not move with the figures. The result is that the pattern alignment will remain stable relative to the LUC origin, but will change relative to the outlines of the filled figures. If you need to keep the pattern alignment stable relative to the figures, you can move the pattern alignment origin along with the figures with `msgDcAlignPattern`. This will tile the pattern relative to the LUC coordinate you specify, rather than to the LUC origin.

**Determining Filled Areas**

For a simple figure such as an ellipse, it is obvious what the filled area should be. However, with a complex irregular polygon of crossing lines, it is not so obvious. There are two ways to determine what falls inside and should be filled. The default for `clsSysDrwCtx` is even/odd fill, but you can set the `sysDcWindingFill` flag in `SYSDC_MODE` with `msgDcSetMode` to use a winding fill calculation.

**Raster Operations**

`clsSysDrwCtx` departs from a true stencil/paint imaging model in that you can specify how new drawing operations combine with existing pixels. For any drawing operation, `clsSysDrwCtx` figures out the value of each affected pixel, and then in a separate step combines this value with the current value in that pixel. The way in which pixels combine is called the raster operation, or *raster op*. The default raster op is `sysDcRopCopy`, which replaces the destination pixel’s value with the computed (source) value; this is what you would expect a painting operation to do.

*GO strongly discourages the use of the raster op facility. It is documented here for completeness.*
Using msgDcSetRop, you can set the raster op to combine source and destination pixels in other ways. The possibilities are values of the enum SYSDC_ROP. They are named after the logical operations which they perform.

**Dynamic Drawing**

On displays more than one bit deep, it is difficult to predict graphic behavior with raster ops other than sysDcRopCopy. This difficulty makes raster op capabilities most useful for one-bit displays. The XOR raster op provides a non-destructive draw-erase sequence on a one-bit display, but is difficult to set up properly for displays deeper than one bit. Because the XOR behavior is very useful, PenPoint provides a **dynamic drawing** mechanism to achieve the XOR effect independent of display depth.

If you set the sysDcDrawDynamic flag in SYSDC_MODE, then the raster op is overridden to XOR and the foreground color is set up to XOR properly. This ensures that (if you draw figures using the foreground color) they will be visible no matter what is underneath; if you then repeat the same drawing messages, your drawing will disappear and leave the original pixel values restored. This means you don't have to save the contents of pixels when you draw over them; the only drawback is that colors drawn over others will be different (e.g. black lines on a black background are inverted). If you want to draw dynamically, use the sysDcDrawDynamic flag instead of only setting the raster op to XOR.

You can also set the sysDcDrawFast mode flag to reduce the time it takes to draw. This gives up certain ImagePoint features such as thick lines, patterns, and so on, in exchange for greater drawing speed. The precise set of features given up in sysDcDrawFast mode is device-dependent.

**Drawing Operations**

The actual set of drawing messages is quite small. It is by manipulating the DC state that you can achieve a wide variety of effects. There are two kinds of figures, open and closed.

**Open Figures**

Open figures are made up of lines drawn with the line pattern and line attributes.

- msgDcDrawPolyline draws a line made up of multiple segments. It takes a pointer to a SYSDC_POLYLINE, in which you specify a pointer to an array of XY32 points (points) and the number of points in the array (count).

- msgDcDrawBezier draws a single Bezier curve through four control points. It takes a pointer to an array of four XY32 points.

- msgDcDrawArcRays draws a single arc of an ellipse. It takes a pointer to a SYSDC_ARC_RAYS, in which you specify the enclosing rectangle of the ellipse, and two points. The arc drawn is the portion of the ellipse demarcated by rays from the center of the ellipse to the two points.
Closed Figures

These are drawn with a line, and filled. You can draw only the line by setting the fill pattern to sysDcPatNil (or by having a transparent fill pattern). You can fill without outlining by setting the line width to 0.

msgDcDrawRectangle draws a rectangle. It takes a pointer to a RECT32 which specifies the rectangle. You can change the radius in the line style to a non-zero value to get a rounded rectangle.

Note that the lines of a rectangle have thickness. msgDcDrawRectangle touches every screen pixel touched by the rectangle's coordinates in LUC. Contrast this to msgWinDirtyRect, which simply rounds the LUC coordinates to the nearest pixel. This can sometimes result in msgWinDirtyRect dirtying a rectangle that is smaller than the rectangle msgDcDrawRectangle draws, even when the specified coordinates are the same.

One way to avoid rectangle rounding problems is to convert LUC to LWC for msgWinDirtyRect coordinates, then to increase the size of the rectangle by one pixel in each direction.

msgDcDrawEllipse draws an ellipse. It takes a pointer to a RECT32 which specifies a rectangle enclosing the ellipse.

For both msgDcDrawRectangle and msgDcDrawEllipse, if the line thickness is non-zero, half of the line will fall outside the rectangle.

msgDcDrawPolygon draws a filled polygon. It takes a P_SYSDC_POLYGON, which is the same structure as the P_SYSDC_POLYLINE described above. It closes the polygon for you.

msgDcDrawSectorRays takes the same P_SYSDC_ARC_RAYS message arguments structure as msgDcDrawArcRays, and computes the arc of the ellipse edge to draw in the same manner. It creates a filled pie wedge shape by drawing lines to the center of the ellipse from the end points of the arc.

msgDcDrawChordRays takes the same P_SYSDC_ARC_RAYS message arguments structure as msgDcDrawSectorRays, and computes the arc of the ellipse edge to draw in the same manner. It creates a filled shape by drawing a line between the end points of the arc.

Filling a Window

msgDcFillWindow draws a rectangle exactly the same size as a window. If the line thickness is greater than zero, the lines will fall half within and half outside of the window. Thus, if you wanted to get a three pixel border around your window using this message, you must set the line width in the DC to six (assuming you sent msgDcUnitsDevice to set LUC to pixels).
Color

Class clsSysDrwCtx draws with a foreground/background color metaphor. Most drawing occurs in the foreground color, a few drawing operations draw in the foreground and background colors simultaneously. The background color can be transparent.

There are two ways to choose colors; as RGB (red, green, and blue) values, or as colors in a hardware-dependent palette. The former style is the default and is compatible with printers and future displays.

RGB Color Values

You can set the foreground and background colors to any color using msgDcSetForegroundRGB and msgDcSetBackgroundRGB; there are equivalent messages to get the colors. These take a pointer to a SYSDC_RGB structure specifying red, green, blue, and transparency values between zero and 255. The higher the value, the brighter the color component. ImagePoint will find the closest available color on the display device. The colors are automatically recalculated if you connect the DC to a different device.

Transparency can have only the extreme values of zero and 255. Use a transparency of 0 for opaque colors; a transparency of 255 leads to a transparent color (the red, green, and blue values are ignored). You cannot assign transparency to the foreground color.

The macro SysDcGrayRGB creates a SYSDC_RGB value for an opaque shade of gray from a single gray value.

Palette Colors

There is a separate mechanism to select a particular palette index, using msgDcSetForegroundColor and msgDcSetBackgroundColor. These take a 16-bit SYSDC_COLOR. On a four-bit display, all requested colors will map to the following predefined colors: black, white, light gray and dark gray (sysDcInkBlack, sysDcInkWhite, sysDcInkGray33 and sysDcInkGray66). There is also sysDcInkTransparent which stops the background color from painting anything (the foreground color cannot be transparent).

Palette colors are hardware-dependent. You should use RGB color specifications.

Inverting Colors

msgDcInvertColors exchanges the foreground and background colors. This works whether colors are specified as RGB or palette colors. If the background is transparent, msgDcInvertColors will cause an error.

Color Compatibility

Using RGB values to specify colors is compatible with printers. If you bind your DC to a different device, such as a printer, the DC will automatically recompute the closest match for the current foreground and background RGB values.
Using palette colors is marginally faster than specifying RGB values, but it is highly device-dependent. It is likely on a printer that the palette index of, say, black will not be zero. If you need to change between different foreground and background colors frequently, you can use `msgDcMatchRGB` to return the SYSDC COLOR palette indices which best match the RGB values you want to use, and then use `msgDcSetForegroundRGBColor` and `msgDcSetBackgroundColor` to switch colors. So long as you redo the matching when you bind the DC to a different device, this should be compatible with any printer.

**Planes**

A display device may support several bitmap planes in hardware. However, to support certain special effects a DC is normally limited to drawing into a subset of the hardware planes.

The DC is limited to drawing in certain planes by the `plane mask`. It is possible to change the planes you draw in by altering the plane mask, although you should generally avoid doing so because it can result in device-dependent code. `msgDcPlaneNormal` sets the plane mask to the normal drawing plane(s); this is the default.

**The Pen Plane**

`msgDcPlanePen` sets the plane mask to the plane(s) used for pen ink. This pen plane is the plane(s) where pen ink is usually dribbled by the pen tracking software. It is called the acetate layer because it acts like a transparent layer of acetate on which the pen tracking software lays down dribbles of ink as the user moves the pen, then erases the ink.

You should avoid using the pen plane unless your window has ink dribbling disabled (`inputInk` in its input flags). Otherwise your drawing in it will conflict with pen tracking. Even so, the pen tracking software in the input system uses the pen plane extensively. It will dribble ink in your window if the user starts writing with the pen tip down in another window which does enable pen tracking, and it will frequently erase the entire pen plane. As a general rule, it is best to adhere to the default of `msgDcPlaneNormal` unless you have compelling reasons not to.

**Other Planes**

`msgDcSetPlaneMask` gives you precise control over which planes which will be affected by drawing. This takes a SYSDC PLANE MASK (an unsigned number) as its argument. Each bit in the plane mask enables a particular plane. If you disable all planes (null value), no drawing takes place.

`msgDcPlaneNormal`, `msgDcPlanePen`, and `msgDcSetPlaneMask` all return the current plane mask (instead of STATUS) if they succeed.

Manipulating planes is inherently device-dependent, and may cause problems when you port to other PenPoint computers. Not all PenPoint machines will have the same number of planes or plane group divisions.
Sampled Images

Most ImagePoint drawing operations are device- and resolution-independent. You send a message to draw an idealized polyline, or filled ellipse, or text, and ImagePoint figures out which pixels to turn on in the destination window.

However, sometimes you have to deal with images which have already been broken down into pixels, such as scanner output. Or, you may want to work with pixels for performance reasons or to get the best visual appearance at low resolution. Such images are called sampled images rather than pixel images because the image may not be stored as a set of pixel values, but in some other format which produces a set of sample values.

ImagePoint provides several messages which deal with images made of samples.

- `msgDcDrawImage` reduces samples which you supply to gray values and renders the image in shades of gray, without regard for the current foreground and background colors.
- `msgDcDrawImageMask` reduces the samples which you supply to a bitmap of ones and zeros, which it renders in the current foreground and background colors.
- `msgDcCacheImage` renders an image into a chunk of memory so that it can later be quickly put on-screen.
- `msgDcCopyImage` copies a cached image into a window.

Related Messages

ImagePoint also has messages to copy pixels from one part of a window to another, described in Chapter 23, The Window Class.

Another technique to boost performance is to draw into an off-screen window, then copy pixels from this onto a visible window. Off-screen windows live on image devices, which are described in Chapter 24, Window Device Classes.

Sampled Image Operator

The image operator `msgDcDrawImage` draws an image from sampled image data (pixel values). You can use this to draw pixelmaps in windows. The TIFF object (clsTiff) uses `msgDcDrawImage` to draw TIFF images on screen.

`msgDcDrawImage` takes a pointer to a SYSDC_IMAGE_INFO structure as its message arguments. In this you specify:

- `dstRect` the size and position of the destination image.
- `srcSize` the number of source samples.
- `callback` an optional function or object to supply samples one row at a time.
- `pBuffer` a pointer to the sample data.
pClientData an optional pointer to client data to keep track of state if you use callbacks.

flags various bit flags, described below.

msgDcDrawImage can draw images from sample data at different resolutions and different numbers of bits per sample than the destination. Thus you can scale a sampled image to fill an area. The destination can either be a rectangle (dstRect) or the entire window (set sysDcImageFillWindow in flags).

For example, to display an image from 8-bit samples, you would set sysDcImage8BPS in flags, and pass data, one byte (eight bits) per sample, in which 0 represents black, and 255 represents white.

**Filter**

The filter flag controls how msgDcDrawImage filters the source samples when reducing the size of the image. The possible values are:

- sysDcImageNoFilter no filter, very low quality.
- sysDcImageLoFilter faster than sysDcImageHiFilter, but lower quality.
- sysDcImageHiFilter high-quality filter.

Use sysDcImageHiFilter for the most part. sysDcImageNoFilter favors speed over fidelity to the original.

**Run-Length Encoding**

Black and white (1 bit per sample) sampled images can be passed in a simple run-length encoded format to save space (sysDcImageRunLength). This encodes runs of black or white pixels, which usually results in space savings. Patterns of alternating black and white pixels may be larger when run-length encoded. msgDcDrawImage always converts black and white images to run-length encoding internally, so run-length encoding is always the fastest format to use for black and white images.

**Encoding Format**

The high bit of each byte indicates whether the run is black or white (0 is black, 1 is white). The other 7 bits is a count of the number of samples of that color. For example, a run of 300 white pixels would be represented by the hexadecimal values FF FF AC (FF hex=128 white, FF hex=128 white, AC hex=44 white).

The count is only for the current line—runs of samples don’t continue onto the next line.

**Drawing**

Sampled image drawing ignores the raster op, and the foreground and background colors. It looks directly at the windowing device’s palette to figure out what to put in each pixel in the destination window to generate colors matching those in the image. It assumes normal drawing planes.
**Call Backs**

You can either supply all the samples in `pBuffer`, or you can have `msgDcDrawImage` call you back to supply the samples. In the latter case you can either have it call a function you supply (set `sysDcImageCallBack`), or send `msgDcGetSrcRow` to an object (set `sysDcImageCallObject`). `ImagePoint` will ask your function or object to supply the next row of samples by passing it the `SYS_DC_IMAGE_INFO` structure. Your function or object passes back the samples in `pBuffer`; it can keep track of its state in `pClientData`. If `pBuffer` is null, that means that the sampled image operator does not in fact need the next line of samples.

**Rendering Colors**

`msgDcDrawImage` assumes that input sample values from 0 to the maximum sample value map to shades of gray increasing linearly from black to white. It ignores the current foreground and background colors when rendering, using only shades of gray. `msgDcDrawImageMask`, on the other hand, reduces the samples to ones and zeros. By default, `msgDcDrawImageMask` renders ones in the current foreground color and zeros in the current background color. If you set the `sysDcPolarityFalse` flag in the `SYS_DC_IMAGE_INFO` argument, `msgDcDrawImageMask` reverses its default color scheme, rendering ones in the current background color and zeros in the current foreground color.

**Image Model**

The representation of pixelmaps for sample image data is similar to what Postscript calls an image or a image mask. This is the representation that you should use to import and represent pixel images, and to draw small images (such as icons) on-screen. However, large areas of pixels in this representation cannot be rendered quickly. To create and manipulate large pixelmaps, you should create an image device and maintain windows in its off-screen memory.

**Cached Images**

When you use `msgDcDrawImage` to draw sampled images on-screen, `ImagePoint` handles the current LUC transformations, different device resolutions, different pixel aspect ratios, different number of planes, dithering, etc. `msgDcDrawImage` is powerful (see the Fax Viewer), but it's somewhat slow (see the Fax Viewer). If you need to draw a sampled image repeatedly on the same device at the same scale, what you would like to do is go through all these transformations once to arrive at the desired pixelmap, and then copy this pixelmap to the screen.

`ImagePoint` supports this. You use `msgDcCacheImage` to load a sampled image into a pixelmap in advance. The resulting pixelmap is called a **cached image**. You use `msgDcCopyImage` to quickly copy the pixels in the cached image to a window. `msgDcCopyImage` ignores most transformation and device issues and puts the pixels in a cached image straight into the DC's window. You can think of
msgDcCacheImage as converting a sampled image which draws slowly into a
cached image which draws fast.

Cache is somewhat of a misnomer. There is no cache of these images, each is
simply allocated by ImagePoint and it is up to the client to manage each one. In
other words, it is up to you to cache your images.

A cached image is a rectangular pixmap, but it can maintain an associated mask
which controls which parts of the image appear. This is used for icons and cursors,
which do not have rectangular appearances. A cached image also has a hot spot,
which specifies the origin of the image. The move icon uses the hot spot to appear
with its center over the pen tip.

The Bitmap Editor creates bitmaps which you can load as cached images. It is
described in the PenPoint Development Tools volume.

Creating Cached Images

msgDcCacheImage creates a cached image from the sampled image you specify.
It takes a SYSDC_CACHE_IMAGE structure for its message arguments. In this you
specify:

- Image an array of two SYSDC_IMAGE_INFO structures, one for the image
  and one for its optional mask.
- hasMask a Boolean indicating if the image has a mask.
- hotSpot the hot spot's coordinates relative to the lower-left of the image.
- pCache ImagePoint passes back a pointer to the cached image.

The image array contains the information for an image, and if hasMask is true,
the information for the image's mask as well. msgDcCacheImage shares
SYSDC_IMAGE_INFO with msgDcDrawImage.

ImagePoint renders the sampled image into a pixmap which it allocates. The
pixmap is compatible with the windowing device of the window to which the
DC is bound. ImagePoint uses msgDcDrawImage to generate the image, so the
sampled image is scaled and sampled. The hotSpot is interpreted in LUC at this
point.

Drawing a Cached Image

To get the cached image to appear in a window, send msgDcCopyImage. This
takes a pointer to a SYS_DC_COPY_IMAGE structure for its message arguments. In
this you specify:

- xy the location in LUC where the cached image's hot spot should be drawn.
- pCache a pointer to the cached image.

pCache is the same pointer passed back from msgDcCacheImage. ImagePoint
draws the cached image so that its hotSpot is at xy in LUC.
Mask
The mask controls which pixels in the destination are affected. Where pixels in the mask are zero (black), the pixels in the destination are not affected. Only where pixels in the mask are 1 (white) does something appear. If there is no mask, the entire pixmap of the cached image appears. Of course, the drawing is still affected by the current update region and clipping region.

As with other sampled image operations, msgDcCopylmage ignores the current foreground and background colors, and current raster op.

Invalidating a Cached Image
Cached images are valid for the current windowing device. When you create a cached image with msgDcCacheImage, the current LUC, windowing device pixel aspect ratio, orientation, plane depth, etc. are taken into account. It is up to you to detect that the windowing device has changed (for example, the user changes from portrait to landscape) and to free (with OSHeapBlockFree()) and then recreate the cached image. Also, if your LUC transformation changes, and you want the cached image to scale, you should free and re-create it.

Cached Images vs Image Devices
The only thing you can render into a cached image is a sampled image. If you want to draw other types of figures (text, for example) into an off-screen image, you must use an image device. If all you want to do is put some sampled pixels on the screen, you can use cached images.

Related Classes
clsBitmap stores samples; it includes a message to load a cache from its information. clsTiff is an object you associated with a sampled image in a TIFF file; you can send it a message to draw its contents. See Chapter 28, Bitmaps and TIFFs for more information on clsBitmap and clsTiff.

Printing
Printers are represented as objects of clsWinDev and are similar to the Screen device. Drawing on a printer can be exactly the same as drawing on-screen.

Since printers are clsWinDev devices, you can either draw on the root window of the printer (the page), or you can create child windows on it. The support for windows on printing devices may seem curious. It exists because windows can be used as drawing tools (viewports), thus, it is possible to create images using several windows that could not be created otherwise. While printer windows can be moved and resized, there is no real point in doing so beyond the usual layout episode.
Application Printing

When the user prints one of your application’s documents, a copy of the documented will be created in the Outbox in a queue for the current printer. When the PenPoint computer is connected to that printer, the document will be started up with its application mainWin inserted as a child window on the printer.

The printing framework will send the application msgWinRepaint. If your application creates a standard set of windows within its frame and paints them in a device-independent way in response to msgWinRepaint, then it should print its window on the printer without any further work. The same arrangement of windows will appear on the page as on the screen. Because of memory constraints, the print device may ask each window to paint itself several times in order to produce the page in a series of bands; thus like all windows, printer windows must respond to msgWinRepaint on demand.

If you want to print a window differently on the printer, your window subclass can intercept msgWinStartPage. This is sent to each window advising it that it is on a printer (as opposed to the screen) and that printing is about to commence. Your window class could set an instance variable to false at creation, set it to true if it receives msgWinStartPage, and when responding to msgWinRepaint, it would check this variable and modify drawing accordingly.

ImagePoint Font Support

Chapter 25, Graphics Concepts, describes some of the features of the ImagePoint font model. This section explains the programming interface to the model.

How You Use the API

Hello World is a sample program that draws text in its window. It is described in the PenPoint Application Writing Guide; its source code is in the SDK sample code directory \PENPOINT\SDK\SAMPLE\HELLO.

Here are the steps involved in drawing text:

1. Send msgDcOpenFont to your DC, specifying the attributes of the font you want to use (Modern Roman Bold Italic, for example).
2. Send msgDcScaleFont to your DC to scale the font to your desired size.
3. Send msgDcDrawText to your DC to output a text string.

Text and the Drawing Context

Just as your drawing context includes the current line width, line style, fill pattern, etc., it also includes the current font and current font scale. When you save and restore the graphics state with msgDcPush and msgDcPop, you save and restore this font information. If all you are doing is changing to a different font, you can use msgDcPushFont and msgDcPopFont to save and restore only the font information.
What is a Font?

Fonts exist as files produced by the PenPoint font editor FEDIT. At cold-boot time PenPoint loads the fonts in `\BOOT\PENPOINT\FONT` into memory; later on the user can install and remove fonts using the Installer. Thus different computers have different sets of fonts installed.

Font IDs and Font Names

Each font file contains the instructions to create a specific font—a typeface with a particular look. Each font has a unique 16-bit ID which GO assigns, so that a font ID refers to the same font on all PenPoint computers for all time.

The font ID is a condensed version of a four-character string used to identify fonts. The function `SysDcFontID` performs this algorithm, returning a 16-bit font ID of the four-character string you pass it. Conversely, `SysDcFontString` takes a pointer to a buffer and a 16-bit font ID, and passes back a null-terminated string.

As an example of a short font name, the font string of GO's Sans Serif is HE55. By convention, the DOS font file name is this four-character string with the extension `.PCK`. There is no length checking on the buffer; it must be at least 5 bytes long to hold the string.

The user-visible name of the font is the PenPoint file name of the font; in this example, SANS SERIF (URW).

Font Attributes

Each font has attributes which indicate its visual appearance. These include:

- **group** the typeface group (Venetian, Old style, Transitional, Script, etc.).
- **weight** the font weight (light, normal, bold, extra-bold, etc.).
- **aspect** the font aspect ratio (condensed, normal, extended, etc.).
- **angle** the font obliqueness (whether it is italic or not).

The idea is that the font attributes describe the style of the font in great detail. Note that, unlike in some other window systems, the size of the font is not part of its attributes.

Opening a Font

You might expect to open a font simply by supplying its ID. However, different PenPoint computers will have different fonts on them, and users can load and remove fonts. There's no guarantee that the font ID which you want to use exists on a computer.

So you also specify a desired set of font attributes, and ImagePoint will pick the closest font matching those attributes. It can also *synthesize* some font attributes from an existing font. For example, it can create a bold or italic version of a font.

You specify both a font ID and the attributes you desire at the same time that you send `msgDcOpenFont`. If the font ID you request exists on the computer,
ImagePoint will use it (even if it doesn't remotely match the attributes you've supplied). If the font ID is not available, ImagePoint finds the font which matches most closely the attributes which you specified.

In both cases, ImagePoint then transforms the font it's using to fit the attributes you have given. If the attributes match those of the font, this step doesn't involve any work.

This scheme guarantees that the user will get the closest match possible, no matter what fonts are installed on his or her PenPoint computer. However, it has some important programming implications which you need to be aware of:

- If the font ID you request is available and you supply font attributes which do not match it, ImagePoint will alter the font to try to fit the attributes.
- If the font ID is not installed, ImagePoint does not know what kind of font it is, so you must always supply the group attribute.

If you're not too concerned about which font to use, you can leave the font ID field in SYSDC_FONT_METRICS as 0 and only specify the group. ImagePoint will find (and if necessary, forge) a font close to what you want. The essential point is, you must always supply font attributes.

**User Choice of Font**

You can avoid the difficulty of specifying a font altogether by letting the user pick fonts. This also avoids limiting your application to a fixed set of fonts. The UI Toolkit provides two components for this purpose, font list boxes and pop-up font choices. Both of these display descriptive names of the installed fonts. You can control whether they display non-alphanumeric fonts such as the GO symbol and gesture fonts. See Part 4: UI Toolkit for more information.

**Font Enumeration**

You can find out yourself what fonts are installed by querying the installed font manager, described in Part 12: Installation.

**Storing Fonts**

The combined font ID (a U16) and attributes (a SYSDC_FONT_ATTR structure) form a 32-bit quantity (a SYSDC_FONT_SPEC). You can file this with text to indicate its formatting, and on subsequent occasions just pass the stored value to msgDcOpenFont; you are guaranteed to get the best match possible.

**Font IDs and Font Strings**

Font ID numbers are not random. A font ID is an encoded version of a four-character font string, such as HESS. The two letters identify the font family, while the ending two digits indicate the type of the font (extra-bold, oblique, symbol, etc.). Unless you are creating fonts with GO's font editor, you don't need to know the details of the font encoding scheme. Always refer to a font in source
code by its four-character name, and use SysDcFontId to convert this to the id required by msgDcOpenFont.

**Font Attributes**

The attributes of a font are in a SYSDC_FONT_ATTR structure. This contains:

- **group**: the type of the font.
- **weight**: how bold the font is.
- **aspect**: how wide or tall the characters in it appear.
- **angle**: whether the font is italic.
- **monospaced**: whether the font is monospaced.
- **encoding**: the type of encoding map to use.

**Group**

The group of a font is its broad classification; it categorizes a font according to whether it looks like a newspaper headline, or writing in a book, or handwriting, etc. The font group names are:

- sysDcGroupDefault
- sysDcGroupUserInput
- sysDcGroupVenetian
- sysDcGroupOldStyle
- sysDcGroupTransitional
- sysDcGroupModernRoman
- sysDcGroupEgyptian
- sysDcGroupSansSerif
- sysDcGroupDisplayRoman
- sysDcGroupScript
- sysDcGroupGraphic
- sysDcGroupTypewriter

Many monospace fonts are part of the sysDcGroupTypewriter category, irrespective of the appearance of their glyphs.

**Default Font**

If you want to use the same font that is used in the Notebook, you can specify the attr.group in the SYSDC_FONT_SPEC as sysDcGroupDefault. This is a pseudo-group which the font machinery maps to the current system font. Similarly, sysDcGroupUserInput is a group which maps to the font for user input. Note that the binding is set at msgDcFontOpen. Thus if the system font changes, your DC will still be bound to the old system font until another msgDcFontOpen. Applications which want to respond to changes in the default font need to observe the system preferences (theSystemPreferences).
The system and user input fonts and their sizes are system preferences which the user can change using the Preferences application on the Bookshelf. Thus you can also find out the system font by sending msgResReadID to theSystemPreferences. This returns a SYSDC_FONT_SPEC which you pass to msgDcFontOpen.

**Weight**

The weight of a font indicates the thickness of its characters’ strokes. The possible values are:

- sysDcWeightLight
- sysDcWeightNormal
- sysDcWeightBold
- sysDcWeightExtraBold

**Aspect Ratio**

The aspect ratio of a font indicates the height to width ratio of the font; roughly speaking, this determines how tall or wide the characters in it appear. The possible values are:

- sysDcAspectCondensed
- sysDcAspectNormal
- sysDcAspectExtended

**Other Attributes**

An italic font often has slanted (oblique) characters.

The characters in a monospaced font all have the same width.

**Common Fonts**

There is no default font set in PenPoint. The user can always remove or add fonts. However, there is a standard set of fonts on the PenPoint boot disks:

<table>
<thead>
<tr>
<th>PenPoint Name</th>
<th>Short Name</th>
<th>Group</th>
<th>User-Visible Fonts</th>
<th>System Fonts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Courier</td>
<td>CR65</td>
<td>sysDcGroupTypewriter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sans Serif</td>
<td>HE55</td>
<td>sysDcGroupSansSerif</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Roman</td>
<td>TR55</td>
<td>sysDcGroupTransitional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gestures</td>
<td>GS80</td>
<td>none applicable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dingbats</td>
<td>PI80</td>
<td>sysDcGroupGraphic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Symbols</td>
<td>GO55</td>
<td>none applicable</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
All the user fonts have normal weights, and aspects, except that Courier is sysDcWeightBold and is monospace.

**Encoding**

Encoding is not a property of the font, it's an indication to ImagePoint of how you are going to use the font. The encoding maps the 8-bit characters in your text strings to the glyphs in the font. In one encoding, character 165 in a text string is the Yen currency symbol (¥), in another it is the Ntilde symbol (ヌ).

The possible font encodings are:

- sysDcEncodeLinear
- sysDcEncodeAdobeStandard
- sysDcEncodeAdobeSymbol
- sysDcEncodeIBM850
- sysDcEncodeGoSystem
- sysDcEncodeHWX850
- sysDcEncodeUnicode

Some of the font encodings are documented in the Font Editor chapter of *PenPoint Development Tools* manual.

**Transforming Fonts**

ImagePoint can transform an existing font as necessary to match most of the requested attributes you pass in to msgDcFontOpen. It will thicken a font’s characters to create a boldface weight font, expand or condense a font’s characters and character spacing to the requested aspect, slant a font’s characters if italic is set, and make all the font’s character widths the same if monospaced is set.

ImagePoint will not attempt to change a font so that its appearance is more like the group specified.

The encoding of a font is not part of its definition, it is just something you specify when you open the font. ImagePoint always honors it.

**Font Metrics**

After opening a font, you can draw text with it. Often to do this you need to know additional information about the font, such as how wide a space is in it, how much its characters go over or under the base line, and so on. msgDcGetFontMetrics returns information about the current font in LUC in a SYSDC_FONT_METRICS structure.

SYSDC_FONT_METRICS contains a SYSDC_FONT_ATTR structure holding the font ID and attributes of the font that ImagePoint actually uses to honor your msgDcFontOpen request. If you asked for certain attributes, and ImagePoint had to use another font or synthesize the font, the font attributes returned are the natural attributes of the font it’s using. For example, if you ask for Roman condensed (sysDcAspectCondensed), and only Roman normal is installed, then the aspect attribute passed back by msgDcGetFontMetrics will be sysDcAspectNormal.
even though ImagePoint will synthesize a condensed font. SYS_DC_FONT_METRICS also contains additional information:

- **name** the name of the font (up to 80 characters).
- **spaceWidth** the width of a space in the font.
- **underThickness** the best thickness to use for an underline.
- **underPos** the best distance from the baseline to use for an underline.
- **xPos** the height of a lowercase x in the font.
- **ascenderPos** the ascender height for the font.
- **descenderPos** the descender depth for the font.
- **em** the em size for the font.
- **maxY** the highest coordinate touched by any character in the font (relative to the baseline).
- **minY** the lowest coordinate touched by any character in the font (relative to the baseline).

All the dimensions are passed back rounded to the nearest whole number in LUC.

To do precise work with font metrics, you need to have small enough units and large enough font scale in effect. **msgDcUnitsTwips** is a good choice (a twip is one twentieth of a point).

Also, all the dimensions are for the synthesized font. To use the same example, if you ask for condensed and get a regular font, the character widths passed back will be the narrower ones which ImagePoint synthesizes.

**Name**

This is the font identification for ownership and copyright purposes. It is not the name displayed to the user, that name is the PenPoint name of the font file.

**Character Geometry**

The rest of the parameters are best shown graphically.
The values for these dimensions are for the current x and y scaling of the current font, and are in the current LUC (Logical Unit Coordinates) of your DC. Note that they are integers, hence are subject to gross round-off errors at small scales and coarse units.

\textbf{descenderPos}, \textbf{minY}, and \textbf{underPos} are negative numbers.

\textbf{ascenderPos} and \textbf{descenderPos} give the typographic line height. They indicate the guidelines the font designer used in designing the font (similar to \textbf{xHeight}). You might use these to draw a box around characters, or to adjust the gap between lines, or \textbf{leading}, with different fonts on them.

\textbf{maxY} and \textbf{minY} give the height of the tallest character and the depth of the deepest character in the font. You can use these to compute which lines of text fall in an update region in order to minimize repaint.

\textbf{underPos} and \textbf{underThickness} indicate the best position and thickness for underlines for characters. If you specify an underline when you draw text, \textbf{ImagePoint} positions and sizes it using this information.

Note that the locations of \textbf{xPos}, \textbf{ascenderPos}, and \textbf{descenderPos} are aesthetic judgements of the font designer, as are \textbf{underThickness} and \textbf{underPos}.

The \textbf{em} is a \textbf{SIZE16} that indicates what the font is scaled to in the current LUC and font scale. In other words, if you scale a font to 18 and scale LUC by 2 and 3, \textbf{em.w} would be 36 and \textbf{em.y} would be 54. It's useful if you haven't been keeping track of your LUC transformations and font scale.

\section*{Ems and Text Widths}

An em is a square whose sides are, roughly speaking, the distance from the lowest descender to the highest ascender in the font. This is often approximately the width of an M in the font, hence the name. Em size is the scale of the font: if the font is 14 point, then \textbf{em.h} and \textbf{em.w} are both 14 with a default DC. Thus when you open a font with a default DC, its \textbf{em.w} and \textbf{em.h} are both 1.

As a rule of thumb, the width of a string 20 characters long is about 10 em widths in many fonts. Ten characters are almost guaranteed to fit into a space 10 em-widths wide (or 80 layout units in the UI Toolkit).

\section*{Character Widths}

The rules of thumb relating the em-width of a font to text widths are useful to figure out roughly how wide to make a text region, but if you really care you should add up individual character widths. \textbf{msgDcGetFontWidths} helps you do the former; it returns the widths of all 256 characters in the current font (based on the current encoding specified in the font attributes). These widths are the designer's guidelines for the visual widths of each character; it's possible for a character's pixels to extend beyond its widths (for example the top of an italic T might extend beyond its nominal width.)
The character widths are constants and don’t take into account how particular letter combinations look better closer or further apart than their nominal widths would indicate. Adjusting character combinations is called **kerning**. ImagePoint does not currently support automatic kerning.

`msgDcGetFontWidths` will not work well with future large character sets (such as Unicode), so it’s better to use `msgDcMeasureText` or `msgDcGetCharMetrics`.

### Character Metrics

You can get width information about particular characters in the current font by sending `msgDcGetCharMetrics`. This message takes a pointer to a `SYSDC_CHAR_METRICS` structure. You supply a pointer to a text string (`pText`), and the length of it (`len`). Note that the string can have nulls in it, since null may be a valid character. ImagePoint passes back a set of text extents (`pExtents`). An extent is a `SYSDC_EXTENTS16` structure which gives the minimum and maximum x-y locations of each character (min and max, both XY16). The x and y coordinates are for each individual character relative to the origin—the characters aren’t rendered next to each other.

Unlike `msgDcGetFontWidths`, which passes back the type designer’s aesthetic sense of character widths, `msgDcGetCharMetrics` passes back the actual bounding box of each character. Thus it’s possible for any of the coordinates (min.x, max.x, min.y, and max.y) to be positive or negative, depending on how the character is designed. For example, an italic g might be designed to run slightly underneath the character to its left.

### Scaling a Font

When you open a font, ImagePoint sets it up so that the point size of the font (its **em height** is one unit in your DC’s current logical unit coordinates (LUC). Hence the size of the current font is affected by the units and scaling of the DC.

You send `msgDcScaleFont` to your DC to scale fonts separately from other drawing operations. For example, if you have sent `msgDcUnitsPoints` to set your DC’s coordinates to points, and you then send `msgDcScaleFont` with x and y of 12, you will set the font size to be 12 points with a normal aspect ratio. If you change the DC scale with `msgDcScale`, the font size will be affected. The font scaling is concatenated with the DC scaling.

Moreover, `msgDcScaleFont` is cumulative. In the previous example, if you sent `msgDcScaleFont` a second time with x and y of 9, you would end up drawing 108-point characters, not 9-point characters.

The message `msgDcIdentityFont` resets the font matrix scale to the default of 1 unit in LUC. Note that `msgDcIdentity` changes LUC (and thereby the size of characters), but doesn’t change the font scale itself.

If your scale is very small, obviously characters will be hard to read. Also, if you use radically different x and y scales for your LUC and you do not adjust for them...
with `msgDcScaleFont`, your text will appear stretched or squashed. Sometimes it’s appropriate to maintain two drawing contexts; one for graphics and the other for pleasing text.

**Drawing Text**

You send `msgDcDrawText` to your DC to draw text. `ImagePoint` renders your text by filling the pixels of characters with the current foreground color, using the current raster op. You can set foreground color with `msgDcSetForegroundColor` (the default foreground color is `sysDcRGBBlack`). Characters are not filled with the current fill pattern, only the current foreground color.

`msgDcDrawText` takes a `SYSDC_TEXT_OUTPUT` structure as its message arguments. You must initialize the structure before drawing text.

```c
typedef struct
{
    U16          alignChr;  // use sysDcAlignChr...
    U16          underline; // use 0,1, or 2
    U16          strikeout; // use 0 or 1
    P_U8         pText;
    U16          lenText;   // in (and out for measure)
    XY32         cp;        // in and out, where to place string
    COORD32      stop;      // used by msgDcMeasureText
    U16          spaceChar; // code for space, usually 32
    COORD16      spaceExtra; // added to width of space
    otherExtra;  // added to width of every char
} SYSDC_TEXT_OUTPUT, FAR * P_SYSDC_TEXT_OUTPUT;
```

`msgDcDrawText` draws the text pointed to by `pText` starting at the location `cp` in LUC. You can make the text appear underlined or thickly underlined by setting `underline`, and can make it appear struck out by setting `strikeout`. The size of the underline and strike-out rules comes from the font metrics. It is independent of the current line style in the DC. Spaces between underlined or stuck out words are underlined and struck out.

You specify the number of characters to draw in `lenText`. Characters are drawn from left to right with respect to LUC. You can can make text appear at an angle by rotating your DC. Characters can be null, so you must specify `lenText`.

`cp` is the starting point for the text.

`alignChr` determines where characters are positioned vertically relative to the origin specified in `cp`. The possible alignments are:

- `sysDcAlignChrTop` align the ascenderPos of the font with the origin.
- `sysDcAlignChrCenter` align the midpoint of the x height of the font (xPos) with the origin (this is not the same as half the height of an uppercase letter).
- `sysDcAlignChrBaseline` align the baseline of the font on the origin.
- `sysDcAlignChrDescender` align the descenderPos of the font on the origin.

The default alignment is `sysDcAlignChrBaseline`. ImagePoint fonts are outlines, but you can’t stroke a single glyph outline or fill it with patterns as you can in PostScript.
Characters in Text

Text is composed of encoded characters. The mapping from an encoded character to a glyph in the font is determined by the value you specified for the encoding field when you opened the current font. Most encodings have the same glyphs for the standard typewriter ASCII characters, but different encodings produce different glyphs for “control characters” and non-English characters.

PenPoint is migrating to the Unicode universal encoding standard. PenPoint 2.0 will use the 16-bit Unicode encoding standard for all characters in use around the world.

The space character has a width. Displaying a space advances the current point by the font’s space width. There are no other carriage control codes in text. ASCII codes such as tab, newline, carriage return, and form feed may have characters in them, but none of them move the current position in any other way. Even character 0 may have a glyph (for example, the happy face in the IBM 850 encoding).

Spacing Text

ImagePoint uses the character widths in the font to determine how much to advance between characters. otherExtra is an additional amount of space which it inserts between each character. spaceExtra is an additional amount of space which it will add to the space character. You could use these to spread out or shrink lines of text to produce justified right margins; using otherExtra stretches out every character, while spaceExtra only stretches out the gaps between words.

Moreover, spaceChar indicates the character in pText to which spaceExtra will be added.

otherExtra and spaceExtra can be positive (to expand space) or negative (to tighten letter spacing).

Measuring Text

You can ask for the widths of the characters in the current font with msgDcGetFontWidths or msgDcCharMetrics, and compute the width of a text string using this. ImagePoint computes character locations for text strings in LUC units.

When you send msgDcDrawText, you pass in the starting point for the text in cp. Upon return, cp contains the location of the next character to draw. You can use this to determine more exactly the length of a string after you’ve drawn it.

If you want to compute the length of a string before drawing it, use msgDcMeasureText. This takes the same SYSDC_TEXT_OUTPUT message argument structure as msgDcDrawText. However, it draws nothing and only returns the new cp value.
Determining How Much Text Fits

A related problem to computing the width of text is computing how much text fits in a line. For `msgDcMeasureText`, you can supply a stop coordinate indicating the point beyond which text should not appear. This is in effect the right margin for the text. Upon return from sending `msgDcMeasureText` to your DC, `lenText` will indicate the offset of the last character which entirely fits in the space available. `cp` will be the ending coordinate of the part of the string up to `lenText` which does fit. You can then send `msgDcDrawText` with the original `cp` position and new `lenText` value, and know that the text will not overshoot the right margin.

`msgDcMeasureText` doesn't handle the space character specially; if you don't want spaces at the start of the new line, you must check yourself to see if the character(s) after the line break are spaces. `msgDcMeasureText` is identical to `msgDcDrawText` except that the former doesn't draw and the latter does not use the stop message argument.

Splicing Text

`msgDrawText` and `msgMeasureText` do not include the optional `otherExtra` spacing after the last character. This means that isolated lines of text will have cosmetically correct lengths, and also that underlines and strikeout lines will not extend beyond the last character. However, if you want to splice text together to make continuous runs of characters, use the equivalent messages `msgDcDrawTextRun` and `msgDcMeasureTextRun`. Text views use these when repainting so that they can redraw part of a line of text without leaving any gaps.

Finding a Glyph

When you issue a command to draw or measure text, ImagePoint has to find the glyphs. The meaning of the bytes in the text string depends on the font encoding you specified when you opened the font.

In total, all the encodings call for hundreds of different glyphs, far more than 256. All the standard glyphs have a well-known unique number, the same in all fonts. This allows a font to have all the glyphs in the standard encodings in it. Not all characters for each encoding are present in all fonts; for example, the copyright symbol is not present in the Roman font. However, ImagePoint will search in its system fonts for the missing characters.

Font Search Path

ImagePoint searches for missing glyphs using a standard path. If it can't find the missing character, it tries to find it in Sans Serif, Courier, and Symbol. If it still can't find the character, it displays the missing symbol glyph (a square with a white diamond in it).

The font search mechanism is very useful. By selecting the appropriate encoding you can open a single font, draw text with glyphs that aren't in that font, and rely on the font machinery to find those characters.
It's also nice when you create your own font, since it only needs to contain the few characters you want to handle specially. The font machinery will find any characters missing from your font: they may not blend aesthetically with your font, but they won't be missing.

**Metrics and Encoding**

The font metrics passed back by `msgDcGetFontMetrics` do not take into account the sizes of characters which will be substituted from other fonts. For example, if you open Sans Serif using the `sysDcEncodeIBM850` encoding and ask for the font metrics, some of the characters will be very tall and deep on account of the line draw characters, but this won't be accounted for by the metrics because those characters are only available in a monospaced font.

However, `msgDcGetFontWidths` and `msgDcGetCharMetrics` do take into account the characters that are substituted from the font search path.

**Loading the Font Cache**

ImagePoint creates bitmaps of glyphs at the right size and with the right attributes from font outlines. After rendering a bitmap, the font machinery keeps the bitmap in a font cache.

The time it takes to create a bitmap from an outline is noticeable: the time to draw a character from the font cache is much smaller. If you want to get text to appear without delay, you can preload the characters in the font cache. For example, in a presentation program, you might be willing to have a delay between prepared images, but then want text to draw quickly onto the screen.

You do this by sending `msgDcPreloadText` to your DC. `msgDcPreloadText` takes the same `SYSDC_TEXT_OUTPUT` structure as `msgDcDrawText`. It prepares the text and loads it into the cache, but it doesn't render the characters (nor does it compute the new starting point). If the message arguments or `pText` are null, it preloads a default set of upper- and lowercase Latin characters.

If PenPoint runs low on memory, it discards the font cache. The font machinery will allocate a new font cache the next time it needs to render a character.

**Bitmap Fonts**

Some fonts also have bitmaps of characters for some low resolutions. If a bitmap is available at the right size, ImagePoint will use it instead of the outline. The font machinery still renders the bitmap into the cache.

You can use FEDIT, the PenPoint Font Editor, to add bitmaps to an outline file. You should not modify the built-in fonts. If you're interested in acquiring or modifying other commercial fonts for PenPoint, contact GO Developer Support for more information.

---

26.12.15.1
Improving Performance

Font State

If you need to switch between different fonts frequently, you can use msgDcPushFont and msgDcPopFont. These are similar to msgDcPush and msgDcPop, except they get and restore only the font state (current font and font scale). Open one font, establish the font scale, then store the font state by declaring a SYSDC_FONT_STATE structure and passing its address to msgDcPushFont. Repeat this for the other fonts and sizes needed, and then use msgDcPopFont to switch to a font. This is faster than sending msgDcOpenFont.
Chapter 27 / The Picture Segment Class

When you draw in a window, you send a stream of messages to its drawing context telling it what to draw. Often, to repaint or print your window’s contents, you must send the same stream of messages. Picture segments are drawing contexts which remember the drawing messages they receive, thus saving you from much of the burden of redrawing. Picture segments also let you reorder or change their elements. You can store a picture segment for later use, and the same picture segment can be drawn into many different windows.

`clsPicSeg` inherits from `clsSysDrwCtx`. Thus you bind a picture segment to a window and draw in it exactly as you would with a DC. The picture segment stores certain drawing messages it receives.

The unbundled DrawingPaper™ application uses a component that uses a picture segment to store and redisplay graphic objects, and also to perform some editing operations. The Fax Viewer, another unbundled application, uses a picture segment to draw the FAX image and the user’s notations. The FAX image itself in a separate TIFF file maintained by a separate TIFF object, which the picture segment asks to paint. Table 27-1 summarizes the messages `clsPicSeg` defines.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_PIC_SEG_NEW</td>
<td>Initializes a PIC_SEG_NEW structure to default values.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_PIC_SEG_NEW</td>
<td>Creates a new picture segment.</td>
</tr>
<tr>
<td>msgPicSegGetMetrics</td>
<td>P_PIC_SEG_METRICS</td>
<td>Passes back the metrics of the picture segment.</td>
</tr>
<tr>
<td>msgPicSegSetMetrics</td>
<td>P_PIC_SEG_METRICS</td>
<td>Sets the metrics of the picture segment.</td>
</tr>
<tr>
<td>msgPicSegSetFlags</td>
<td>S32</td>
<td>Sets the picture segment flags.</td>
</tr>
<tr>
<td>msgPicSegGetFlags</td>
<td>P_S32</td>
<td>Gets the picture segment flags.</td>
</tr>
<tr>
<td>msgPicSegDelta</td>
<td>P_PIC_SEG_GRAFIC</td>
<td>Changes the current grafic.</td>
</tr>
<tr>
<td>msgPicSegGetGrafic</td>
<td>P_PIC_SEG_GRAFIC</td>
<td>Gets the current grafic.</td>
</tr>
<tr>
<td>msgPicSegSetCurrent</td>
<td>S32</td>
<td>Sets the current grafic index.</td>
</tr>
<tr>
<td>msgPicSegGetCurrent</td>
<td>P_S32</td>
<td>Gets the index of the current grafic.</td>
</tr>
<tr>
<td>msgPicSegGetCount</td>
<td>P_S32</td>
<td>Gets the number of grafics in the picture segment.</td>
</tr>
<tr>
<td>msgPicSegSizeof</td>
<td>P_PIC_SEG_GRAFIC</td>
<td>Returns the size of the specified grafic’s pData, in bytes.</td>
</tr>
</tbody>
</table>

Table 27-1  
`clsPicSeg` Messages

continued
### Table 27-1 (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgPicSegScaleUnits</td>
<td>MESSAGE</td>
<td>Scales all coordinates in the picture segment from the old units to the new units, then sets the units of the picture segment to the new units.</td>
</tr>
<tr>
<td>msgPicSegTransform</td>
<td>MAT</td>
<td>Transforms all coordinates in the picture segment database with the provided matrix.</td>
</tr>
<tr>
<td>msgPicSegPaint</td>
<td>pNull</td>
<td>Redraws all of the graphics in the picture segment.</td>
</tr>
<tr>
<td>msgPicSegDrawSpline</td>
<td>P_PIC_SEG_SPLINE</td>
<td>Adds a spline graphic to the end of the picture segment display list and draws it by sending msgPicSegPaintObject.</td>
</tr>
<tr>
<td>msgPicSegDrawObject</td>
<td>P_PIC_SEG_OBJECT</td>
<td>Adds an arbitrary object to the picture segment display list and draws it by sending msgPicSegPaintObject.</td>
</tr>
<tr>
<td>msgPicSegPaintObject</td>
<td>P_PIC_SEG_PAINT_OBJECT</td>
<td>The picture segment sends this message to its graphics to cause them to draw themselves.</td>
</tr>
<tr>
<td>msgPicSegDrawGraphic</td>
<td>P_PIC_SEG_GRAFIC</td>
<td>Draws a specified graphic.</td>
</tr>
<tr>
<td>msgPicSegDrawGraphicIndex</td>
<td>S32</td>
<td>Sets the current graphic index to the specified value and draws the graphic associated with the index.</td>
</tr>
<tr>
<td>msgPicSegDrawGraphicList</td>
<td>P_PIC_SEG_LIST</td>
<td>Draws all the graphics referred to by the list.</td>
</tr>
<tr>
<td>msgPicSegAddGraphic</td>
<td>P_PIC_SEG_GRAFIC</td>
<td>Adds a graphic to the picture segment and draws the graphic.</td>
</tr>
<tr>
<td>msgPicSegErase</td>
<td>pNull</td>
<td>Deletes all graphics.</td>
</tr>
<tr>
<td>msgPicSegDelete</td>
<td>S32</td>
<td>Deletes the graphic having the specified index, and sends msgDestroy to the graphic object.</td>
</tr>
<tr>
<td>msgPicSegRemove</td>
<td>S32</td>
<td>Deletes a graphic, takes a graphic index. Does not send msgDestroy to the graphic object.</td>
</tr>
<tr>
<td>msgPicSegMakeInvisible</td>
<td>S32</td>
<td>Makes the given graphic invisible.</td>
</tr>
<tr>
<td>msgPicSegMakeVisible</td>
<td>S32</td>
<td>Makes the given graphic visible.</td>
</tr>
<tr>
<td>msgPicSegCopy</td>
<td>OBJECT</td>
<td>Copies the contents of the specified picture segment to self.</td>
</tr>
<tr>
<td>msgPicSegHitTest</td>
<td>P_PIC_SEG_HIT_LIST</td>
<td>Passes back the index of the first graphic that intersects a specified rectangle.</td>
</tr>
<tr>
<td>msgPicSegChangeOrder</td>
<td>S32</td>
<td>Changes the order of the graphics in the display, moving the current graphic to the given index.</td>
</tr>
</tbody>
</table>

### Developer's Quick Start

You create a picture segment much as you create a drawing context, and bind it to a window as usual using `msgDcSetWindow`. The picture segment's own `msgNew` arguments are flags indicating if the picture segment should paint and if it should store (both are set by default).
Thereafter, just send ordinary clsSysDrwCtx messages to the picture segment. They will appear in the window it is bound to, and the picture segment will remember them. When you wish to paint or repaint the picture segment’s window, send it msgWinBeginPaint or msgWinBeginRepaint as appropriate, and then send msgPicSegPaint to the picture segment.

Grafics

A picture segment stores each figure drawing message sent to as a grafic. A grafic is a compressed representation of some drawing operation which actually puts something on the screen. Each grafic is composed of an opcode identifying the drawing operation, and a pointer to the data required by that drawing operation. Different drawing operations require different data, just as different DC messages take different message arguments. Here’s the structure of a PIC_SEG_GRAFIC:

```c
typedef struct {
    OP_CODE             opcCode;
P_UNKNOWN             pData;
} PIC_SEG_GRAFIC, FAR * P_PIC_SEG_GRAFIC;
```

Since drawing operations correspond to DC messages, for the most part there is a one-to-one correspondence between DC drawing messages and grafics. As a picture segment receives DC drawing messages, it stores them as grafics and passes them to its ancestor (so that they appear in the window). clsPicSeg also responds to a few drawing messages of its own. Here are the drawing messages which add a grafic to a picture segment, together with the grafic’s opcode and data type:

<table>
<thead>
<tr>
<th>Message</th>
<th>Opcode</th>
<th>Storage Data Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgDcDrawEllipse</td>
<td>OpCodeEllipse</td>
<td>P_PIC_SEG_ELLIPSE</td>
</tr>
<tr>
<td>msgDcDrawRectangle</td>
<td>OpCodeRectangle</td>
<td>P_PIC_SEG_RECT</td>
</tr>
<tr>
<td>msgDcDrawPolyline</td>
<td>OpCodePolyLine</td>
<td>P_PIC_SEG_POLYLINE</td>
</tr>
<tr>
<td>msgDcDrawPolygon</td>
<td>OpCodePolygon</td>
<td>P_PIC_SEG_POLYGON</td>
</tr>
<tr>
<td>msgDcDrawSectorRays</td>
<td>OpCodeSectorRays</td>
<td>P_PIC_SEG_ARC_RAYS</td>
</tr>
<tr>
<td>msgDcDrawArcRays</td>
<td>OpCodeArcRays</td>
<td>P_PIC_SEG_ARC_RAYS</td>
</tr>
<tr>
<td>msgDcDrawChordRays</td>
<td>OpCodeChordRays</td>
<td>P_PIC_SEG_ARC_RAYS</td>
</tr>
<tr>
<td>msgDcDrawBezier</td>
<td>OpCodeSpline</td>
<td>P_PIC_SEG_SPLINE</td>
</tr>
<tr>
<td>msgDcDrawText</td>
<td>OpCodeText</td>
<td>P_PIC_SEG_TEXT</td>
</tr>
<tr>
<td>msgDcDrawImage</td>
<td>not implemented</td>
<td></td>
</tr>
<tr>
<td>msgPicSegDrawSpline</td>
<td>OpCodeSpline</td>
<td>P_PIC_SEG_SPLINE</td>
</tr>
<tr>
<td>msgPicSegDrawObject</td>
<td>OpCodeObject</td>
<td>P_PIC_SEG_OBJECT</td>
</tr>
</tbody>
</table>
The data storage structure is the way the information is stored in the gragic, not the argument to the drawing message. The drawing messages take the usual arguments. The storage type is interesting if you want to edit picture segments.

The picture segment drawing message msgPicSegDrawSpline is a special messages implemented by picture segments to draw splines. This is an enhancement to the drawing messages of ImagePoint™. You can use clsPicSeg instead of clsSysDrwCtx solely to gain access to the additional drawing capabilities.

Picture segments provide a means to draw other figures besides the various ImagePoint primitives and splines. You can add an object to a picture segment using msgPicSegDrawObject, so long as that object responds to msgPicSegPaint.

Picture segments do not support msgDcDrawImage.

**Coordinates**

Most grafics require one or more coordinates as part of their data. DC drawing messages take 32-bit coordinates (s32). Picture segments convert these to 16-bit coordinates (s16) to save memory.

If you use small units and scales and computed coordinates, you end up with coordinate values greater than maxS16 or less than minS16 (32767 and -32768, respectively). These values won't work as expected if you switch from a drawing context to a picture segment.

**DC State**

Messages which change DC state, such as msgDcSetLineThickness, msgDcSetForegroundRGB, and msgDcSetFillPat, are not stored as grafics in a picture segment. Instead, every grafic encodes some of the graphics state along with its drawing operation. The advantage is that you can reorder grafics and draw individual grafics without having to set up the DC state up for each one. The entire DC state is not stored with each grafic because of the size of the DC state (hundreds of bytes).

**Paint**

Most grafics encode interesting DC state in a PICSEG_PAINT structure:

```
typedef struct {
  U16  linePat,
       fillPat;
  SYSDC_RGB  foregroundRGB,
             backgroundRGB;
  U16  lineThickness;
} PICSEG_PAINT, FAR * P_PICSEG_PAINT;
```

Note that line parameters such as join, cap, and miterLimit are not in this structure, since they are only relevant for polylines, polygons, splines, and beziers. The data types for these line drawing operations store this information in a PICSEG_PLINE_TYPE structure.
Some DC State Not Stored

Picture segments don’t record several aspects of DC state, including:
- Plane control.
- Raster ops.
- Drawing context mode.

It is not a good idea to use these in the midst of sending drawing messages to a picture segment, since they will not be recorded in the picture segment. The picture segment inherits these parts of the graphic state. You should set these once for the window to which the picture segment is bound, and then leave them alone while drawing. Note that the picture segment stores LUC matrix and units with each graphic, so you can change the LUC while recording graphics.

Colors

You should not use hardware color palette indices (SYSDC_COLOR) when setting foreground and background colors. These are hardware-dependent and won’t work when printing, so picture segments store RGB values in PIC_SEG_PAINT and not SYSDC_COLORs. In other words, use msgDcSetForeground/BackgroundRGB instead of msgDcSetForeground/BackgroundColor.

Text Attributes

clsPicSeg does not store all of the text-specific DC state with text graphics. It stores the font spec, alignment, underline, and strikeout, but to save space it does not store more obscure text attributes such as spaceChar, spaceExtra, otherExtra. It uses the default values for these.

Although clsPicSeg does not store DC scaling, it does store text scaling set by msgDcScaleFont and msgDcIdentityFont. It does this by storing the current size (width and height) of the text in LUC at the time of msgDcDrawText. It remembers this as a SIZE16, not as an SCALE32; thus the picture segment drops fractional text scales and won’t work at all if the scale is below 1.

To help clients display grab handles around text quickly, clsPicSeg calculates the bounding rectangle of the text and stores it in the rectangle field of PIC_SEG_TEXT.

Picture Segment Storage

A picture segment stores its graphics (both opcodes and data) in the object’s heap. You can specify a heap in the object.heap of the PIC_SEG_NEW structure when you create a picture segment. By default the heap is the OSProcessHeap of the object’s owner.
Using Picture Segments

Creating a Picture Segment

The msgNew argument for clsPicSeg is a pointer to a PIC_SEG_NEW structure. This includes the msgNew fields for clsSysDrwCtx, and a PIC_SEG_METRICS structure. The PIC_SEG_METRICS structure contains:

- **flags** flags include:
  - `picSegAdd` whether the picture segment should remember drawing operations by adding graphics (default is true).
  - `picSegDraw` whether the picture segment should paint by passing drawing messages to its ancestor (default is true).

- Many attributes of the DC, including the fill pattern, line pattern, units, foreground and background RGB values, line style, and font spec. The picture segment sets these (remember, it is itself a drawing context), and new graphics will use these values until you change them.

Drawing in a Picture Segment

clsPicSeg passes all clsSysDrwCtx messages that it receives to its ancestor. As explained above, it stores drawing operations and keeps track of most the current graphics state.

If you need to paint or repaint the entire contents of a picture segment:

- Send its window `msgWinBeginRepaint` (if you are repainting damage) or `msgWinBeginPaint` (if you are painting outside a repaint handler).
- Send `msgPicSegPaint` to the picture segment.

The picture segment replays its graphics in the order you sent the original drawing messages.

To change the contents of a picture segment, you can send it `msgPicSegErase` to delete all its graphics, then draw new ones. This chapter later describes how you can edit a picture segment rather than erase and start over.

If the window changes size, or if you are drawing the picture segment in part of a window and wish to draw it elsewhere, you can change the scale, translation, and rotation of the DC before redrawing the picture segment.

Picture Segment Drawing Messages

There are a few special drawing messages supported by clsPicSeg which aren't implemented by clsSysDrwCtx. These end up sending standard DC drawing messages to self, but are more convenient to use.
msgPicSegDrawSpline

This message draws a spline in the window. A spline is a series of Bezier curve control points. SysDC's msgDcDrawBezier allows you to draw a spline, but is limited to a single curve with four control points; to draw a continuing path with it you must send multiple msgDcDrawBezier messages.

Drawing Other Objects in Picture Segments

Picture segments provide a means to draw other figures besides the various SysDC primitives and splines. You can add an object to a picture segment using msgPicSegDrawObject, so long as that object responds to msgPicSegPaintObject. This takes a PIC_SEG_OBJECT as its message arguments. In the PIC_SEG_OBJECT, you specify:

- paint the paint to use.
- object the object to draw and store.
- rectangle the rectangle in which to draw the object.

The picture segment sends msgPicSegPaintObject to object, whose arguments include a DC to use to draw itself. If object responds to msgPicSegPaintObject, it can be stored in a picture segment and will redraw properly.

Of course, if you subclass clsPicSeg and you recognize the class of the object, you can manipulate it directly.

Drawing Objects Outside Picture Segments

If an object responds to msgPicSegPaint, msgPicSegGetGraphic, and msgPicSegDelta, then you can send it these messages to draw itself outside a picture segment. For example, you can tell a TIFF object to draw in your own window without using a picture segment.

Building up a Picture Segment

It's possible to add graphics to a picture segment without drawing them, using msgPicSegSetMetrics or msgPicSegSetFlags to reset the picSegDraw flag. Conversely, you can draw in a window without adding to a picture segment by clearing the picSegAdd flag. The latter is useful to draw a grid or background in the window.

Using Picture Segments in Graphics Applications

clsPicSeg supports several features which enable it to do much of the work needed for a vector drawing application component. The picture segment acts as a display list of the graphic display objects in the drawing, and the drawing program provides a user interface for the addition, reorganization, and modification of these objects.
Editing a Picture Segment

A picture segment is an ordered list of graphics. It's like a stack of cards where each card is a graphic. You can count cards, shuffle cards, move cards, and delete cards. Each graphic has an associated index, a signed 32-bit number. You can send msgPicSegGetCount to determine how many graphics are in a picture segment.

For all messages which take a graphic index as a message argument, if you supply picSegTopGrafic, the message affects the last graphic in the picture segment. picSegTopGrafic is a constant with a special meaning to clsPicSeg. It is not the maximum number of graphics you can store in a picture segment (which is limited only by available memory).

You can repaint a particular graphic by sending msgPicSegDrawGraficIndex (afterwards the current graphic is set to that graphic). You can delete a particular graphic by sending msgPicSegDelete (afterwards, the current graphic is the topmost graphic again).

The Current Graphic

clsPicSeg always adds new graphics at the end of the display list. However, picture segments keep track of the current graphic. You can set the current graphic using msgPicSegSetCurrent.

msgPicSegGetGrafic retrieves the current graphic in a PIC_SEG_GRAFIC structure. This includes a pointer (pData) to the data structure appropriate for the type of graphic.

Note that clsPicSeg allocates the memory for the graphic-dependent data structure from the process heap, but it is up to the client to free it with OSHeapBlockFree().

msgPicSegDelta changes the current graphic. It also takes a P_PI C _SEG _GRAFIC as its message arguments.

To edit a graphic you must:

- Send msgPicSegSetCurrent to set the current graphic to the one you wish to edit.
- Send msgPicSegGetGrafic to retrieve that graphic.
- Cast data pointer (pData) into the appropriate type (e.g. P_PI C _SEG _ELLIPSE) and modify it.
- Send msgPicSegDeltaGrafic.
- Free the data pointed to by pData using OSHeapBlockFree()..

If you have a PIC_SEG_GRAFIC, you can paint it by sending msgPicSegDrawGrafic. This never adds the graphic to the picture segment, it only paints it in the window (if picSegDraw is set). This is useful if you want to redraw a graphic slightly differently.
To replace a graphic, you need not retrieve it, only:

- Send `msgPicSegSetCurrent` to set the current graphic to the one you wish to replace.
- Declare or allocate a data structure of the appropriate type and set it up as desired.
- Send `msgPicSegSetGraphic`, specifying the opcode and `pData`, to replace the graphic.

`msgPicSegChangeOrder` moves the current graphic to the index you specify. This effectively moves a graphic in front of or in back of other graphics.

**Drawing by Adding Graphics**

You can use `msgPicSegAddGraphic` to add a graphic to a picture segment. This will also draw the graphic, so it accomplishes the same thing as using a SysDC drawing message. `msgPicSegAddGraphic` is useful if you already have the opcode and data for a graphic available and want to add it. For example, you might be duplicating graphics, or copying graphics between picture segments.

**Correct Opcode**

When you use `msgPicSegSetGraphic`, `msgPicSegInsert`, or `msgPicSegAddGraphic`, you must ensure that the opcode and data are correct for each other and that the opcode is valid. If `clsPicSeg` does not recognize an opcode, it ignores it or returns an error.

**Hit Testing a Picture Segment**

You can send `msgPicSegHitTest` to a picture segment to determine if a graphic falls within a rectangle. You can use this to see if something is selected by a user's pen tap or selection rectangle. `msgPicSegHitTest` takes a pointer to a `PIC_SEG_HIT_LIST` structure. In this you specify:

- `rect` rectangle for the hit detection.
- `index` the index of the starting graphic.

`clsPicSeg` traverses the picture segment from top to bottom, starting at the `index` you supply (use `picSegTopGraphic` to start at the very top). It checks each graphic to see if it intersects the `RECT32` you supply. It passes back the index of the first graphic it finds that intersects your rectangle. It returns the same status values as do drawing messages when hit testing is on:

- `stsHitOn` if the rectangle intersects the edge of the figure.
- `stsHitIn` if that graphic's fill is non-transparent and the rectangle intersects its center.
- `stsHitOut` if no graphic intersects the rectangle.

You must send `msgWinBeginPaint` to the window before using `msgPicSegHitTest`, and `msgWinEndPaint` after it.
**Invisible Graphics**

msgPicSegMakeInvisible marks a graphic so that the picture segment does not draw it during msgPicSegPaint.

Invisibility is controlled by the opCodeMaskInvisible flag bit in the graphic's opcode. To find out if a graphic is invisible, use msgPicSegGetGrafic to get it and check the opCodeMaskInvisible bit.

**Converting to Other Formats**

You can convert a picture segment into some other vector drawing file format by getting each graphic from the picture segment in turn and converting it to the other format.

**What Picture Segments DON’T Do**

clsPicSeg does not do everything needed for a graphics application. Here is an incomplete list of what it does not support:

- Selections.
- Highlighting.
- Selection handles.
- Input processing, gesture translation (these would be up to the window to which you bind the picture segment).

**Moving and Copying Picture Segments**

You can move and copy graphics in picture segments using the selection manager and transfer mechanism. clsPicSeg does none of the work for this, since it doesn’t handle selections. However, if you do provide a way for the user to select, delete, and add graphics in your picture segment, you should probably implement this protocol so that the user can cut and paste between it and drawing paper.

The following sections discuss the selection transfer of picture segment graphics. The discussion describes what the sender of the selected graphics and the receiver of the selected graphics each must do to implement a move or copy.

There is a special transfer type for picture segments, xferPicSegObject. If as sender you are asked for transfer types msgXferList, your XferAddIds() response should include xferPicSegObject as well as any other formats in which you can provide the selection.

**Sender Responsibilities**

As sender of the selected graphics, you receive msgXferGet. The way to transfer the selected graphics is to pass a picture segment to the receiver. Since the receiver may be in a different process, you need to create a global heap and create a global picture segment which uses that heap. This is the intermediate holder for the transferred graphics.
Set the id in msgXferGet’s XFER_BUF structure to xferPicSegObject, and set the uid field to the UID of the transfer picture segment.

You should put the selected graphics in the provided transfer picture segment. To do this, for each selected graphic send msgPicSegSetCurrent and msgPicSegGetGrafic to your normal picture followed by msgPicSegAddGrafic to the transfer picture segment.

If the operation was a move, the producer will receive msgSelDelete, and should delete the selected graphics.

**Receiver Responsibilities**

As the receiver, you send msgXferList to the producer when you receive msgSelMoveSelection or msgSelCopySelection. You can use XferMatch() to do this and to check to see that one of the types is xferPicSegObject. Then send msgXferGet, specifying an id of xferPicSegObject. As described above, the producer should pass back a global intermediate picture segment.

You need to scale the transferred graphics to the units of your picture segment. Send msgPicSegScaleUnits to the intermediate picture segment, passing in the message indicating your picture segment’s units (e.g. msgDcUnitsPoints). To perform the copy, self-send msgPicSegCopy, specifying the intermediate picture segment as the message argument.
Chapter 28 / Bitmaps and TIFFs

Everything which appears on screen is an instance of clsWin or of one of its descendant class. For example, every class in the UI Toolkit is a descendant of clsWin. Another popular window descendant class is clsView, which is a window which observes another Class Manager object. The UI Toolkit is documented in Part Four, and clsView is documented in Part 2: PenPoint Application Framework.

This chapter documents some other clsWin descendants and other classes which provide additional functionality to the basic window:

- Bitmaps, which store sample values.
- TIFF objects, which display the contents of a TIFF bitmap file.

**Bitmaps**

clsBitmap provides a way to store the information used by msgDcDrawImage and msgDcCacheImage. This information is primarily the samples which these sampled image operators draw. clsBitmap also supports a limited range of operations on the bitmap information, mainly to support the PenPoint Bitmap Editor. This tool is on the SDK Goodies disk and is documented in the PenPoint Development Tools volume.

GO may change the format of the bitmap pixels, so you should not access them in a way that relies on their format.

The image in an icon (a UI toolkit gadget of clsIcon) can be specified as a bitmap object. Table 28-1 summarizes the messages clsBitmap defines:

### Table 28-1

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_BITMAP_NEW</td>
<td>Initializes BITMAP_NEW structure to default values.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_BITMAP_NEW</td>
<td>Creates a bitmap.</td>
</tr>
<tr>
<td>msgBitmapGetMetrics</td>
<td>P_GET_METRICS</td>
<td>Gets bitmap metrics.</td>
</tr>
<tr>
<td>msgBitmapSetMetrics</td>
<td>P_BITMAP_METRICS</td>
<td>Sets bitmap metrics.</td>
</tr>
<tr>
<td>msgBitmapSetSize</td>
<td>P_SIZE16</td>
<td>Sets bitmap size, resizing heap block if necessary.</td>
</tr>
<tr>
<td>msgBitmapInvert</td>
<td>pNull</td>
<td>Inverts the colors of the bitmap.</td>
</tr>
<tr>
<td>msgBitmapLighten</td>
<td>pNull</td>
<td>Lightens the colors of the bitmap by 1/4.</td>
</tr>
<tr>
<td>msgBitmapFill</td>
<td>RGB</td>
<td>Fills bitmap pixels with specified RGB value leaving mask alone.</td>
</tr>
<tr>
<td>msgBitmapCacheImageDefaults</td>
<td>P_SYSDC_CACHE_IMAGE</td>
<td>Prepares argument structure for msgDcCacheImage.</td>
</tr>
</tbody>
</table>

continued
Creating a New Bitmap

You send `msgNew` to `clsBitmap` to create a bitmap. This takes a `BITMAP_NEW` structure which you should initialize with `msgNewDefaults`. You also specify:

- **style** various style flags.
- **size** the size of the bitmap.
- **pPixels** a pointer to the samples.
- **clsBitmap** copies the samples from `pPixels` into memory which it allocates.

Using a Bitmap

In fact, you rarely create bitmaps yourself. Usually you read bitmaps from resource files, since the Bitmap Editor saves bitmaps as resources. The tag for bitmaps is `bitmapResId`.

Having created a bitmap, you usually want to get it on the screen in some form. The way to do this is to send `msgBitmapCacheDefaults`. This takes a pointer to the `SYSDC_IMAGE_INFO` structure used by `msgDcDrawImage`. `clsBitmap` fills in the structure with default values so that you can send it to `msgDcDrawImage`, and the sampled image stored in the bitmap will be drawn in the window.

Modifying

You can change a bitmap's size using `msgBitmapSetSize`. `clsBitmap` frees the memory for the bitmap.

You can change bitmap metrics by sending `msgBitmapSetMetrics`. This is probably a bad thing to do since it (currently) doesn't resize or reallocate the bitmap.

Bitmaps are not windows. You can't bind a DC to them. However, it is possible to perform some graphic operations on them:

- You can invert a bitmap by sending it `msgBitmapInvert`.
- You can lighten a bitmap by 25% by sending it `msgBitmapLighten`.
- You can fill a bitmap with a color by sending it `msgBitmapFill`.

You can't lighten or fill a one bit per sample (black and white) bitmap.
clsBitmap defines several observer notification messages which it sends to the observer of a bitmap when it changes: msgBitmapPixChange, msgBitmapChange, and msgBitmapMapChange. These are mainly used by the Bitmap Editor.

clsTiff

clsTiff displays TIFF (Tagged Image File Format) images. An instance of clsTiff refers to a pathname to a TIFF file, so the file must be in that location in order for the clsTiff object to display its image. The class displays the TIFF image in the file.

TIFF objects are not windows. clsTiff inherits from clsObject, not clsWin.

The TIFF specification encompasses many different flavors of TIFF files. clsTiff handles many (but not all) variants. It handles Intel and Motorola byte order, packed data (1, 2, 4, or 8 samples per byte), fill order, orientation, photometric interpretation, and dot size. It supports Type 1 (packed data), Group 3 FAX encodings (types 2 and 3), and pack bits run-length encoding (type 32773).

clsTiff assumes the image is black and white or gray-scale; it does not support color images or colormap (pseudocolor) images.

Table 28-2 summarizes the messages clsTiff defines.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_TIFF_NEW</td>
<td>Initializes a TIFF_NEW structure to default values.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_TIFF_NEW</td>
<td>Creates a new TIFF object, and optionally opens its associated file.</td>
</tr>
<tr>
<td>msgTiffGetMetrics</td>
<td>P_TIFF_METRICS</td>
<td>Passes back the metrics of the clsTiff object.</td>
</tr>
<tr>
<td>msgTiffSetMetrics</td>
<td>P_TIFF_METRICS</td>
<td>Sets the metrics of the clsTiff object.</td>
</tr>
<tr>
<td>msgTiffSetGroup3Defaults</td>
<td>P_TIFF_SAVE</td>
<td>Sets the clsTiff object metrics to Group 3 compression type 2 defaults.</td>
</tr>
<tr>
<td>msgTiffGetSizeMils</td>
<td>P_SIZE32</td>
<td>Provides the actual size of the TIFF image in mils (0.001 inch).</td>
</tr>
<tr>
<td>msgTiffGetSizeMM</td>
<td>P_SIZE32</td>
<td>Provides the actual size of the TIFF image in millimeters.</td>
</tr>
<tr>
<td>msgPicSegPaintObject</td>
<td>P_PIC_SEG_PAINT_OBJECT</td>
<td>Renders the TIFF image with a specified drawing context object.</td>
</tr>
<tr>
<td>msgTiffSave</td>
<td>P_TIFF_SAVE</td>
<td>Saves the image as a TIFF file.</td>
</tr>
<tr>
<td>msgTiffGetRow</td>
<td>U32</td>
<td>Sent to the client of the TIFF_SAVE to get the next row of the image.</td>
</tr>
</tbody>
</table>
Creating a New TIFF Object

You send msgNew to clsTiff to create a new TIFF object. Before doing this you should probably have a TIFF file around, one that you have created yourself or are importing.

You declare a TIFF_NEW structure. In the usual manner, one of the fields in this is tiff, a TIFF_NEW_ONLY structure, in which you specify:

- pName the pathname to the TIFF file.
- imageFlags a subset of SYS_DC_IMAGE_FLAGS.
- rectangle the origin and size you want the TIFF image to be.

The only valid SYS_DC_IMAGE_FLAGS are sysDcImageFillWindow and sysDcImageLoFilter/sysDcImageHiFilter. clsTiff passes these on to msgDcDrawImage when it paints the TIFF object.

By default only sysDcImageFillWindow is set in tiff.imageFlags, and the origin and size of the image are zero.

clsTiff creates a TIFF object. If tiff.pName is not null, it tries to open the named TIFF file. If sysDcImageFillWindow is set, clsTiff passes back the natural size of the TIFF image in Mils in tiff.rectangle. (Remember, the TIFF image is not associated with a window or DC, so there's no obvious LUC or LWC coordinates to use.)

TIFF Image Metrics

msgTiffGetMetrics passes back information about the TIFF file. This takes a pointer to a TIFF_METRICS structure. As well as the file name, rectangle, and flags, clsTiff also passes back a wealth of information about the TIFF image in the file, including:

- orientation.
- width and length.
- xResolution and yResolution, and the resolutionUnits.

This information is very technical and you'll need the complete TIFF file specification to make sense of it. The TIFF specification is maintained by and available from Aldus Corporation of Seattle, Washington (USA). You can contact Aldus through their forum on CompuServe or their developer's bulletin board on AppleLink.

You can set the metrics of the TIFF object and its image using msgTiffSetMetrics. Note that indiscriminately modifying TIFF parameters without changing the TIFF file will undoubtedly cause problems. The only safe TIFF file metrics to change are the orientation, pDocumentName, and a few others.

Repaint

A TIFF object repaints when it receives msgPicSegRedraw. Since a TIFF object isn't a window and isn't bound to one, you must pass in a drawing context as the
message argument to msgPicSegRedraw. You should bracket the send of
msgPicSegRedraw with either a msgWinBeginRepaint/msgWinEndRepaint or
msgWinBeginPaint/msgWinEndPaint pair sent to the DC's window.

To paint the TIFF image, clsTiff uses msgDcDrawImage. It sets
msgDcDrawImage's sysDclmageFillWindow, sysDclmageLoFilter, and
sysDclmageHiFilter flags to the values you specify. If you set
sysDclmageFillWindow, the TIFF image will fill whatever window the DC is
bound to. Otherwise, the TIFF image will be painted at the location and size you
specified in its metrics (rectangle). The coordinates in rectangle are interpreted in
the DC's LUC.

sysDclmageLoFilter and sysDclmageHiFilter control how much filtering the
system drawing context performs when it draws the pixels of the TIFF image. See
Chapter 25, The Drawing Context Class, for more information on
msgDcDrawImage.

 TIFF Images in Picture Segments

You can use TIFF objects with picture segments. msgPicSegDrawObject will add
and draw a TIFF object in the picture segment. So that it works in a picture
segment, clsTiff responds to msgPicSegPaintObject by drawing itself.

 Filing

clsTiff saves and restores its instance data. However, it does nothing with the
separate TIFF file. It is up to the client of the TIFF object to maintain this file,
copying it as necessary.

 Destroying

Likewise, when a TIFF object receives msgFree, it does not delete the separate
TIFF file. If the TIFF file is in the document's directory, then the PenPoint
Application Framework will delete it when the document is deleted from the
Notebook.
Chapter 29 / ImagePoint Rendering Details

This chapter discusses some low-level implementation details of the ImagePoint™ rendering mechanism. The information is technical and presented primarily for developers working on graphics applications. The chapter covers the following topics:

- Logical unit coordinates.
- Logical device coordinates.
- Line, rectangle, and polygon rendering.
- Line thickness issues.
- Differences from earlier releases.

**Logical Unit Coordinate (LUC)**

A logical unit coordinate system is an integer grid with the integer coordinate pair positioned at the intersection of a pair of gridlines. The gridlines have no thickness. Lengths measured from a given point starts at such an intersection. This model applies to both figure rendering and clipping region construction.

ImagePoint defines an ellipse by the origin and size of its bounding rectangle. Figure 29-1 illustrates drawing an ellipse whose bounding rectangle has origin (1, 1), width 5, and height 4.

![Figure 29-1](image)
Logical Device Coordinate (LDC)

Throughout this chapter, it will be helpful to consider a coordinate system that is similar to the window system's LWC, but whose orientation is that of the LUC. This system is the logical device coordinate system, or LDC. It is obtained by transforming an LUC by the LTM and USC, where:

- LTM is the logical transformation matrix computed as a result of an application issuing any number of msgDcIdentity, msgDcRotate, msgScale, msgDcScaleWorld, msgDcTranslate, and msgDcSetMatrix to the DC in question.

- USC is a scaling matrix computed as a result of an application issuing any one of the msgDcUnits... messages. The USC establishes the LUC scaling to that of the device, measured by the width of a pixel.

You can think of the LDC as a virtual display surface mapped onto your LUC where one unit equals one hardware pixel. In device unit scaling, the LDC is identical to the LUC.

An LUC not in device units usually produces fractional values when transformed to the LDC. To arrive at an integer LDC value a LUC coordinate is transformed into fractional LDC and then snapped to an integer LDC in the direction of (toward) the LDC origin. Put simply, ImagePoint truncates toward the LDC (hence LUC) origin. One can think of the LDC system as one in which the integer coordinates are placed at the the lower-left hand corner of pixel boundaries. More precisely, a pixel at \((x, y)\) is a half-open region containing the points \([x, x+1) \times [y, y+1)\).

The rendering details discussed in the following sections will use the LDC as the frame of reference. In other words, where a drawing primitive takes LUC parameters, the result figure will be invariant under different rotations of the screen.

A word of caution: ImagePoint uses fixed point arithmetic in the construction of these matrices as well as coordinate transformations. A fixed point, \(f\), is a 32-bit number encoding the real number \(\frac{f}{2^{36}}\). Fixed point arithmetic is fast but limited in precision and prone to propagation of errors due to rounding. Use as few steps as possible in the construction of the LTM. For example, suppose you wish to draw a series of squares along the x axis with unit dimensions. You can move the origin of the square from \((0, 0)\), to \((1, 0)\), \((2, 0)\), and so on, or you may choose to translate your coordinate system by \((1, 0)\) at each step with pre-multiplication and always position the origin of the square at \((0, 0)\). However, you will quickly find that the pre-multiplication method results in gaps between the squares after a few translations.
Chapter 29 / Imagepoint Rendering Details

29.3

Line Drawing

In drawing a line with a thickness of one pixel in LUC, the line is transformed to LDC. In this transformation, the LUC endpoints of the ideal line are not considered. That is, if the LUC endpoint of the ideal line is the only part of the line that intersects an LDC pixel, that pixel is not set, as illustrated in Figure 29-2.

Figure 29-2
End Points in One-Pixel Lines

Horizontal and vertical lines which fall exactly between two LDC pixels set the pixels on the increasing (that is, most positive) side of the LUC. For example, in Figure 29-3, a four-pixel line drawn along the horizontal axis sets the pixels on the increasing side of the horizontal axis. In figure 29-3, the LUC on the left increases its vertical coordinates in the opposite direction of LDC (screen coordinates), while the LUC on the right increases its vertical coordinates in the same direction as LDC.

Figure 29-3
LDC Rounding to Positive Side of LUC

29.3.1

Rectangular Figures Without Borders

Rectangular figures are described by an origin along with a width and a height. They include rectangles, ellipses, sectors, and chords. A rectangular figure without a border is one where the line width is set to zero. During rendering, the origin of the rectangle is transformed to LDC using the procedure described in the previous sections. The device coordinates of the opposing corner are obtained by adding the width and height to the origin in LUC, and only then applying the matrix transformation.

The system draws the pixels bound by the left and the bottom edges and up to but not including the right and the top edges. The terms top, left, right, and bottom are all relative to the LUC, not device space. This holds true for all rotations of the LUC. This scheme allows you to line up adjacent rectangular figures without overlapping pixels.
The implication of this model is that locations are favored over pixel metrics. Consider an LUC in which both axes are scaled 0.35 and draw the rectangle at (0, 0) with width and height 10. The pixels drawn will be the region \([0, 2] \times [0, 2]\). Now draw the same rectangle at (2, 2), the pixels drawn will be the region \([0, 3] \times [0, 3]\). Thus pixel metrics vary with positions (this is a change of drawing semantics from the Developer's Release of PenPoint™).

Clipping rectangles obey the same rule of coordinate transformations and rendering as described above as long as the DC is rotated by one of \(0^\circ, 90^\circ, 180^\circ,\) or \(270^\circ\). If the DC is rotated by an arbitrary angle, the resulting clipping rectangle is the minimal bounding box of the rotated rectangle (instead of the rotated rectangle itself—this is because ImagePoint does not yet support non-rectangular clipping regions). More precisely, a clipping rectangle with origin \((x, y)\), width \(w\) and height \(h\) is transformed into the 4 corners using the LUC coordinates \((x + w, y)\), \((x + w, y + h)\), and \((x, y + h)\). The minimal bounding rectangle is then computed using these 4 transformed points in device space.

The image operator msgDcDrawImage obeys the same semantics. However, in the case of an arbitrarily rotated DC, ImagePoint does not guarantee that rectangles or images adjacent to the image have no overlapping pixels.

**Rectangular Figures With Borders**

Rectangular figures fall into three categories: those with no border, with a one pixel border, or with a border of more than one pixel. This section discusses rectangular figures with one pixel borders. "Rendering Geometric Shapes With Thick Borders," later in this chapter, discusses rectangular figures with borders of more than one pixel.

The **border** is the lines connecting the transformed vertices. The same rules of transformations are applied for each, but the pixels hit vary. Figure 29-4 shows the difference between drawing a rectangle without a border and one having a one pixel border.
The origin depicted in both figures would have the $x$ in $[1...2)$, and the $y$ in $[1...2)$. The opposite corner depicted in both figures would have the $x$ in $[6...7)$, and the $y$ in $[5...6)$.

### Polygons

The difference between drawing a polygon with and without a border is quite similar to that of rectangles. In the case of drawing a polygon without a border, the “right-hand-side” edges are not drawn. A right-hand-side edge is defined as follows: draw a horizontal line starting at any point on the edge except the two end points, extending to positive infinity. If the edge in question is a horizontal line, we use a vertical test ray extending to negative infinity. If the test ray intersects an even number of edges (or an odd number of edges in the case of a horizontal edge) on the polygon, it is considered a right-hand-side edge. This test is conducted in LUC space.

![Figure 29-5 Left-Hand and Right-Hand Edges](image)

Figure 29-5 illustrates the test. The purpose of this scheme, as in the case of rectangles, is to allow adjacent polygons with common edges to line up without overlapping pixels.

The concept of “right-handedness” breaks down with self-intersecting polygons (such as a star-shaped polygon) because a test ray from an edge intersects a different number of edges depending on where we start the test ray. If you wish to draw such a polygon with predictable results, you should always draw it with a border.

### Line Width and Corner Radius Scaling

The messages `msgDcSetLineThickness` and `msgDcSetLine` set the line width and corner radius for subsequent drawing. This section describes how they are scaled. The scaling algorithm for both drawing parameters are identical, therefore the following discussion refers only to line width.

A DC maintains a logical line width scale which starts out being unity. It is scaled by `msgDcScale` (unless explicitly held by `msgDcHoldLine`). It is reset to unity by `msgDcIdentity`. Other DC transformations such as rotation do not affect the line width. When `msgDcScale` is applied to coordinates, the $x$ and $y$ scaling
components are applied independently. However, in the case of line width, the effective scale is obtained by averaging the two scaling components. This aspect of ImagePoint is markedly different from other imaging systems such as Adobe’s PostScript, where the line width is indeed scaled independently.

We now give an example to show why ImagePoint’s approach is useful. Consider a square drawn in a point coordinate system with some fixed dimensions. Now consider scaling the DC with the factors 2 and 1 for x and y respectively. That is a square becomes twice as wide along the x axis while retaining the same height along the y axis. In a model where the line width is scaled separately for the x and y components, we would also observe that the vertical sides of the rectangle are much thicker than the horizontal sides. While this is mathematically “pure,” it isn’t particularly useful. We prefer a model in which the line width scales identically in both dimensions, drawing lines with uniform thickness regardless of coordinate transformations—thus the averaging of the scale factors.

The logical line width is multiplied by the logical line width scale to obtain the line width measured by the width of a pixel. This number, called the physical line width, is the line width with which the device level imaging system renders geometric figures.

**Rendering Geometric Shapes with Thick Borders**

This section pertains to renderings of geometric figures. The messages applicable to this section are msgDcDrawPolyline, msgDcDrawBezier, msgDcDrawArcRays, msgDcDrawRectangle, msgDcDrawEllipse, msgDcDrawPolygon, msgDcDrawSectorRays, and msgDcDrawChordRays. All of these messages take a set of points in LUC as parameters.

Closed figures rendered with a zero line width are drawn in much the same manner as rectangles without borders, as described above. Figures rendered with a line thickness of one pixel similarly follow the model for rectangles with one pixel borders. In the cases where the line width is bigger than one pixel, we’ll discuss the different effects of the line thickening algorithm for the two broad categories of messages:

- Those described by a bounding rectangle: msgDcDrawArcRays, msgDcDrawRectangle, msgDcDrawEllipse, msgDcDrawSectorRays and msgDcDrawChordRays.
- Those described by a point path: msgDcDrawPolyline, msgDcDrawBezier, and msgDcDrawPolygon.

**Figures Described by a Bounding Rectangle**

Figure 29-6 illustrates the line thickening algorithm with a rectangle originating at \( p_0 \) with width and height \( w \) and \( h \) respectively, all in LUC, and with a logical line width of \( l \). Assume the LTM is the identity matrix and the device is not rotated.
The sequence of computations is as follows:

1. Compute the point \( p_1 = (p_{0x} + w, p_{0y} + h) \) in LUC.
2. Transform \( p_0 \) and \( p_1 \) using the CTM, yielding \( p_0' \) and \( p_1' \), which are now the upper left and lower right corners of the rectangle respectively.
3. Transform the logical line width \( l \) as described in the previous section yielding \( l_x \)—the thickness of the vertical edges in device units.
4. The outer vertical edges of the thickened rectangle is offset from the frame at \( p_0' \) by a quantity of \( \text{Floor}(l_x/2) \). This is \( l_{x1} \) in the figure.
5. The inner vertical edges are offset from the frame at \( p_0' \) by \( \text{Ceiling}(l_x/2) \). This is \( l_{x0} \) in the figure. In other words, where a line width transformed into an odd number of pixels, the inner frame gets the extra pixel.
6. The thickness of the horizontal edges is computed by \( \text{Round}(l_x \times \text{asp}) \), where \( \text{asp} \) is the aspect ratio of the device.
7. The offsets for the inner and outer horizontal edges are computed similarly to step 4 and 5.
8. The pixels between the two frames, including those on the boundary of the inner frame, are filled with the line pattern. The pixels enclosed by the inner frame, excluding those on the boundary of the inner frame, are filled with the fill pattern.

By offsetting each pair of edges with the same amounts for the inner and outer frames, the distribution of thickness is invariant under different rotations of the device.

We will not go into the elaborate mathematics of rendering frames with arbitrary rotations. It is sufficient to assert that the same rule of thickness distribution applies.

Drawing elliptical figures follows the same algorithm above, with the outer and inner frame bounding two ellipses. The pixels between the two ellipses, including those pixels on the boundary of the inner ellipse, are drawn with the line pattern.
Figures Described by a Point Path

Figures with thick line widths traced out by a point path are treated in the same manner as figures described by a bounding rectangle, with the exception that these figures do not possess the symmetry property of a rectangle with respect to line width distribution. As mentioned above, line widths with an odd number of pixels create an unequal distribution of pixels to the two sides of the line. In a rectangular shape, this poses no problem since each edge of the rectangle always has an opposing edge. Point path figures, however, do not necessarily an opposing edge to which to allocate the extra pixel.

In the PenPoint Developer’s Release, ImagePoint allocated the extra pixel in the direction of increasing coordinate value in device space. This produced an anomaly in which a thick segment did not have a consistent appearance under different rotations of the screen. ImagePoint under PenPoint 1.0 solves this problem by allocating the extra pixel in the direction of increasing coordinate value in LUC.

Differences from Earlier Releases

We have described the major ImagePoint rendering details implemented in PenPoint 1.0. Some of the rendering semantics differ from those implemented in the earlier PenPoint Developers’ Release. We summarize them as follows:

- For rectangular figures without a border, the image operator, and clipping rectangles, the top and the right edges are excluded from rendering.
- For polygons without a border, the right-hand-side edges are not drawn.
- For polylines, the “ceiling side exclusion” semantics is implemented.
- For polylines with an odd pixel thickness, the extra pixel is allocated in the direction of increasing coordinate value in LUC.
Part 4 / UI Toolkit
Chapter 30 / Introduction
Overview 30.1 349
Organization of This Part 30.2 350
Developer's Quick Start 30.3 352
Creating a Choice 30.3.1 352
Creating a Menu Bar 30.3.2 353
Creating a Tabular Layout Window 30.3.3 353
Creating a Custom Layout 30.3.4 353
Creating a Button 30.3.5 354
Other Sources of Information 30.4 356

Chapter 31 / Concepts and Terminology
Toolkit Classes 31.1 357
Four Kinds of Classes 31.2 361
Borders 31.2.1 361
Layout 31.2.2 361
Nested Components 31.3 362
How Menus Work 31.3.1 363
Instance Creation and Defaults 31.4 364
Toolkit Tables 31.4.1 364
Filed Representation 31.5 364
One Approach to Filing UI Toolkit Components 31.5.1 365
Layout Speedup 31.5.2 365
Other Benefits of Using Resource Files 31.5.3 365
UI Toolkit Programming Details 31.5.4 366

Chapter 32 / Toolkit Ancestors
Objects 32.1 367
Windows 32.2 367
Repaint 32.2.1 368
Filing 32.2.2 368
Gesture Windows 32.3 368
Gesture Propagation 32.3.1 368
Responding to Gestures 32.3.2 369
Help IDs 32.3.3 370
Embedded Windows 32.4 370
Borders 32.5 370

Chapter 33 / Border Windows
clsBorder Messages 33.1 371
Creating a Border Window 33.2 373
The bsUnitsLayout Measurement 33.3 375
Painting a Border Window 33.4 375
Painting the Border 33.4.1 375
Painting the Background 33.4.2 376
Painting Background and Foreground 33.4.3 376
Descendants and Colors 33.4.4 376
Providing Custom Backgrounds 33.4.5 377
Painting the Shadow 33.4.6 377
Resizing, Dragging, Topping 33.5 377
Delta Window 33.5.1 377
Layout 33.6 378
Avoiding Repaint and Relayout 33.6.1 378
Propagating and Notifying Visuals 33.7 378
Border Geometry 33.8 379
Outer Offsets 33.8.1 380
Subclassing clsBorder 33.8.2 380

Chapter 34 / Layout Classes
Window Layout 34.1 381
clsTableLayout and clsCustomLayout 34.1.1 382
Coordinate System 34.2 382
Table Layout 34.3 383
clsTableLayout Messages 34.4 384
Table Layout Structure 34.4.1 384
Specifying the Table Layout 34.4.2 385
Table Layout Constraints 34.4.3 386
Layout Baseline 34.4.4 387
Layout Example: a Calculator 34.4.5 388
Locations 34.4.6 388
Custom Layout 34.5 389
Custom Layout Example: the Frame 34.5.1 389
Creating a Custom Layout Window 34.6 390
Specifying Constraints 34.6.1 390
Four Child Window Constraints 34.6.2 391
Custom Layout Dimensions 34.6.3 391
Constraints 34.6.4 392
Picking an Edge for Alignment Constraints 34.6.5 393
Aligning Width and Height Dimensions 34.6.6 393
Additional Constraint Flags 34.6.7 395
Constraints and Shrink-Wrap 34.7 395
Relative Window 34.7.1 396
Value 34.7.2 396
Custom Layout Initialization 34.7.3 396
More on msgCstmLayoutGetChildSpec 34.7.4 396
Shrink-Wrap 34.8 397
Lazy Layout 34.9 397
Layout Loops 34.10 397
Shrink-Wrap and Parent-Relative Sizing 34.10.1 398
Capturing vs. Layout 34.11 398

Chapter 35 / Controls
Filing Controls 35.1 399
Message Dispatching 35.2 399
Presentation and Interaction Behavior 35.3 400
clsControl Messages 35.4 401
Creating a Control 35.5 402
Control Style 35.5.1 402
Control Defaults 35.5.2 403
Values 35.6 403
Dirty Controls 35.6.1 403
Control Enable 35.7 404
Evaluating Control Enable 35.7.1 404
Dynamic Control Enable 35.7.2 405
Internal Notification 35.8 405
When Are Preview Messages Generated? 35.8.1 406
Previewing 35.8.2 407
Gesture Notification 35.9 408
Special Gesture Handling 35.9.1 408

**Chapter 36 / Labels**

csLabel Messages 36.1 410
Creating a Label 36.2 411
Label Styles 36.2.1 411
Label Strings and Special Characters 36.3 413
Layout 36.4 414
No Notification 36.5 414
Painting 36.6 414
Child Windows 36.7 415
Layout 36.7.1 415
Painting 36.7.2 416
Field Support 36.8 416
Insertion Point 36.8.1 416
Character Positions 36.8.2 416

**Chapter 37 / Buttons**

csButton Messages 37.1 417
Other Kinds of Buttons 37.2 418
Creating a Button 37.3 419
Button Defaults 37.3.1 419
Notification 37.4 420
Simple Activation 37.4.1 420
Unwelcome Notification 37.4.2 421
Painting 37.5 422
Value 37.6 422
Control Dirty 37.6.1 422
Creating Many Buttons 37.7 422
Advanced Button Notification Techniques 37.8 423
Manager Objects 37.8.1 423
How clsButton and clsTkTable Respond to Button Previewing Message 37.8.2 424

**Chapter 38 / Toolkit Tables**

csTkTable Messages 38.1 427
Other Kinds of Toolkit Tables 38.2 427
Creating a Toolkit Table 38.3 428
Common Creation Information 38.3.1 428
Class-Dependent Creation Information 38.3.2 429
Creating the Child Windows 38.3.3 434
Changing Defaults in a Toolkit Table 38.3.4 434
Modifying a Toolkit Table 38.4 436
Creating and Adding Your Own Items 38.4.1 436
Modifying Items in a Toolkit Table 38.4.2 436
Toolkit Tables and Window Tags 38.4.3 436
Painting 38.5 436
Layout 38.6 437
Notification 38.7 438
Control Enable 38.7.1 438
Managers 38.8 438
Button Manager Notification Details 38.8.1 438
Menu Management 38.8.2 439
Manager Classes 38.8.3 439
Displaying Installed Fonts in Tables 38.9 441
Removing Items from a Toolkit Table 38.10 441
Subclasses of csTkTable 38.11 442
Toggle Tables 38.12 442
Modifying Toggle Tables 38.12.1 442
Choices 38.13 442
Creating a Choice 38.13.1 443
Notification 38.13.2 443
Choice Manager 38.13.3 443
Choice Value 38.13.4 443

**Chapter 39 / Menus and Menu Buttons**

Menu Buttons 39.1 445
Creating a Menu Button 39.2 446
Menu Button Style 39.2.1 446
Notification 39.2.2 446
Painting 39.2.3 447
Menus 39.3 447
Creating a Menu 39.3.1 447
Creating Submenus 39.3.2 448
Displaying a Menu 39.3.3 448
How a Menu Button Displays its SubMenu 39.4 448
Dynamic SubMenu 39.4.1 449
Control Enable 39.4.2 449
Notification 39.4.3 450
### Chapter 40 / Scrollbars

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layou</td>
<td>453</td>
</tr>
<tr>
<td>Painting</td>
<td>453</td>
</tr>
<tr>
<td>Notification</td>
<td>453</td>
</tr>
<tr>
<td>Providing Information</td>
<td>454</td>
</tr>
<tr>
<td>Client Notification</td>
<td>454</td>
</tr>
<tr>
<td>Normalizing the Scroll</td>
<td>456</td>
</tr>
<tr>
<td>Scroll Windows</td>
<td>457</td>
</tr>
<tr>
<td>Creating a Scrollwin</td>
<td>458</td>
</tr>
<tr>
<td>Repaint</td>
<td>459</td>
</tr>
<tr>
<td>Layout</td>
<td>460</td>
</tr>
<tr>
<td>Notification</td>
<td>461</td>
</tr>
<tr>
<td>Updating</td>
<td>462</td>
</tr>
<tr>
<td>Toolkit Tables</td>
<td>462</td>
</tr>
</tbody>
</table>

### Chapter 41 / List Boxes

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating a List Box</td>
<td>464</td>
</tr>
<tr>
<td>Style Fields</td>
<td>464</td>
</tr>
<tr>
<td>List Box Contents</td>
<td>464</td>
</tr>
<tr>
<td>List Box Entries</td>
<td>465</td>
</tr>
<tr>
<td>Supplying Entries</td>
<td>465</td>
</tr>
<tr>
<td>Scrolling</td>
<td>466</td>
</tr>
<tr>
<td>Free Mode</td>
<td>466</td>
</tr>
<tr>
<td>Pre-Loading a List Box</td>
<td>467</td>
</tr>
<tr>
<td>Modifying</td>
<td>467</td>
</tr>
<tr>
<td>Inserting and Removing Entries</td>
<td>467</td>
</tr>
<tr>
<td>State</td>
<td>467</td>
</tr>
<tr>
<td>Notification</td>
<td>467</td>
</tr>
<tr>
<td>Gestures</td>
<td>468</td>
</tr>
<tr>
<td>Painting</td>
<td>468</td>
</tr>
<tr>
<td>Scrolling</td>
<td>468</td>
</tr>
<tr>
<td>Entry Visibility</td>
<td>468</td>
</tr>
<tr>
<td>Layout</td>
<td>469</td>
</tr>
<tr>
<td>Filing</td>
<td>469</td>
</tr>
<tr>
<td>Miscellaneous Messages</td>
<td>469</td>
</tr>
<tr>
<td>Toolkit Tables versus List Boxes</td>
<td>470</td>
</tr>
</tbody>
</table>

### Chapter 42 / Fields

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Style Flags</td>
<td>475</td>
</tr>
<tr>
<td>Messages</td>
<td>476</td>
</tr>
<tr>
<td>Creating a Field</td>
<td>477</td>
</tr>
<tr>
<td>Style Flags</td>
<td>477</td>
</tr>
<tr>
<td>Custom Handwriting Translation</td>
<td>479</td>
</tr>
<tr>
<td>Access to Field Properties</td>
<td>480</td>
</tr>
<tr>
<td>Component Creation</td>
<td>480</td>
</tr>
<tr>
<td>Activation and Deactivation</td>
<td>481</td>
</tr>
<tr>
<td>Input Processing</td>
<td>482</td>
</tr>
<tr>
<td>Delayed Input</td>
<td>483</td>
</tr>
<tr>
<td>Input Validation</td>
<td>484</td>
</tr>
<tr>
<td>Layout</td>
<td>485</td>
</tr>
<tr>
<td>User Interface</td>
<td>485</td>
</tr>
<tr>
<td>Toolkit Tables</td>
<td>486</td>
</tr>
</tbody>
</table>

### Chapter 43 / Notes

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standardized Messages</td>
<td>487</td>
</tr>
<tr>
<td>Kinds of Notes</td>
<td>487</td>
</tr>
<tr>
<td>clsNote Messages</td>
<td>488</td>
</tr>
<tr>
<td>Creating a Note</td>
<td>488</td>
</tr>
<tr>
<td>Flags</td>
<td>489</td>
</tr>
<tr>
<td>Contents from Resource Files</td>
<td>489</td>
</tr>
<tr>
<td>System Modal vs. Application Modal</td>
<td>490</td>
</tr>
<tr>
<td>Using a Note</td>
<td>490</td>
</tr>
<tr>
<td>Filter</td>
<td>490</td>
</tr>
<tr>
<td>Notification</td>
<td>491</td>
</tr>
<tr>
<td>What Comes Back</td>
<td>491</td>
</tr>
<tr>
<td>Note Dismissal</td>
<td>491</td>
</tr>
<tr>
<td>Painting</td>
<td>492</td>
</tr>
<tr>
<td>Layout</td>
<td>492</td>
</tr>
<tr>
<td>System Modal vs. Application Modal</td>
<td>493</td>
</tr>
<tr>
<td>Standard Message Interface</td>
<td>493</td>
</tr>
<tr>
<td>System and Application Dialogs</td>
<td>493</td>
</tr>
<tr>
<td>Progress Notes</td>
<td>493</td>
</tr>
<tr>
<td>Application Errors</td>
<td>494</td>
</tr>
</tbody>
</table>
## Chapter 44 / Frames

- Using clsFrame Messages
- Creating a Frame
  - Multiple Windows, Multiple Frames
  - Modifying a Frame
- Frame Layout
- Notification
  - Selection
  - Zoom
  - Close, Float, Bring-to-Front, Delete
- Filing
  - Frame Menus
- PenPoint Application Framework and Frames
  - Application Main Window
  - Standard Application Menus
- Subclasses of Frames

## Chapter 45 / Frame Decorations

- Close Box
  - Creating a Close Box
  - Notification
- Title Bars
  - Creating a Title Bar
- User Interface
- Tab Bars
  - Creating a Tab Bar
  - Adding Items
  - Layout
- Command Bar
- Page Number
- Shadow
  - Creating a Shadow

## Chapter 46 / Option Sheets

- clsOption
- clsOption Messages
- Creating an Option Sheet
  - Option Sheet Styles
- Manipulating Cards
  - Layout
  - Painting
  - Notification
  - Destruction
- Performance
- Option Sheet Protocol
- Check Gesture Processing
  - What the Card Client Does
  - Multiple Card Types per Object Type
  - Indicating Mixed Attributes
  - Nested Components
  - Multiple Option Sheets Up at Once
  - Dimmed Controls
  - Selection Interaction
- Option Tables
- Command Sheets
  - Creating a Command Sheet

## Chapter 47 / Icons

- Messages
- Creating
  - Bitmap Picture
  - Pixelmap Picture
- Painting
  - Invalidation
- Notification
- Layout

## Chapter 48 / Trackers and Grab Boxes

- Drawing
- Notification
- Destruction
- Grab Boxes

## Chapter 49 / Progress Bars

- Progress Bar Concepts
- Progress Bar Style and Metrics
  - The PROGRESS_STYLE Structure
  - Progress Bar Metrics
- Progress Bar Messages
  - Creating a Progress Bar
  - Manipulating Style and Metrics Fields
- Manipulating Region Appearance
  - Structures for Manipulating Regions
  - Modifying Region Color and Pattern
  - Determining Region Bounds
- Responsibilities of Progress Bar Clients
  - Providing Custom Labels
- Useful Inherited Messages
List of Figures

30-1 UI Toolkit Components
31-1 UI Toolkit Classes Not Inheriting from clsControl
31-2 UI Toolkit Classes Inheriting from clsControl
31-3 UI Toolkit Class Outline
31-4 A Choice and its Component Buttons
31-5 Nesting of Controls in Menus
33-1 Sample Border Windows
33-2 Layout Units
33-3 Regions of a Border Window
34-1 Sample Table Layouts
34-2 Horizontal Window Baseline Alignment in an Option Sheet
34-3 Calc's Positioning of Child Window Using clsTableLayout
34-4 Sample Custom Layouts
34-5 Layout of Adjacent Windows by clsCustomLayout
36-1 Sample Labels
37-1 Sample Buttons
38-1 Sample Toolkit Tables
38-2 Toolkit Table from TkDemo Application
40-1 A Scrollbar
42-1 Sample Fields
43-1 Sample Application-Modal Note
44-1 Different Frame Styles
46-1 Option Sheets and Option Cards
49-1 Progress Bars

List of Tables

33-1 clsBorder Messages
33-2 BORDER_NEW Styles
34-1 clsTableLayoutMessages
34-2 Table Layout Flags
34-3 Table Layout Constraints
34-4 clsCustomLayout Messages
34-5 Layout Constraints for clsCustomLayout
35-1 clsControl Messages
35-2 CONTROL_STYLE Fields
35-3 Values for Different Subclasses of clsControl
35-4 Previewing in Different Subclasses of clsControl
35-5 Control Messages Sent in Response to Events
36-1 clsLabel Messages
36-2 LABEL_STYLE Fields
37-1 clsButton Messages
37-2 BUTTON_STYLE Styles
37-3 clsButton Previewing Messages
38-1 clsTkTable Messages
38-2 Interpretation of TK_TABLE_ENTRY Fields
38-3 TK_TABLE_ENTRY Flag Values
38-4 clsChoiceMgr Messages
38-5 clsSelChoiceMgr Messages
38-6 clsChoice Messages
38-7 clsMenuMessages
38-8 MENU_BUTTON_STYLE Fields
38-9 clsMenuMessages
38-10 clsPopupChoice Messages
40-1 Scrollbar Actions
40-2 clsScrollwin Messages
40-3 SCROLLWIN_STYLE Styles
41-1 clsListBox Messages
41-2 clsStringListBox Messages
41-3 STRLB_STYLE Fields
42-1 clsField Messages
42-2 Field Style Values
42-3 Component Creation Messages
42-4 Activation and Deactivation Messages
42-5 Input Processing Messages
42-6 Delayed Input Messages
42-7 Input Validation Messages
43-1 clsNote Messages
44-1 clsFrameMessages
44-2 Frame Action Messages
46-1 clsOption Messages
47-1 clsIcon Messages
48-1 clsTrack Messages
48-2 clsGrabBox Messages and Procedures
49-1 PROGRESS_STYLE Styles
49-2 clsProgressBar Messages
49-3 Useful Inherited Messages

List of Examples

30-1 A Simple Menu with Nested Buttons and a Choice
36-1 Creating an Annotation Label
37-1 Creating a Button
38-1 TK_TABLE_ENTRY Array for an Option Card
Chapter 30 / Introduction

Overview

The VI Toolkit provides classes that lay out windows to form user interface elements such as buttons, tabs, handwriting fields, labels, icons, menus, frames, and option sheets. These elements are called UI components. UI components send messages among themselves and to their clients when the user interacts with them.

The VI Toolkit implements the middle layer of the appearance and functionality of the user interface architecture in PenPoint. It calls on the Windows & Graphics subsystem to draw these windows. In turn, the PenPoint Application Framework and the internal classes implementing the Notebook User Interface (NUI) use VI Toolkit objects extensively.

You use VI Toolkit components whenever you want to provide these standard user interface components to the user. If you want to provide alternate styles or interactions, you can leverage the VI Toolkit code by subclassing at various class levels.

PenPoint uses UI components for menus, option sheets, window decorations, the default application frame, and many application windows. Nearly every window that has a label or responds to the pen is a UI component of some sort.

Usually, you use more than one UI component. You can lay out components inside other windows to form tables, control panels, and simple forms. The VI toolkit uses this layout facility to create frames and option sheets.

Menus, command bars, and choices are themselves also UI components that consist of groupings of buttons and labels.

Higher level application components built using UI components include option sheets, search and replace, and the spell check dialog box. The overall Notebook user interface appearance is largely the result of using standard UI components.
Organization of This Part

Part 4 provides some simple examples of how to create common user interface combinations.

Chapter 31, Concepts and Terminology, explains the important concepts in the UI Toolkit, and shows how these are implemented by several families and layers of classes in the toolkit class hierarchy.

The rest of the chapters in Part 4 follow the layered implementation of the UI Toolkit. They describe the UI Toolkit classes by functional area in more detail. The toolkit’s layered design provides a flexible and powerful toolkit, but it makes the reference documentation quite complex for the beginning reader, since the creation and behavior of a UI component is spread across several classes and,
hence, several chapters. You have to read something about the ancestors of a class to understand how you can use their functionality, yet a lot of what an ancestor does is used internally by each class and doesn’t need to be touched by the developer.

- Chapter 32, Toolkit Ancestors, explains how UI Toolkit classes use features provided by their common ancestor classes.

- Chapter 33, Borders, covers clsBorder, which supports some common features of toolkit window repainting, such as margins, and background and foreground colors.

- Chapter 34, Layout Classes, covers general layout issues, and clsTableLayout and clsCustomLayout in particular. These classes implement two approaches for positioning and sizing child windows.

- Chapter 35, Controls, covers clsControl, which implements the translation of window input messages into messages sent to self and other objects.

- Chapter 36, Labels, covers clsLabel, which implements the display of a string or decorative window. Buttons, menu buttons, toggle buttons, and frame title bars all inherit this behavior from clsLabel.

- Chapter 37, Buttons, covers buttons, labels that the user activates by applying a gesture such as a tap.

- Chapter 38, Toolkit Tables, covers toolkit tables. clsTkTable supports the initialization, layout, and notification management for groups of toolkit components (such as buttons). The capability to organize components in a group is used by several subclasses: this chapter describes toggle tables and choices, two particular subclasses of clsTkTable.

- Chapter 39, Menus and Menu Buttons, covers menus (clsMenu) and menu buttons (clsMenuButton). Menu buttons are special buttons that display a menu; menus are special toolkit tables that often group several menu buttons.

- Chapter 40, Scrollbars, covers scrollbars (clsScrollBar) and clsScrollWin, a special descendant of clsBorder which handles a lot of the work of scrolling for you.

- Chapter 41, List Boxes, covers list boxes (clsListBox). List boxes are scrolling windows that support very large numbers of entries. Unlike table elements, only those entries currently visible in the list box need have a window. Descendants of clsListBox provide scrolling lists specifically for strings (clsStringListBox) and for font names (clsFontListBox).

- Chapter 42, Fields, covers text fields (clsField), which are labels that you can handwrite in. It also describes subclasses of clsField that have additional semantics to support integer fields, date fields, fixed-point fields, and so forth.

- Chapter 43, Notes, covers notes (clsNote), which present transient information to the user. Standard messages are a set of standard procedural interfaces for displaying error information that use clsNote.
Chapter 44, Frames, explains frames (clsFrame), which manages a collection of other UI components and a client window. Most applications use frames for their main windows, dialog boxes, and pop-up windows.

Chapter 45, Frame Decorations, explains some of the decorations of frames: close boxes, title bars, tab bars, and command bars.

Chapter 46, Option Sheets, describes the classes that support the option sheet user interface. clsOption provides the option sheet, while the cards in the sheet are usually instances of clsOptionTable. Since the user can leave an option sheet on-screen, the interaction between option sheets, option cards, and the selection can be quite complex.

Chapter 47, Icons, covers icons (clsIcon), which show an iconic menu button made up of a bitmap image and a string.

Chapter 48, Trackers and Grab Boxes, covers clsTrack, which grabs input and draws transient rubber-banding figures in response to pen movements. The toolkit uses this to provide feedback when the user drags or resizes items. In particular, grab boxes (clsGrabBox) are the resize handles on frames and scrollbars which use clsTrack internally.

Chapter 49, Progress Bars, describes how to use clsProgressBar. clsProgressBar implements a dynamic, bar-graph representation of a value. You typically use a progress bar to indicate how far a long process has progressed, or to show where a value falls in a range of possible values.

Developer's Quick Start

Here are the basic steps in creating a typical control panel. They skip a lot of details and aren't applicable to all UI components.

The best approach to take is to try to use toolkit tables (clsTkTable) as much as possible. If you can organize your windows in a tabular arrangement, you can create a single toolkit table and statically specify its contents. This saves you from specifying the NEW structures for each component, worrying about the defaults to override, and assembling them. Choices, menus, tab bars, command bars, and option tables are all descendants of toolkit tables.

Creating a Choice

1. Statically define a TK_TABLE_ENTRY array. Each item in this array describes one component in a toolkit table. In the case of clsChoice, the components are instances of clsButton. The value of the first three fields of each TK_TABLE_ENTRY item varies, depending on the class of component (see PENPOINT\SDK\INC\TKTABLE.H for details). For clsButton components, the first three fields of the TK_TABLE_ENTRY specify the string to appear in the button, the message it sends, and 32 bits of data to send as the message argument.
2 The rest of the fields for each toolkit table entry are a window tag, some flags, and an override class, and an ID for Quick Help. The window tags are important if you want to get to a particular component in the toolkit table after you have created it: you can use 0, 1, 2, ... or MakeTag to define tags. clsChoice sets the flags correctly for standard toggles. You can leave class as objNull since all the buttons have the same class.

Send msgNewDefaults to clsChoice.

3 In CHOICE_NEW, specify tkTable.client as the UID of the client who will respond to button messages.

4 Set tkTable.pEntries to the address of the static TK_TABLE_ENTRY array you defined. You don't need to set tkTable.class to clsButton because that is the default for clsChoice. You can also set the value of the choice by setting the "on" button's tag in choice.value.

Send msgNew to clsChoice, with the modified CHOICE_NEW structure as its argument.

Creating a Menu Bar

1 Specify the components of the menu in a TK_TABLE_ENTRY array.

2 However, the components in the menu need not all be buttons; they can also be choices or menu buttons. These entries themselves are toolkit tables, so they take an array of TK_TABLE_ENTRYs. The first two fields for a menu button TK_TABLE_ENTRY are its string and the TK_TABLE_ENTRY array for its pull-down menu.

Define an entire menu tree by nesting static TK_TABLE_ENTRY arrays. The same goes for option sheets, and so on. Chapter 38, Toolkit Tables, discusses toolkit table nesting in more detail.

Creating a Tabular Layout Window

1 Send msgNewDefaults to clsTableLayout.

2 Specify the constraints in tableLayout.

3 Send msgNew to clsTableLayout.

4 Use the normal msgWinInsert message (remember that all UI Toolkit classes inherit from clsWin) to insert your UI Toolkit components in the table layout window.

Creating a Custom Layout

1 Send msgNewDefaults and msgNew to clsCustomLayout.

2 Use the normal msgWinInsert message to insert your UI component in the custom layout window.
As you insert each component window, set up the appropriate constraints for it in the metrics field of a CSTM_LAYOUT_CHILD_SPEC, put its UID in the child field, and send the custom layout window msgCstmLayoutSetChildSpec.

### Creating a Button

If you need to create a free-standing UI component (instead of one inside a toolkit table), here's how you would go about creating a button:

1. Send msgNewDefaults with the appropriate structure to clsButton.
2. Fill in the button.msg and button.data fields with the message and data you want the button to send.
3. Set the control.client to the UID of the application instance (assuming the application is the object handling button notifications). Set the label.pString to a suitable name. Send msgNew to clsButton.
4. Insert the button into a window to make it visible.
5. The client must handle the button’s message, so mention the message in the application’s method table and write a message handler for it.

You often create a main window in response to msgAppOpen and destroy it at msgAppClose, since your application only needs a user interface while it is on screen. See “Filed Representation” in Chapter 31, Concepts and Terminology, for more information on when to create toolkit components and whether to file them.
The relatively simple toolkit table example below creates the menu bar in the WriterApp sample program (in \PENPOINT\SDK\SAMPLE\WRITERAP\WRITERAP.c).

The menu bar has two menu buttons:

- **Clear** operates immediately (it has no submenu)
- **Translator** has a pull-down submenu with a choice in it.

This is non-standard; a real application would normally have the Standard Application Menus (Document and Edit).

The code defining the menu as a set of nested toolkit tables is as follows:

```c
// Data structure for the menu
static const TK_TABLE_ENTRY menuBar[] = {
    {"Clear", msgWriterAppClear, 0},
    {"Translator", 0, 0, 0, tkMenuPullDown},
    {0, 0, 0, 0, clsChoice},
    {"Word", msgWriterAppTranslator, 0, 0, tkButtonOn},
    {"Number", msgWriterAppTranslator, 1},
    {"Text", msgWriterAppTranslator, 2},
    {pNull}, // This pNull ends the clsChoice table
    {pNull}, // This pNull ends the "Translator" tkMenuPullDown
    {pNull} // This pNull ends the tkTable
};
```

Later on, in its msgAppInit method, WriterApp creates the menu bar:

```c
// WriterApp creates its own custom window
...
// Create the menu bar
ObjectCall(msgNewDefaults, clsMenu, &mNew);
ObjectCall(msgNewDefaults, clsMenu, &mNew);
mNew.tkTable.pEntries = menuBar;
mNew.tkTable.client = self;
ObjectCallRet(msgNew, clsMenu, &mNew, s);

// Set the client window and menu bar in the application frame
ObjectCallRet(msgFrameGetMetrics, am.mainWin, &fm, s);
fm.clientWin = sNew.object.uid;
fm.style.menuBar = true;
fm.menuBar = mNew.object.uid;
ObjectCallRet(msgFrameSetMetrics, am.mainWin, &fm, s);
...
```
Other Sources of Information

Nearly every class in the UI Toolkit is a descendant of clsObject, clsWin, clsGWin, and clsEmbeddedWin. Chapter 32, Toolkit Ancestors, explains how the toolkit classes use these classes. You don’t need in-depth knowledge of these classes to use the toolkit, but you need some familiarity with them. They are described in:

- Part 1: Class Manager (clsObject).
- Part 2: PenPoint Application Framework (clsApp).
- Part 5: Input and Handwriting Translation (clsGWin).

Similarly, UI component behavior is largely determined by window input processing flags. However, if you haven’t read Part 5: Input and Handwriting Translation, you’ll still be able to understand and use the UI Toolkit.

The PenPoint Application Framework is closely involved with the UI Toolkit. It can supply applications with a default frame and a Standard Application Menu, and can file and restore an application’s UI. It is described in Part 2: PenPoint Application Framework.

clsField and its descendants in the UI Toolkit let you create text controls for hand-written input. These are built from the complex functionality of handwriting translation, scribble collection, insertion pads, translation objects, and so on, which are explained in detail in Part 5: Input and Handwriting Translation.

The Application Writing Guide takes a tutorial approach towards some of the ideas in Part 4. The Tic-Tac-Toe program developed in the tutorial performs the following tasks:

- Creates a menu bar of debugging commands.
- Creates a scroll window to scroll its board.
- Creates an option sheet containing numerous UI components.

In addition, sample programs that look like they create user interface components probably do create UI toolkit objects or subclass UI Toolkit classes. For example, the Calc program uses many UI Toolkit windows and nothing else for its user interface. The Clock sample program (which is also part of the standard set of tools) uses several approaches to layout classes to implement the clock display desired by the user.

The source to all these programs is in subdirectories of \PENPOINT\SDK\SAMPLE. You can put together many different UI Toolkit component layouts. PenPoint UI Design Reference suggests appropriate ways to use the UI Toolkit in keeping with PenPoint’s Notebook user interface.
Chapter 31 / Concepts and Terminology

Toolkit Classes

The UI Toolkit class hierarchy is large because there are so many different kinds of toolkit windows and controls. Figures 31-1 and 31-2 show the hierarchy of many of the UI Toolkit classes. Some UI Toolkit classes are not shown in these figures. For completeness, Figure 31-3 provides an outline view of the class hierarchy of all UI Toolkit classes.

The layering of the class hierarchy of UI components and their fluid design provide a framework for constructing a wide variety of user interface tools. While this makes it possible to create new and bizarre kinds of tools, deviations from the standard UI components can confuse users. For this reason, you should restrict yourself to the standard classes in the PenPoint UI Toolkit unless you can determine that none of them meet your needs.

Similarly, the UI Toolkit does not preclude other combinations of components besides those you see in PenPoint. For example, you can easily put a menu bar in the middle of an option sheet (not the right place for a menu bar). See the PenPoint UI Design Reference for information regarding the correct use of UI Toolkit components and suggested ways of combining them consistently.
UI Toolkit Classes Not Inheriting from clsControl
Figure 31-2
UI Toolkit Classes Inheriting from clsControl

other subclasses of clsBorder are on previous page
Four Kinds of Classes

There are many classes involved in UI components because the UI Toolkit separates the implementation of user interface components into different functional parts, with corresponding classes:

- Window border
- Layout
- Message dispatching
- Presentation/interaction behavior

Borders

clsBorder supports the border styles of windows in the PenPoint UI:

- None
- Black
- Double
- Gray
- Rounded
- Shadow
- Resizing from the shadow or drag handles
- Inner margins

The clsBorder shadow is not a true shadow with transparent corners; see clsShadow for a perfect shadow.

Layout

clsWin defines various layout messages and flags, plus related shrink-to-fit flags, but it does not itself implement layout policies for windows. clsCustomLayout and clsTableLayout implement two useful styles of layout. They do not determine how user interface components look; instead they specify how collections of user interface components are arranged. A layout window lays out its child windows according to these specifications. For example, a menu bar is a layout window that lays out its elements in a row.

Usually, you create layout windows to lay out the components of your dialog boxes and other windows. Although this use of layout windows is normally the most convenient, you can insert UI component windows in any kind of window. Conversely, you can use layout windows to lay out other kinds of windows than UI components.
Nested Components

The UI Toolkit allows you to combine components together to create interesting new kinds of components. For example, a scrollwin groups two scrollbars with a client window so that the user can scroll the client window. As another example, a string list box is a kind of scrollwin whose contents are a set of string labels.

The most common form of nested components is a toolkit table. The descendants of clsTkTable group multiple windows (often buttons) together to form more complex toolkit components. For example, a choice component is a group of buttons, each set up to provide feedback with a check mark.

In Figure 31-4, the Contents Layout option card includes a choice of two buttons: Icons and Buttons. The choice buttons preview in the normal way. And because only one button in the set can be on at any time, the choice must also manage the set of buttons. The buttons send preview messages (msgButtonBeginPreview, msgButtonUpdatePreview, msgButtonRepeatPreview, and msgButtonCancelPreview) to their manager, which in this case is their parent, the choice. The choice maintains a hidden manager object which responds to a choice button's preview message by turning off the currently selected button. When the user finally raises the pen, the button sends its client its msgButtonNotify or other specified message, and notifies its manager so that the manager knows which button is active.

Note that this choice is itself embedded in another window, the option card. The choice itself notifies its manager when the user raises the pen, and this sets the option.

Also note the many forms that a choice can take. By inserting different kinds of buttons in a choice and changing the choice's table alignment, you can create a choice like the Alignment choice in the MiniText Paragraph option card shown below the Layout card in Figure 31-4. It has the same manager protocol despite its different appearance.
Separating the manager object from the parent component supports other kinds of complex components, such as menus and selected choices. The details of this complex layered interaction are described in Chapter 38, Toolkit Tables.

**How Menus Work**

PenPoint's pull-right and pull-down menus may appear to be menus nested inside other menus inside other menus, but the reality is simpler. A menu bar is a toolkit table of several menu buttons (instances of `clsMenuBar`, a subclass of `clsButton`), and the latter may each be associated with another menu made up of other kinds of components. For example, Figure 31-5 shows the UI Toolkit components involved when you choose About... from the Document menu of MiniText.

The application menu bar is a toolkit table of several menu buttons. In MiniText, the Document menu button has a menu associated with it, so it responds to previewing by displaying its menu. The Document menu is a separate toolkit table. Its manager is the Document menu button, but it is otherwise a separate window containing its own set of components. One of these components is the About menu button.

When the user taps on About, the button notifies its manager (the menu manager). The Document menu notifies its manager (the Document button), which sends a message which removes the Document menu from the screen. The About... button then notifies its client (the application) by sending it its message (msgAppAbout). Thus, there are two communication paths:

- Up the management chain.
- Directly to the client.

Note that in this example, the menu button, the menu bar, and the default application response are all handled by the PenPoint Application Framework. Even the simplest applications with the Standard Application Menus provided by the Application Framework display this menu and respond to it, without any special coding on your part.
Instance Creation and Defaults

As you can see from the class diagram, UI Toolkit classes do not always behave as expected. Many descendant classes only work as documented by their many ancestors. For example, buttons may have certain input flags set. However, UI Toolkit classes do not automatically send msgNew arguments to ancestors, and they do not perform operations that might adversely affect their operation for performance reasons, and also because they are designed to be subclassed, and a subclass implementation might well want radically different behavior.

You must always send msgNewDefaults to a UI Toolkit class before sending it msgNew to create a new instance. This will ensure reasonable behavior from the component. The header files in \PENPOINT\SDK\INC for the various classes document the defaults initialization performed in response to msgNewDefaults; the same information is in the PenPoint API Reference.

You need to fill in some of the dozens of ancestor fields in the msgNew and msgNewDefaults structure, but tinkering with others may lead to unpredictable results. The header files describe the defaults for each class. You may find that to get a component to work in a way it was not intended to work, it may not be enough to change its style fields; you may need to create a new class.

Toolkit Tables

Toolkit tables provide a very different approach to creating UI components. You can declare the contents of a toolkit table statically in an array that specifies things like the string of a label, the number of columns in a field, and so on. However, you can also create components in the usual way by sending them msgNew, and then nest them in the toolkit table by sending it messages like msgTkTableAddAsLast. clsTkTable knows how to create standard instances of various components. See the example from the WriterApp sample program in Chapter 30.

This approach of static specification reduces code size and is much easier to maintain. If it's possible to create all or part of your UI as toolkit tables, you should do so. Toolkit tables foster a certain homogeneity among components, which is often what you want for standard application elements, such as menus and option sheets.

Filed Representation

Toolkit components file themselves. They also file their descendants. However, you may not want to take advantage of this.

There is a paradox at work in the UI Toolkit: toolkit components do an excellent job of filing their state, which is good, but to conserve memory, you don't want lots of static toolkit components around (either as live objects or filed) when your application is not on-screen.
Another paradox: toolkit components should reflect the state of the application, yet if you let them file, they duplicate the state of the application. For example, if your application has a choice with which the user picks either circle mode or rectangle mode, then when that choice is told to file, it will store the mode the user chose. However, your application should probably also be storing its current mode in its instance data or some other object. Keeping two copies of application state is wasteful and can lead to internal inconsistencies.

**One Approach to Filing UI Toolkit Components**

It is difficult to generalize, but here is a scheme for creating and filing UI Toolkit components that takes advantage of filing:

- Create your user interface in advance, for example in your application’s INIT.DLL.
- File it as one or more well-known resources.
- You can then read your UI from the resource list in each document.
- Don’t file UI components per document.
- File application state separately from UI.
- Set UI state from application state.

This approach avoids duplicate filed copies of UI components in each document.

**Layout Speedup**

You can also file the UI in advance to speed up layout. When windows file, they store their bounds. If, when a window restores, it is positioned with the same bounds as when it was filed (and if some other conditions are met), it need not lay out, nor lay out its children—the sizes computed for the windows are valid. This layout saving can speed up performance, especially when the parent window has many descendant windows.

Windows filed with dirty layouts (wsLayoutDirty set) will require layout whenever they are restored, reducing the efficiency of your application. To prevent this, send msgWinLayout to the topmost window before filing the windows. Another window style flag, wsFileLayoutDirty, ensures that the wsLayoutDirty flag is set whenever the window is restored from a file.

**Other Benefits of Using Resource Files**

There are other benefits to using resource files:

- If the environment is the same when the UI is read from the resource file, its windows won’t have to lay out again—the sizes computed for the windows that were filed with them will be correct. However, if the top-most window is a frame or client window, it may still be laid out as a result of the user floating, zooming, or resizing the window.
The code that uses the UI can be independent of the code that creates it. The code that uses the UI just reads it from a resource file, without concern for the appearance of the UI. Someone could, in theory, replace the filed UI with a different resource, and the application would still work the same. (Note that the application often has to reach in and send specific messages to controls in the UI, so this degree of independence is hard to achieve in practice.)

The code to create the UI can, in fact, be deleted after instance zero runs it. The document instances never create the UI, they read it from the application’s resource list, so as part of installation you could deinstall the installation code. One way to do this is to have your UI creation code in a separate INIT.DLL that the Installer automatically runs at installation and then immediately deinstalls.

**UI Toolkit Programming Details**

In instance zero of your application, create the UI. You could do this in a separate INIT.DLL; when its DLLMain routine is called by the Installer, create the UI.

After creating the UI, send msgWinLayout to the highest level component (such as the frame, or the option sheet) of each subtree. Then file the components you want in the application’s resource file. You can get the application resource file by sending msgAppMgrGetMetrics to the class of your application.

In the other instances of your application (each document), respond to msgAppOpen by reading in the filed instance information. Instead of reading the information from the application resource file, just get the application’s resource list from its application metrics, and read the resource from that. This allows the default UI to be overridden in one document or another.

Make sure you turn off the wsSendFile window flag of components in your main window that came from a resource file so that they aren't filed by clsWin when the PenPoint Application Framework files your application’s frame.
Chapter 32 / Toolkit Ancestors

As you've probably noticed, there are many classes in the UI Toolkit. Some of them provide obvious visual components. Others, such as clsControl and clsTableLayout, provide vital support for other classes. Moreover, all classes in the UI Toolkit are descendants of clsGWin.

All the functionality provided by these ancestor classes is available to you. The msgNew arguments for any visual toolkit component include OBJECT_NEW_ONLY, WIN_NEW_ONLY, GWIN_NEW_ONLY and EMBEDDED_WIN_NEW_ONLY fields. You can send object, window, gesture window, and embedded window messages to toolkit components.

However, be aware that toolkit components rely on certain behavior from their ancestor classes. Changing around ancestors may produce unexpected results. You should rely on msgNewDefaults to set up the ancestor fields appropriately. Only change them if a component does not operate as you want.

This chapter explains how toolkit components use features provided by ancestor classes. It's intended to illustrate rather than explain in detail.

**Objects**

All toolkit components are objects.

Often, toolkit windows are embedded inside other toolkit windows. The parent component may not know in advance what kind of components are inside itself. Thus, toolkit sometimes ask their nested components what their class is (msgClass), or whether they inherit from a particular class (msgIsA).

Some of the machinery of the toolkit is implemented by objects you might not know about. Many toolkit classes allow you to plug in other objects as their children, managers, or clients. Never assume that you know the class of an object passed back by the toolkit.

**Windows**

All visual toolkit components are windows. They use clsWin messages and functionality for obvious things such as painting and UI component resizing.

The UI Toolkit nests components inside other components (such as buttons in menus, labels in tables, fields in option sheets, and so on) by inserting them as child windows of the parent component. Thus, the hierarchy of toolkit components corresponds to the window tree. The UI Toolkit takes advantage of this correspondence to perform the following tasks:

- List nested components using msgWinEnum.
Find nested components by tag using `msgWinFindTag`.

Send messages from nested components to their ancestors using `msgWinSend`.

Propagate and respond to the layout messages `msgWinLayoutSelf` and `msgWinGetDesiredSize`.

This nesting of components isn’t always what you might expect. For example, the client window inside a scrollwin is not a child of the scrollwin: it is actually a child of a child of the scrollwin. The pop-up menu associated with a menu button isn’t a child of it, but a child of the the root window.

**Repaint**

The UI Toolkit handles window repainting itself, so you rarely need to intercept `msgWinRepaint` yourself. The UI Toolkit carries out all its drawing operations using a private toolkit drawing context shared between all processes. You cannot access this DC. However, several of the toolkit component classes let you affect the drawing of components by setting certain metrics, such as background ink and border thickness.

**Filing**

The UI Toolkit does an excellent job of filing its state. All (well, nearly all) components are windows, so you can use the window hierarchy filing provided by `clsWin` to file your entire user interface.

**Gesture Windows**

Most UI Toolkit components support gestures, in which the user handwrites a symbol in the component. For example, the user can flick a vertical scrollbar up or down by drawing an up or down stroke. `clsGWin` supports gestures in windows; it handles the pen input, deciding when the user has completed the gesture, and the translation of the gesture. It self-sends `msgGWinGesture` announcing the translation of the gesture. The interpretation of the gesture is up to descendant classes.

To disable gesture support, set `gestureEnable` in `GWIN_STYLE` to `false`.

**Gesture Propagation**

In the PenPoint user interface, gestures should propagate up the window tree whenever possible. For example, a scrolling flick made on a label inside a table inside a scrollwin should be passed up to the scrollwin, which will scroll the entire table. Also, a question mark over one button of an exclusive choice should generate help on the choice, rather than Help on the particular button of the choice.

How this works in practice is `clsGWin` has a forwarding flag in `GWIN_STYLE` (`gestureForward`) set by default. When the user makes a gesture over a gesture
window, it self-sends `msgGWinGesture`. If no descendant class handles `msgGWinGesture`, the message ends up at `clsGWin`. If the gesture is not the Help gesture, `clsGWin` returns `stsRequestForward`. If `clsGWin` gets `stsRequestForward` back from sending `msgGWinGesture`, it assumes that this object did not handle the gesture.

If `gestureForward` is set, `clsGWin` then propagates the message by self-sending `msgWinSend` with the sent message field set to `msgGWinForwardedGesture`. `msgWinSend` propagates up the window tree. If it arrives at `clsGWin`, `clsGWin` unpacks the `msgGWinForwardedGesture` message and self-sends it. If `msgGWinForwardedGesture` returns `stsRequestForward`, `msgWinSend` goes to the ancestor class.

This scheme allows components to be embedded in other windows (possibly non-toolkit windows) without regard for tying them to a client, and still let some window pick up their gestures. For example, all the frame components have their gestures passed on via `msgWinSend` of `msgGWinForwardedGesture`, and the frame would pick them up.

The `GWIN_GESTURE` structure for the message arguments to `msgGWinGesture` and `msgGWinForwardedGesture` includes a `uid` field. `clsGWin` sets this field to the UID of the window that received the gesture. Thus, if you have a parent window that handles the gestures in child windows, it can still figure out which child window received the gesture.

### Responding to Gestures

If you create your own UI component class and want your class to respond to gestures, your class should intercept `msgGWinGesture`. If your class recognizes the gesture, it should respond to it and return `stsOK`. If your class doesn't recognize the gesture, it should pass the gesture to its ancestor class. If none of the ancestors handle the gesture, `clsGWin` will propagate the gesture to the instance's parent window (if `gestureForward` is set).

If you create your own UI component class and want to respond to gestures made in child windows embedded in instances of your class, your class needs to respond to `msgGWinForwardedGesture`. This message takes the same arguments as `msgGWinGesture`. If your class doesn't distinguish between gestures in its own windows and the gestures of their children, your class can handle both gesture messages in the same procedure. You can self-send `msgGWinTransformGesture` to transform the gesture bounds and hot point to the local window coordinate system from the coordinates of the original window.

If your class uses instances of `clsControl` or one of its many descendant classes, your class can get involved in gesture handling by setting up one of its objects (such as the application) to be the client of a control. As explained in Chapter 35, Controls, `clsControl` forwards gestures to its client. A control is sent gesture messages, which it sends to its ancestor class, and then passes to its client object, and then propagates to its parent window.
Help IDs

clsGWin does interpret one gesture: the Help ? gesture. If no other class handles xgsQuestion during all the propagation described above, when it reaches clsGWin, it will try to display Quick Help by sending self msgGWinHelp. You can assign a Quick Help ID to any gesture window (in clsGWin’s metrics), and clsGWin will find the Quick Help resource for that window and display it in the Quick Help window using msgQuickHelpShow. If the window does not have a Quick Help ID, clsGWin propagates msgGWinForwardedGesture in the manner described above.

For more information on creating Quick Help resources, see Part 9: Utility Classes.

Embedded Windows

In PenPoint, you can freely move and copy applications and components into and out of each other. The result that one application or component is embedded inside another. clsEmbeddedWin supports the protocol for moving and copying applications and components by moving their windows.

All UI Toolkit windows are embedded windows. However, the UI Toolkit does not use the features of clsEmbeddedWin much itself, since UI components cannot, in fact, be moved using the press-hold-and-drag technique.

Descendants of toolkit window classes that implement real components, such as text views and application windows, do use clsEmbeddedWin functionality. clsEmbeddedWin uses special move or copy icons during move and copy operations.

For more information on embedded windows and the move/copy protocol, see Part 2: PenPoint Application Framework.

Borders

UI Toolkit components are descendants of clsBorder, which implements much of the rendering and layout ability of UI toolkit windows. See Chapter 33, Border Windows, for more information on clsBorder.
Chapter 33 / Border Windows

Many UI components have similar visual elements, such as background colors, shadows and outlines. To share code, all UI Toolkit classes are descendants of clsBorder (a descendant of clsWin), which draws these elements.

Figure 33-1
Sample Border Windows

<table>
<thead>
<tr>
<th>Tk Demo: Borders</th>
</tr>
</thead>
<tbody>
<tr>
<td>White Background</td>
</tr>
<tr>
<td>White Background, All Black Borders</td>
</tr>
<tr>
<td>Bottom border</td>
</tr>
<tr>
<td>Double Grey Border</td>
</tr>
<tr>
<td>Round Border</td>
</tr>
<tr>
<td>Black Border with Shadow</td>
</tr>
<tr>
<td>Round Border with Shadow</td>
</tr>
<tr>
<td>Grey Background</td>
</tr>
<tr>
<td>Black Border, Shadow, and Resize Handle</td>
</tr>
</tbody>
</table>

**clsBorder Messages**

Table 33-1 summarizes the messages clsBorder defines. The sections following discuss these message in more detail.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_BORDER_NEW</td>
<td>Initializes the BORDER_NEW structure to default values.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_BORDER_NEW</td>
<td>Creates a border window.</td>
</tr>
<tr>
<td>msgBordergetStyle</td>
<td>P_BORDER_STYLE</td>
<td>Passes back the current style values.</td>
</tr>
<tr>
<td>msgBorderSetStyle</td>
<td>P_BORDER_STYLE</td>
<td>Sets all of the style values.</td>
</tr>
<tr>
<td>msgBorderSetStyleNoDirty</td>
<td>P_BORDER_STYLE</td>
<td>Sets all of the style values.</td>
</tr>
<tr>
<td>msgBorderGetLook</td>
<td>P_U16</td>
<td>Passes back value of style.look.</td>
</tr>
</tbody>
</table>

continued
### Table 33-1 (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgBorderSetLook</td>
<td>U16 (bsLook...)</td>
<td>Sets style.look as in msgBorderSetStyle.</td>
</tr>
<tr>
<td>msgBorderSetPreview</td>
<td>BOOLEAN</td>
<td>Sets style.preview as in msgBorderSetStyle.</td>
</tr>
<tr>
<td>msgBorderGetPreview</td>
<td>P_BOOLEAN</td>
<td>Passes back value of style.preview.</td>
</tr>
<tr>
<td>msgBorderSetSelected</td>
<td>BOOLEAN</td>
<td>Sets style.selected as in msgBorderSetStyle.</td>
</tr>
<tr>
<td>msgBorderGetSelected</td>
<td>P_BOOLEAN</td>
<td>Passes back value of style.selected.</td>
</tr>
<tr>
<td>msgBorderPropagateVisuals</td>
<td>pNull</td>
<td>Propagates visuals to children.</td>
</tr>
<tr>
<td>msgBorderSetDirty</td>
<td>BOOLEAN</td>
<td>Recursively sends msgBorderSetDirty to each child.</td>
</tr>
<tr>
<td>msgBorderGetDirty</td>
<td>P_BOOLEAN</td>
<td>Passes back true if any child responds to msgBorderGetDirty with true, otherwise passes back false.</td>
</tr>
<tr>
<td>msgBorderGetForegroundRGB</td>
<td>P_SYSDC_RGB</td>
<td>Passes back foreground RGB to use given current visuals.</td>
</tr>
<tr>
<td>msgBorderGetBackgroundRGB</td>
<td>P_SYSDC_RGB</td>
<td>Passes back background RGB to use given current visuals.</td>
</tr>
<tr>
<td>msgBorderInkToRGB</td>
<td>P_SYSDC_RGB</td>
<td>Maps ink value (bsInkGray66, for example) to RGB.</td>
</tr>
<tr>
<td>msgBorderRGBToInk</td>
<td>P_SYSDC_RGB</td>
<td>Maps RGB value to ink value.</td>
</tr>
<tr>
<td>msgBorderSetVisuals</td>
<td>P_BORDER_STYLE</td>
<td>Sets only the visual fields from pArgs.</td>
</tr>
<tr>
<td>msgBorderGetBorderRect</td>
<td>P_RECT32</td>
<td>Passes back the rect on the border.</td>
</tr>
<tr>
<td>msgBorderInsetToBorderRect</td>
<td>P_RECT32</td>
<td>Assumes given rect is window bounds, insets to border rect as in msgBorderGetBorderRect.</td>
</tr>
<tr>
<td>msgBorderGetInnerRect</td>
<td>P_RECT32</td>
<td>Passes back the rect after the inner margin.</td>
</tr>
<tr>
<td>msgBorderInsetToInnerRect</td>
<td>P_RECT32</td>
<td>Assumes given rect is window bounds, insets to inner rect as in msgBorderGetInnerRect.</td>
</tr>
<tr>
<td>msgBorderGetMarginRect</td>
<td>P_RECT32</td>
<td>Passes back the rect after the border.</td>
</tr>
<tr>
<td>msgBorderInsetToMarginRect</td>
<td>P_RECT32</td>
<td>Assumes given rect is window bounds, insets to margin rect as in msgBorderGetMarginRect.</td>
</tr>
<tr>
<td>msgBorderGetOuterSize</td>
<td>P_SIZE32</td>
<td>Passes back the sum of the border, margin and shadow sizes for width and height.</td>
</tr>
<tr>
<td>msgBorderGetOuterSizes</td>
<td>P_RECT32</td>
<td>Passes back the breakdown of the outer size requirements.</td>
</tr>
<tr>
<td>msgBorderGetOuterOffsets</td>
<td>P_RECT32</td>
<td>Passes back the distance from the outer edge to the border rect in each dimension.</td>
</tr>
</tbody>
</table>

#### Border Geometry Messages

- **msgBorderGetBorderRect**: Passes back the rect on the border.
- **msgBorderInsetToBorderRect**: Assumes given rect is window bounds, insets to border rect as in msgBorderGetBorderRect.
- **msgBorderGetInnerRect**: Passes back the rect after the inner margin.
- **msgBorderInsetToInnerRect**: Assumes given rect is window bounds, insets to inner rect as in msgBorderGetInnerRect.
- **msgBorderGetMarginRect**: Passes back the rect after the border.
- **msgBorderInsetToMarginRect**: Assumes given rect is window bounds, insets to margin rect as in msgBorderGetMarginRect.
- **msgBorderGetOuterSize**: Passes back the sum of the border, margin and shadow sizes for width and height.
- **msgBorderGetOuterSizes**: Passes back the breakdown of the outer size requirements.
- **msgBorderGetOuterOffsets**: Passes back the distance from the outer edge to the border rect in each dimension.

#### Rendering Messages

- **msgBorderXOR**: Sets the raster-op to XOR and paints the background.
- **msgBorderPaint**: Paints the border, background, shadow, and so on using msgWinBeginPaint.
- **msgBorderFlash**: Flashes self's window by drawing a thick border and erasing it.
- **msgBorderTop**: Brings the border window to the front with optional UI feedback.
- **msgBorderConvertUnits**: Converts values from one unit to another.

**continued**
Creating a Border Window

To create a border window, you send msgNew to clsBorder or one of its many descendants. This takes a pointer to a BORDER_NEW structure. In BORDER_NEW you specify:

- **style**: several style flags

These style flags are (currently) the only field in BORDER_NEW_ONLY. The flags are in a BORDER_STYLE structure and are summarized in Table 33-2.

These are among the most important clsBorder flags. See the PenPoint API Reference and BORDER.H for more information on clsBorder flags.

You can set the style flags after sending msgNew with msgBorderSetStyle.

The border itself is not a separate window, it is just additional graphics drawn inside UI component windows by their common ancestor, clsBorder.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgBorderProvideDeltaWin</td>
<td>P_WIN</td>
<td>Receiver must provide window to be dragged, resized or topped.</td>
</tr>
<tr>
<td>msgBorderProvideBackground</td>
<td>P_BORDER_BACKGROUND</td>
<td>Receiver must provide rect and ink for drawing background.</td>
</tr>
<tr>
<td>msgBorderPaintForeground</td>
<td>VOID</td>
<td>Receiver must paint the foreground, if any.</td>
</tr>
</tbody>
</table>

### Table 33-2

**BORDER_NEW Styles**

<table>
<thead>
<tr>
<th>Styles/Style Flags</th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>edge</td>
<td>Defines which edges have borders. Can be bsEdgeAll, bsEdgeNone, or a combination OR'd together.</td>
</tr>
<tr>
<td>bsEdgeLeft</td>
<td></td>
</tr>
<tr>
<td>bsEdgeLeft</td>
<td></td>
</tr>
<tr>
<td>bsEdgeRight</td>
<td></td>
</tr>
<tr>
<td>bsEdgeTop</td>
<td></td>
</tr>
<tr>
<td>bsEdgeBottom</td>
<td></td>
</tr>
<tr>
<td>join</td>
<td>Describes how borders edges are joined.</td>
</tr>
<tr>
<td>bsJoinRound</td>
<td></td>
</tr>
<tr>
<td>bsJoinSquare</td>
<td></td>
</tr>
<tr>
<td>bsJoinEllipse</td>
<td></td>
</tr>
<tr>
<td>lineStyle</td>
<td>Describes line border thickness.</td>
</tr>
<tr>
<td>bsLineSingle</td>
<td></td>
</tr>
<tr>
<td>bsLineDouble</td>
<td></td>
</tr>
<tr>
<td>bsLineMarquee</td>
<td></td>
</tr>
<tr>
<td>bsLineDashed</td>
<td></td>
</tr>
<tr>
<td>bsLineDoubleMarquee</td>
<td></td>
</tr>
<tr>
<td>bsLineDoubleDashed</td>
<td></td>
</tr>
<tr>
<td>borderInk</td>
<td>Describes ink to draw border.</td>
</tr>
<tr>
<td>backgroundInk</td>
<td>Describes ink to fill background.</td>
</tr>
<tr>
<td>Styles/Style Flags</td>
<td>Functional Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>shadow</td>
<td>Describes border edge shadow style.</td>
</tr>
<tr>
<td></td>
<td>Use shadowThickness and shadowInk.</td>
</tr>
<tr>
<td>bottomMargin, leftMargin, rightMargin, topMargin</td>
<td>Describes the four inner margin sizes, in marginUnits (described below), of the component’s window. Value can be anywhere from 0-15; some common values predefined.</td>
</tr>
<tr>
<td></td>
<td>No inner margin.</td>
</tr>
<tr>
<td></td>
<td>One margin unit.</td>
</tr>
<tr>
<td></td>
<td>Two margin units.</td>
</tr>
<tr>
<td></td>
<td>Eight margin units.</td>
</tr>
<tr>
<td>marginUnits</td>
<td>Units for the four inner margins.</td>
</tr>
<tr>
<td></td>
<td>Use layout units (described below).</td>
</tr>
<tr>
<td></td>
<td>Use device units (pixels).</td>
</tr>
<tr>
<td></td>
<td>Use twips (0.05 points).</td>
</tr>
<tr>
<td></td>
<td>Use points (1/72 inch).</td>
</tr>
<tr>
<td></td>
<td>Use system font line size.</td>
</tr>
<tr>
<td></td>
<td>Use rules (0.05 lines).</td>
</tr>
<tr>
<td></td>
<td>Use units of 0.01 millimeters.</td>
</tr>
<tr>
<td></td>
<td>Use mils (0.001 inch).</td>
</tr>
<tr>
<td></td>
<td>System font line size.</td>
</tr>
<tr>
<td></td>
<td>Use rules (0.05 lines).</td>
</tr>
<tr>
<td></td>
<td>Compute units at layout time.</td>
</tr>
<tr>
<td></td>
<td>Compute units at layout time and preserve aspect ratio.</td>
</tr>
<tr>
<td>shadowGap</td>
<td>Determines whether the shadow contains notches.</td>
</tr>
<tr>
<td></td>
<td>No corner gaps.</td>
</tr>
<tr>
<td></td>
<td>Clear gaps to white.</td>
</tr>
<tr>
<td></td>
<td>Don’t paint gaps.</td>
</tr>
<tr>
<td>resize</td>
<td>Determines if UI component window has resize handles. A set of flags.</td>
</tr>
<tr>
<td></td>
<td>No resize handles; the default.</td>
</tr>
<tr>
<td></td>
<td>Lower right corner handle. User can resize in both dimensions.</td>
</tr>
<tr>
<td></td>
<td>Bottom edge handle. User can resize vertically.</td>
</tr>
<tr>
<td></td>
<td>Right edge handle. User can resize horizontally.</td>
</tr>
<tr>
<td></td>
<td>A combination of bsResizeBottom, bsResizeCorner, and bsResizeRight.</td>
</tr>
<tr>
<td>drag</td>
<td>Determines if and how user can drag UI component.</td>
</tr>
<tr>
<td></td>
<td>No dragging.</td>
</tr>
<tr>
<td></td>
<td>Drag when user press-holds down pen.</td>
</tr>
<tr>
<td></td>
<td>Drag when user touches pen.</td>
</tr>
<tr>
<td></td>
<td>Drag when user moves pen while down beyond threshold.</td>
</tr>
<tr>
<td>top</td>
<td>Determines if and how user can bring border window to front of its siblings.</td>
</tr>
<tr>
<td></td>
<td>Never top window.</td>
</tr>
<tr>
<td></td>
<td>Top window when user lifts pen.</td>
</tr>
<tr>
<td></td>
<td>Top window after user drags it.</td>
</tr>
</tbody>
</table>
**The bsUnitsLayout Measurement**

The bsUnitsLayout unit is scaled so that eight horizontal units are the width of a typographer's em in the system font, and eight vertical units are the height of an em.

For example, if the user sets the system font to eight points, then a bsUnitsLayout unit is one point; if the user sets the system font to 16 points, a bsUnitsLayout unit is two points. Also, an em is always square, regardless of whether any character in the font is square. Therefore, bsUnitsLayout units are always square. For more information on font geometry, see *Part 3: Windows and Graphics*.

---

**Painting a Border Window**

In repaint, first clsBorder draws the background and border, then descendant classes draw the contents of the window. If the window’s size is fixed and the contents are too large, they may overflow the margin, border, and shadow. Descendants should send `msgBorderGetInnerRect` to figure out the area in which they should draw.

**Painting the Border**

`msgDCDrawRect` draws borders with square or rounded corners and the specified thickness. If you specify that only some edges should be drawn, clsBorder draws only those edges, maintaining the join style. If the join style is `bsJoinEllipse`, clsBorder uses `msgDCDrawEllipse` to draw the joint.
Painting the Background

3clsBorder fills the background with backgroundInk when it calls msgDcDrawRect.

In a rounded-corner border window, there are gaps in the corners that the border line will not touch. If the border window shares clipping with its parent (wsParentClip is set to true) and the parent can write on its children (wsClipChildren is false), clsBorder relies on the parent filling in these dog-ears. Otherwise, clsBorder clears the entire background rectangle to the background ink. If the background ink is bsInkTransparent, the background is not drawn at all.

Painting Background and Foreground

If a clsBorder instance isn't selected or being previewed, it uses the backgroundInk you specify. If selected or previewing, then the background ink is changed to look like the window is selected or previewing. What this looks like depends on which subclass of clsBorder you are working with.

You can't set the foreground ink directly. Instead, you choose a look, which gives you either a black or gray foreground color. Rather than directly controlling the foreground color and style of painting, you set certain properties of the border window in its BORDER_STYLE field that affect painting. These properties are called the visuals of the bordered window. They are the look, preview, selected, and backgroundInk fields in the border style.

The look (bsLookActive or bsLookInactive), preview, and selected fields all affect the foreground color. The look takes precedence.

Descendants and Colors

It's entirely possible for you to create your own toolkit component classes. However, if these are to work well with option sheets, menus, and other existing components, they need to respond to changes in visuals. Rather than reimplement all the border messages, it's easiest to inherit from clsBorder and let it handle these messages.

It is much better to change visuals rather than to force windows to use a different color; using the colors suggested by clsBorder will ensure that your own UI gadgets have a similar interaction behavior to PenPoint's own components.

If, when you receive msgWinRepaint, you do inherit from clsBorder, you should probably use the same colors for drawing as all other buttons and labels use. clsBorder provides several messages that help you translate its current state into specific colors to use, without having to hard-code the relationships between different visuals, selected state, and colors. To get the correct foreground and background colors, send msgBorderGetForegroundRGB and msgBorderGetBackgroundRGB. clsBorder also provides msgBorderInkToRGB, which maps the ink value of the line used for the border to an RGB value, and its inverse, msgBorderRGBToInk.
Providing Custom Backgrounds

clsBorder provides another way for its descendants to adjust background painting. When style.preview or style.selected is on, clsBorder self-sends msgBorderProvideBackground during repaint. Subclasses can intercept this message and alter the background ink and the rectangle used by clsBorder (or both) during repaint of the background. This takes a pointer to a BORDER_BACKGROUND structure, in which you specify these parameters:

- **ink** the background ink to use, for example, bsInkBlack.
- **rect** the rectangle that clsBorder should fill with the background ink.

clsBorder fills these in with the ink and color it would use before sending msgBorderProvideBackground to self.

Painting the Shadow

The shadow is also part of the window. It is drawn on the bottom and right of the main part of the window. If the shadowGap style field is bsGapNone, the shadow extends to the left and top of the rest of the window. Otherwise, there are two gaps at the top and left of the shadow. The other values of the shadowGap style field control how these gaps are painted. If the shadowGap is bsGapWhite, they are cleared to white. If it is bsGapTransparent, they are not painted, and what is behind the border window shows through the gaps.

Resizing, Dragging, Topping

If resize is on, the user can grab onto a resize corner or edge of a toolkit window. If drag is set to a style other than bsDragNone, the user can hold the pen down and then drag the window, and if top is set to a style other than bsTopNone, the user can bring the window to the front of its siblings (or push it to the back if it is already at the front). Normally, these actions affect the size and location of self—the user resizes this window, and it changes size.

Delta Window

However, often the window with drag, resize, and/or top enabled is actually inserted in another window. For example, the part of a frame with a resize handle may actually be a child of another window that contains the shadow and tabs, and it is the latter that needs to be dragged or resized when the user adjusts the size. To support this (if getDeltaWin is on), clsBorder uses msgWinSend to send msgBorderProvideDeltaWin up the window tree to get the window dragged, resized, or topped. When clsBorder receives msgBorderProvideDeltaWin, it returns self's UID if getDeltaWin is set to false; otherwise, it lets the message continue to its parent. The result is that the first window found without getDeltaWin set is the one that gets dragged, resized, and topped.
Layout

If you turn on borders and shadows after creating a border, you increase the space needed by a toolkit window, so you should lay out the window again. `clsBorder` dirties the window's layout when you change its style, but it doesn't actually send it `msgWinLayout`. You need to send `msgMinLayout` to the border window or its parent window.

Avoiding Repaint and Relayout

Normally, when you send `msgBorderSetStyle`, the border window figures out whether the change requires laying out the window again or repainting the window. If it repaints one or the other, the border window self-sends `msgWinDirtyRect` and `msgWinSetLayoutDirty`. The former would usually cause the border window to repaint, and the latter would result in the border window being relaid out in the next layout episode. It's up to you to make sure that, at some point, you send `msgWinLayout`. However, if you send `msgBorderSetStyleNoDirty`, neither of these is sent to self, even if the new pixels would dirty the window. This is an optimization for cases where you want to avoid the computation and messages related to calculating the dirty region.

Propagating and Notifying Visuals

To support the nesting of components, you want to create high level components out of other components, and have the high level component work as though it were a single control. Thus, you want the components nested within it to share the same visual appearance. By setting `propagateVisuals`, any changes in visuals (look, preview, selected, and background ink) are propagated to child border windows using `msgBorderSetVisuals`.

If you only want this to happen on demand, you can send `msgBorderPropagateVisuals` to the border window to update child border windows only once.

This works if the components are nested windows. However, if they are not, you can still keep changes in sync by making components observers of the border window and set `notifyVisuals`. Whenever the visuals of the border window change, the observers are sent `msgBorderSetVisuals` (via `msgNotifyObservers`).
**Border Geometry**

There are four regions to each border window, as illustrated in Figure 33-3:

- The outer size of the border window, including border and shadow.
- The size of the bordered area, excluding the shadow.
- The size of the margin area.
- The size of the inner area, excluding the border, shadow, and margin.

The size of the inner rectangle is the size of the margin rectangle less the left, right, top, and bottom margins. You control the size of the margins by setting `leftMargin`, `rightMargin`, `bottomMargin`, and `topMargin` in `BORDER_STYLE` to some value. The contents of the window are drawn by some subclass of `clsBorder`.

The default margin units are layout units, so the margins scale with the size of the system font.

Although a border window is a single window, you can find out the size of the various parts of it using `msgBorderGetBorderRect`, `msgBorderGetInnerRect`, and `msgBorderGetOuterSize`. These all pass back the appropriate dimensions in LWC (device units relative to the lower left corner of the window) as a `RECT32` or `SIZE32`.

Often, the current size of the border window is not the final size, so you want to compute the size of the border rectangle and inner rectangle. `msgBorderInsetToBorderRect` and `msgBorderInsetToInnerRect` compute the border and inner rectangles for the border window as if it were the size of the rectangle passed in.

Conversely, you can self-send `msgBorderGetOuterSize` to get the total size (including the border, margin, and shadow) for a particular size of border window. This is useful for subclasses of `clsBorder` that are trying to compute their size in response to `msgWinLayoutSelf`. 
**Outer Offsets**

The shadow and its border take up some amount of the outside edge of a window. To zoom a window to fill another appropriately, you need to know where the inner part of the window is so you can push all this border out of the way. If you send `msgBorderGetOuterOffsets` to a border window, it passes back a `RECT32`. These aren’t the coordinates of a rectangle; they are the lengths of the offset of this inner area from the four edges. For example, the `origin.x` is the distance from the left edge to the left-most pixel of the inner area, and the `size.h` is the distance from the top edge to the top-most pixel of the inner area.

**Subclassing clsBorder**

If you are implementing a descendant of `clsBorder`, you need to draw inside the rectangle passed back by `msgBorderGetInnerRect`. To respect visuals, you should use `msgBorderGetForegroundRGB` to figure out what color to use for foreground.
Chapter 34 / Layout Classes

The layout window classes orchestrate the recursive layout of a subtree of windows. Broadly speaking, clsTableLayout positions windows in a tabular format, while clsCustomLayout aligns windows relative to each other. You can use these to create pleasing high level user-interface constructs, such as an option sheet. The UI Toolkit itself uses table layout for menus, choices, and other toolkit tables. The built-in frame class is a particular kind of custom layout window.

clsWin defines the layout protocol, its messages, and window flags; see Part 3: Windows and Graphics for a full explanation. As part of a layout episode, windows are told to lay themselves out with msgWinLayoutSelf. Layout uses a model in which the parent gets desired sizes from child windows, then decides how to position and size the children.

You can position windows of any kind using layout windows, not just control windows. For example, if you have two or more windows in your application, you can create a layout window as the frame’s client view, and use the layout window to position them.

Clients ordinarily tell a layout window to position its offspring by sending it msgWinLayout. The window system figures out what subtree of the window hierarchy will be affected by the change in layout, and coaxes windows into position by sending them msgWinLayoutSelf.

**Window Layout**

When a layout window receives msgWinLayoutSelf, it asks each child for its desired size with msgWinGetDesiredSize and, using algorithms from its class and data supplied when it was created, it positions and resizes its children. As a side effect of asking for the child’s desired size, the window system may, in turn, ask the child to lay itself out with msgWinLayoutSelf.

This message is defined by clsWin and is fully explained in Part 3: Windows and Graphics, but here is a review. msgWinLayoutSelf tells a window to determine how best to lay itself out, then tells it to go and do so. Re-laying out usually involves moving child windows around. msgWinLayoutSelf takes a WIN_METRICS structure that contains a flag indicating whether the window can change its own size (wsLayoutResize).

The two layout window classes generally work as follows:

- You specify instructions on how a layout window should lay out its child windows. The layout window instance stores this information, not the children.
- You add child windows to the layout window using msgWinInsert.
When a layout window receives `msgWinLayoutSelf`, it:

- Gets the sizes of its child windows (using `msgWinEnum`).
- Computes new locations (and possibly sizes) for its child windows based on their sizes, possibly their desired sizes, and its specifications for them.
- Sends each child `msgWinDelta` to position it.
  - If positioning the child changed the child’s size, the window system sets the window’s `wsLayoutDirty` bit. Later, when the window system sends the child `msgWinLayoutSelf`, the dirty window will lay itself out to fit the new space.
- If `wsLayoutResize` is set, the layout window can change its own size. You can set up a layout window to **shrink-wrap** around its children by setting its `wsShrinkWrapWidth` and `wsShrinkWrapHeight` window style flags.

### clsTableLayout and clsCustomLayout

Note that the layout window does not have to pay attention to the desired size of child windows. The parent can simply tell them where to go. In the UI Toolkit, most **leaf** controls (non-nesting controls such as labels, buttons, and so on) respond to `msgWinLayoutSelf` by recomputing their size based on the current system font.

**clsTableLayout** lays out its children in an invisible tabular grid. It takes constraints for the row and column dimensions of the grid (for example, three rows tall with each column as wide as the widest entry). By looking at the size of its child windows, it computes the grid layout and positions its children. You can tell **clsTableLayout** to resize its child windows to match the space it computes for them.

**clsCustomLayout** takes constraints for the size and position of each child window (for example, align on left edge, position after left of window A, extend height to top edge of parent) and figures out how to position and size them when it receives `msgWinLayoutSelf`. **clsFrame** uses **clsCustomLayout** to provide a consistently decorated top-level frame window for applications.

Because **clsCustomLayout** requires you to specify layout constraints for each dimension of every window in the layout, it is best suited for a small, relatively fixed number of child windows, such as the **decoration windows** that implement the controls on an application frame.

### Coordinate System

Layout classes take lengths and positions in any of the measurement units that **clsBorder** supports. The lower left corner is the origin, as standard in PenPoint graphics.
Table Layout

`clsTableLayout` lays out child windows in tabular format. You specify constraints for the number of rows and columns, the height of each row, and the width of each column. The table layout window does not store any constraints for particular items, rows, or columns. However, you can tell it to adjust the size of each column to fit its members. For example, you can specify layouts:

- Child windows in three rows and two columns.
- Each row 10 units tall as many as will fit; three columns each as wide as the widest entry.
- Two columns of menu buttons, the buttons in each column to be the same width as the widest button in that column.

All of the UI Toolkit's nested control windows—choices, menus, toggle tables, and so on—are toolkit tables, and `clsTkTable` is a descendant of `clsTableLayout`.

Figure 34-1 shows some sample table layouts.

![Sample Table Layouts](image-url)
**clsTableLayout Messages**

Table 34-1 summarizes the messages clsTableLayout defines.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_TBL_LAYOUT_NEW</td>
<td>Initializes the TBL_LAYOUT_NEW structure to default values.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_TBL_LAYOUT_NEW</td>
<td>Creates a table layout window.</td>
</tr>
<tr>
<td>msgTblLayoutGetMetrics</td>
<td>P_TBL_LAYOUT_METRICS</td>
<td>Passes back current metrics.</td>
</tr>
<tr>
<td>msgTblLayoutSetMetrics</td>
<td>P_TBL_LAYOUT_METRICS</td>
<td>Sets current metrics.</td>
</tr>
<tr>
<td>msgTblLayoutGetStyle</td>
<td>P_TBL_LAYOUT_STYLE</td>
<td>Passes back current style values.</td>
</tr>
<tr>
<td>msgTblLayoutSetStyle</td>
<td>P_TBL_LAYOUT_STYLE</td>
<td>Sets style values.</td>
</tr>
<tr>
<td>msgTblLayoutXYToIndex</td>
<td>P_TBL_LAYOUT_INDEX</td>
<td>Determines a child zero-based index from an xy position.</td>
</tr>
<tr>
<td>msgTblLayoutAdjustSections</td>
<td>BOOLEAN</td>
<td>Adjusts the border edges and margins of children to correctly reflect a sectioned table.</td>
</tr>
<tr>
<td>msgTblLayoutComputeGrid</td>
<td>P_TBL_LAYOUT_GRID</td>
<td>Computes the table grid parameters given the current constraints.</td>
</tr>
<tr>
<td>msgTblLayoutComputeGridXY</td>
<td>P_TBL_LAYOUT_GRID</td>
<td>Computes the table grid start xy given the other grid parameters.</td>
</tr>
<tr>
<td>msgTblLayoutFreeGrid</td>
<td>P_TBL_LAYOUT_GRID</td>
<td>Frees any storage allocated by msgTblLayoutComputeGrid.</td>
</tr>
</tbody>
</table>

**Table Layout Structure**

The TBL_LAYOUT_STYLE structure lets you specify a variety of flags which control the behavior of the table layout. Table 34-2 summarizes the flags and their values.

<table>
<thead>
<tr>
<th>Fields/Field Values</th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tblXAlignment,</td>
<td>Determines the alignment of the table. You can specify the alignment of the table grid as a whole within the table layout window. Possible values are the same as for child window alignments, described below.</td>
</tr>
<tr>
<td>tblYAlignment</td>
<td></td>
</tr>
<tr>
<td>childXAlignment</td>
<td>Determines the x alignment of the child windows within their grid cells. If grid cell size is larger than child window, child will be aligned according to this value. You cannot specify different alignments for child windows.</td>
</tr>
<tr>
<td>tAlignLeft</td>
<td>Position child window to the left in its cell.</td>
</tr>
<tr>
<td>tAlignCenter</td>
<td>Center the child window in its cell.</td>
</tr>
<tr>
<td>tAlignRight</td>
<td>Position child window to the right in its cell.</td>
</tr>
<tr>
<td>tAlignBaseline</td>
<td>Position the baseline of the child on the baseline of the cell (described in “Layout Baseline,” below).</td>
</tr>
</tbody>
</table>
### Fields/Field Values

<table>
<thead>
<tr>
<th>Field</th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>childYAlignment</td>
<td>Determines the y alignment of the child windows within their grid cells. If grid cell size is larger than child window, child will be aligned according to this value. You cannot specify different alignments for different child windows.</td>
</tr>
<tr>
<td>tAlignBottom</td>
<td>Place child window at the bottom of its cell.</td>
</tr>
<tr>
<td>tAlignCenter</td>
<td>Center the child window in its cell.</td>
</tr>
<tr>
<td>tAlignTop</td>
<td>Place child window at the top of its cell.</td>
</tr>
<tr>
<td>tAlignBaseline</td>
<td>Position the baseline of the child on the baseline of the cell (described in “Layout Baseline,” below).</td>
</tr>
<tr>
<td>growChildWidth</td>
<td>Determines whether the table layout should resize the width of the child windows to fill their grid cells. For example, to maintain the same highlight width for each button in a menu, menus set growChildWidth so that all menu buttons in a menu are the same width.</td>
</tr>
<tr>
<td>growChildHeight</td>
<td>Determines whether the table layout should resize the height of the child windows to fill their grid cells.</td>
</tr>
<tr>
<td>placement</td>
<td>Determines whether table layout places child windows in the grid cells. Table layout also enumerates child windows in back-to-front order to place children into the grid.</td>
</tr>
<tr>
<td>tlPlaceRowMajor</td>
<td>Top row fills before next row is started.</td>
</tr>
<tr>
<td>tlPlaceColMajor</td>
<td>Left-most column fills before column to the right is started.</td>
</tr>
<tr>
<td>tlPlaceStack</td>
<td>All children are stacked one on top of the other in the first (top left) grid cell.</td>
</tr>
<tr>
<td>tlPlaceOrientation</td>
<td>Table layout uses tlPlaceColMajor if the table's window device is in portrait mode, or tlPlaceRowMajor if it is in landscape mode.</td>
</tr>
<tr>
<td>reverseX, reverseY</td>
<td>Determines placement order of child windows. By default, clsTableLayout places child windows from left to right, top to bottom. Setting reverseX or reverseY reverses the order of placement. For example, tab bars use this to get tabs to overlap appropriately. Bookshelf also uses reverse to lay out icon from bottom of screen upwards.</td>
</tr>
<tr>
<td>wrap</td>
<td>Determines whether to wrap long row or column to next row or column.</td>
</tr>
<tr>
<td>widthExtra, heightExtra</td>
<td>Determines what to do with extra space.</td>
</tr>
<tr>
<td>tExtraNone</td>
<td>Leave extra space as is.</td>
</tr>
<tr>
<td>tExtraFirst</td>
<td>Put extra space before first row or column.</td>
</tr>
<tr>
<td>tExtraAfterFirst</td>
<td>Put extra space after first row or column.</td>
</tr>
<tr>
<td>tExtraLast</td>
<td>Put extra space after last row or column.</td>
</tr>
<tr>
<td>tExtraBeforeLast</td>
<td>Put extra space before last row or column.</td>
</tr>
<tr>
<td>tExtraAll</td>
<td>Divide extra space and add evenly to each row or column.</td>
</tr>
<tr>
<td>tExtraBetweenAll</td>
<td>Divide extra space and add evenly after each row or column.</td>
</tr>
</tbody>
</table>

**clsTableLayout** positions each child in the grid cell formed by the row/column intersections. If you set `growChildWidth` or `growChildHeight`, the table layout window resizes child windows to fit their grid cells in that dimension. Otherwise, child windows are aligned within their grid cells according to the table’s `childXAlignment` and `childYAlignment` styles. If a child window is too big it will overlap some edge of its cell.

### Specifying the Table Layout

You specify the number of rows and columns in two `TBL_LAYOUT_COUNT` structures, one for rows (numRows) and one for columns (numCols). In this, you specify:

- **constraint** a table constraint, as described above in Table 34-2.
- **value** an optional absolute number of rows or columns.
You specify row height and column width in two TBL_LAYOUT_SIZE structures, one for all rows (rowHeight), another for all columns (colWidth). In this you specify:

- **constraint**: a table constraint, as described in Table 34-2.
- **value**: an optional absolute height or width.
- **gap**: the gap to put between rows or columns.
- **baseline**: an optional absolute baseline alignment, as described below.

PenPoint 1.0 implements only limited support for baseline alignment.

- **valueUnits**: the units in which to interpret the value field. This can be any unit understood by clsBorder (bsUnitsLayout, for example). See Chapter 33, Border Windows, for more information on border units.

The gap is only between rows and columns.

### Table Layout Constraints

Table 34-3 lists the possible values for a TBL_LAYOUT_CONSTRAINT. The same constraints, more or less, can apply to both the number of rows and columns (in a TBL_LAYOUT_COUNT structure) and the size of rows and columns (in a TBL_LAYOUT_SIZE structure). If you specify a constraint that does not apply, you will get back stsTblLayoutBadConstraint from msgWinLayoutSelf.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Meaning for numRows/Cols</th>
<th>Meaning for ColWidth/RowHeight</th>
</tr>
</thead>
<tbody>
<tr>
<td>tlAbsolute</td>
<td>Fixed number of rows or columns given by value.</td>
<td>Fixed size given by value.</td>
</tr>
<tr>
<td>tlChildrenMax</td>
<td>Not applicable.</td>
<td>Size is max of all children in table.</td>
</tr>
<tr>
<td>tlGroupMax</td>
<td>Not applicable.</td>
<td>Size is max of children in column or row.</td>
</tr>
<tr>
<td>tlMaxFit</td>
<td>As many as will fit.</td>
<td>As wide or as high as possible.</td>
</tr>
<tr>
<td>tlInfinite</td>
<td>Unbounded number of rows or columns.</td>
<td>Not applicable.</td>
</tr>
</tbody>
</table>

Using **tlMaxFit** for the number of rows or columns means as many rows or columns as will fit given the height or width, row or column gap, and parent size. Using **tlMaxFit** for row height or column width means as high or wide as possible, given the number of rows or columns, row or column gap, and parent size.

If you OR **tlBaselineBox** with **tlGroupMax**, the constraints will take the maximum of the ascender and descender of each child in the group. OR-ing **tlMaxFitIfConstrained** with any of the colWidth/rowHeight constraints gives you the equivalent of using **tlMaxFit**, but only if the width or height is constrained during layout (for example, if one of wsLayoutResize, wsShrinkWrapWidth, or wsShrinkWrapHeight is not set).

When it receives **msgWinLayoutSelf**, clsTableLayout uses the grid size and row and height information to create a grid of child window locations. It enumerates the table layout window’s children using **msgWinEnum** and lays them out on this grid. It places children in the table in window-depth order (bottom-most child
window to top-most child window). To later reorder the child windows, you must use msgWinInsert, msgWinInsertSibling, or msgWinSort.

If the rowHeight or colWidth is tlChildrenMax or tlGroupMax, clsTableLayout asks each child window in the table (or only in the row or column) for its desired size (by sending it msgWinGetDesiredSize). It computes the largest dimension in the table (or only in the row or column) and uses that as the size. Note that this is independent of growChildHeight or growChildWidth. The latter tells clsTableLayout to size each child window to fit its cell in the table, but this takes place after clsTableLayout computes the grid for the table.

If the table layout window has more children than space in the grid, clsTableLayout does not lay out the children past the last column of the last row, so their positions are left unchanged.

### Layout Baseline

Windows have the notion of a **baseline**, a line that windows should be positioned on to look good, just as there is a baseline for text along which individual letters are positioned. clsTableLayout lets you position child windows relative to this baseline so that they align pleasingly.

The message to determine the baseline for a window is msgWinGetBaseline. This passes back the vertical offset of the baseline and a horizontal offset of the baseline; the default for clsWin is (0, 0). The horizontal baseline allows label decorations (such as the check mark on a toggle) to hang in the margin of a vertical set of labels, as in the Clock option sheet in Figure 34-2.

**Figure 34-2**

**Horizontal Window Baseline Alignment in an Option Sheet**
Using tlAlignBaseline for Table Layout

If childXAlignment or childYAlignment are set to tlAlignBaseline, the table layout positions child windows with their baselines on the baseline of the row or column. In the current PenPoint release, the only way to determine the row or column baseline is to have the table compute it when it receives msgWinGetBaseline.

In order to support this behavior, you must OR the tlBaselineBox flag into the TBL_LAYOUT_CONSTRAINT setting for the rowHeight or colWidth constraints. Baseline alignment makes sense only in conjunction with tlGroupMax or tlChildrenMax. Normally, if the constraint is tlGroupMax or tlChildrenMax, clsTableLayout gets the size of each child window and computes the largest size. If tlBaselineBox is OR'd in, clsTableLayout gets the baseline of each child window as well as its size, and computes the row height or column width as the sum of the largest ascender offset and largest descender offset of all the child windows. It doesn't really make sense not to set the child alignment to tlAlignBaseline if you do this.

Layout Example: a Calculator

Menus are simple table layout windows, which arrange their child windows (menu buttons) in a row (for example, your application’s menu bar) or in columns. A more complex example is the Calc sample program in the SDK directory PENPOINT\SDK\SAMPLE\CALC. The Calc program’s main window positions its calculator buttons in a matrix with gaps between them.

Locations

To convert an x-y location in a table layout window to an index in the table, send the table msgTblLayoutXYToIndex. This takes a pointer to a TBL_LAYOUT_INDEX structure. In this message, you specify the location in the local window coordinates of the table (xy, an XY32 structure). clsTableLayout passes back an index, such that if a child window were inserted there and the table laid out, the new child would be at the specified x-y location. index is the zero-based count of the new window in the child hierarchy of the table.

Calc’s Positioning of Child Window Using clsTableLayout
Note that neither clsWin nor clsTableLayout defines a message to insert a new window as the Nth child of another. It is up to you to figure out the UID of the window at index and specify this in msgWinInsertSibling. However, clsTkTable, a descendant of clsTableLayout used in the UI Toolkit, provides a msgTkTableAddAt message.

The message is useful if you want to respond to gestures in a table layout window by creating a new window in the table.

**Custom Layout**

clsCustomLayout is the opposite of clsTableLayout. It does not take any overall specifications, except shrink-to-fit (see below). Instead, you specify constraints for each child window. You can achieve consistent effects, such as placing one window adjacent to another or making one window the same width as another. Or, you can be strict, and specify the absolute size and position of a window.

**Custom Layout Example: the Frame**

The most common example of a custom layout in PenPoint is a frame. Most applications appear in a page-sized frame, a container that surrounds (or decorates) the application’s own views with a title bar, a resize corner, a frame header, scrollbars, and so on. clsFrame is a descendant of clsCustomLayout and uses custom layout to position its child windows. There are no hard-coded locations for any of the child windows in clsFrame, so frames can be resized while preserving their appearance. This is important because the user can resize a floating frame at will, and the frame must accommodate whatever size the user sets.
Table 34-4 summarizes the messages clsCustomLayout defines.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_CSTM_LAYOUT_NEW</td>
<td>Creates a custom layout window.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_CSTM_LAYOUT_NEW</td>
<td>Initializes the CSTM_LAYOUT_NEW structure to default values.</td>
</tr>
<tr>
<td>msgCstmLayoutGetMetrics</td>
<td>P_CSTM_LAYOUT_METRICS</td>
<td>Passes back current metrics.</td>
</tr>
<tr>
<td>msgCstmLayoutSetMetrics</td>
<td>P_CSTM_LAYOUT_METRICS</td>
<td>Sets current metrics.</td>
</tr>
<tr>
<td>msgCstmLayoutGetStyle</td>
<td>P_CSTM_LAYOUT_STYLE</td>
<td>Passes back current style values.</td>
</tr>
<tr>
<td>msgCstmLayoutSetStyle</td>
<td>P_CSTM_LAYOUT_STYLE</td>
<td>Sets style values.</td>
</tr>
<tr>
<td>msgCstmLayoutGetChildSpec</td>
<td>P_CSTM_LAYOUT_CHILD_SPEC</td>
<td>Sets layout spec for given child.</td>
</tr>
<tr>
<td>msgCstmLayoutRemoveChildSpec</td>
<td>WIN</td>
<td>Removes the spec for the specified child window.</td>
</tr>
<tr>
<td>CstmLayoutSpecInit()</td>
<td>P_CSTM_LAYOUT_SPEC</td>
<td>Initializes the P_CSTM_LAYOUT_SPEC to zero.</td>
</tr>
</tbody>
</table>

### Creating a Custom Layout Window

You send msgNewDefaults and then msgNew to clsCustomLayout. In the CSTM_LAYOUT_NEW_ONLY structure, the only field (currently) is CSTM_LAYOUT_STYLE, and in this the only field (currently) is limitToRootWin, which limits the size of a custom layout window so that it fits on-screen.

### Specifying Constraints

When you insert a child window into a custom layout window, at some point you will need to specify how the custom layout window positions the child window. The next sections explain the form of this specification. There are two ways to specify child window positioning:

- Specify a static constraint for the child window using msgCstmLayoutSetChildSpec. The custom layout window stores the constraint for the child window.
- Subclass clsCustomLayout and respond to msgCstmLayoutGetChildSpec by supplying a constraint for the child window.
CHAPTER 34 / LAYOUT CLASSES
Creating a Custom Layout Window

The first method lets you create instances of clsCustomLayout directly. However, the constraint information for each child window takes up memory (in addition to the code filling in the constraint), so if you have a lot of windows in a custom layout, the information uses up a lot of memory. The second method requires you to create a descendant of clsCustomLayout that responds to
msgCstmLayoutGetChildSpec. The advantage is that you save memory.

Both messages take the same CSTM_LAYOUT_CHILD_SPEC structure. In both cases, someone gives clsCustomLayout the information it needs to position the child window; the difference is that in one a client (your application) is sending a message telling clsCustomLayout what the child constraint specification is, while in the other, your descendant of clsCustomLayout must respond to a message.

Four Child Window Constraints

CSTM_LAYOUT_CHILD_SPEC specifies the child window and its
CSTM_LAYOUT_SPEC. The latter is the structure in which you specify
four separate constraints for the child window’s origin (x and y of its lower
left corner) and for its width and height. The possible constraints for these
dimensions (they are really two dimensions and an origin) are the same,
and each is stored in a CSTM_LAYOUT_DIMENSION structure, described later.

The hierarchy of the nested structures is as follows:
A CSTM_LAYOUT_CHILD_SPEC specifies a child window and its
CSTM_LAYOUT_SPEC.
The CSTM_LAYOUT_SPEC specifies four CSTM_LAYOUT_DIMENSIONs.
Each CSTM_LAYOUT_DIMENSION specifies the constraints for a particular
dimension.
CSTM_LAYOUT_CHILD_SPEC also includes parentShrinkWrapWidth
and parentShrinkWrapHeight. When clsCustomLayout self-sends
msgCstmLayoutGetChildSpec, it sets these two booleans to indicate whether
the custom layout window (the parent) is shrink-to-fit in either dimension. In
some situations, you want the child to have different constraints if the parent is
going to adjust its size.

Custom Layout Dimensions

Each of the four CSTM_LAYOUT_DIMENSION structures (for the x-y, width,
and height of a child window) specifies the following values:

  constraint a particular constraint enumerated in
        CSTM_LAYOUT_CONSTRAINT (with additional optional
        specifications OR’d in).
  value an optional offset (or absolute value). valueUnits, below, specifies
         the unit of measurement.
  valueUnits the unit of measurement for value. valueUnits can be any unit
         that clsBorder recognizes, such as bsUnitsLayout.
relWin the window relative to which clsCustomLayout applies the constraint (if applicable). If objNull, then constraint is relative to the parent window (the custom layout window itself).
relWinTag the tag of the window relative to which clsCustomLayout applies the constraint (if applicable).

**Constraints**

The constraint field indicates the kind of constraint you want to apply to the child window's x-y, width, or height. clsCustomLayout supports many different layout techniques within the limitations of C structure syntax, so the specification of constraints can be difficult to explain, but if you understand how to set them up, you can specify almost any window layout. One way to learn is by looking at other programs that use custom layout.

The six basic types of constraints are listed in Table 34-5.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Meaning for x/y</th>
<th>Meaning for width/height</th>
</tr>
</thead>
<tbody>
<tr>
<td>clAsIs</td>
<td>Leave unchanged.</td>
<td>Use desired size.</td>
</tr>
<tr>
<td>clAbsolute</td>
<td>Value gives absolute offset.</td>
<td>Value gives absolute size.</td>
</tr>
<tr>
<td>clPctOf</td>
<td>Value * relWin edge / 100.</td>
<td>Value * relWin size/100.</td>
</tr>
<tr>
<td>clBefore</td>
<td>Position one pixel less than relWin's.</td>
<td>Size one pixel less than relWin's.</td>
</tr>
<tr>
<td>clSameAs</td>
<td>Position same as relWin's.</td>
<td>Size same as relWin's.</td>
</tr>
<tr>
<td>clAfter</td>
<td>Position one pixel greater than relWin's.</td>
<td>Size one pixel greater than relWin's.</td>
</tr>
</tbody>
</table>

If the constraint for a dimension is clAsIs, do not change the dimension's value. For width and height, the custom layout window sends msgWinGetDesiredSize to the child, and uses the values the child passes back. value and relWin are ignored.

If the constraint clAbsolute is set for x or y, the value specifies an absolute position relative to the parent window in border units. If clAbsolute is set for width or height, the value specifies a width or height.

If the constraint is clPctOf, value specifies a percentage (for example 50% if value is 50) relative to relWin.

The three remaining constraints all align this window with relWin. Sometimes you want the windows to touch, other times you want one window to be exactly adjacent (one pixel before or after) to the border of the other window. Hence, the three flavors, clBefore, clSameAs, and clAfter. The alignment isn't strictly with the edge of the other window; clsCustomLayout sends msgBorderGetOuterOffsets to the child window to find out the inset (if any) from the child window's outer edge to its border rectangle.
Picking an Edge for Alignment Constraints

When you align windows, you are saying "I want this edge of this window to touch (or be next to) this edge of the other." Rather than #define dozens of possible alignments, clsCustomLayout provides a macro, CIAlign(), which generates the appropriate constraint information with three parameters:

- The child edge to align.
- The type of alignment.
- The edge of relWin to align with.

For both the child window and the relWin, clsCustomLayout lets you pick from several different edges to align:

- clMinEdge the left or bottom edge.
- clCenterEdge the middle of the side.
- clMaxEdge the right or top edge.
- clBaselineEdge the baseline of the window as reported by msgWinGetBaseline.

The alignment type can be clBefore, clSameAs, clAfter, or clPctOf (with some restrictions). The two edges in the macro can be independently set to clMinEdge, clCenterEdge, or clMaxEdge. CIAlign() returns a bit pattern, which you set as the x or y constraint in CSTM_LAYOUT_DIMENSION. For example:

```
metrics.x.constraint = CIAlign(clMaxEdge, clBefore, clMaxEdge);
```

will place this window's right edge on the pixel to the left of relWin's right edge as defined by msgBorderGetOuterOffsets. You might use this to place a component right-adjusted inside a window without obscuring the window's border.

Aligning Width and Height Dimensions

Aligning width or height is similar. You can tell clsCustomLayout to extend the width or height of a window so that it aligns with the edge of another window. The CIExtend() macro generates the constraint for you. For example:

```
metrics.w.constraint = CIExtend(clAfter, clMinEdge);
```

will extend the width of a window so that it ends one pixel to the right of relWin's left edge as defined by msgBorderGetOuterOffsets.

It is possible for constraints to conflict or be unresolvable. For example, the above two constraints fail to determine an x position for this window if relWin's x position is the same as relWin's width.
Figure 34-5 shows some examples of interesting layout constraints.

relWin is parentWin and value is 0 unless otherwise indicated. The width constraint is clAsIs, in most cases.

1  \[ x = \text{clAlign(clMaxEdge, clBefore, clMaxEdge)} \]
   \[ y = \text{clAlign(clMinEdge, clAfter, clMaxEdge)} \]

2  \[ x = \text{clAlign(clMinEdge, clAfter, clMinEdge)} \]
   \[ y = \text{clAlign(clMaxEdge, clBefore, clMaxEdge)} \]

3  \[ x = \text{clAlign(clMinEdge, clSameAs, clMaxEdge)} \]
   \[ y = \text{clAlign(clMaxEdge, clSameAs, clMaxEdge)} \]

4  \[ x = \text{clAlign(clCenterEdge, clSameAs, clCenterEdge)} \]
   \[ y = \text{clAlign(clMaxEdge, clBefore, clMinEdge)} \]
   with relWin set to the window marked 2 in the illustration, and width set to clPctOf, value 32.

5  \[ x = \text{clAlign(clMaxEdge, clSameAs, clMinEdge)} \]
   \[ y = \text{clAlign(clMaxEdge, clSameAs, clMinEdge)} \]
   with relWin set to the window marked 4 in the illustration.

6  \[ x = \text{clAlign(clMinEdge, clSameAs, clMinEdge)} \]
   \[ y = \text{clAlign(clMinEdge, clSameAs, clMinEdge)} \]

7  \[ x = \text{clAlign(clMinEdge, clAfter, clMaxEdge)} \]
   \[ y = \text{clAlign(clMaxEdge, clBefore, clMinEdge)} \]
**Additional Constraint Flags**

There are several additional flags that you can OR in to any `CSTM_LAYOUT_CONSTRAINT` to produce additional effects. Some combinations of constraints, edge specifications, and flags are meaningless or nonsensical.

You can OR the `clOpposite` flag with `clBefore`, `clSameAs`, or `clAfter` to refer to the opposite dimension. Usually, you align the x position with the x position of another window, but it is possible to align with the y position instead. You can use this to ensure square windows, for example, to set the height of a window to be the same as its width:

```plaintext
metrics.w.constraint = clSameAs | clOpposite;
metrics.w.relWin = self;
```

**Constraints and Shrink-Wrap**

If the custom layout window is set to shrink-wrap, it will shrink to the minimum size necessary to show all of its children, then lay itself out. If all of the children are sized relative to the layout window, this process will shrink them all out of existence.

If you OR in `clAsIsIfShrinkWrap`, `clsCustomLayout` will check to see if the parent is set to shrink to fit in the relevant dimension, and if so, it will use the child’s size. `clsFrame` uses the `clAsIsIfShrinkWrap` bit in specifying the width and height of the `clientWin`, so that if the frame is shrink-to-fit it will leave the client window its desired size, but if the frame is forced to a particular size (for example, when it’s a page in a Notebook), then the child window will fill it.

Another variant is that if the custom layout is shrink-to-fit, it should leave a child window no smaller than its desired size. You can do this by OR'ing in `clAtLeastAsIs`. An example is the command bar at the bottom of an option sheet. If the frame is shrink-to-fit, it should not clip off the end of the command bar, but if other components in the option sheet lead to a wider frame, then the command bar should be expanded to the width of the frame.

Finally, if you wish to exclude a width or height constraint from the shrink-wrap computation, OR `clShrinkWrapExclude` with the dimension you wish to exclude. This is useful if you have overlapping children.

Other points to watch out for with shrink-wrap:

- If the parent is shrink-wrap, any margins around child windows (specified with `value`) will be shrink-wrapped out of existence.

- If the parent is shrink-wrapped, and one or more children have width or height constraint of `clAsIs` and have very large desired sizes, the parent window may be too large to fit on-screen. You can set the custom layout window’s style flag `limitToRootWin` to ensure that it will never be sized larger than the display root window.
Relative Window

Every constraint but clAsIs and clAbsolute takes a window to which it is relative. If it is more convenient, you can specify the relative window by its tag instead of its UID by filling in relWinTag instead of relWin. clsCustomLayout looks up the tag if relWin is objNull. relWin and relWinTag must refer to a sibling window. This means that, for example, you can't align a window with the bookshelf. You have to write your own layout class, which queries the window metrics of the bookshelf, then converts them to your parent's window coordinates.

If relWin and relWinTag are both objNull, the constraint is relative to the parent window. This is the default. If you want relWin to see the parent window, don't use the parent's UID, use objNull.

Value

For all constraints other than clAbsolute, clAsIs, and clPctOf, clsCustomLayout adds the value you specify in CSTM_LAYOUT_DIMENSION to the computed value of the dimension as an offset. For example, if you want to position a child five units to the right of another child, use cIAfter with value set to 5.

For clPctOf, value is the percentage of the relative dimension. For all other constraints, value is an offset or value in some units. Value can be negative.

Custom Layout Initialization

You should always use CstmLayoutSpecInit to initialize a CSTM_LAYOUT_SPEC. This initializes all the constraints to clAsIs, value to zero, and valueUnits to bsUnitsLayout.

More on msgCstmLayoutGetChildSpec

When clsCustomLayout requires the constraint specifications for a child, it always self-sends msgCstmLayoutGetChildSpec. If you subclass clsCustomLayout and respond to this message, you can save on memory, especially if you use a lot of windows. When you receive this, child is the UID of the window that needs a layout spec. metrics has been initialized by CstmLayoutSpecInit.

If you specified a static custom layout constraint, the custom layout window still receives msgCstmLayoutGetChildSpec. If you call your ancestor first, you'll get the static constraint, which you can modify. Your msgCstmLayoutGetChildSpec method should never ObjectCallAncestor after you modify the CSTM_LAYOUT_CHILD_SPEC, since it will ignore and overwrite your changes.
Shrink-Wrap

The window style flags of clsWin include two shrink-to-fit or shrink-wrap flags: wsShrinkWrapWidth and wsShrinkWrapHeight. If a window with either of these flags set receives msgWinLayoutSelf, it can compute a size to fit its contents (it can expand as well as contract).

If a table layout or custom layout window has shrink-wrap turned on when it lays out, it will lay out its children as usual, then it will shrink or expand horizontally or vertically to fit around the bounding box of its child windows. The bounding box is the smallest rectangle that encloses all the child windows. The layout window leaves enough room to support its border (inner margin, border itself, and shadow).

Lazy Layout

In general, the UI Toolkit does not try to guess when windows need laying out. If you change the size of a child window, add a new child window, or add a margin around a child window, the layout window sets its wsLayoutDirty bit, but will not automatically reposition its children. It is up to the application, or possibly the task that has altered the window layout, to decide when to send msgWinLayout. They rely on someone telling them to lay out. After altering a window, send it msgWinLayout to cause a layout episode.

If you want to change several windows at once before laying out, send each msgWinSetLayoutDirty before finally sending msgWinLayout (many changes to the style of UI Toolkit windows do, in fact, mark a window's layout as dirty for you, but they don't directly cause layout).

The exceptions to this lazy layout are scrollwins and option sheets, which self-send msgWinLayout at appropriate times (for example, when changing to a new card on an option sheet).

Layout Loops

You should not specify self-referential layout constraints, that is, constraints that loop. If msgWinLayout determines that it is in a layout loop, it will return stsWinInfiniteLayout, stsTblLayoutLoop or stsCstmLayoutLoop.

In clsTableLayout, if you set the number of rows to tMaxFit (fit in as many rows as possible) and set the height of each row to tMaxFit (make the row height as high as possible), then clsTableLayout cannot determine either.

In clsCustomLayout, because you can use clsSameAs to set the dimension of one window to be the same as another, you can have circular dependencies between windows.
Shrink-Wrap and Parent-Relative Sizing

You can have the extent of the child windows depend on the parent's size in both table layout windows and custom layout windows. If the parent is set to shrink-wrap around its children, you might think that this creates a layout loop—the child is as wide as the parent, but the parent contracts to fit around the child.

Instead, the order of processing is as follows (size here refers to either width or height).

1. The parent lays out its children using a size of (0,0).
2. The parent sizes itself (shrink-wraps) around its children.
3. The parent lays out its children a second time. At this point, it should have shrink-wrapped around all its children of a fixed size, and its children whose size depends on it should have adjusted to that size. The layout episode will continue until the parent window size becomes stable. If the size does not become stable within a fixed number of iterations, the parent returns stsCstmLayoutLoop.

The net effect is that the UI Toolkit can satisfy some requests for both a child that fills its parent and a parent that wraps around its children, so long as there is a window around with a definite size. A simple example of an erroneous situation is a shrink-to-fit custom layout window with a child window whose width or height constraint is 150% of its parent's size.

Capturing vs. Layout

Part 3: Windows and Graphics describes how you can set up a window (using msgWinDeltaOk) so that it can modify or even veto its children's sizes and insert requests. This approach is best when you need to guard against unfriendly child windows, such as embedded applications, which may mess up your window.

On the other hand, the layout semantics described here allows a parent window to position and direct its child windows. Capture is better for incremental layout, in which the parent window handles the insertion and size changes of different child windows. Layout is better for static window organizations that are periodically re-laid out.
Chapter 35 / Controls

clsControl is an important ancestor class for many kinds of UI components. Although its descendant classes have very different visual appearances, clsControl provides them with a similar programmatic interface. clsControl mainly acts to translate user inputs into notification messages which it sends to its client.

Here's a quick sketch of how you use controls in your programs. clsControl is ancestor to a wide variety of toolkit elements: buttons, menu buttons, handwriting fields, page numbers, scrollbars, and so on.

- You create the control you want.
- You specify a client for the control. The control notifies the client when the user activates the control with a gesture (such as a tap). The form of the notification is determined the particular subclass of clsControl that you use.
- Some controls have the notion of a current value which you can set and retrieve.
- You can ask for more detailed notification of the user’s interaction with a control before he or she finally activates it (if at all). As a result, the control self-sends additional preview messages.
- Each notification message (if any) sent by a control is triggered by an intermediate message which clsControl self-sends in response to user input.

Filing Controls

clsControl responds to msgSave. It does not file its client object. However, if the control’s client is the application object (it uses OSThisApp() to check), then during restore, clsControl will set the control’s client to the new application object passed back by OSThisApp(). Otherwise, the control’s client will be objNull after msgRestore, and you will have to go in and reset the client using msgControlSetClient.

Message Dispatching

clsControl defines a set of messages and the circumstances under which controls send them to their clients. clsControl converts pen input event messages in a control window into intermediate control messages which clsControl sends to self. Descendants of clsControl are free to respond to these intermediate messages as they wish.

If the user activates a control, the final message that the control sends itself is msgControlAcceptPreview. Before the control sends this message, the user has been interacting with the control. For example, the user might move the pen tip
in and out of an item in a menu without choosing it. The menu items flicker on and off during this interaction, even though the user hasn’t really initiated any action. This stage is called **previewing**, during which controls can self-send more detailed messages, such as msgControlBeginPreview, msgControlCancelPreview, and so on.

The main descendant of clsControl is clsButton. This responds to msgControlAcceptPreview by notifying the control’s client. A control’s client can be any object; typically it is the one that created the control. The button’s message to the client can be either msgButtonNotify, indicating that the user wants something to happen, or an arbitrary message of your choice, such as msgMyAppQuit.

Some components act like controls even though they do not inherit from clsControl. This state of affairs arises because PenPoint classes inherit from only one line of descent. For example, an exclusive choice responds to msgControlGetValue and msgControlSetValue, but it is not truly a subclass of clsControl.

### Presentation and Interaction Behavior

clsLabel is the workhorse class that depicts controls on-screen. It draws the text, the highlight, and label decoration.

The descendants of clsLabel tell it what presentation style to use for their particular style of label, such as button, toggle, or menu button. They also determine the interaction behavior of the control and tell clsLabel how to draw the various states of previewing.

The descendants of clsControl provide the interaction behavior of particular kinds of controls. Buttons support various kinds of previewing (checking or unchecking, highlighting, and so on).
## clsControl Messages

Table 35-1 summarizes the messages defined by clsControl:

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew Defaults</td>
<td>P_CONTROL_NEW</td>
<td>Initializes the CONTROL_NEW structure to default values.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_CONTROL_NEW</td>
<td>Creates a control window.</td>
</tr>
<tr>
<td>msgControlGetMetrics</td>
<td>P_CONTROL_METRICS</td>
<td>Passes back the current metrics.</td>
</tr>
<tr>
<td>msgControlSetMetrics</td>
<td>P_CONTROL_METRICS</td>
<td>Sets the metrics.</td>
</tr>
<tr>
<td>msgControlGetStyle</td>
<td>P_CONTROL_STYLE</td>
<td>Passes back the current style values.</td>
</tr>
<tr>
<td>msgControlSetStyle</td>
<td>P_CONTROL_STYLE</td>
<td>Sets the style values.</td>
</tr>
<tr>
<td>msgControlGetClient</td>
<td>P_UID</td>
<td>Passes back metrics.client.</td>
</tr>
<tr>
<td>msgControlSetClient</td>
<td>UID</td>
<td>Sets metrics.client.</td>
</tr>
<tr>
<td>msgControlGetDirty</td>
<td>P_BOOLEAN</td>
<td>Passes back true if the control has been altered since dirty was set false.</td>
</tr>
<tr>
<td>msgControlSetDirty</td>
<td>BOOLEAN</td>
<td>Sets style.dirty.</td>
</tr>
<tr>
<td>msgControlGetEnable</td>
<td>P_BOOLEAN</td>
<td>Passes back style.enable.</td>
</tr>
<tr>
<td>msgControlSetEnable</td>
<td>BOOLEAN</td>
<td>Sets style.enable.</td>
</tr>
<tr>
<td>msgControlGetValue</td>
<td>P_S32</td>
<td>Passes back the current value of the control.</td>
</tr>
<tr>
<td>msgControlSetValue</td>
<td>S32</td>
<td>Sets the current value of the control.</td>
</tr>
<tr>
<td>msgControlBeginPreview</td>
<td>P_INPUT_EVENT</td>
<td>Self-sent when msgPenDown is received.</td>
</tr>
<tr>
<td>msgControlUpdatePreview</td>
<td>P_INPUT_EVENT</td>
<td>Self-sent when msgPenMoveDown is received.</td>
</tr>
<tr>
<td>msgControlRepeatPreview</td>
<td>P_INPUT_EVENT</td>
<td>Self-sent if style.repeatPreview is true. Initial delay is 600ms, then immediate repeat until msgPenUp.</td>
</tr>
<tr>
<td>msgControlCancelPreview</td>
<td>P_INPUT_EVENT</td>
<td>Self-sent when style.previewGrab is false and msgPenExitDown is received. Clients or subclasses can send this to a control to cancel existing preview.</td>
</tr>
<tr>
<td>msgControlAcceptPreview</td>
<td>P_INPUT_EVENT</td>
<td>Self-sent when msgPenUp is received when gestureEnable is false, or when xgs1Tap is received when gestureEnable is true.</td>
</tr>
<tr>
<td>msgControlProvideEnable</td>
<td>P_CONTROL_PROVIDE_ENABLE</td>
<td>Sent out to client or specified object during processing of msgControlEnable.</td>
</tr>
<tr>
<td>msgInputEvent</td>
<td>P_INPUT_EVENT</td>
<td>Notification of an input event.</td>
</tr>
</tbody>
</table>

### Attribute Messages

### Intermediate Preview Messages

### Notification Messages
The following sections go through this interface in a top-down order. You usually
do not need to use the lower level API for controls described towards the end.
Moreover, much of the machinery provided by clsControl is used by descendant
classes, such as clsButton, which take care of it for you.

Creating a Control

Controls are an abstract class; there's no reason why you would create an
instance of clsControl itself. However, clsControl handles much of control
interaction and housekeeping, and there are important control initializations
you specify at msgNew time. This information is in the CONTROL_NEWONLY
structure (currently the same as the CONTROL_METRICS structure). The
CONTROL_METRICS structure contains two fields:

- **style** various style fields for the control, described below.
- **client** the client object that will receive all notifications from the
  control. A control has only one client and generally does not notify
  observers—adding observers to a control won't make the control
  send messages to other objects.

You can get and set these with msgControlGetMetrics and
msgControlSetMetrics. clsControl also defines msgControlGetClient
and msgControlSetClient to get and set the client alone, and similar
msgControlGetStyle and msgControlSetStyle messages.

Control Style

The control style is a CONTROL_STYLE structure. Table 35-2 summarizes the
possible fields:

<table>
<thead>
<tr>
<th>Fields/Field Values</th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enable</td>
<td>Describes how the control responds to input.</td>
</tr>
<tr>
<td>dynamicEnable</td>
<td>Describes how the enable value is determined.</td>
</tr>
<tr>
<td>csDynamicNone</td>
<td></td>
</tr>
<tr>
<td>csDynamicClient</td>
<td></td>
</tr>
<tr>
<td>csDynamicObject</td>
<td></td>
</tr>
<tr>
<td>csDynamicArgs</td>
<td></td>
</tr>
<tr>
<td>previewEnable</td>
<td>Determines whether the control should self-send previewing messages.</td>
</tr>
</tbody>
</table>
| previewGrab         | Determines whether previewing starts an input grab. The default is it does not
                     | grab input. |
| previewRepeat       | Determines whether the control sends out additional messages
                     | (msgControlRepeatPreview after a certain period of time. The default is no repeat. |
| previewing          | Determines whether the control is previewing at this moment. The default is no
                     | previewing. |
| dirty               | Determines whether the control is dirty (has been altered). Up to subclasses to maintain
                     | this information. The default is not dirty. |
| showDirty           | Determines whether the control should indicate that it is dirty. Up to subclasses to
                     | make use of this. Default is it does indicate dirty. |
Control Defaults

By default (in the CONTROL_NEW_ONLY structure passed back by msgNewDefaults), a control is enabled, does not grab input, does not repeat, and is not previewing or dirty, and shows dirty (although the last is up to subclasses to implement).

Values

For many descendants of clsControl, it makes sense for the control to have an arbitrary value. For example, buttons are on or off, and scrollbars have a numeric offset. clsControl defines msgControlGetValue and msgControlSetValue, but it is up to descendant classes to respond to these. The value is a signed 32-bit value (§32). Table 35-3 shows examples of values for a number of standard subclasses of clsControl.

The interpretation of the value is up to the descendant class. Some classes may return stsBadParam if you set a value that the descendant doesn’t like; other control classes have no notion of a value and will send self msgNotUnderstood if you try to get or set it.

<table>
<thead>
<tr>
<th>Descendant Class</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>0 for off, 1 for on.</td>
</tr>
<tr>
<td>Label</td>
<td>None.</td>
</tr>
<tr>
<td>Scrollbar</td>
<td>Scrollbar offset.</td>
</tr>
<tr>
<td>Field</td>
<td>Not applicable—the client must get the value Using msgLabelGetString.</td>
</tr>
<tr>
<td>Integer field</td>
<td>Integer value of field (can also use msgLabelGetString).</td>
</tr>
<tr>
<td>Fixed field</td>
<td>Integer value of the field in hundredths.</td>
</tr>
<tr>
<td>Date field</td>
<td>Date value of field as a YYYYMMDD numeric string.</td>
</tr>
</tbody>
</table>

Dirty Controls

The dirty bit field in CONTROL_STYLE indicates if a control’s value has changed. clsControl doesn’t check this; it is up to subclasses to interpret. You can set and get it separate from the other style fields using msgControlSetDirty and msgControlGetDirty. Note that this is different from a dirty window (which needs repainting) or a dirty window layout (which needs to be relaid out).

The related style field showDirty indicates to subclasses whether they should change the way they paint an instance to indicate that it is dirty. For example, clsLabel draws its decoration in black if it is dirty, and in gray if it is not dirty. By default, showDirty is set.

Descendants of clsControl use the dirty bit to indicate those controls in option sheets that the user has fiddled with.
**Control Enable**

A control responds to user input only if its enable bit is set. If the enable bit is clear, the control is read-only. You can disable or enable a control by sending it `msgControlSetEnable` with a message argument of `true` or `false` (or, to modify enable along with other style values, you can send `msgControlGetStyle`, assigning the values, and sending `msgControlSetStyle`). One way for subclasses to disable or enable input is to set the `inputDisable` window input flag to `true` or `false`.

To indicate that a control is inactive, `clsControl` sends `msgBorderSetLook`, with an argument of either `bsLookInactive` or `bsLookActive`. Most controls change from black to gray ink if they are not enabled.

Whether a control is enabled or not is often dynamic. For example, the Move command in the Edit menu should be disabled if there is no selection. The UI Toolkit provides a protocol for the client to ask whether each control should be enabled. This protocol allows items to have their `enable` state either static, or dynamically based on criteria such as:

- A decision from the item's client.
- A decision from the current selection owner.
- Whether the current selection owner's process matches the application's process.

The protocol works between several classes. `clsControl` defines the messages and provides some default responses. Toolkit tables, described in Chapter 38, Toolkit Tables, provide some semantics to check whether controls are enabled when creating a tree of controls. Menu buttons, described in Chapter 39, Menus and Menu Buttons, provide additional semantics for changing the enabled status of the controls in their associated menus.

**Evaluating Control Enable**

When a control receives `msgControlEnable`, it should reevaluate whether it is enabled. `clsControl` does different things based upon the value of its `dynamicEnable` style field. `msgControlEnable` takes a pointer to a `CONTROL_ENABLE` structure as its message arguments. In this the client should specify the following values:

- `root` the window originating the enable check.
- `object` an object to query if the control's style is `csDynamicObject`.
- `enable` a new `enable` value to use if the control's `dynamicEnable` style is `csDynamicPargs`.

If the control's `dynamicEnable` style is:

- `csDynamicNone` then the control returns `stsOK`.
- `csDynamicClient` then the control sends `msgControlProvideEnable` to its client, and then changes its `enable` value to what the client passed back (if the client changed it).
csDynamicObject  then the control sends the same
msgControlProvideEnable message, but it sends it to the object specified
in the message arguments of msgControlEnable.
csDynamicPargs  then the control sets its enable value to match the enable
value specified in the message arguments of msgControlEnable.

**Dynamic Control Enable**

The msgControlProvideEnable message that dsControl sends out if
dynamicStyle is csDynamicClient or csDynamicObject takes a
CONTROL_PROVIDE_ENABLE as its message arguments. In this, the control
specifies between one to four arguments:

- **root**  the window originating the check. This is the same window as specified in msgControlEnable.
- **control**  the message’s UID.
- **tag**  the message’s window tag.
- **enable**  its current enable value.

The receiver of msgControlProvideEnable should pass back the new enable value
of the control. Since the control itself is passed to the receiver of
msgControlProvideEnable, the receiver could choose to alter other attributes of the control at that time.

The Toolkit Demo sample program creates a menu of buttons with different enable styles. See \PENPOINT\SDK\SAMPLE\TKDEMO\TKDEMO.C.

---

**Internal Notification**

You need only read this subsection if you intend to create your own subclasses
of dsControl.

Many descendants of dsControl end up sending the client (the object stored in
the control’s metrics) an activation message of some sort when the user activates them. For example, a vertical scrollbar sends msgScrollbarVertScroll to client.

If you set **previewEnable** in CONTROL_STYLE, then underneath this interaction, dsControl self-sends a stream of various notification messages. These are called previewing messages because a control generates them while the user is working with the controls but before the user activates the control (or decides not to activate it). These messages are important for descendant classes because many controls need to change as the user toys with them, mainly to provide visual feedback.

For example, an individual button must highlight itself or show some kind of decoration when preview begins, and de-highlight when previewing ends. The messages that dsControl self-sends all have names of the form msgControl...Preview. Descendant classes respond to these messages as they see fit. For example, clsButton highlights in response to msgControlBeginPreview.
Remember that a control sends these messages to itself. If you create an instance of a control descendant, and make your application its client, your application will not receive any of these messages. It is up to the control to notify its client in some way.

The messages are:

- **msgControlBeginPreview**: The user has begun previewing.
- **msgControlUpdatePreview**: The user continues to preview in a manner which requires the client to update. For example, if a slider and a text field both indicate a value, then it might be appropriate for the client to change the text field as the user moves the slider.
- **msgControlRepeatPreview**: The previewing has repeated. If a control’s `previewRepeat` style bit is set, then it will send `msgControlRepeatPreview` after a timer notification. This is true for scrollbars and the page turn buttons.
- **msgControlCancelPreview**: The user has finished previewing this control but hasn’t activated it.
- **msgControlAcceptPreview**: The user has activated this control.

The term **activated** is slightly misleading in this context. A button only sends a message to its client when self receives **msgControlAcceptPreview**, but scrollbars do something on **msgControlUpdatePreview** and **msgControlRepeatPreview**. **clsControl**’s definition of activation may not be the same as a given descendant’s.

Every control should respond to **msgControlBeginPreview** by highlighting in some way to indicate that the user has chosen it. Most controls invert, but others do something else. For example, **clsScrollbar** moves the thumb up and down when it receives **msgControlBeginPreview** over the thumb. Table 35-5 summarizes previewing behavior in some subclasses of **clsControl**.

<table>
<thead>
<tr>
<th>Subclass</th>
<th>Previewing Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal button</td>
<td>Inverts foreground and background colors.</td>
</tr>
<tr>
<td>Toggle button</td>
<td>Displays decoration.</td>
</tr>
<tr>
<td>Menu button</td>
<td>Inverts foreground and background colors.</td>
</tr>
<tr>
<td>Label</td>
<td>Ignores preview messages (previewEnable is false).</td>
</tr>
<tr>
<td>Scrollbar</td>
<td>Thumb bar inverts, arrow buttons use <strong>msgControlRepeatPreview</strong>, thumb creates a tracker that moves the thumbing bar.</td>
</tr>
<tr>
<td>Field</td>
<td>Depends on style fields; may display input focus indicator.</td>
</tr>
</tbody>
</table>

**Table 35-4**

**When Are Preview Messages Generated?**

**clsControl** self-sends these **msgControl...Preview** messages when it receives certain user input. The user input is delivered to **clsControl** in two forms: as pen input events and as gestures. Pen input events are sent to the control from the input system. The control calls its ancestor with all input events. Because one of
its ancestors is clsGWin, clsGWin receives pen events in the control. If the control
has gestures enabled, then clsGWin recognizes pen motions as gestures, and as a
result, self-sends msgGWinGesture. Also, if resizing or dragging are permitted in
clsBorder, clsBorder will grab certain input events.

The mapping from input events to these messages is as shown in Table 35-6.

<table>
<thead>
<tr>
<th>Event</th>
<th>Message Sent to Self</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgPenDown</td>
<td>msgControlBeginPreview</td>
</tr>
<tr>
<td>msgPenMoveDown</td>
<td>msgControlUpdatePreview</td>
</tr>
<tr>
<td>msgPenExitDown</td>
<td>msgControlCancelPreview (if gestures disabled and control is not grabbing input)</td>
</tr>
<tr>
<td>msgPenUp</td>
<td>msgControlAcceptPreview (if gestures disabled)</td>
</tr>
<tr>
<td>xgsITap</td>
<td>msgControlAcceptPreview (if gestures enabled)</td>
</tr>
<tr>
<td>msgTimerNotify</td>
<td>msgControlRepeatPreview (if previewRepeat set)</td>
</tr>
</tbody>
</table>

If the control has gestures enabled (gestureEnable in GWIN_STYLE), then
clsControl looks for gestures, and accepts preview on receipt of xgsITap. If not,
then it accepts preview on msgPenUp and cancels preview on msgPenExitDown
(if previewGrab isn't enabled).

The mapping from window input flags to msgPen... input events is explained in
Part 5: Input and Handwriting Translation. clsControl doesn't set window input
flags. If you want different input flags than clsGWin, you need to set them.

The message argument for every msgControl...Preview message is either pNull, or
a pointer to the INPUT_EVENT message arguments structure for the event that
caused the message. Control subclasses can use the information in the event to
determine where the pen was.

Previewing

When msgPenDown input arrives at clsControl, it triggers previewing. clsControl
sets the previewing bit in CONTROL_STYLE to true. It then self-sends the
corresponding msgControlBeginPreview message.

Stopping Preview

msgControlCancelPreview sets previewing to false when it arrives at clsControl.
msgControlAcceptPreview also sets previewing to false. The user can usually
cancel previewing by dragging the pen out of the control.

Descendants can terminate previewing at any time by not returning stsOK in
response to msgControl...Preview messages. stsControlCancelPreview is a status
value defined for this purpose; it cancels preview but it is not negative, so
clsControl does not interpret it as an error.
Preview Grab

If `previewGrab` is set in the control's style bits, previewing starts an input grab. The control will continue previewing even if the user drags the pen out of it; normally, the control would send itself `msgControlCancelPreviewing`. The user can still cancel previewing by drawing some unrecognizable gesture then lifting the pen, or the descendant class can return some status other than `stsOK`.

Scrollbars set `previewGrab` so that the user doesn't have to keep the pen in them while adjusting.

Preview Repeat

If `previewRepeat` is set in the control's style bits, `msgControlBeginPreview` starts a timer by sending `msgTimerRegister` to `theTimer`. 600 milliseconds later, `theTimer` sends a `msgTimerNotify` to the control. If previewing is still on (the control is still previewing) and `previewRepeat` hasn't been reset, the control generates `msgControlRepeatPreview`. Its message arguments are the event that started the preview.

After self-sending `msgControlRepeatPreview`, the control registers a new timer with `theTimer`. This has a zero timeout. The result is that controls that repeat start repeating after 600 milliseconds, then repeat as fast as events can be delivered.

Gesture Notification

When `clsControl` receives `msgGWinGesture` or `msgGWinForwardedGesture`, `clsControl` sends `msgGWinForwardedGesture` to the control's client. `clsControl` returns whatever status is returned by its client. If the client does not return `stsOK`, the gesture reaches `clsGWin`, which returns `stsRequestForward` to propagate the message up the window tree. See "Gesture Windows" in Chapter 32, Toolkit Ancestors, for more information on how gestures are forwarded and propagated in general.

Special Gesture Handling

Single Taps

Note that `clsControl` unconditionally steals away the `xgs1Tap` gesture and maps it into `msgControlAcceptPreview`. This allows the user's tap on a button to execute the button rather than send `msgGWinForwardedGesture` to the button's client.

Quick Help

`xgsQuestion` is one gesture that `clsGWin` handles itself. If the user makes the Quick Help ? gesture, no one responds to it, and the control has a Quick Help ID specified in its gesture metrics, `clsGWin` will display the Quick Help text directly.
Chapter 36 / Labels

clsLabel handles the presentation of most controls (exceptions include clsScrollbar and clsProgressbar). Buttons, menu buttons, individual toggles, and frame title bars are all descendants of clsLabel. A label displays either a string you specify or a window you provide. You can specify that it display other decorative elements, such as check marks and arrows, next to the string or window. clsLabel does not provide any notification or preview behavior at all.

Figure 36-1
Sample Labels
clsLabel Messages

Table 36-1 summarizes the messages defined by clsLabel.

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
<th>Class Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>Creates a label window.</td>
<td></td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>Initializes the LABEL_NEW structure to default values.</td>
<td></td>
</tr>
<tr>
<td>msgLabelGetStyle</td>
<td>Passes back the current style values.</td>
<td></td>
</tr>
<tr>
<td>msgLabelSetStyle</td>
<td>Sets the style fields.</td>
<td></td>
</tr>
<tr>
<td>msgLabelGetString</td>
<td>Passes back current string.</td>
<td></td>
</tr>
<tr>
<td>msgLabelSetString</td>
<td>Sets the label string.</td>
<td></td>
</tr>
<tr>
<td>msgLabelGetWin</td>
<td>Passes back the child window.</td>
<td></td>
</tr>
<tr>
<td>msgLabelSetWin</td>
<td>Sets the child window.</td>
<td></td>
</tr>
<tr>
<td>msgLabelGetFontSpec</td>
<td>Passes back the font spec.</td>
<td></td>
</tr>
<tr>
<td>msgLabelSetFontSpec</td>
<td>Sets the font spec.</td>
<td></td>
</tr>
<tr>
<td>msgLabelGetScale</td>
<td>Passes back the font scale.</td>
<td></td>
</tr>
<tr>
<td>msgLabelSetScale</td>
<td>Sets the font scale.</td>
<td></td>
</tr>
<tr>
<td>msgLabelGetRows</td>
<td>Passes back the number of rows the label will size itself to.</td>
<td></td>
</tr>
<tr>
<td>msgLabelSetRows</td>
<td>Sets the number of rows the label will size itself to.</td>
<td></td>
</tr>
<tr>
<td>msgLabelGetCols</td>
<td>Passes back the number of columns the label will size itself to.</td>
<td></td>
</tr>
<tr>
<td>msgLabelSetCols</td>
<td>Sets the number of columns the label will size itself to.</td>
<td></td>
</tr>
<tr>
<td>msgLabelGetBoxMetrics</td>
<td>Passes back the current box metrics.</td>
<td></td>
</tr>
<tr>
<td>msgLabelResolveXY</td>
<td>Resolves a point to a character in the string.</td>
<td></td>
</tr>
<tr>
<td>msgLabelGetRects</td>
<td>Computes the rectangle for each given character index.</td>
<td></td>
</tr>
</tbody>
</table>

In its default usage, clsLabel paints the label string that you set with msgNew or msgLabelSetString aligned at the bottom left corner of its window. You can set the label style so that the label displays one of a fixed set of glyphs to the left or right of the label string, and to control other aspects of the label's appearance.

Since the behavior of a label is quite different when its contents are a window instead of a string, the details of labels with child windows are presented separately, in Section 7.6.
Creating a Label

You send `msgNew` to `clsLabel` to create a label. This takes a `LABEL_NEW` structure as its message arguments; part of this is a `LABEL_NEW_ONLY` structure, as usual. In this you specify:

- **style** various style fields, described below.
- **pString** the string for the label, or its child window, or its resource ID.
- **font** the `SYSDC_FONT_SPEC` to open to draw the label string. Ignored if `style.fontType` is set to `lsFontSystem` or `lsFontUser`.
- **fontName** the name of the font to draw the label string. Ignored if `style.fontType` is set to `lsFontSystem` or `lsFontUser`.
- **scale** the scale of the label in `scaleUnits`.
- **rows** the number of rows in the label. Each row has the height of one typographer’s em in the label’s font. Ignored if `style numRows` is `lsNumAsNeeded`.
- **cols** the number of columns in the label (zero indicates no limit). Each column is one em (eight layout units) wide. Because PenPoint typically uses a proportionally spaced font, it is impossible to predict what will fit in a label with `cols` specified. Ignored if `style.numCols` is `lsNumAsNeeded`.

Label Styles

The various styles are used by label subclasses to form the familiar UI Toolkit components. `LABEL_STYLE` incorporates several categories of label appearance, shown in Table 36-2:

<table>
<thead>
<tr>
<th>Fields/Field Values</th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>infoType</td>
<td>Label information type.</td>
</tr>
<tr>
<td>IsInfoString</td>
<td>Label information is a string.</td>
</tr>
<tr>
<td>IsInfoStringID</td>
<td>Label information is a resource ID.</td>
</tr>
<tr>
<td>IsInfoWindow</td>
<td>Label information is a window UID.</td>
</tr>
<tr>
<td>xAlignment</td>
<td>Horizontal alignment of label.</td>
</tr>
<tr>
<td>IsAlignLeft</td>
<td>Align with left edge.</td>
</tr>
<tr>
<td>IsAlignRight</td>
<td>Align with right edge.</td>
</tr>
<tr>
<td>IsAlignCenter</td>
<td>Center horizontally.</td>
</tr>
<tr>
<td>IsAlignCustom</td>
<td>Self-send <code>msgLabelAlign</code> to get offset.</td>
</tr>
<tr>
<td>yAlignment</td>
<td>Vertical alignment of label.</td>
</tr>
<tr>
<td>IsAlignBottom</td>
<td>Align with bottom edge.</td>
</tr>
<tr>
<td>IsAlignTop</td>
<td>Align with top edge.</td>
</tr>
<tr>
<td>IsAlignCenter</td>
<td>Center vertically.</td>
</tr>
<tr>
<td>IsAlignCustom</td>
<td>Self-send <code>msgLabelAlign</code> to get offset.</td>
</tr>
</tbody>
</table>

Table 36-2

LABEL STYLE Fields
### Fields/Field Values

<table>
<thead>
<tr>
<th>Field</th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rotation</td>
<td>Degree of rotation of the text.</td>
</tr>
<tr>
<td>IsRotateNone</td>
<td>Do not rotate label text.</td>
</tr>
<tr>
<td>IsRotate90</td>
<td>Rotate text 90 degrees.</td>
</tr>
<tr>
<td>IsRotate180</td>
<td>Rotate text 180 degrees.</td>
</tr>
<tr>
<td>IsRotate270</td>
<td>Rotate text 270 degrees.</td>
</tr>
<tr>
<td>underline</td>
<td>Underline style of text.</td>
</tr>
<tr>
<td>IsUnderlineNone</td>
<td>Do not underline label text.</td>
</tr>
<tr>
<td>IsUnderlineSingle</td>
<td>Single underline.</td>
</tr>
<tr>
<td>IsUnderlineDouble</td>
<td>Double underline.</td>
</tr>
<tr>
<td>strikeOut</td>
<td>Determines whether text has a strikeout line through it.</td>
</tr>
<tr>
<td>decoration</td>
<td>Type of label decoration.</td>
</tr>
<tr>
<td>IsDecorationNone</td>
<td></td>
</tr>
<tr>
<td>IsDecorationBlank</td>
<td></td>
</tr>
<tr>
<td>IsDecorationExclusiveOn</td>
<td></td>
</tr>
<tr>
<td>IsDecorationExclusiveOff</td>
<td></td>
</tr>
<tr>
<td>IsDecorationNonExclusiveOff</td>
<td></td>
</tr>
<tr>
<td>IsDecorationNonExclusiveOn</td>
<td></td>
</tr>
<tr>
<td>IsDecorationCircle</td>
<td></td>
</tr>
<tr>
<td>IsDecorationBox</td>
<td></td>
</tr>
<tr>
<td>IsDecorationCheckedBox</td>
<td></td>
</tr>
<tr>
<td>IsDecorationCheckedCircle</td>
<td></td>
</tr>
<tr>
<td>IsDecorationHollowLeft</td>
<td></td>
</tr>
<tr>
<td>IsDecorationHollowRight</td>
<td></td>
</tr>
<tr>
<td>IsDecorationSolidLeft</td>
<td></td>
</tr>
<tr>
<td>IsDecorationSolidRight</td>
<td></td>
</tr>
<tr>
<td>IsDecorationButtonOn</td>
<td></td>
</tr>
<tr>
<td>IsDecorationButtonOff</td>
<td></td>
</tr>
<tr>
<td>IsDecorationPopup</td>
<td></td>
</tr>
<tr>
<td>IsDecorationCustomLeft</td>
<td></td>
</tr>
<tr>
<td>IsDecorationCustomRight</td>
<td></td>
</tr>
<tr>
<td>numCols, numRows</td>
<td>Style for the number of rows and columns.</td>
</tr>
<tr>
<td>IsNumAbsolute</td>
<td>Fixed number of rows or columns specified in rows or columns.</td>
</tr>
<tr>
<td>IsNumAsNeeded</td>
<td>As many rows or columns as needed.</td>
</tr>
<tr>
<td>box</td>
<td>Style of box around characters in a label.</td>
</tr>
<tr>
<td>IsBoxNone</td>
<td>No box around characters.</td>
</tr>
<tr>
<td>IsBoxSquare</td>
<td>Square box, for fields.</td>
</tr>
<tr>
<td>IsBoxTicks</td>
<td>Ticks or combs, for fields.</td>
</tr>
<tr>
<td>IsBoxInvisible</td>
<td>Don’t draw specified boxes.</td>
</tr>
<tr>
<td>wordWrap</td>
<td>Determines whether label text should wrap to the next row at the end of a word.</td>
</tr>
<tr>
<td>fontType</td>
<td>Style of label font.</td>
</tr>
<tr>
<td>IsFontSystem</td>
<td>Use the system font.</td>
</tr>
<tr>
<td>IsFontUser</td>
<td>Use the user font.</td>
</tr>
<tr>
<td>IsFontCustom</td>
<td>Use custom font specified by font or name.</td>
</tr>
<tr>
<td>scaleUnits</td>
<td>Specifies scale of units.</td>
</tr>
<tr>
<td>bsUnitsLayout</td>
<td>Use the UI Toolkit layout units.</td>
</tr>
<tr>
<td>bsUnitsPoints</td>
<td>Use points (1/72 inch).</td>
</tr>
<tr>
<td>bsUnits...</td>
<td>Use any other border unit (see BORDER.H).</td>
</tr>
<tr>
<td>stringSelected</td>
<td>Determines whether the string should show the selected visual.</td>
</tr>
</tbody>
</table>
In general, each of the settings within a category is exclusive. A label can only have one decoration; it can't have a pull-right arrow as well as a check box.

The UI Toolkit's label descendent classes set the LABEL_STYLE flags to achieve different effects. They also make use of the drawing styles provided by clsBorder to control their outline (none, rectangular, or round-cornered rectangle, with different shades) and background; border windows are described in Chapter 33, Border Windows.

The fact that labels do not respond to input might lead you to think that clsLabel is an abstract class. However, this lack of behavior is perfect for annotations in dialog boxes or option sheets.

Example 36-1

Creating an Annotation Label

The following code from the Hello World (Toolkit) sample program (in \PENPOINT\SDKSAMPLE\HELLOTK\HELLOTK1.C) puts the words Hello, World in the center of the window. It uses label styles to make the label scale to the window. Hello World (Toolkit) creates this label in its msgAppInit handler.

```
LABEL NEW
STATUS
...
// Create the Hello label window.
ObjCallRet(msgNewDefaults, clsLabel, &ln, s);
ln.label.style.scaleUnits = bsUnitsFitWindowProper;
ln.label.style.xAlignment = IsAlignCenter;
ln.label.style.yAlignment = IsAlignCenter;
ln.label.pString = "Hello World!";
ObjCallRet(msgNew, clsLabel, &ln, s);
// Insert ln.object.uid in some window.
...
```

Label Strings and Special Characters

By default, label strings appear in the current system font. This font is used in the notebook tabs, message box, and frame headers, as well as the UI Toolkit. The encoding for the system font is sysDcEncodeGoSystem, which has several special GO symbols in its low ASCII area, including symbols for label decorations. To draw decorations, clsLabel merely draws a string of characters. If SysDC does not find a special GO glyph in the font cache, it looks for the glyph in the current system font. If the character is missing from the system font, SysDC looks for it in a special GO glyph font. No matter what font you use, the UI Toolkit will display decorations matching its size.

You can use some other font either by setting the fontName in LABEL_NEW_ONLY to the system name of a font (such as HESS), or by supplying a SYSDC_FONT_SPEC in the font field of LABEL_NEW_ONLY. The fontName takes priority unless it is pNull (the default).
**Layout**

String labels are leaf windows—they don’t have any child windows. Thus, a string label has nothing in it to lay out in response to msgWinLayoutSelf. But it does contain a string, which may change in size when you send msgLabelSetString. Sometimes you want the label to change its size to fit around the string, other times you want the label to remain a fixed size.

Normally, a window recomputes its size and changes its size if necessary when it receives msgWinLayoutSelf with wsLayoutResize set. You can control whether a label changes size by setting its wsShrinkWrapWidth and wsShrinkWrapHeight window style flags. If these are false, the label will not change size.

If either is true, the label will recomputes its width or height and size itself so that it is just big enough to display its current label string and decorations in the style you’ve specified inside the border you’ve specified. The exception to this is if you’ve specified a fixed number of rows or columns (IsNumAbsolute), in which case the label changes its size to fit the number of rows or columns in the label font. The default is that both wsShrinkWrapWidth and wsShrinkWrapHeight are true, so that labels change size to fit their contents.

If you have a label with a changing string, you have several choices:

- Keep the label’s size constant and run the risk that all of its string may not be visible.
- Set the scaleUnits style to either IsScaleFitWindow or IsScaleFitWindowProper so that the scale of the label’s contents changes to fit inside the label.
- Make the label grow and shrink.

In the latter case, you need to send msgWinLayout to the label with wsLayoutResize set whenever you change its string.

**No Notification**

clsLabel does not turn on previewEnable in clsControl, or set any window input flags. Thus, an instance of clsLabel does not respond to input, and clsControl never generates any messages. By default, input in a label won’t even be passed to its parent window. It is the descendents of clsLabel that turn on different combinations of window input flags and thereby receive msgControl... messages.

**Painting**

A label paints itself as best as it can within the rectangle of its window. It paints its:

- Border (by clsBorder).
- Label string.
- Label decoration.
- Insertion point.
When you change the label style, clsLabel tries to be smart about painting, and paints only what has changed.

clsLabel responds to msgBorderPaintForeground by painting its decoration and string.

Child Windows

This section describes how labels use child windows. You need read it only if you want to put a window in a label.

If you set a label's style.infoType to IsInfoWindow, it assumes pString is the UID of a window. Or, if you send msgLabelSetWin to a label, this changes the infoType to IsInfoWindow. You need to be careful not to set the infoType to IsInfoString when displaying a child window, and vice-versa. Set pString to pNull first, then change infoType, then set pString to the new value.

Instead of painting a string of text, clsLabel inserts the window as a child of the label. It never paints the child window; it relies on the child window repainting itself in response to msgWinRepaint. Everything else about label window behavior is unchanged.

If the child window's parent is not the label, clsLabel makes the label the child's parent. If the label doesn't clip its children, then clsLabel sets the child window to share the parent's clipping and not clip its own child windows.

If you know your child window will draw within its boundary, then you can turn on wsParentClip in its window style flags, and set backgroundInk to bsInkTransparent | bsInkExclusive. This will let the child window share the update region of its parent (the label), which saves space and speeds repaint.

Layout

As explained in Section 7.3, labels will resize themselves when they receive msgWinLayoutSelf if both the following hold:

- wsLayoutResize is set in the message arguments.
- One or both of wsShrinkWrapWidth and wsShrinkWrapHeight are set in the label's window style flags.

In the last case, the label will fit around the child window's current size.

If a window label does change size, it will ordinarily fit around the size of its child window or string. However, if bsUnitsFitWindow or bsUnitsFitWindowProper is set in LABEL_STYLE, the label will resize its child window so that it fits within the current size of the label. If your child window can repaint to fit its size, this is fine. However, if your child window needs to be a particular size, you should not set bsUnitsFitWindow or bsUnitsFitWindowProper. You might also want to reset the label's wsShrinkWrapWidth and wsShrinkWrapHeight window style flags so that the label will not change its size. If you specify a fixed-size child window, note that it won't handle the system font changing very well—all other labels will
change size except the one with a child window. It’s best to let the label size the child window.

If you want to change the size of your child window, you should send msgWinLayout to the label it’s in to make the label change size also.

**Painting**

Your child window will receive msgWinRepaint when the label needs to repaint. The label takes care of drawing the border and decorations, and if the child window has wsParentClip set, then the label will clear the child window’s area.

**Field Support**

Fields allow handwritten and keyboard input. They inherit from clsLabel (see Chapter 42, Fields, for more information). clsLabel does the drawing for fields, so it has some internal messages that clsField uses.

**Insertion Point**

clsLabel draws the boxes for the insertion point that indicates where the next character from the keyboard will appear. To find out where it should draw the insertion point, clsLabel self-sends msgLabelProvideInsPt. Descendants can respond by passing back the zero-based character offset of the insertion point.

If the label style is non-boxed, clsLabel draws the insertion point as a vertical line before this character; if the label style is boxed, clsLabel highlights the baseline of the box around this character. If the message gets to clsLabel, it passes back the value –1, which tells itself not to draw the insertion point.

**Character Positions**

clsField handles gestures over fields. To figure out which character is closest to the hot point of a gesture, it self-sends msgLabelGetRects. This computes the bounding rectangles for a set of characters. It takes a pointer to an array of LABEL_RECT structures. In each structure, the sender specifies the location in the string (index). As in msgLabelProvideInsPt, characters are specified by a zero-based index in the string. clsLabel computes the bounding rectangle for the character in its string at index, and passes it back as a RECT32 in device units (rect). clsLabel knows it has reached the end of the array of LABEL_RECTs when the index specified is the value –1.


Chapter 37 / Buttons

Buttons are labels with input behavior—the user can tap a button. When the user activates a button, the button sends its client (maintained by clsControl) information that you specify.

![Sample Buttons](image)

### clsButton Messages

Table 37-1 summarizes the messages defined by clsButton.

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>Creates a button window.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>Initializes the BUTTON_NEW structure to default values.</td>
</tr>
<tr>
<td>msgButtonGetMetrics</td>
<td>Passes back the current metrics.</td>
</tr>
<tr>
<td>msgButtonSetMetrics</td>
<td>Sets the metrics.</td>
</tr>
<tr>
<td>msgButtonGetStyle</td>
<td>Passes back the current style values.</td>
</tr>
<tr>
<td>msgButtonSetStyle</td>
<td>Sets the style values.</td>
</tr>
<tr>
<td>msgButtonGetMsg</td>
<td>Passes back metrics.msg.</td>
</tr>
<tr>
<td>msgButtonSetMsg</td>
<td>Sets metrics.msg.</td>
</tr>
</tbody>
</table>

37.1

Table 37-1 clsButton Messages

**Class Messages**

**Instance Messages**

continued
Table 37-1 (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgButtonGetData</td>
<td>Passes back metrics.data.</td>
</tr>
<tr>
<td>msgButtonSetData</td>
<td>Sets metrics.data.</td>
</tr>
<tr>
<td>msgButtonSetNoNotify</td>
<td>Sets the value of the button without notifying.</td>
</tr>
<tr>
<td>msgButtonDone</td>
<td>Sent via msgWinSend to the manager when button receives msgControlAcceptPreview.</td>
</tr>
<tr>
<td>msgButtonBeginPreview</td>
<td>Sent via msgWinSend to the manager when button receives msgControlBeginPreview.</td>
</tr>
<tr>
<td>msgButtonUpdatePreview</td>
<td>Sent via msgWinSend to the manager when button receives msgControlUpdatePreview.</td>
</tr>
<tr>
<td>msgButtonRepeatPreview</td>
<td>Sent via msgWinSend to the manager when button receives msgControlRepeatPreview.</td>
</tr>
<tr>
<td>msgButtonCancelPreview</td>
<td>Sent via msgWinSend to the manager when button receives msgControlCancelPreview.</td>
</tr>
<tr>
<td>msgButtonAcceptPreview</td>
<td>Sent via msgWinSend to the manager when button receives msgControlAcceptPreview.</td>
</tr>
<tr>
<td>msgButtonNotifyManager</td>
<td>Sent to self when button wants to notify its manager.</td>
</tr>
<tr>
<td>msgButtonNotify</td>
<td>Sent to self when button wants to notify its client.</td>
</tr>
</tbody>
</table>

To create a button:

- Specify a client for the button in the control part of its BUTTON_NEW structure.
- Specify the visual appearance of the button in the BUTTON_STYLE part of its BUTTON_NEW_ONLY structure.
- Specify a message and data that the button will send to its client when the user activates it.

There are several different styles of button. The default button responds to input, sets its border style to a rectangle with a shadow, centers its label. Buttons respond to msgControlBeginPreview and msgControlAcceptPreview by providing some kind of visual feedback. For example, the default button previews by becoming gray.

### Other Kinds of Buttons

**Menu buttons** Menu buttons are a special kind of button. They display an associated menu when the user taps on them. Menu buttons and menus are explained in Chapter 39, Menus and Menu Buttons.

**Pop-up choices** Pop-up choices are buttons which display a current value of chosen from a selection of values. The user can flick to move to other values, or tap to display a pop-up menu showing all the choice values. Pop-up choices are also documented in Chapter 39, Menus and Menu Buttons.

**Icons** Icons are buttons which can display a bitmap as well as a string. Icon toggles are icons which maintain an on or off state, displaying a different picture for each of the two states.
Creating a Button

clsButton's class-specific msgNew arguments (BUTTON_NEW_ONLY) are the same as its metrics (BUTTON_METRICS). The structure contains three fields:

- **style**: various style bits for the button, described below.
- **msg**: the message to send when the button is activated. This may just be raw data, see Table 37-2.
- **data**: 32 bits of data. If style.pArgs is bsPargsData, then this data will be part of the message arguments sent to the client when the button is activated.

You can get and set all of these with msgButtonGet/SetMetrics. In addition, clsButton defines other messages to get and set individual elements of the metrics: msgButtonGet/SetStyle, msgButtonGet/SetMsg, and msgButtonGet/SetData.

Button Defaults

By default (that is, in the BUTTON_NEW structure passed back by msgNewDefaults), a button notifies its client by sending the msg you define for it. In clsControl, a button is enabled, does not grab input, does not repeat, does not send detailed previewing messages to its client, and is not previewing or dirty.

clsButton supports different kinds of controls: pushbuttons, toggles, and lock-on buttons. It also supports different styles of notification that control how the msg and data in the button's metrics are sent to the button's client. The look and behavior are controlled by style flags in BUTTON_STYLE, shown in Table 37-2.

<table>
<thead>
<tr>
<th>Styles/Style Flags</th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contact</td>
<td>Describes how the button behaves when pushed.</td>
</tr>
<tr>
<td>bsContactMomentary</td>
<td>Button is changed momentarily to on, then off when the button is released.</td>
</tr>
<tr>
<td>bsContactToggle</td>
<td>Button is toggled on and off between each push.</td>
</tr>
<tr>
<td>bsContactLockOn</td>
<td>Button is set to on, but stays on after the user lets go. Used in choices where user can’t turn off a button by tapping on it.</td>
</tr>
<tr>
<td>feedback</td>
<td>Describes how the current value (on or off) should display.</td>
</tr>
<tr>
<td>bsFeedbackNone</td>
<td>No visual feedback.</td>
</tr>
<tr>
<td>bsFeedbackInvert</td>
<td>Invert the button.</td>
</tr>
<tr>
<td>bsFeedbackDecorate</td>
<td>Use onDecoration or offDecoration to draw button.</td>
</tr>
<tr>
<td>bsFeedbackNone</td>
<td>No feedback.</td>
</tr>
<tr>
<td>bsFeedback3D</td>
<td>Use three-dimensional shadow effect.</td>
</tr>
<tr>
<td>bsFeedbackBox</td>
<td>Outline the button with a box.</td>
</tr>
<tr>
<td>on</td>
<td>Determines whether button is on or off; the button’s value.</td>
</tr>
<tr>
<td>onDecoration</td>
<td>Use when button is on if feedback is set to bsFeedbackDecorate.</td>
</tr>
<tr>
<td>offDecoration</td>
<td>Use when button is off if feedback is set to bsFeedbackDecorate.</td>
</tr>
<tr>
<td>pArgs</td>
<td>Determines the kind of button notification message arguments.</td>
</tr>
<tr>
<td>bsPargsData</td>
<td>Message argument is the data in the button's metrics.</td>
</tr>
<tr>
<td>bsPargsValue</td>
<td>Message argument is the button's current value.</td>
</tr>
<tr>
<td>bsPargsUID</td>
<td>Message argument is the button’s UID.</td>
</tr>
<tr>
<td>notifyDetail</td>
<td>Determines whether the button sends messages to its manager for its detailed previewing messages.</td>
</tr>
</tbody>
</table>
**Notification**

**Simple Activation**

The model for the simplest use of buttons is:

1. User interacts with button.
2. Button sends message to manager.
3. Button sends message to client at end of interaction if the user actually activated the button.

Note that the button will send a message to its client even if the value of the button (on in BUTTON_STYLE) is the same as it was before. It sends the message whether the user or the API sets its value.

At this level, clsControl self-sends `msgControlAcceptPreview`. clsButton responds to this by sending self `msgButtonNotify` (the next sections describe how a button comes to send itself this message). The arguments to this are in a `BUTTON_NOTIFY` structure. clsButton responds to `msgButtonNotify` as follows:

- The `notifyWithMsg` button style field determines what clsButton sends to the client when it receives `msgButtonNotify`. If set, the message that the button sends to its client is the message you specified in the button's metrics (`msg`). If `msg` is `null`, clsButton won't send any message.

- However, if `notifyWithMsg` is `false`, then the message that the button sends to its client is the same generic `msgButtonNotify`. The message arguments are a `BUTTON_NOTIFY` structure containing the `msg` and `data` you specified in the button's metrics, and the UID of the button sending the notification message. In this case, the button does not interpret the `msg` field of `BUTTON_METRICS`, so it, like `data`, could be any 32-bit data.

---

<table>
<thead>
<tr>
<th>Styles/Style Flags</th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>notifyWithMsg</td>
<td>Determines whether button's notification message is specified in <code>msg</code> or the generic message <code>msgButtonNotify</code>.</td>
</tr>
<tr>
<td>halfHeight</td>
<td>Determines if button has partial border when not previewing or executing and a full-round; fill when previewing.</td>
</tr>
<tr>
<td>manager</td>
<td>Determines the kind of manager the button has.</td>
</tr>
<tr>
<td></td>
<td>Button has no manager.</td>
</tr>
<tr>
<td></td>
<td>Button's parent window is the manager.</td>
</tr>
<tr>
<td></td>
<td>Button's control client is the manager.</td>
</tr>
</tbody>
</table>
As an example of the first style, if your application has a button labeled Recalc, you could set notifyWithMsg and specify msgMyAppRecalc as its message and the application UID as the client. When the user taps this button, the application would receive msgMyAppRecalc. The nice thing here is that the application is isolated from its user interface. You could change the button or add another way to recalculate, or the application could receive the message from an agent process without requiring any changes to the application.

```c
BUTTON_NEW bn;
APP myApp;
STATUS s;

myApp = OSThisApp();

ObjCallRet(msgNewDefaults, clsButton, &bn, s);
bn.win.parent = parentWin;
bn.win.options = wsPosTop;

bn.label.pString = "Recalc";
bn.control.client = myApp;
bn.button.msg = msgMyAppUserRecalc;
ObjCallRet(msgNew, clsButton, &bn, s);
ObjCallRet(msgWinInsert, bn.object.uid, &bn.win, s);
```

The application object myApp will receive msgMyAppUserRecalc with P_ARGS of null when the user activates the button. This example doesn't use the data field because all the information the client needs is the msgMyAppRecalc itself. You could associate a different message with every button in your application. However, the cost of defining separate messages for every button is that the client must respond to dozens of messages. The data field is useful when you want several buttons to share the same message, but want to distinguish between them. For example, you might have several buttons to zoom in and out of a diagram. Each could have the same msg field (msgMyAppZoom), and a different data corresponding to the different zoom factors or styles associated with each button.

With notifyWithMsg set (as in this example), the client doesn't automatically get the UID of the button sending the message. You can get around this by specifying in the BUTTON_STYLE that its data be the UID of the button (set pArgs to bsPargsUID).

### Unwelcome Notification

A button normally notifies its client whenever someone sets its value. This is true even if the new value is the same as the old value. This may cause problems. For example, you have a toggle in your application that indicates the zoomed state of the application. When you restore the application, you want to set the toggle to show whether the view is zoomed or not. But if you send msgControlSetValue to the toggle, it will notify its client that it has a new value.

If you want to temporarily avoid receiving these kinds of notifications, you can either use msgButtonSetNoNotify to set the value of the button without notifying its manager and client, or don't set the button's client until after you have set it up.
Painting

If halfHeight is set in BUTTON_STYLE, then buttons draw a half-height border when not previewing or executing, and a full-round fill when previewing.

The bsFeedback3D style sets up many border metrics to provide the three-dimensional effect.

clsButton responds to msgBorderGetForegroundRGB with the appropriate RGB color to use for the foreground, given the current visuals. Usually, it calls its ancestor, but if the button’s style.look is bsFeedback3D, it computes the appropriate color to use.

Value

The value of a button is either 1 (on) or 0 (off). You can set a button’s value with msgControlSetValue and test it with msgControlGetValue. You can also manipulate the value by setting and getting the BUTTON_STYLE fields, which include on.

Control Dirty

When a button’s value changes (either programmatically or by the user changing it), the button self-sends msgControlSetDirty, so that if its showDirty control style bit is set, it will indicate that its value is different. Someone else (probably the button’s client) is responsible for resetting the button’s control dirty. For example, the client might reset control dirty after it completes the action requested by the button.

If a button’s contact style is not bsContactMomentary, then as soon as the user taps it (and it thereby receives msgControlBeginPreview), it sends itself msgControlSetDirty with value true. Thus, a button is dirty as soon as the user touches it. The intent of control dirty is to allow option tables and other collections of controls to determine which of their nested controls has been altered. If a control is dirty, then the change will be applied. The assumption is that if a user has touched a button, then the user wants its value applied, even if the user never changed its state.

Creating Many Buttons

If you need to create many buttons, then you can create them in a toolkit table. clsTkTable lets you create all the nested buttons at once. clsToggleTable and clsChoice are collections of buttons with additional semantics. See Chapter 38, Toolkit Tables, for more information on creating a set of buttons at once.
Advanced Button Notification Techniques

Most developers need not be concerned with the internals of button notification. A single button works as expected, as do the many classes that combine groups of buttons. However, in the rare case that you must create a completely new class of button, you will have to write code to handle low level button notification and management protocols.

If you set the notifyDetail button style field, the button's manager will receive a host of additional messages. You set the manager at creation time or in the BUTTON_METRICS structure. The manager is an object that receives notification messages from the button. These messages are called previewing messages because a button generates them while the user is interacting with the button, but before the user activates the button.

The manager can alter the behavior of the button or of other buttons in a group, as for example, a choice does to coordinate its toggles. This is useful if you want to change things around as the user toys with buttons. For example, you might want to change the style of some sample text as the user touches the pen on a choice, not just after the user makes his final choice.

The messages are all of the form msgButton...Preview and an extra message, msgButtonManagerNotify. These messages to the button manager correspond to the msgControl...Preview messages that clsControl sends to the button. The button sends them using msgWinSend, so that if its manager doesn't respond, the message propagates to the manager's ancestor window, and on up the window tree.

Unless you need to create a completely new class of button, don't read this section!

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgButtonBeginPreview</td>
<td>The user has begun previewing.</td>
</tr>
<tr>
<td>msgButtonUpdatePreview</td>
<td>The user continues to preview in a manner that requires the client to update. For example, if a slider and a text field both indicate a value, then it might be appropriate for the client to change the text field as the user moves the slider.</td>
</tr>
<tr>
<td>msgButtonRepeatPreview</td>
<td>The previewing has repeated. If a button's previewRepeat control style bit is set, then it will start repeating the last action after a while. This is true for scrollbars and the page turn buttons.</td>
</tr>
<tr>
<td>msgButtonCancelPreview</td>
<td>The user has finished previewing this button but hasn't activated it.</td>
</tr>
<tr>
<td>msgButtonAcceptPreview</td>
<td>The user has activated this button.</td>
</tr>
</tbody>
</table>

Manager Objects

There are some common styles of button management, such as the exclusive-on buttons in choices and multiple selectable buttons in a list box. The UI Toolkit includes manager classes that do the housekeeping for the preview messages from these sets of buttons: clsManager, clsChoiceMgr, and clsSelChoiceMgr. They are described in more detail in Chapter 38, Toolkit Tables.
How clsButton and clsTkTable Respond to Button Previewing Message

If the manager receiving these messages is itself a button or toolkit table, and it calls ObjectCallAncestor, clsButton and clsTkTable respond to them by notifying the manager's own manager if the manager's notifyDetail flag is set. In other words, the previewing messages continue to flow up the chain of managers if they all set notifyDetail.

Examples of Previewing

clsChoice is one built-in class that responds to msgButton...Preview messages. Choices are made up of several ordinary buttons (clsChoice is a toolkit table; this type of class is described below), but the choice must ensure that only one of the buttons is on at once. When the user previews another button by holding the pen tip over it, the choice must turn off the current button. To implement this, a choice sets the button style.manager of its buttons to bsManagerParent (the parent of the buttons, that is, the choice itself), and uses a hidden object to handle the button previewing messages. See Section 9.6 for more information.

As another example, if you were going to be a client of a joystick-like control, and you wanted to update a view as the user moves the joystick or keeps it sideways, you would set notifyDetail and update your view on receiving msgButtonUpdatePreview and msgButtonRepeatPreview. The pseudo-joystick control would need to be a subclass clsButton to generate these messages.

Unwanted Manager Notification

Since the button's value is the on field in its BUTTON_STYLE, one side effect of sending msgButtonSetStyle or msgButtonSetMetrics is that the button sends itself a msgButtonNotifyManager message and also notifies its client.
Chapter 38 / Toolkit Tables

clsTkTable groups a collection of UI Toolkit components, typically buttons or
descendants of clsButton, into a table. It creates all the items at once. The items
can be from different classes, and you can specify particular initialization for each
one. The items can even be toolkit tables themselves. Thus, not only can you
create homogeneous tables of buttons, such as a bank of switches, you can also put
choices, arrays of buttons, fields, submenus, and any other kind of UI Toolkit
component in the same toolkit table. For example, many of the option cards in
PenPoint option sheets are created from a single toolkit table specification.

Menus, choices, option tables, tab bars, and command bars are all toolkit tables
that are composed of collections of labels and buttons. Each item in the menu, or
possible choice in the choice, or toggle in the toggle table, is a toolkit component
nested inside the toolkit table as a child window. These descendants of clsTkTable
are described in later sections.

The menu, choice or toggle table manages its component buttons, arranging them
and managing their previewing and notification. The arrangement aspect is best
handled by clsTableLayout, so clsTkTable inherits from clsTableLayout. To alter
the layout of buttons in toolkit tables, you can use the messages and structures
provided by clsTableLayout.
### Sample Toolkit Tables

**Tk Demo: TkTables**

<table>
<thead>
<tr>
<th>Document</th>
<th>Edit</th>
<th>Options</th>
<th>Demo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>This</td>
<td>That</td>
<td>Other</td>
</tr>
<tr>
<td>Menu</td>
<td>This</td>
<td>That</td>
<td>Other</td>
</tr>
<tr>
<td>Menu with Pull-Right</td>
<td>Counter</td>
<td>Helvetica</td>
<td>Other</td>
</tr>
<tr>
<td>Menu Bar</td>
<td>Font</td>
<td>Edit</td>
<td></td>
</tr>
<tr>
<td>Exclusive Choice</td>
<td>Small</td>
<td>Medium</td>
<td>Large</td>
</tr>
<tr>
<td>Exclusive Choice with Complex Choice</td>
<td>Small</td>
<td>Medium</td>
<td>Large</td>
</tr>
<tr>
<td>Taggle Table (Non-Exclusive Choice)</td>
<td>Bold</td>
<td>Italic</td>
<td>Underline</td>
</tr>
<tr>
<td>Tab Bar</td>
<td>Card 3</td>
<td>Card 2</td>
<td></td>
</tr>
</tbody>
</table>
### clsTkTable Messages

Table 38-1 summarizes the messages defined by clsTkTable.

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>Creates a toolkit table window.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>Initializes the TK_TABLE_NEW structure to default values.</td>
</tr>
<tr>
<td>msgTkTableGetStyle</td>
<td>Passes back the current style values.</td>
</tr>
<tr>
<td>msgTkTableSetStyle</td>
<td>Sets the style values.</td>
</tr>
<tr>
<td>msgTkTableGetClient</td>
<td>Passes back the client of the first child in the table.</td>
</tr>
<tr>
<td>msgTkTableSetClient</td>
<td>Sets the client of each child in the table to pArgs.</td>
</tr>
<tr>
<td>msgTkTableGetManager</td>
<td>Passes back the manager.</td>
</tr>
<tr>
<td>msgTkTableSetManager</td>
<td>Sets the manager.</td>
</tr>
<tr>
<td>msgTkTableGetMetrics</td>
<td>Passes back the metrics.</td>
</tr>
<tr>
<td>msgTkTableSetMetrics</td>
<td>Sets the metrics.</td>
</tr>
<tr>
<td>msgTkTableChildDefaults</td>
<td>Sets the defaults in pArgs for a common child.</td>
</tr>
<tr>
<td>msgTkTableAddAsFirst</td>
<td>Inserts pArgs as the first child in the table.</td>
</tr>
<tr>
<td>msgTkTableAddAsLast</td>
<td>Inserts pArgs as the last child in the table.</td>
</tr>
<tr>
<td>msgTkTableAddAsSibling</td>
<td>Inserts pArgs-&gt;newChild in front of or behind pArgs-&gt;sibling.</td>
</tr>
<tr>
<td>msgTkTableAddAt</td>
<td>Inserts pArgs-&gt;newChild table at zero-based index pArgs-&gt;index.</td>
</tr>
<tr>
<td>msgTkTableRemove</td>
<td>Extracts specified window from the table.</td>
</tr>
<tr>
<td>msgTkTableInit</td>
<td>Sent to TK_TABLE_ENTRY.class after default mappings from entries to pChildNew.</td>
</tr>
</tbody>
</table>

### Other Kinds of Toolkit Tables

#### Toggle Tables

A toggle table (clsToggleTable) is a table of independent toggle buttons. clsToggleTable maintains a 32-bit bitmask encoding the values (on or off) of the first 32 buttons in the table. This chapter describes toggle tables in more detail after describing clsTkTable.

#### Choices

A choice (clsChoice) is a table of buttons of which only one button may be on. When the user activates a button in a choice table, whatever button had been previously activated is deactivated. This chapter describes choices in more detail after describing clsTkTable.
**Menus**

A menu is a toolkit table that organizes its nested components in a column. Menus are documented in Chapter 39, Menus and Menu Buttons.

**Tab Bars**

A tab bar (clsTabBar) arranges a set of tab buttons vertically, and let the user slide tabs around to hide and expose tabs. Frames and option sheets can have attached tab bars.

**Command Bars**

A command bar (clsCommandBar) presents a centered set of buttons. Option sheets have a standard command bar containing Apply and Apply & Close buttons.

**Option Tables**

An option table (clsOptionTable) presents the standard two-column layout of pairs of bold labels and settings inside option sheets.

### Creating a Toolkit Table

When you create a toolkit table, you indirectly specify the msgNew arguments for the components in it as part of clsTkTable's own msgNew arguments. TK_TABLE_NEW_ONLY permits a mass initialization of the nested components. You can either specify msgNew arguments for each individual component, or a global msgNew, or neither, and just set some flags for the common classes of components. clsTkTable uses this information to create the child components itself—it sends msgNew over and over to create each child, so you don't have to.

This means that there are two distinct ways to create most UI Toolkit components: by sending msgNew, or by specifying it as a table entry when creating a toolkit table.

### Common Creation Information

Rather than requiring you to separately create each component in the table by sending msgNew over and over, you supply some information common to all the components, and a pointer to an array of information for each child (P_TK_TABLE_ENTRY). The common information in TK_TABLE_NEW_ONLY is:

- **style** the table style (currently empty).
- **client** the client for controls in the table.
- **pEntries** a pointer to the array of per-child information.
- **pButtonNew** a default msgNew structure for all of the nested items (if needed).
- **manager** a manager to notify..
Class-Dependent Creation Information

The array of information that you specify for each button individually consists of a null-terminated array of TK_TABLE_ENTRY structures. The fields in a TK_TABLE_ENTRY are as follows:

- **arg1, arg2, arg3**: clsTkTable's interpretation of the fields depends on the class of the item. These are a P_UNKNOWN and two U32s, respectively.
- **tag**: a window tag. Because clsTkTable creates the items for you, the easiest way to retrieve an object is to give it a tag (using MakeTag) and find it with msgWinFindTag.
- **flags**: a set of flags that encode common variants of items in a toolkit table, such as toggle-style button, wide border margins, and a menu button with a pull-right menu. Many of the flags only apply to certain classes of items.
- **class**: the class of the item. This is only needed to override the default class defined in pButtonNew.
- **helpId**: a Help ID for the item, used by clsGWin to support Quick Help.

The three unknown fields and the flags field allow a single structure to define most varieties of many different kinds of nested components, without having to supply a custom msgNew structure for each component. For buttons, you usually only set the string, message, and data. For a label, you only set the string.

Now for a menu button with a sub-menu, you want to specify the menu button's string, and then information for each of the items in the sub-menu. This is possible: the second argument for a menu button with a sub-menu is the address of another TK_TABLE_ENTRY array. The values of the fields for different classes are as shown in Table 38-2.

<table>
<thead>
<tr>
<th>Class</th>
<th>arg1</th>
<th>arg2</th>
<th>arg3</th>
</tr>
</thead>
<tbody>
<tr>
<td>clsLabel</td>
<td>pString</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>clsButton</td>
<td>pString</td>
<td>msg</td>
<td>data</td>
</tr>
<tr>
<td>clsMenuButton (no submenu)</td>
<td>pString</td>
<td>msg</td>
<td>data</td>
</tr>
<tr>
<td>clsMenuButton (with submenu)</td>
<td>pString</td>
<td>pEntries</td>
<td>N/A</td>
</tr>
<tr>
<td>clsChoice</td>
<td>pEntries</td>
<td>num of rows or cols</td>
<td>N/A</td>
</tr>
<tr>
<td>clsToggleTable</td>
<td>pEntries</td>
<td>num of rows or cols</td>
<td>N/A</td>
</tr>
<tr>
<td>clsPopupChoice</td>
<td>pEntries</td>
<td>num of rows or cols</td>
<td>N/A</td>
</tr>
<tr>
<td>clsPopupChoice (displaying fonts)</td>
<td>prune (see below)</td>
<td>num of rows or cols</td>
<td>N/A</td>
</tr>
<tr>
<td>clsField</td>
<td>pString</td>
<td>num of cols</td>
<td>N/A</td>
</tr>
<tr>
<td>clsListBox</td>
<td>num of entries</td>
<td>num of entries visible</td>
<td>N/A</td>
</tr>
<tr>
<td>clsFontListBox</td>
<td>num of entries</td>
<td>num of entries visible</td>
<td>look</td>
</tr>
</tbody>
</table>
Wherever this table calls for a `pEntries` in a `TK_TABLE_ENTRY` field, you supply the address of another `TK_TABLE_ENTRY` array (or `pNull`, and `clsTkTable` will read the embedded table’s entries in-line).

Wherever this table calls for number of rows or columns in a `TK_TABLE_ENTRY` field, the interpretation of what you are setting depends on whether the flag `tkTableVertical` or `tkTableHorizontal` is OR’d into `flags`. For example, for a choice in a toolkit table, the default orientation is vertical and the default number of columns is one. However, if you set the `tkTableHorizontal` flag and set `arg2` to 3, you make the choice wrap onto three columns. If neither flag is set, `arg2` is ignored (flags are described below). Note that these flags apply to the nested table item, not to the parent toolkit table; you would change the orientation of the parent toolkit table by changing its `TABLE_LAYOUT_NEW_ONLY` fields.

`prune` is a `FIM_PRUNE_CONTROL` flags word, described in more detail in "Displaying Installed Fonts in a Table," later in this chapter.

### Defining Table Entries Statically

You usually declare the `TK_TABLE_ENTRY` array statically:

```c
static TK_TABLE_ENTRY myMenuBar = {
    {itemlarg1, itemlarg2, itemlarg3, itemlarg4, itemflag1, itemclass, itemhelpid},
    {item2arg1, 0, item2arg3, item2tag, item2flags, 0, 0},
    ...
    ...
    {pNull}
};
```

This saves the code required to initialize each of the entries. You can leave trailing fields out and the C compiler will fill them with zeros.

Furthermore, wherever a particular component class calls for a pointer to another `TK_TABLE_ENTRY` (a `pEntries` in the above table), instead of declaring a separate array, you can embed a new array in the static definition. To do this enter `pNull` where the `pEntries` pointer would go, and follow that entry with the entries for the embedded `TK_TABLE_ENTRY`. `clsTkTable` will create the second toolkit table from these entries.

If you do embed `TK_TABLE_ENTRY`s inside others, make sure that each array ends with a null entry (`{pNull}`). `clsTkTable` has to find the empty entry to know where the embedded table ends.

### Toolkit Table Flags

The possible `flags` values in `TK_TABLE_ENTRY` allow you to set common styles for the existing label and button classes without having to resort to subclassing or supplying a `msgNew` structure. The flags also work for classes descending from the existing label and button classes. The flag values are listed in Table 38-3.
### Table 38-3

<table>
<thead>
<tr>
<th>Flag Values</th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tkLabelEntry</td>
<td>arg1 is a P_TK_TABLE_ENTRY.</td>
</tr>
<tr>
<td>tkLabelStringId</td>
<td>arg1 is a string resident.</td>
</tr>
<tr>
<td>tkLabelBold</td>
<td>Use bold system font in label.</td>
</tr>
<tr>
<td>tkLabelWordWrap</td>
<td>Word wrap the label string.</td>
</tr>
<tr>
<td>tkButtonPargsValue</td>
<td>Send value instead of data.</td>
</tr>
<tr>
<td>tkButtonPargsUID</td>
<td>Send UID instead of data.</td>
</tr>
<tr>
<td>tkButtonOn</td>
<td>Turn on the button.</td>
</tr>
<tr>
<td>tkButtonHalfHeight</td>
<td>Use half-height button border.</td>
</tr>
<tr>
<td>tkButtonManagerNone</td>
<td>Set button manager to bsManagerNone.</td>
</tr>
<tr>
<td>tkButtonToggle</td>
<td>Make button a toggle.</td>
</tr>
<tr>
<td>tkButtonBox</td>
<td>Use bsFeedbackBox.</td>
</tr>
<tr>
<td>tkMenuPullRight</td>
<td>arg2 is pEntries for a pull-right menu.</td>
</tr>
<tr>
<td>tkMenuPullDown</td>
<td>arg2 is a pEntries for a pull-down menu.</td>
</tr>
<tr>
<td>tkContentsSection</td>
<td>arg2 is a pEntries for section contents.</td>
</tr>
<tr>
<td>tkInputDisable</td>
<td>Disable input.</td>
</tr>
<tr>
<td>tkBorderEdgeTop</td>
<td>Turn on top border.</td>
</tr>
<tr>
<td>tkBorderEdgeBottom</td>
<td>Turn on bottom border.</td>
</tr>
<tr>
<td>tkBorderMarginNone</td>
<td>Turn off all margins.</td>
</tr>
<tr>
<td>tkBorderLookInactive</td>
<td>Make entry inactive.</td>
</tr>
<tr>
<td>tkTableHorizontal</td>
<td>Table is horizontal.</td>
</tr>
<tr>
<td>tkTableVertical</td>
<td>Table is vertical.</td>
</tr>
<tr>
<td>tkNoProto</td>
<td>Do not use prototypical pButtonNew.</td>
</tr>
<tr>
<td>tkNoClient</td>
<td>Do not copy client fields.</td>
</tr>
<tr>
<td>tkPopupChoiceFont</td>
<td>Use current font name.</td>
</tr>
<tr>
<td>tkControlDynamicClient</td>
<td>Set dynamicEnable equal to csDynamicClient.</td>
</tr>
<tr>
<td>tkControlDynamicObject</td>
<td>Set dynamicEnable equal to csDynamicObject.</td>
</tr>
<tr>
<td>tkControlDynamicPargs</td>
<td>Set dynamicEnable equal to csDynamicPargs.</td>
</tr>
<tr>
<td>tkControlCallSel</td>
<td>tkControlDynamicObject.</td>
</tr>
<tr>
<td>tkControlSelLocal</td>
<td>tkControlDynamicPargs.</td>
</tr>
<tr>
<td>tkMenuButtonGetMenu</td>
<td>Send msgMenuButtonProvideMenu.</td>
</tr>
<tr>
<td>tkMenuButtonEnableMenu</td>
<td>Send msgMenuControlEnable.</td>
</tr>
</tbody>
</table>
As an example of a complex nested toolkit table, the following example shows the code from TkDemo for its sample option table (in `\PENPOINT\SDK\SAMPLE\TKDEMO\OPTABLES.C`).

**Example 38-1**

**TK_TABLE_ENTRY Array for an Option Card**

**Option cards**, the client windows in option sheets, are usually instances of `clsOptionTable`, a descendant of `clsTkTable`. The only differences between `clsOptionTable` and `clsTkTable` is that the defaults for the former are a two-column table of bold labels. You can often specify every component in the option card in a deeply nested static `TK_TABLE_ENTRY` array.

To show off the flexibility of toolkit tables, two of the items nested in the table are identical custom-aligned On/Off/Show groupings. The Header & Footer and the Background Doc options use this same item, so it is specified as a separate `TK_TABLE_ENTRY` (`onOffChoice`) to avoid duplicating a static array. (If it weren't for this, the entire option table could be specified in one `TK_TABLE_ENTRY` array.)

```c
static const TK_TABLE_ENTRY onOffChoice[] = {
    {0, 0, 0, 0, tkLabelEntry},
    {0, 0, 0, 0, tagNestedTbl, tkTableWideGap | tkTableHorizontal | tkInputDisable, clsTkTable},
    {"On", 0, 0, 0, tkInputDisable | tkBorderMarginNone, clsLabel},
    {"Show", msgButtonNotify, 0, 0, tkButtonManagerNone, clsButton},
    {pNull},
    {"Off", 0, 0, 0, tkButtonOn},
    {pNull}
};
```

Here’s the main `TK_TABLE_ENTRY` for the card.

```c
static const TK_TABLE_ENTRY sampleCard[] = {
    {"Pages:"}, {0, 0, 0, 0, tagPagesChoice, tkNoClient, clsChoice},
    {"All"},
    {0, 0, 0, 0, tkLabelEntry | tkButtonOn},
    {0, 0, 0, 0, tagNestedTbl, tkTableHorizontal, clsTkTable},
    {"From", 0, 0, 0, tkBorderMarginNone, clsLabel},
    {"1", 2, 0, 0, 0, clsIntegerField},
    {"to", 0, 0, 0, tkBorderMarginNone, clsLabel},
    {"1", 2, 0, 0, 0, clsIntegerField},
    {pNull},
    {pNull},
    {"No. of Copies:"}, {"1", 0, 0, 0, 0, clsIntegerField},
    {"Name:"}, {0, 10, 0, 0, 0, clsTextField},
    {"Paper Size:"}, {0, 0, 0, 0, tkNoClient, clsChoice},
    {"8 1/2 X 11"},
    {"8 1/2 X 14", 0, 0, 1, tkButtonOn},
    {pNull},
    {"Printer:"}, {0, 0, 0, 0, tkNoClient, clsChoice},
    {"HP"},
    {"Diconix", 0, 0, 1},
    {"Apple", 0, 0, 2, tkButtonOn},
    {pNull},
    {"Header & Footer:"}, {onOffChoice, 0, 0, tagHFChoice, tkNoClient, clsChoice},
    {"Background Doc:"}, {onOffChoice, 0, 0, tagBDChoice, tkNoClient, clsChoice},
    {pNull}
};
```
The example entry above is a static definition outside of any function. When the time comes to create the toolkit table, you specify the top level TK_TABLE_ENTRY array as an argument to clsTkTable (or one of its descendants). clsTkTable creates all the nested components, including other toolkit tables. In the case of TkDemo, the application calls an internal routine (AddOptionTableExamples) to create the option sheet as part of another window (parent).

```
STATUS LOCAL AddOptionTableExamples (WIN parent) {
    STATUS s;
    OPTION_TABLE_NEW otn;
    AddLabel(parent, "Standard Option Table, with Nested Tables");
    OjbCallRet(msgNewDefaults, clsOptionTable, &otn, s);
    otn.border.style.edge = bsEdgeAll;
    otn.tableLayout.style.childYAlignment = tlAlignBaseline;
    otn.tableLayout.rowHeight.constraint = tlGroupMax | tlBaselineBox;
    otn.tkTable.pEntries = sampleCard;
    OjbCallRet(msgNew, clsOptionTable, &otn, s);
    StsRet(AdjustTblAlignment((WIN) ObjectCall(msgWinFindTag, otn.object.uid, (P_ARGS) tagPagesChoice)), s);
    StsRet(AdjustTblAlignment((WIN) ObjectCall(msgWinFindTag, otn.object.uid, (P_ARGS) tagHFChoice)), s);
    StsRet(AdjustTblAlignment((WIN) ObjectCall(msgWinFindTag, otn.object.uid, (P_ARGS) tagBDChoice)), s);
    otn.win.parent = parent;
    OjbCallRet(msgWinInsert, otn.object.uid, &otn.win, s);
    return stsOK;
} /* AddOptionTableExamples */
```

Figure 38-2 shows what the complete table looks like.

**Figure 38-2**

**Toolkit Table from TkDemo Application**
Creating the Child Windows

When you send msgNew, clsTkTable creates the child windows specified in P_TK_TABLE_ENTRY. It has to create a msgNew structure for each one. In this structure, it sets the win.parent to self, sets the win.options to wsPosTop, and sets the button.style.manager to bsManagerParent (if the child window is a button). It sets the other defaults using appropriate values from the prototypical msgNew structure in TK_TABLE_NEW_ONLY (pButtonNew), the class specified, and the values and flags in the TK_TABLE_ENTRY. The process of establishing these defaults is quite complex, but often you can ignore it and just specify as few parameters as possible.

clsTkTable then sends msgNew with the resulting structure to the class of the nested component to create that child, and inserts it as a child window.

Changing Defaults in a Toolkit Table

When you create a toolkit table with a separate static TK_TABLE_ENTRY array, you eliminate all the code from your program, which declares the msgNew structures, fills them in, and sets their values. Instead, clsTkTable creates the entries in the toolkit table for you. If you want the default items in your toolkit table, you can often get by with setting no more than the label strings and button messages, leaving everything else to zero.

If you do want the entries to be different, there are several ways to change them. clsTkTable uses several levels of defaulting to establish the appropriate msgNew arguments when it creates the entries. On top of this are the changes made by the common descendants of clsTkTable, such as menus and choices.

Using the Default Item Class

If you don't specify a class for an item in its TK_TABLE_ENTRY, clsTkTable uses the default class for the nested items. The default class for the nested items comes from the pButtonNew field in TK_TABLE_NEW_ONLY as follows:

- pButtonNew should point to a default msgNew structure for the default child item in the table.
- When msgNewDefaults gets to clsClass, it sets pButtonNew->object.class to the UID of the class that received msgNewDefaults.
- clsTkTable looks at pButtonNew->object.class to see what class to create.
- Since clsTkTable sets up pButtonNew by sending msgNewDefaults to clsButton, its default child item is a button. However, you can tell clsTkTable to use another class by declaring a _NEW structure for another class, sending msgNewDefaults to that class, and passing the address of your structure as pButtonNew.

If the class in an item's TK_TABLE_ENTRY is not set, then clsTkTable sends msgNew to the default class for the child items, passing it the pButtonNew structure. Thus, you can give different defaults for all items in the table by
modifying pButtonNew. For example, if your class wanted to have a large margin around every child window in the table, you could send msgNewDefaults yourself to the class of the child windows, change border.style.margin to bsMarginLarge, and set pButtonNew to the address of your custom msgNew arguments structure.

**Specifying the Item Class**

If class in an item’s TK_TABLE_ENTRY is set, then clsTkTable does not use the pButtonNew structure in TK_TABLE_NEW. Instead, it sends msgNewDefaults to the entry’s class to initialize it to default values. Thus, you can give different defaults for a particular child in the table by specifying a class for it.

After getting a filled-in msgNewDefaults structure for the entry’s class, clsTkTable then self-sends msgTkTableChildDefaults so that it can modify this msgNew structure. This allows clsTkTable and its subclasses to tinker with any component placed inside them. For example, toolkit tables tinker with the border windows and parent clip. For another example, clsMenu changes button items so that the button’s manager is its parent (bsManagerParent) so that it can take down the menu.

clsTkTable and its subclasses determine how much memory to allocate the msgNew and msgNewDefaults structures by sending msgNewArgsSize to the class of the entry.

**Using Flags to Modify Items**

Another way to change the defaults for a particular child item is to set appropriate flags in the item’s TK_TABLE_ENTRY structure to get the desired behavior.

TKTABLE.H defines flags for many common adjustments to items, including such things as making labels bold (tkLabelBold), making buttons send their UIDs (tkButtonPargsUID), giving a menu button a pull-right menu (tkMenuPullRight), giving a child a bottom edge (tkModeEdgeBottom), and so on. clsTkTable checks to make sure that the class matches the flag set, so that if you set, say, tkButtonPargsValue in an entry’s flags and the entry isn’t a button, it won’t try to change the entry.

**Using Low-Level Customization**

If all these adjustments fail, you can supply the address of a custom msgNew arguments structure in the first field of a particular child item’s TK_TABLE_ENTRY. If you do this, you must set tkPNew in the entry’s flags so that clsTkTable knows about this change. This is similar to supplying a tkTable.pButtonNew, but the former supplies custom msgNew arguments for every entry that doesn’t specify a class, whereas the latter is for a single entry only.

Finally, you can always let clsTkTable create a standard item, and then you can locate it using msgWinFindTag and modify it as you desire.
Modifying a Toolkit Table

Creating and Adding Your Own Items

You can add new child items to a toolkit table using msgTkTableAddAsFirst, msgTkTableAddAsLast, msgTkTableAddAsSibling (relative to a window UID in the table), and msgTkTableAddAt (at a zero-based index).

If you do this, you probably want the items to have the same appearance and behavior as other items in the toolkit table. To ensure this, after sending msgNewDefaults to the class of the item, send msgTkTableChildDefaults to the table, passing it the child's initialized _NEW structure. clsTkTable responds by setting some window flags in the child window to improve performance; descendants of clsTkTable make other changes. Send msgNew to the child's class, passing the modified _NEW structure, then insert the child window.

Modifying Items in a Toolkit Table

After creation, you can get the UID of a nested child within a toolkit table using msgWinFindTag to find a particular tagged window. Having retrieved the window, you can modify its metrics. It's a bad idea to change those metrics, which a toolkit table requires its nested items to have in order to work properly—know what you're doing. For example, if you found a button in a menu and changed its manager style, the user tapping on the button might no longer take down the menu.

Toolkit Tables and Window Tags

You can use the window tag facility to locate items in toolkit tables without having to remember the UIDs of every window.

Since msgWinFindTag propagates down into child windows, it's important that you define unique tags for your components so as not to clash with other components in the UI Toolkit. You can either use low numbers, or define your tags using MakeTag() and one of your classes, hence guaranteeing unique tags. GO publishes the tags of many items in system option sheets and menus so that adventurous developers can find and modify them. The tags definitions are in header files in \PENPOINT\SDK\INC, such as APPTAG.H and TV_TAGS.H.

Painting

A toolkit table may contain many child windows, which may contain their own child windows. Using dozens of windows this way is very flexible, but can lead to slow repainting. clsTkTable tinkers with window style flags and border styles in order to improve repainting. You need only be concerned about these changes if you are putting special windows in a toolkit table, trying to draw in it yourself, or otherwise making unusual use of clsTkTable.

Ordinarily, a child window is prevented from painting pixels outside its rectangle on its parent, and if it is covered by sibling windows it can't paint pixels on them.
Similarly, a parent is prevented from painting pixels within its child windows. The result is that windows don’t paint over each other. The cost is that the window system has to re-compute the area to draw on for each window.

For windows in a toolkit table, all this protection is usually unnecessary. Labels do not draw outside their boundaries. clsTkTable places child windows so they do not overlap. clsTkTable paints the background for its children but does not otherwise paint over them.

Consequently, when clsTkTable creates its child windows, it changes their window style flags. It turns on wsParentClip in each child window, and turns off wsClipSiblings and wsClipChildren. In its own window style flags, it turns off wsClipChildren. For more information on the effects of these window flags, see Part 3: Windows and Graphics.

If a child window is a border window, clsTkTable sets its background ink to transparent (border.style.backgroundInk set to bsInkExclusive). When the toolkit table paints its background, it will fill its child windows, so the border windows don’t need to fill their own backgrounds.

**Layout**

clsTkTable inherits from clsTableLayout, and toolkit tables use table layout functionality to position their nested components. By default (in response to msgNewDefaults), clsTkTable sets its table layout to a single row and multiple columns by setting:

- `tableLayout.numRows.constraint` to `tlAbsolute`.
- `tableLayout.numRows.value` to 1.
- `tableLayout.colWidth.constraint` to `tlGroupMax`.
- `tableLayout.style.growChildWidth` to `false`.
- `tableLayout.rowHeight.constraint` to `tlChildrenMax`.
- `tableLayout.style.growChildHeight` to `true`.

As a result, the nested components are all the same height but they are of different widths. Descendant classes can override this when handling either `msgNew` or `msgNewDefaults`. For example, a menu choice has one and many rows.

clsTkTable does not place any limitation on the size of its nested components, which can cause problems if your application creates menus or choices out of lists of strings that are not predefined. If one of the strings is very long, the toolkit table will be very large and ill-proportioned. You can solve this problem by putting the toolkit table in a scrollwin, described in Chapter 40, Scrollbars.
You set a client for the toolkit table in its `msgNew` arguments. By default, `clsTkTable` copies this client into each control within it, and within nested toolkit tables within it. You can turn off this behavior if you set the flag `tkNoClient` in an entry's `TK_TABLE_ENTRY`. `clsTkTable` also intercepts `msgControlSetClient` and does the same copying into each of its controls.

A toolkit table simply relays notification messages from its nested controls to its manager using `msgWinSend`.

`clsTkTable` does not set `notifyDetail` or respond to any `msgButtonControl`...

**Control Enable**

When a toolkit table receives `msgControlEnable`, it recursively enumerate itself and forwards the `msgControlEnable` to every instance that is a control within itself, passing the original message arguments. Since a menu is a toolkit table, if the menu receives `msgControlEnable`, the controls in the menu will also receive the message.

There are two `TK_TABLE_ENTRY` flags that set the `dynamicEnable` control style field to common values:

- `tkControlDynamicClient` sets it to `csDynamicClient`.
- `tkControlDynamicPargs` sets it to `csDynamicPargs`.

By setting these flags, you get dynamic control over whether a control in the toolkit table is enabled, as explained in Chapter 35, Controls.

The `tkMenuButtonGetMenu` `TK_TABLE_ENTRY` flag sets a menu button's `getMenu` style flag. The `tkMenuButtonEnableMenu` `TK_TABLE_ENTRY` flag sets a menu button's `enableMenu` style flag. By setting these flags, you get dynamic control over a menu button's sub-menu, as explained in Chapter 39, Menus and Menu Buttons.

**Managers**

When you have a set of objects whose interactions need to be coordinated, you can have a manager for them. `clsTkTable` maintains a `manager` field. The manager gets certain notifications from buttons in the toolkit table whose `button.style.manager` is `bsManagerParent`.

**Button Manager Notification Details**

If a button is in a toolkit table, it uses `msgWinSend` to send protocol messages (in this case `msgButtonDone`, `msgButtonAcceptPreview`, and so on) to its manager so long as `button.style.manager` is not `bsManagerNone`. `clsWin` responds to `msgWinSend` by propagating the message to the instance's parent window.
When someone sets the value of a button, it sends `msgButtonDone` to its manager using `msgWinSend`.

For example, a toolkit table sets up the buttons that are its children so that their `button.style.manager` is `bsManagerParent`. In other words, the buttons pass manager notifications to the toolkit table (their parent) via `msgWinSend`. However, the toolkit table intercepts `msgWinSend`, and if the toolkit table has a `manager` defined in its metrics, it jumps out of the window tree send and just `ObjectCalls` its manager. If the manager returns `stsManagerContinue`, then the button will continue propagation via `msgWinSend`.

The point of all this is that you can arbitrarily nest button components and have them be managed by some object. You can have a button in a custom layout inside another button send protocols up the tree and get to the appropriate manager.

**Menu Management**

For menus, the manager is the menu button that puts up the menu. The menu button takes down the menu when the user activates one of the buttons in the menu, as follows:

- Button sends `msgButtonDone` using `msgWinSend`.
- Menu gets this and sends `msgMenuDone` to its manager, the menu button.
- Manager always receives `msgButtonDone`, even if it doesn’t get all the detail messages such as `msgButton...Preview`.

The menu is doing a grab; it looks at the destination of input event, and if it’s not in a descendant, it sends manager `msgMenuDone`.

**Manager Classes**

Other classes require more sophisticated managers, so there are a few classes of specialized manager objects, including `clsChoiceMgr` and `clsSelChoiceMgr`. The former manages the settings in a choice so that only one button in it is ever on; the latter manages objects that are selectable. Both these classes inherit from `clsManager`. `clsManager` is an abstract class, which means that it is not useful to create instances of `clsManager`. `clsManager` exists simply to define details that are common to all its subclasses.

**Choice Management**

`clsChoice` creates an instance of `clsChoiceMgr` and sets that as the manager of the toolkit table. The choice manager ensures that, at most, only one button is on.

Table 38-4 summarizes the messages defined by `clsChoiceMgr`: 
Selection Choice Management

clsSelChoiceMgr goes further, and helps its client acquire the selection when one of its objects is activated.

No classes in the UI Toolkit use selection choice managers, but other PenPoint components that have to track selection choices do.

Table 38-5 summarizes the messages defined by clsSelChoiceMgr:

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>Creates a selChoiceMgr object.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>Initializes the SEL_CHOICE_MGR_NEW structure to default values.</td>
</tr>
<tr>
<td>msgSelChoiceMgrGetClient</td>
<td>Passes back the client UID held by the receiver.</td>
</tr>
<tr>
<td>msgSelChoiceMgrSetClient</td>
<td>Sets the client UID held by the receiver.</td>
</tr>
<tr>
<td>msgSelChoiceMgrGetId</td>
<td>Passes back the ID held by the receiver.</td>
</tr>
<tr>
<td>msgSelChoiceMgrSetId</td>
<td>Sets the ID held by the receiver.</td>
</tr>
<tr>
<td>msgSelChoiceMgrNullCurrent</td>
<td>Tells the receiver to clear the visuals and state of the choice.</td>
</tr>
<tr>
<td>msgChoiceMgrGetOnButton</td>
<td>Gets the current on button. Passes back objNull if no button is on.</td>
</tr>
<tr>
<td>msgChoiceMgrSetOnButton</td>
<td>Sets the current on button.</td>
</tr>
<tr>
<td>msgSelChoiceMgrAcquireSel</td>
<td>Sent to the client whenever a different button is selected.</td>
</tr>
<tr>
<td>msgSelChoiceMgrNullSel</td>
<td>Sent to the client whenever a different button is selected.</td>
</tr>
</tbody>
</table>

Another example of the use of managers is if you have two toolkit tables, but want them to behave as one exclusive choice, you can specify the same manager for both of them.
Displaying Installed Fonts in Tables

Often, you want to create a toolkit table displaying the fonts available on the system. `TkTableFillArrayWithFonts` is a utility function in TKCOMPDLL that allocates a `TK_TABLE_ENTRY` array filled in with the set of currently installed fonts. It fills in `arg1` of each entry with the name of the font and sets the `tag` field to the `FIM_SHORT_ID` of the corresponding font. You can use this array to create a choice, menu, set of labels, or whatever, of fonts.

To use `TkTableFillArrayWithFonts`, you pass in a heap from which the toolkit allocates the `TK_TABLE_ENTRY` array and strings, and a `U16` controlling whether the toolkit should prune the font list. Pruning refers to the removal of system and other hidden fonts from the font list. `TkTableFillArrayWithFonts` passes back a pointer to the allocated array.

After creating the toolkit table, you must call `TkTableFreeArray` to free the array created by `TkTableFillArrayWithFonts`.

Pruning and `FIM_SHORT_ID`s are explained in Part 12: Installation. Briefly, prune is a set of flags. It is defined to be a `U16` in `TKTABLE.H` so as not to require including lots of extraneous header files, but is actually a `FIM_PRUNE_CONTROL` structure defined by the installed font manager in `<FONTMGR.H>`. This lets you reduce the number of fonts displayed. You can set `prune` to `fimNoPruning` to get a full list, or `OR` in the following flags to reduce the font list:

- `fimPruneDupFamilies` remove fonts in the same family, such as Swiss Bold, Swiss Italic.
- `fimPruneSymbolFonts` remove symbol fonts.

If you want to provide a pop-up choice showing the installed fonts, there’s an even more direct way to get the installed fonts. OR the flag `tkPopupChoiceFont` into the `flags` field of a `clsPopupChoice` entry. The pop-up choice created will reflect the list of installed fonts. In this case, you specify the pruning parameter in place of `pEntries` in the first field of the entry.

`FIM_SHORT_ID`s are compact identifiers for fonts. `clsFontInstallMgr` provides a message, `msgFIMGetNameFromId`, which provides the font name of an ID.

There’s yet another way to create a list of installed fonts: using `clsFontListBox`. This is described in Chapter 41, List Boxes.

Removing Items from a Toolkit Table

To remove a nested item from a toolkit table, send the table `msgTkTableRemove`, specifying the UID of the child window to remove. This extracts the window, and subclasses can clean up any references to it. Then you can destroy the removed window.
Subclasses of clsTkTable

There are several subclasses of clsTkTable. Some of these add a lot of functionality; all change the toolkit table defaults and the defaults for nested components to produce a different visual effect. Using a descendant of clsTkTable further reduces the amount of defaults overriding you need to do. For example, to create a vertical column of buttons with their text rotated 90 degrees using clsTkTable, you would have to override the default child structure. But, if you use clsTabBar, these settings are the default, and you are back to specifying only the string, message, and data of each entry.

The rest of the sections in this chapter document the various descendants of clsTkTable. Menus are covered in the next chapter.

Toggle Tables

Toggle tables are a group of independent toggle buttons. clsToggleTable sets the button style during msgNew to toggle style (bsContactToggle).

Its value is a 32-bit bitmask giving the values (on or off) of the first 32 toggles nested in itself. msgControlSetValue and msgControlGetValue can be used to set or get the mask value. If you place more than 32 toggles in a toggle table, you have to enumerate or find the other toggles to get their UIDs in order to set and get their values.

Modifying Toggle Tables

Toggle tables also respond to msgControlSetEnable, msgControlGetEnable, msgControlSetDirty, and msgControlGetDirty in a similar fashion. For the Get messages, clsToggleTable passes back a bitmask indicating which of the first 32 toggles have that style set. For the Set messages, you specify to clsToggleTable which of the first 32 buttons should be on and off by turning bits on and off in the bitmask.

The only reason to use a toggle table is so that you can set and get the values and styles of several buttons at once. You don’t have to use clsToggleTable to create a group of buttons. You can always create a vanilla toolkit table by sending msgNew to clsTkTable, specifying tkButtonToggle in the flags of each entry’s TK_TABLE_ENTRY.

Choices

clsChoice is a descendant of clsTkTable. It sets child windows in the toolkit table to be buttons with the bsContactLockOn style. However, clsChoice defines additional semantics such that only one button can be on at any given time.

Table 38-6 summarizes the messages defined by clsChoice:
CHAPTER 38 / TOOLKIT TABLES

Choices

Table 38-6

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>Creates a choice (and its nested button windows).</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>Initializes the CHOICE_NEW structure to default values.</td>
</tr>
<tr>
<td>msgChoiceGetStyle</td>
<td>Gets the style of the receiver.</td>
</tr>
<tr>
<td>msgChoiceSetStyle</td>
<td>Sets the style of the receiver.</td>
</tr>
<tr>
<td>msgChoiceSetNoNotify</td>
<td>Like msgControlSetValue, but without button notifications.</td>
</tr>
</tbody>
</table>

Creating a Choice

You send msgNew to clsChoice to create a choice. This takes a CHOICE_NEW structure. The most important field in this is tkTable.pEntries, a pointer to an array of TK_TABLE_ENTRYs that specify the buttons in the choice.

You specify the contents of the choice in the TK_TABLE_ENTRY array. The child windows can be any kind of window. By default they are buttons. The TK_TABLE_ENTRY fields for a button are pString, message, and data (and the standard tag, flags, class, and helpId).

There are no fields currently used in the CHOICE_NEW_ONLY part of the CHOICE_NEW structure.

Notification

The buttons within a choice notify their client(s) in the usual way. The buttons also send previewing messages to their manager.

clsChoice is the only descendant of clsTkTable that turns on notifyDetail in its nested buttons. It responds to their msgButtonChild...Preview messages by ensuring that the current choice is de-highlighted while the user previews other items.

Choice Manager

clsChoice internally uses a clsChoiceMgr to ensure that, at most, only one button is on. You can access the choice manager by sending the choice msgTkTableGetManager.

Choice Value

By default, a choice sets up its child buttons to have the button style bsContactLockOn set. This results in the behavior that the choice always has one button on. You set the on button at creation by setting the flag tkButtonOn in that buttons TK_TABLE_ENTRY array.
To allow no button to be on, you need to change the choice so that its buttons have the style `bsContactToggle`, by changing `pButtonNew->button.style контакt` to `bsContactToggle` in the `CHOICE_NEW` structure.

`clsChoiceMgr` defines messages to get and set the current on button (`msgChoiceMgrGet/SetOnButton`), but you can also get the choice's value by sending the choice `msgControlGetValue` directly. Note that a choice is not a control, but it responds to this message anyway. The value of a choice is the window tag of the on button, not the index of the button in the choice or the button's UID. If no button is on, `clsChoice` returns `stsChoiceNoValue`. Conversely, you can set the value of a choice using `msgControlSetValue`.

To set the value of the choice to be no value, send its manager `msgChoiceMgrSetOnButton` with a message argument of `objNull`.

When you set the value of the choice, the button turned on sends its notification message. Your code may be unprepared to deal with messages from buttons before its user interface is fully realized. One way to avoid this is not to set up a client until your code is ready to receive messages. You can also use `msgChoiceSetNoNotify` (or `msgChoiceMgrSetNoNotify` to the choice's manager) to change a choice's value without notification.

If you don't intend to set or get the value of the choice, you need not specify tags for its buttons.
Chapter 39 / Menus and Menu Buttons

A menu button may have an associated pop-up menu. If it does not, it acts like other buttons. If the menu button does have a menu, the menu button displays its menu when the user taps on the menu button. The menu can, in turn, have menu buttons nested in it that pop up submenus.

clsMenuButton inherits from clsButton. clsMenu inherits from clsTkTable.

Menu Buttons

clsMenuButton implements a menu button. Menu buttons behave similarly to regular buttons, but they can have a submenu that they pop up and take down.

Table 39-1 summarizes the messages defined by clsMenuButton:

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>Creates a menu button window.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>Initializes the MENU_BUTTON_NEW structure to default values.</td>
</tr>
<tr>
<td>msgMenuButtonGetStyle</td>
<td>Passes back the current style values.</td>
</tr>
<tr>
<td>msgMenuButtonSetStyle</td>
<td>Sets the style values.</td>
</tr>
<tr>
<td>msgMenuButtonGetMenu</td>
<td>Passes back the pull-right or pull-down menu, objNull if none.</td>
</tr>
<tr>
<td>msgMenuButtonSetMenu</td>
<td>Sets the pull-right or pull-down menu.</td>
</tr>
<tr>
<td>msgMenuButtonProvideWidth</td>
<td>Self-sent when style.getWidth is true.</td>
</tr>
<tr>
<td>msgMenuButtonShowMenu</td>
<td>Puts up or takes down the menu.</td>
</tr>
<tr>
<td>msgMenuButtonInsertMenu</td>
<td>Self-sent when menu is down and the action specified by style.menuAction is detected.</td>
</tr>
<tr>
<td></td>
<td>Puts up the submenu by inserting it as sibling window of the menu button or by sending msgMenuShow(true).</td>
</tr>
<tr>
<td>msgMenuButtonExtractMenu</td>
<td>Self-sent when menu is au and the action specified by style.menuAction is detected.</td>
</tr>
<tr>
<td></td>
<td>Takes down the submenu by extracting its window or by sending msgMenuShow(false).</td>
</tr>
<tr>
<td>msgMenuButtonPlaceMenu</td>
<td>Self-sent whenever a menu button needs to position its associated menu.</td>
</tr>
<tr>
<td>msgMenuButtonProvideMenu</td>
<td>Sent to the client if style.getMenu is true.</td>
</tr>
<tr>
<td>msgMenuButtonMenuDone</td>
<td>Sent to the client if style.getMenu is true.</td>
</tr>
</tbody>
</table>
Creating a Menu Button

Using Toolkit Tables Instead

You can create a free-standing menu button, but usually you use menu buttons as nested controls in menus. Menus are toolkit tables, and permit you to create all their menu buttons at once from a set of TK_TABLE_ENTRIES. clsMenu is explained in the next section.

Send msgNew to clsMenuButton to create a menu button. This takes a MENU_BUTTON_NEW structure, which includes a MENU_BUTTON_NEW_ONLY structure. In this, you specify:

- **style**: several menu button style fields, described below.
- **menu**: the submenu of the menu button, if it has one.

Menu Button Style

The menu button **style** is a MENU_BUTTON_STYLE structure. Table 39-2 summarizes the structures field elements:

<table>
<thead>
<tr>
<th>Fields/Field Values</th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>submenuType</td>
<td>Determines type of button submenu.</td>
</tr>
<tr>
<td>mbMenuNone</td>
<td>No button submenu type defined.</td>
</tr>
<tr>
<td>mbMenuPullDown</td>
<td>Brings up submenu as a pull-down menu.</td>
</tr>
<tr>
<td>mbMenuPullRight</td>
<td>Brings up submenu as a pull-right menu.</td>
</tr>
<tr>
<td>mbMenuPopup</td>
<td>Brings up submenu as a pop-up menu.</td>
</tr>
<tr>
<td>mbMenuSibling</td>
<td>Inserts submenu as a sibling window of the menu button.</td>
</tr>
<tr>
<td>getWidth</td>
<td>Specifies whether the menu button should dynamically compute the width of the submenu.</td>
</tr>
<tr>
<td>getMenu</td>
<td>Specifies whether the menu button's client should dynamically provide the submenu.</td>
</tr>
<tr>
<td>enableMenu</td>
<td>Specifies whether the submenu should dynamically determine if its nested controls should be enabled.</td>
</tr>
<tr>
<td>menuAction</td>
<td>Determines if the submenu is brought up on a single tap or a double tap.</td>
</tr>
<tr>
<td>menuIsUp</td>
<td>Determines whether submenu is visible (read-only).</td>
</tr>
</tbody>
</table>

If getWidth, getMenu, or enableMenu is set, the menu button self-sends additional messages to determine these values; the messages are explained in more detail in "Menus," later in this chapter. You can also set and get the style separately from msgNew using msgMenuButtonSetStyle and msgMenuButtonGetStyle.

Notification

A menu button will invert itself when it receives msgControlBeginPreview. Even if the button has a submenu (submenuType is mbMenuNone), it does normal control notification by sending its client its msg and data on msgControlAcceptPreview (just as an instance of clsButton does). A menu button's client is usually the application.
Painting

If subMenuType is mbPullRight, the menu button sets its decoration style to
IsDecorationPullRight and clsLabel draws it with a pull-right arrow decoration to
the right of the string.

Menus

clsMenu defines the class for child windows in the toolkit table to be
clsMenuButton. You can set the menu style to be a horizontal menu bar
(msMenuBar), or a vertical menu (msMenu). Submenus off a menu bar and
pull-right menus are both vertical menus. It sets self’s table layout constraints to
be either one row, many columns (if self’s style is msMenuBar) or one column,
many rows (style msMenu).

Table 39-3 summarizes messages defined by clsMenu.

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>Creates a menu window, together with the child windows specified in pEntries.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>Initializes the MENU_NEW structure to default values.</td>
</tr>
<tr>
<td>msgMenuGetStyle</td>
<td>Passes back the current style values.</td>
</tr>
<tr>
<td>msgMenuSetStyle</td>
<td>Sets the style values.</td>
</tr>
<tr>
<td>msgMenuShow</td>
<td>Puts up or takes down the menu by inserting or extracting it as a child of theRootWindow.</td>
</tr>
<tr>
<td>msgMenuDone</td>
<td>Sent via msgWinSend to the manager when the menu is “done.”</td>
</tr>
</tbody>
</table>

msgTblLayoutAdjustSections now does what msgMenuAdjustSections did in
earlier versions of PenPoint, but is available to all descendants of clsTableLayout.
msgMenuAdjustSections is provided for backward compatibility; you should use
msgTblLayoutAdjustSections in your new clients.

Creating a Menu

You send msgNew to clsMenu to create a menu. This takes a MENU_NEW
structure. The most important field in this is tkTable.pEntries, a pointer to an
array of TK_TABLE_ENTRYs that specify the contents of the menu.

The fields in MENU_NEW_ONLY include:

- style  style fields including shadow and margin style, as well as row and
column constraints.
- menuButtonNew in-line storage for a MENU_BUTTON_NEW structure.

In msgNewDefaults, clsMenu sets up menu.menuButtonNew to be the msgNew
arguments for a default menu button. It does this based on the menu.style.type of
the menu. It changes tkTable.pButtonNew to point to this structure. If you want
to change the defaults for the menu buttons, cast tkTable.pButtonNew to a
P_MENU_BUTTON_NEW; don't modify menu.menuButtonNew.

menuButtonNew only allocates storage for the msgNew structure so that
clsMenu doesn't have to.

You specify the contents of the menu in the TK_TABLE_ENTRY array. The child
windows can be any kind of window. By default, they are menu buttons. The
TK_TABLE_ENTRY fields for a menu button with no submenu are the same as a
button's:

- pString, message, data (and the standard tag, flags, class, and helpId).

Creating Submenus

The menu buttons in the menu may have submenus of their own. You indicate
this by setting either tkMenuPullRight, tkMenuPullDown, or tkMenuPopup in
the flags field of a TK_TABLE_ENTRY. This tells clsMenu that the menu button
has a submenu. It then interprets the second field as a pointer to the array of
TK_TABLE_ENTRYs for the contents of the submenu, and the third field is ignored:
pString, pEntries, ... (and the standard tag, flags, class, and helpId)

As usual, if pEntries is pNull, then clsTkTable assumes it will find the
TK_TABLE_ENTRYs for the nested child windows in-line. You can define an entire
menu hierarchy using TK_TABLE_ENTRYs, and only need explicitly to create the
topmost menu.

Displaying a Menu

If you want to display a pop-up menu yourself, you can send it msgMenuShow.
This takes a Boolean: true to display the menu, false to take it down. Before
sending this to display a menu, you should set it to the desired location by
changing its bounds.origin using msgWinDelta. clsMenu ensures that the menu
is entirely on-screen, repositioning it as necessary, before inserting the menu in the
root window.

How a Menu Button Displays its Submenu

You are free to use a menu as you would any other toolkit component. You could
have a static menu in the middle of a window (TkDemo actually does this).
Typically, however, a menu is displayed by a menu button.

When the user taps (or double taps, depending on the value of style.menuAction)
on a menu button, it sends its button message to its client in the usual manner. In
addition, if its submenuType is mbPullDown, mbPullRight, or mbPopup, on
msgControlAcceptPreview the menu button puts up its submenu, and on the
next msgControlAcceptPreview, it takes down its submenu.
In more detail, the menu button:

1. Sets itself as the manager of the menu (using `msgTkTableSetManager`).
2. Uses `msgWinDelta` to set the menu to the appropriate location relative to itself.
3. Puts up the menu by sending it `msgMenuShow` or, if `style.subMenuType` is `mbMenuSibling`, by inserting the menu as a window tree sibling of the menu button.
4. Takes down the menu when it next receives `msgControlAcceptPreview`.

`clsMenu` ensures that the menu will never be off-screen. While it is up, the menu grabs input so that it can see pen taps outside its window; a pen tap outside the window dismisses the menu. `clsMenu` looks at the destination of the pen tap, and if it is one of its nested buttons, it allows the tap to go to that button.

The menu button sets itself as the manager each time it puts up the submenu, in case it is sharing the menu with another menu button. Menu buttons in the menu typically set their manager to be `bsManagerParent` so that their notification messages get forwarded to the toolkit table’s parent, in this case, the menu.

Usually a menu button with a submenu does not notify its own to its client; its own control `msg` and `data` are unused and the client only gets activation messages from the submenu’s menu buttons.

If you set the `subMenuType` to `mbPullDown`, `mbPullRight`, or `mbPopup`, but the menu button doesn’t have a submenu, the menu button will preview but not do anything.

**Dynamic Submenu**

If the `getMenu` menu button style is set, then when the menu button needs to put up its submenu, it sends `msgMenuButtonProvideMenu` to its client. This takes a `MENU_BUTTON_PROVIDE_MENU` structure as its message arguments, in which the menu button sets `menuButton` as its UID, and `menu` as the UID of its current submenu. The client can either modify the passed `menu` (for example to add, disable, or delete some items) or replace it with the UID of a new menu.

**Control Enable**

c`lsMenuButton` checks the value of the instance’s `enableMenu` style field. If this is `false` (the default), then the menu button just puts up its submenu without further ado (by sending the menu `msgMenuShow`). Otherwise, the menu button sends `msgControlEnable` to its submenu. This takes a `CONTROL_ENABLE` structure, as described in Chapter 35, Controls. `clsMenuButton` fills this out as follows:

1. Sets `root` to self.
2. Sets `object` to the current selection owner.
3. Sets `enable` to `true` or `false`, depending on whether the selection owner’s process is this application’s task. If a menu button doesn’t have some other function as described in Chapter 35, Controls.
way of determining whether it is enabled, this disables it if the selection isn’t in this application.

The menu relies on its ancestor clsTableLayout to propagate msgControlEnable to each control in the menu. The controls in the submenu respond to msgControlEnable by figuring out whether they should be enabled or not. See Chapter 35, Controls, and Chapter 38, Toolkit Tables, for more information.

**Notification**

Notification is the same as clsTkTable. When the user activates a menu button, it notifies its client, then its manager. The menu button’s manager is its menu. If that menu is itself a submenu, then its manager is the menu button that popped it up. The menu button responds to msgButtonAcceptPreview by taking down the menu.

**Pop-Up Choices**

The UI Toolkit includes **choices**, which display several alternatives and allow the user to pick one (clsChoice is explained in more detail in Chapter 38). However, if there are several alternatives, the choice ends up taking up a lot of space on the screen. You can put a choice in a pop-up menu, but then the user has to pop up the menu to see the current value of the choice. clsPopupChoice is an alternative.

clsPopupChoice is a descendant of clsMenuButton. It displays a choice in a menu when you tap it, much like a menu button with a choice. The difference is that the pop-up choice’s label string is a copy of the string of the current choice button. Whenever the choice changes value, the pop-up choice copies the new value’s string to its label.

The user does not have to pop up the choice. The user can instead move to other values by making scrolling gestures directly on the pop-up choice button. Up and down flicks move to the next and previous value in the choice, and double-flicks move to the top or bottom value in the choice.

Table 39-4 summarizes the messages defined by clsPopupChoice.

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>Creates a pop-up choice button.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>Initializes the POPUP_CHOICE_NEW structure to default values.</td>
</tr>
<tr>
<td>msgPopupChoiceGetStyle</td>
<td>Passes back the receiver’s style. (Currently there are no style fields defined in POPUP_CHOICE_STYLE.)</td>
</tr>
<tr>
<td>msgPopupChoiceSetStyle</td>
<td>Sets the receiver’s style.</td>
</tr>
<tr>
<td>msgPopupChoiceGetChoice</td>
<td>Passes back the choice associated with this pop-up.</td>
</tr>
</tbody>
</table>
Creating a Pop-Up Choice

If you create a pop-up choice using the C API, you create it as you would a menu, but set the `menuButton.menu` field to the UID of an instance of `clsMenu`.

The suggested way to create a pop-up choice is to specify it as a static `TK_TABLE_ENTRY`. This approach is described in Chapter 38, Toolkit Tables.

Miscellaneous Pop-Up Choice Messages

You can retrieve the UID of the toolkit table (usually a choice) associated with a pop-up choice by sending it `msgPopupChoiceGetChoice`. Remember, the pop-up choice itself is the button displaying the current value of the choice.

`clsPopupChoice` responds to `msgControlGetValue` by retrieving the tag of the button that is on in the choice, just as `clsChoice` does.
Chapter 40 / Scrollbars

Scrollbars let the user display more of a document than can fit in a window. The user interface of a scrollbar provides information about the size of the document and the offset of the visible part of the document relative to the whole. The scrollbar UI also allows the user to move around in this document. In addition to tapping in the scrollbar, the user can also flick to scroll up and down.

Scrollbars give the user the illusion of moving the window around relative to the document it's displaying. The classic example is a long text document: using the scrollbar, the user can move up and down to see different parts of the text file. Note that the thing displayed in the window does not itself have to be a window, or even an object.

Layout

Scrollbars can either be vertical (the default) or horizontal. It is up to you to position them next to the window that displays the scrolling using, for example, a custom layout window. Usually, you use a scroll window (described later in this chapter in “Scroll Windows”), which positions the scrollbars and can often perform scrolling as well.

Painting

A scrollbar indicates what relative location in the document is visible in the document's window by drawing a drag handle (sometimes called a thumb) in itself, representing the part of the document that is visible. The scrollbar draws this bubble based on the length (or width, for a horizontal scrollbar) of the document and the current offset in the document. However, the scrollbar does not keep track of this information stuff itself.

The bubble indicates the relative position of the offset in the range. For example, if the scrollbar offset is 350 and the range is -100 to 400, then the portion of the thing shown in the window is near the end, and the scrollbar paints its bubble near the bottom. When the user moves the pen into proximity near the bubble, it grabs input and tracks the pen.

Notification

Although scrollbars are intended to manipulate a document in a window, they have no explicit connection with a window. Instead, they communicate with a client, asking it for information so they can display, and sending the client scrolling messages. clsScrollbar inherits from clsControl, which maintains a client. The client can be the document being scrolled, but it need not be.
Figure 40-1 shows a vertical scrollbar along the right hand edge of a window. The components are the same for a horizontal scrollbar.

![A Scrollbar](image)

**Thumbing**

The scrollbar handles thumbing by creating a track object. This grabs input and moves the scrollbar up and down. The scrollbar does not notify its client until the user finishes with the tracker by going out of proximity. Thus, the document does not scroll while the user drags the thumb.

**Providing Information**

The scrollbar client must tell the scrollbar the size of the document, the size of the view with which the scrollbar is associated, and the current offset into it, so that the scrollbar can draw the bubble at the correct location.

Whenever a scrollbar needs to update its display, it sends its client either msgScrollbarProvideVertInfo or msgScrollbarProvideHorizInfo. These both take a pointer to a SCROLLBAR_PROVIDE structure. The scrollbar supplies its UID in sb; it is up to the client to provide the viewLength, docLength, and offset. These are all S32; the client can supply these in any consistent units that make sense. The offset should be the offset of the top (left) of the form.

**Offset**

Note that the offset in a vertical scrollbar increases as it reaches the bottom of the document. This document model is different from the window system’s lower left origin coordinate system.

**Client Notification**

The scrollbar handles previewing internally. Pressing on the up and down arrow generates msgButtonRepeatPreview messages.

A scrollbar sends its client msgScrollbarVertScroll or msgScrollbarHorizScroll, depending on the scrollbar’s orientation. These take a SCROLLBAR_SCROLL structure as their message arguments. This structure includes:

- `sb` the UID of the scrollbar that generated the message.
**What Goes On**

In `sbLineUp` and `sbLineDown`, `offset` is not supplied (zero). The scrollbar client should pass back in `offset` the new desired offset. `sbPageUp` and `sbPageDown` are similar.

In `sbThumbUpDown`, `clsScrollbar` does supply a suggested new `offset`. It does this by converting the location of the tap in the scrollbar to an `offset` in the document.

If your window is slow to repaint, it won't be able to keep up with repeating scrolls. If this is the case, you cannot scroll in response to repeating messages (`sbLineUp`/`sbLineDown`). At the end of the user's scroll, you receive `sbEndScroll` as the action, at which point you can perform the entire scroll at once. You can either keep track of the number of repeating messages received, or normalize the `offset` each time and then perform the final scroll using the `offset` passed in along with `sbEndScroll`.

**Line, Page Scrolling versus Thumbing**

For all scroll actions, the client must interpret the scroll action and redraw the window to comply with the request (although often you can simplify the task using `clsScrollWin`).

There are three main kinds of scrolling: line scroll, page scroll, and thumbing scroll.

With a line or page scroll, the scrollbar has no idea how big a line or page is, so it can't compute the new `offset`. The `offset` in the `SCROLLBAR_SCROLL` structure is the current `offset` (the one returned by `msgScrollbarProvideVertInfo` or `msgScrollbarProvideHorizInfo`). You (the scrollbar client) must interpret how a
line or a page scroll should change the offset of the scrollbar, and pass back the new offset as an out parameter. If you don’t respond to these messages, the position of the bubble doesn’t change.

Typically, in a line scroll, you display a new part of the scrolling offset by a small amount (a “line”), while in a page scroll, you offset the display by the same amount or slightly less than the size of the window.

With a thumbing scroll (scrollbar action is sbThumbUpDown or sbThumbLeftRight), the user taps at a new location to indicate the offset. Hence, the scrollbar can supply the new offset in the offset field in the SCROLLBAR_SCROLL structure (based on the sizes of the view and document).

## Offset Range

When you set the value of the scrollbar, you should set the offset of the top or left edge of the scrolled object from the top or left edge of the window. This is the calculation used by the scrollbar display code in figuring out where to draw the thumb pointer.

Note that the offset range is not 0 – docLength. The offset represents the location of the top of the visible area relative to the document length, so an offset of docLength would imply that the very end of the document is at the top of the view and the rest of the view is empty. Instead, the range is 0 – (docLength – viewLength).

## Updating

If the document view changes for some other reason than user interaction with the scrollbar, you can force a scrollbar update by sending it msgScrollbarUpdate. The scrollbar will ask its client to provide the information when it needs to repaint.

There is no easy way to programmatically cause a scroll. Your code must perform the scroll and then tell the scrollbar to update, rather than simulating a user action which then causes a scroll.

## Normalizing the Scroll

Even if you know in advance what the offset is, you often need to adjust it. For example, if the window is displaying text, then you may want to change the offset slightly to avoid clipping off the tops or bottoms of characters in a line of text. This is called normalizing the scroll.

When the scrollbar client receives either a msgScrollBarVertScroll or msgScrollBarHorizScroll, it can adjust the offset to avoid splitting up important information at the edge of the window. In a msgScrollBarVertScroll or msgScrollBarHorizScroll, the scrollbar specifies an offset. The client can pass back a different offset.

You need to adjust offsets and repainting carefully to get this smooth presentation. You can look at the previous offset field to determine the direction in which the user is scrolling, and use this to normalize the top or bottom edge of the screen.
When the user drags the bubble, the scrollbar sends its client `msgScrollbarProvideVertInfo`. The client should respond with the appropriate `docLength`, `offset`, and so on. Then the scrollbar sends `msgScrollbarVertScroll`, with its best guess for the new offset. The client should normalize before doing the scroll, and can modify the offset. The scrollbar will repaint the thumb in that position.

The scrollbar is a border window. If you only need to scroll the display a small amount, you can use `msgWinCopyRect` to copy the pixels on the screen from one region to another. Then you only need to paint the strip of the scrolled thing that has become visible. See Part 3: Windows and Graphics, for more information on `msgWinCopyRect`.

**Scroll Windows**

The user's model of scrolling is that the window with scrollbars (such as the client window in a frame) provides a porthole looking into the much larger document surface. As he or she uses the scrollbars to scroll around, the porthole moves around to show a different part of that surface.

It's often possible to implement scrolling this way from your program's point of view. In response to a scrolling request, often all you need to do is display a different portion of the same window. `clsScrollWin` lets you do this. An instance of `clsScrollWin` acts as the porthole. It inserts your window as its child, and repositions your window in response to the `msgScrollbarVertScroll` and `msgScrollbarHorizScroll` messages from the scrollbars. Your window is isolated from the scrollbar interface; all it has to do is repaint itself when it receives `msgWinRepaint`, and the correct region of it will appear in the `scrollwin` (scroll window).

The scrollwin has an inner window. Your window, the client window, is a child of this inner window and is clipped by it. `msgScrollWinGetInnerWin` passes back a pointer to the UID of this window. You can have more than one client window in a scrollwin; but only one is visible at any time.

`clsScrollWin` uses your window's size to compute its offset. It handles `sbPageUp` and `sbPageDown` scroll actions by changing the offset by the size of its own window. However, it still can't figure out how much to change the offset for `sbLineUp`, `sbLineDown`, `sbLineLeft`, and `sbLineRight` scroll actions. You can either statically tell it what the delta is for a line motion, or have it send you a message asking you to dynamically compute a new line offset.

A scrollwin can display scrollbars around your window. It can even display a scrollbar only when your window is not entirely visible in that dimension. It responds to the scrollbar request messages `msgScrollbarProvideVertInfo` and `msgScrollbarProvideHorizInfo` by setting the `viewLength` to the size of the inner window to which your window is clipped, and the `docLength` to the size of your window.
Table 40-2 summarizes the messages defined by clsScrollwin.

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>Creates a scrollwin window.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>Initializes the SCROLL_WIN_NEW structure to default values.</td>
</tr>
<tr>
<td>msgScrollWinGetStyle</td>
<td>Returns the current style values.</td>
</tr>
<tr>
<td>msgScrollWinSetStyle</td>
<td>Sets the style values.</td>
</tr>
<tr>
<td>msgScrollWinGetMetrics</td>
<td>Returns the metrics.</td>
</tr>
<tr>
<td>msgScrollWinSetMetrics</td>
<td>Sets the metrics.</td>
</tr>
<tr>
<td>msgScrollWinGetClientWin</td>
<td>Returns the current clientWin.</td>
</tr>
<tr>
<td>msgScrollWinShowClientWin</td>
<td>Sets the current clientWin.</td>
</tr>
<tr>
<td>msgScrollWinAddClientWin</td>
<td>Adds another clientWin.</td>
</tr>
<tr>
<td>msgScrollWinRemoveClientWin</td>
<td>The specified client window is extracted from the scrollWin.</td>
</tr>
<tr>
<td>msgScrollWinGetVertScrollbar</td>
<td>Returns the vertical scrollbar.</td>
</tr>
<tr>
<td>msgScrollWinGetHorizScrollbar</td>
<td>Returns the horizontal scrollbar.</td>
</tr>
<tr>
<td>msgScrollWinGetInnerWin</td>
<td>Returns the inner window of the scrollWin.</td>
</tr>
<tr>
<td>msgScrollWinProvideDelta</td>
<td>Self-sent when style.getDelta is set to true.</td>
</tr>
<tr>
<td>msgScrollWinProvideSize</td>
<td>Self-sent when style.getSize is set to true.</td>
</tr>
<tr>
<td>msgScrollWinCheckScrollbars</td>
<td>Determines whether the on/off state of either scrollbar needs to change.</td>
</tr>
</tbody>
</table>

Creating a Scrollwin

Often, you use a scrollwin with a frame, so you make the scrollwin the frame's client window.

clsScrollWin is a kind of layout window although it doesn't inherit from clsCustomLayout. You pass clsScrollWin your window (scrollWin.clientWin) in msgNew. This is the window on whose behalf it will manage scrolling. You also need to give it the line offset information mentioned above. Either you supply a colDelta and rowDelta (for the horizontal and vertical line scrolls), or you supply a client. The getDelta style flag indicates which method you want to use. You need only set getDelta if you want to normalize line-by-line scrolling, or the line size. For example, in an icon browser, some lines of icons may be much taller than other lines.

The other style flags for a scrolling window are summarized in Table 40-3.
Table 40-3  SCROLLWIN_STYLE Styles

<table>
<thead>
<tr>
<th>Styles/Style Flags</th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDelta</td>
<td>Specifies style of scroll method to use. Use if you want to normalize line-by-line</td>
</tr>
<tr>
<td></td>
<td>scrolling or line size.</td>
</tr>
<tr>
<td>colDelta</td>
<td>Horizontal line scrolling.</td>
</tr>
<tr>
<td>rowDelta</td>
<td>Vertical line scrolling.</td>
</tr>
<tr>
<td>horizScrollbar,</td>
<td>Specifies style of client for the scrollbar.</td>
</tr>
<tr>
<td>verticalScrollbar</td>
<td></td>
</tr>
<tr>
<td>autoVertScrollbar,</td>
<td>Overrides on/off flag; displays the scrollbar if the entire window dimension is not</td>
</tr>
<tr>
<td>autoHorizScrollbar</td>
<td>visible.</td>
</tr>
<tr>
<td>expandChildHeight,</td>
<td>Instructs scrollwin to expand child window to fit the inner window.</td>
</tr>
<tr>
<td>expandChildWidth</td>
<td></td>
</tr>
<tr>
<td>contractChildHeight,</td>
<td>Instructs scrollwin to make client window smaller when the scrollwin gets smaller.</td>
</tr>
<tr>
<td>contractChildWidth</td>
<td></td>
</tr>
<tr>
<td>forward</td>
<td>Determines what messages to forward to the client.</td>
</tr>
<tr>
<td>swForwardNone</td>
<td>Forward nothing.</td>
</tr>
<tr>
<td>swForwardGesture</td>
<td>Forward msgGWinGesture.</td>
</tr>
<tr>
<td>swForwardXlist</td>
<td>Forward msgGWinXlist (can be combined with swForwardGesture).</td>
</tr>
<tr>
<td>xAlignment,</td>
<td>Determines how the client window should be positioned if it is smaller.</td>
</tr>
<tr>
<td>yAlignment</td>
<td></td>
</tr>
<tr>
<td>swAlignLeft/swAlignTop</td>
<td>Client window position combination.</td>
</tr>
<tr>
<td>swAlignCenter</td>
<td>Client window position.</td>
</tr>
<tr>
<td>swAlignRight/swAlignBottom</td>
<td>Client window position combination.</td>
</tr>
<tr>
<td>swAlignSelf</td>
<td>Client window will align itself.</td>
</tr>
<tr>
<td>vertClient,</td>
<td>Specifies style of client for the scrollbar.</td>
</tr>
<tr>
<td>horizClient</td>
<td></td>
</tr>
</tbody>
</table>

The initial offset of a scrollwin is that it is aligned according to alignment. By default, it starts out looking onto the top left of the client window, not the usual lower left origin, since the top left is where the minima of the scrollbars are. To scroll the client window elsewhere programmatically, use msgWinDelta to move it.

**Scrollwin Windows**

A scrollwin is a border window with up to three child windows: the vertical and horizontal scrollbar windows, and an inner window. clsScrollWin inserts your document window to be scrolled as the child of this inner window.

You can retrieve the scrollbar windows and the inner windows using msgScrollWinGetVertScrollbar, msgScrollWinGetHorizScrollbar, and msgScrollWinGetInnerWin. Normally, you set the document window that the scrollwin is positioning at msgNew time, but you can also set and get it using msgScrollWinSetClientWin and msgScrollWinGetClientWin.
Repaint

The scrollwin itself is tiled by the two scrollbars and the inner window. The only area it draws is the square notch at the lower right corner between the two scrollbars. It usually draws this in white; however, clsScrollWin inherits from clsBorder, so you can change the background ink if you want.

The inner window has no border and is normally filled by the client window, so it does no repainting and relies on the client window painting in it. However, if the client window is smaller than the size of the inner window, part of it will be exposed. By default, it is painted white. The inner window is also a border window, so you can change its style after retrieving it with msgScrollWinGetInnerWin.

Although the window in a scrollwin will be clipped to the inner window's boundaries, it's potentially faster to only paint the area that is damaged (this is passed back when you send msgWinBeginRepaint to self before starting to repaint).

Layout

A scrollwin lays out its children in response to msgWinLayoutSelf (unless the scrollwin is shrink-wrapped in either dimension). If the scrollwin is not shrink-wrapped, you can resize it to any size and it will display as much of its inner window as will fit. Thus, scrollwins are useful when you have a limited amount of screen space available for a large or unbounded window, such as a toolkit table with dozens of labels. In fact, since the user can change the system font to be any size, most windows will not be entirely visible at very large font sizes.

If autoVertScrollbar or autoHorizScrollbar are set, clsScrollWin compares the size of your window with the size of the scrollwin. During layout, if the client window would be entirely visible without scrollbars, it takes down the scrollbars. If either is set, then the scrollwin ignores values you give for the corresponding vertScrollbar or horizScrollbar field—you give clsScrollWin responsibility to figure out when to display the scrollbar(s).

You can find out whether the scrollwin is displaying the scrollbar by sending msgScrollWinGetStyle and checking the vertScrollbar or horizScrollbar field. You can also send msgScrollWinCheckScrollbars to see whether the on/off state of either scrollbar needs to change. This takes a pointer to a Boolean. If it passes back true, then you can cause the scrollbars to reappear or disappear as necessary by:

1. Sending msgWinSetLayoutDirty to dirty the layout of the scrollwin.
2. Sending msgWinLayout to get the scrollwin to lay itself out again.

clsScrollWin obeys the window shrink-wrap style bits (wsShrinkWrapWidth and wsShrinkWrapHeight). If set, the scrollwin will size itself around the child window. If the scrollwin is shrink-wrapping, then unless the scrollwin's parent forces it to some size, the scrollwin won't need scrollbars.
Positioning the Client Window

clsScrollWin will not allow the left edge of the child (your window) to extend past the right edge of the scrollwin, nor let the bottom edge of the child go past the top edge of the scrollwin. In other words, it keeps your window in view.

If the client window is smaller than the inner window, clsScrollWin aligns the client window in the inner window according to the settings of xAlignment and yAlignment in SCROLL_WIN_STYLE. If the alignment is swAlignSelf, then it’s up to you to align the child window using msgWinDelta.

If you set expandChildWidth or expandChildHeight in SCROLL_WIN_STYLE, the scrollwin will expand the client window in that direction up to the width or height of its inner window. This makes the alignment settings irrelevant, since the client window will never be smaller than the inner window.

If you set contractChildWidth or contractChildHeight in SCROLL_WIN_STYLE, the scrollwin will shrink the client window in that direction to be the width of its inner window. The scrollbar for that direction then becomes redundant, which is useful when you want a client window to only scroll in one direction.

Notification

For the horizontal and vertical scrolling motions, you can specify what the client of each scrollbar is. The scrollbar sends its standard msgScrollbarProvideVertInfo, msgScrollbarProvideHorizInfo, msgScrollbarVertScroll, and msgScrollbarHorizScroll to that client. The default is scrollwin is the notification client (swClientScrollWin), so scrollwin figures out how to scroll the window using the relationship of the inner window to the scrolled window. When a scrollwin receives scroll notifications, it uses msgWinDelta to move the client window (assuming the client for that direction is swClientScrollWin). This preserves the pixels of your window visible on the screen, thereby minimizing the region that your window needs to repaint.

If the client style is swClientWin, the scrolling is entirely the responsibility of the client window. For example, consider a text view that can’t use a scrollwin to do vertical scrolling, since the text view would need to be a window the height of the entire document. Therefore, the text view itself handles vertical scrolling.

However, documents are generally not very wide, so the text view can make itself as wide as the document and let clsScrollWin scroll it horizontally.

There is an enhancement to this since the scrollwin doesn’t know how big a line is. You can either specify a line or column difference when you create the scrollwin, or you can direct the scrollwin to query its client (or the scrolled window, if this is objNull). However, if getDelta is true, a scrollwin will send msgScrollWinProvideDelta to its client whenever it needs to perform a line scroll. This message takes a SCROLL_WIN_DELTA structure as its message arguments. Many of the fields are the same as in a SCROLLBAR_SCROLL action structure. In addition, there is a viewRect structure which gives (in your window’s coordinates) the position and size of the scrollwin—the part of your window that the porthole
looks onto. You need to figure out where the new origin of viewRect should be and pass this back in the origin field.

**Multiple Windows in a Scrollwin**

You can actually display different client windows in a scrollwin, one at a time. Each of the windows is a child of the scrollwin’s inner window, but only one has its wsVisible window style flag set at any time. The effect is like a deck of cards, with only one on top. To add a new window to a scrollwin, send the scrollwin msgScrollWinAddClientWin, passing it the UID of the window you want inserted as a child of the scrollwin’s inner window. The new child window’s wsVisible flag is turned off, so it is invisible.

To switch the scrollwin to displaying a new client window, send it msgScrollWinShowClientWin, passing it the UID of the window you want shown. If this window isn’t already a child of the scrollwin, the scrollwin will send itself msgScrollWinAddClientWin to add it.

Send msgScrollWinRemoveClientWin to remove one of the client windows. If you remove all the client windows, the inner window will be visible.

msgScrollWinGetClientWin passes back the current client window (the one last shown with msgScrollWinShowClientWin). This takes a pointer to a WIN. You can locate other client windows using msgWinFindTag or msgWinEnum.

**Toolkit Tables**

Because it inherits from clsBorder, clsTableLayout knows how to be the child of a scrollwin; it responds to msgScrollWinProvideDelta by normalizing to align with the rows and columns in the table.
Chapter 41 / List Boxes

List boxes provide a scrollable list of windows. The list can be of arbitrary length (up to `maxS16`), useful when presenting a set of toggles or choices where the total number of items is either very large or unknown.

`clsListBox` inherits from `clsScrollWin`; `clsScrollWin` provides the scrolling ability. When you use a `clsScrollWin` directly, you must supply the client window that is scrolled by `clsScrollWin`. `clsListBox` creates and maintains a client window for you.

There are two standard subclasses of `clsListBox` kinds of list boxes: string list boxes, which display a list of strings that the user can select, and font list boxes, which display a list of the installed fonts. Both are implemented by the descendant classes `clsStringListBox` and `clsFontListBox`. Table 41-1 summarizes the messages `clsListBox` defines.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_LIST_BOX_NEW</td>
<td>Initializes the LIST_BOX_NEW structure to default values.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_LIST_BOX_NEW</td>
<td>Creates a list box (initially empty).</td>
</tr>
<tr>
<td>msgListBoxGetMetrics</td>
<td>P_LIST_BOX_METRICS</td>
<td>Passes back the metrics for a list box.</td>
</tr>
<tr>
<td>msgListBoxSetMetrics</td>
<td>P_LIST_BOX_METRICS</td>
<td>Sets the metrics for a list box.</td>
</tr>
<tr>
<td>msgListBoxAppendEntry</td>
<td>P_LIST_BOX_ENTRY</td>
<td>Appends an entry to the list box after the specified position.</td>
</tr>
<tr>
<td>msgListBoxInsertEntry</td>
<td>P_LIST_BOX_ENTRY</td>
<td>Insert an entry to the list box before the specified position.</td>
</tr>
<tr>
<td>msgListBoxRemoveEntry</td>
<td>U16</td>
<td>Removes an entry from the list box.</td>
</tr>
<tr>
<td>msgListBoxGetEntry</td>
<td>P_LIST_BOX_ENTRY</td>
<td>Gets an entry in a listBox by position.</td>
</tr>
<tr>
<td>msgListBoxSetEntry</td>
<td>P_LIST_BOX_ENTRY</td>
<td>Sets an entry's information.</td>
</tr>
<tr>
<td>msgListBoxFindEntry</td>
<td>P_LIST_BOX_ENTRY</td>
<td>Finds the position of the given entry window/data.</td>
</tr>
<tr>
<td>msgListBoxEnum</td>
<td>P_LIST_BOX_ENUM</td>
<td>Enumerates the entries of a listBox according to the given flags.</td>
</tr>
<tr>
<td>msgListBoxEntryIsVisible</td>
<td>P_LIST_BOX_ENTRY</td>
<td>Passes back the visibility of an entry in a listBox.</td>
</tr>
<tr>
<td>msgListBoxXYToPosition</td>
<td>P_LIST_BOX_POSITION_XY</td>
<td>Gets the position for a given list box window coordinate.</td>
</tr>
<tr>
<td>msgListBoxMakeEntryVisible</td>
<td>P_LIST_BOX_ENTRY</td>
<td>Makes the specified entry visible.</td>
</tr>
</tbody>
</table>

Table 41-1 clsListBox Messages

Class Messages

Attribute Messages

Entry Manipulation Messages

continued
Creating a List Box

You send msgNew to clsListBox to create a new list box. This takes a pointer to a LIST_BOX_NEW structure for its message arguments. The message arguments include:

- The list box style, described below.
- The client to which the list box sends messages (listBox.client).
- The number of entries in the list box (listBox.nEntries).
- The number of entries that should be visible in the list box (listBox.nEntriesToView).

List boxes are not layout windows, so you don't have any control over the size or alignment of the child windows in the list box.

Note that you can't specify the contents of the list box when you create it. The list box asks you to provide entries later, as you will see.

Style Fields

The fields in LIST_BOX_STYLE include:

filing  a filing style. The possible values are IbFileMin, IbFileEntryInfo, and IbFileAll. These are explained in "Filing," later in this chapter.

List Box Contents

The items displayed in a list box must be windows. However, the total number of entries in a list box may be very large, and at any point only a fraction will be displayed on-screen. Having a window for each entry could consume substantial memory. Hence, clsListBox supports a protocol for creating and freeing the windows for its items as it needs them on-screen.

This is very different from toolkit tables. Both have child windows. However, in a toolkit table, you can statically specify every “item” in the table, and the toolkit table maintains every item as an inserted child window. In a list box, you have to dynamically supply the child windows for the list box, and the list box may not maintain the child windows. The “items” in a list box are called list box entries.
This protocol process saves memory. If a list box has many entries, at any point only a few will be visible in the scrollwin's inner window. So the list box only inserts child windows for those entries that are visible. It asks you (or a descendant class) to provide the windows. Since the list box doesn't need a window for every list box entry, you can ask it to destroy windows that are no longer visible.

The next sections explain the protocol of providing list box entries in more detail. If you're using clsStringListBox or clsFontListBox, those classes handle the protocol internally.

### List Box Entries

The entries in a list box are indexed by position, starting at zero. There can be up to maxS16 entries in a list box. clsListBox maintains information about each entry, whether or not it has a child window for that entry. If clsListBox needs to display the entry at some position and does not have a window for it, it sends itself, and then its client, msgListBoxProvideEntry. You can get information about the entry at a position by sending the list box msgListBoxGetEntry, and set information about an entry using msgListBoxSetEntry. All these messages take a pointer to a LIST_BOX_ENTRY structure. This includes the following arguments:

- **listBox** the UID of the list box.
- **position** the entry position.
- **win** the entry's window, if it has one.
- **freeEntry** the free mode of the entry, indicating what clsListBox should do with it when it's no longer visible.
- **state** the state of the entry.
- **data** client data for the entry.
- **arg** a message-specific argument.

Free mode and state are explained in more detail below.

You can also enumerate the entries in a list box with msgListBoxEnum; this retrieves the LIST_BOX_ENTRY information for each entry.

### Supplying Entries

When you create a list box, it has no entries. If you don't do anything special when the list box goes on-screen, clsListBox will self-send msgListBoxProvideEntry. If no descendant responds to this message (clsFontListBox and clsStringListBox both do), clsListBox forwards it to its client. In the LIST_BOX_ENTRY structure, it fills in listBox, and sets position to the desired window position.

You should respond by passing in the UID of the window to use for client. You can also fill in state and data if you're using those to keep track of entries.

When the user scrolls the list box back to a previous position, clsListBox checks to see if it still has a window for that position. If it does, it inserts the window and does not send out a message. If the window is null, clsListBox sends out
msgListBoxProvideEntry as above. The rest of the LIST_BOX_ENTRY fields will be the same as before, so you need only fill in the window.

Scrolling

The same protocol occurs when the user scrolls the list box to a new position, thereby exposing new entries.

If the user scrolls down, clsListBox will ask for an entry past nEntries. You are free to respond with stsFailed if the number of entries is indeed fixed at nEntries. However, if you do supply a window, clsListBox will insert it and update nEntries. This way, you can let the user dynamically extend the list box size by scrolling downwards.

Free Mode

clsListBox will remember the UID of the window you supply. However, if the list box has lots of entries, you want to conserve memory by destroying the windows of off-screen entries.

In LIST_BOX_ENTRY, you can set freeEntry to the desired freeing mode for an entry. The possible values are a set of flags (LIST_BOX_DATA_FREE_MODE):

- lbFreeDataNotVisible free the entry when it’s no longer visible.
- lbFreeDataWhenDestroyed free the entry when the list box itself is destroyed.
- lbFreeDataByMessage free the entry by sending a message.

The default freeEntry mode, lbFreeDataDefault, is lbFreeDataNotVisible | lbFreeDataWhenDestroyed.

lbFreeDataByMessage

By default, clsListBox sends msgDestroy to an entry's window when it is time for it to be destroyed.

If the data field of an entry is a pointer to other storage, then you need to have the list box send you a message, instead of destroying the window directly, so you can free that storage. If you set lbFreeDataByMessage, clsListBox self-sends (and then its client, if no descendant intercepts the message) msgListBoxDestroyEntry. You can then destroy the win and free storage associated with data. Afterwards, clsListBox sets win to objNull.

You may want to receive this free entry message to implement other management schemes for entries. For example, you could have a scheme where you maintain a fixed-size pool of windows for your list box. When the user scrolls the list box, you need to set the window of the entry that's no longer visible to objNull, so that you can provide that window for the new entry that has scrolled into view. (clsListBox frees the child windows scrolled off-screen before sending msgListBoxProvideEntry asking for the newly visible entries.)
Pre-Loading a List Box

To speed up the initial drawing of a list box, you can pre-load the list box entries of the entries that will be visible. For example, if at creation you set listBox.nEntries as 30 and listBox.nEntriesToView as 5, then when you use msgListBoxSetEntry to specify the windows for the first ten entries in the list box, clsListBox will not have to call back to display the initial contents of the list box, nor will it need to when the user scrolls down one “page.”

Modifying

You can send msgListBoxSetEntry to change the state of an entry. You don’t have to supply a window; if the window you give is objNull, clsListBox will ask you to provide one with msgListBoxProvideEntry, as usual.

However, if that position already has a window, and you specify a different window, then clsListBox will extract the old window but will not free it.

Inserting and Removing Entries

Sending msgListBoxSetEntry and responding to msgListBoxProvideEntry fill in an existing entry but don’t change the number of entries in the list box. You can add a new entry to a list box by inserting it at or after a position using msgListBoxInsertEntry or msgListBoxAppendEntry. These also take a pointer to a LIST_BOX_ENTRY for their message arguments.

To remove an entry, reducing the number of entries in the list box, send msgListBoxRemoveEntry. Again, it takes LIST_BOX_ENTRY.

State

In LIST_BOX_ENTRY you can set state to different values. The possible values are a set of flags (LIST_BOX_ENTRY_STATE):

- IB_Selected
- IB_Open
- IB_Busy

The default state, IBStateDefault, is 0, meaning that none of these flags are set.

clsListBox does not interpret state at all; state is a feature for use by subclasses of clsListBox. For example, clsStringListBox, a subclass that displays a list of strings, uses the IB_Selected flag to mark string entries the user has selected.

Notification

List boxes do not (currently) send client notifications. They are passive by design. The user interacts with the list box, and later your application asks the list box which entries are selected.

Like msgListBoxProvideEntry, all of the messages in this section are first sent to the list box itself. This is so that subclasses of clsListBox may intercept these
messages and process them as desired. If these messages reach clsListBox, it forwards them on to the list box's client.

**Gestures**

The list box handles scrolling and other gestures.

In particular, many list boxes allow the user to select multiple items in the list box. The list box does not handle these itself. It passes them to its client. However, the individual entries in the list box themselves may handle certain gestures, such as tap.

Thus, the list box's client receives all non-scrolling gestures that the target window ignored. clsListBox self-sends, then sends its client msgListBoxEntryGesture. Like the other messages, this takes a pointer to a LIST_BOX_ENTRY structure. clsListBox determines the child window over which the user made the gesture, and supplies its list box entry information. When you receive msgListBoxEntryGesture, position is the position of the targeted child window, and win is its UID. The gesture information is in arg.

**Painting**

In msgNewDefaults, clsListBox sets border.style.edge to bsEdgeAll. Consequently, it paints a border around the list box.

Each child window in the list box paints itself as normal.

clsListBox maintains a minimum line height for the list box. It does this by remembering the smallest height of a child window that it encounters. If it does not have a child window for a position, and the descendant or client does not supply one in response to msgListBoxProvideEntry, clsListBox paints an empty “hole” in its place. This region is the minimum height.

**Scrolling**

For line-by-line scrolling, clsListBox uses the minimum line height to figure out how much a line scroll is.

**Entry Visibility**

You can send msgListBoxEntryIsVisible to determine if an entry is currently visible in the list box. Like the other messages, this takes a pointer to a LIST_BOX_ENTRY structure. clsListBox determines whether the entry at the position you specify is visible or not. If the entry is not visible, clsListBox passes back 0L in the arg field, otherwise it passes back a non-zero value. If you set position to maxU16, clsListBox checks the visibility of the window you specify in win.
You can send `msgListBoxMakeEntryVisible` to make a specified entry visible. If
the item at `position` is not visible, `clsListBox` scrolls the list so that item is visible
in the view (you don't have to send `msgWinUpdate` to repaint the list box).
Specifying a `position` of `maxU16` causes the list box to find the given `win`.

### Layout

The list box computes its desired height by multiplying its line height, then
multiplying that by `nEntriesToView`. If it has not yet computed its line height, it
will ask its client to provide one child window (`msgListBoxProvideEntry`) solely
to compute its size. Thus, the list box client may receive `msgListBoxProvideEntry`
before the list box is on-screen.

If the list box has `wsShrinkWrapWidth` set, its width is the width of its widest
child (plus the border and scrollbar region).

### Filing

The setting of the list box `filing` style determines how much state the list box files.
You can set the list box to:

- `lbFileMin` file the minimum data necessary.
- `lbFileEntryInfo` file the entry information for each entry but not the
  windows themselves.
- `lbFileAll` file all the entry information and file the windows of those entries
  that have a window.

After restoring a list box, it will still send `msgListBoxProvideEntry` to its client,
unless `filing` was set to `lbFileAll` and every entry had a valid window UID at the
time of filing.

### Miscellaneous Messages

`msgListBoxXYToPosition` is a utility message that converts an x-y location in the
local window coordinates of the list box to the position of the window underneath
those coordinates. This takes a pointer to a `LIST_BOX_POSITION_XY` structure. In
this you set the location of the coordinates (`place`, an `XY32` structure); `clsListBox`
passes back the `position` of the list box entry at that location. Remember that
`msgListBoxEntryGesture` provides the same information.

`msgListBoxEnum` enumerates list box entries. You can restrict the enumeration to
just those list box entries with a certain state, for example, `lbSelected`. In this way,
a client or descendant can find out which entries in a list box are "selected."
(Remember that the list box does not interpret any gestures as indicating selection;
this would be up to a descendant or client.)
Toolkit Tables versus List Boxes

You can achieve much the same functionality as a list box by embedding a toolkit table in a scrollwin. You can set up the latter in advance and you don’t have to create the windows in response to a callback. Its performance will probably be better than the list box. The disadvantage is that you consume memory for every item in the table, whether it is visible or not.

String List Boxes

clsStringListBox is a descendant of clsListBox. It only supports putting strings in a list box. Consequently, it has a simpler API. Unlike clsListBox, it interprets user gestures on the contents of the list box. You can set up a string list box to behave as a scrolling choice (exclusive or non-exclusive) or as a set of toggles.

Table 41-2 summarizes messages defined by clsStringListBox.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_STRLB_NEW</td>
<td>Initializes the STRLB_NEW structure to default values.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_STRLB_NEW</td>
<td>Creates a string listbox window.</td>
</tr>
<tr>
<td>msgStrListBoxGetStyle</td>
<td>P_STRLB_STYLE</td>
<td>Passes back the style of the receiver.</td>
</tr>
<tr>
<td>msgStrListBoxGetDirty</td>
<td>P_BOOLEAN</td>
<td>Passes back true if the listbox has been altered since dirty was set false.</td>
</tr>
<tr>
<td>msgStrListBoxSetDirty</td>
<td>BOOLEAN</td>
<td>Sets the dirty bit of a string listbox.</td>
</tr>
<tr>
<td>msgStrListBoxGetValue</td>
<td>P_U32</td>
<td>Passes back the value of a string listbox.</td>
</tr>
<tr>
<td>msgStrListBoxSetValue</td>
<td>U32</td>
<td>Sets the value of a string listbox whose role is one of slbRoleChoice.</td>
</tr>
<tr>
<td>msgStrListBoxProvideString</td>
<td>P_STRLB_PROVIDE</td>
<td>This message requests the client (or subclass) to provide a string.</td>
</tr>
<tr>
<td>msgStrListBoxNotify</td>
<td>U32</td>
<td>This message is sent out whenever the value of a string list box changes.</td>
</tr>
</tbody>
</table>

Creating a String List Box

You send msgNew to clsStringListBox to create a new string list box. This takes a pointer to a STRLB_NEW structure for its message arguments. The message arguments are the same as those for clsListBox, with the addition of a STRLB_NEW_ONLY structure. The additional fields include:

- `stringListBox.value`  an initial value (if the string list box is acting like a choice).
- `stringListBox.style` several style fields.
The fields in STRLB_STYLE include:

<table>
<thead>
<tr>
<th>Fields/Field Values</th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>role</td>
<td>The overall behavior of the string list box.</td>
</tr>
<tr>
<td>slbRoleToggle</td>
<td>Act like a toggle table so that zero, one, or more table entries can be activated.</td>
</tr>
<tr>
<td>slbRoleChoice</td>
<td>Act like a choice which has a value, so that one entry is always activated.</td>
</tr>
<tr>
<td>slbRoleChoice01</td>
<td>Act like a choice which can have no value, so that either zero or one entities can be activated.</td>
</tr>
<tr>
<td>look</td>
<td>The visual appearance of the string list box entries when activated.</td>
</tr>
<tr>
<td>slbLookInvert</td>
<td>Invert when activated.</td>
</tr>
<tr>
<td>slbLookDecorate</td>
<td>Decorate when activated.</td>
</tr>
<tr>
<td>dirty</td>
<td>Determines whether the string list box is dirty. The same notion that controls maintain.</td>
</tr>
</tbody>
</table>

### Providing Entries

Just like a standard list box, a string list box has no entries when you create it. When clsListBox self-sends msgListBoxProvideEntry, clsStringListBox responds by sending first self, and then its client, msgStrListBoxProvideString. In response, you provide a string. clsStringListBox creates a button out of the string. The look and role style settings of the string list box determine the appearance of the button.

msgStrListBoxProvideString takes a pointer to a STRLB_PROVIDE structure. This is similar to a LIST_BOX_ENTRY structure, and some of the fields are the same. The fields include:

- strListBox the UID of the string list box.
- position the position of the requested entry.
- pString a 256-byte buffer for the string.
- data client data for the entry.

Ordinarily, you should copy the appropriate string for position into pString. If you have a static string definition, you can pass a pointer to it in pString, but it had better not change. You can supply a U32 of arbitrary data in data; clsStringListBox copies this into clsListBox's LIST_BOX_ENTRY for the entry.

clsStringListBox does not remember any information about the entry beyond the LIST_BOX_ENTRY information maintained by clsListBox. clsListBox needs to display a window for an entry, so it self-sends msgListBoxProvideEntry. clsStringListBox sends out msgStrListBoxProvideString and creates a button from that string which it passes back to clsListBox.

clsStringListBox does not do fancy tricks to improve performance, such as caching empty button windows, reusing button windows, or storing strings in a heap.

If you subclass clsStringListBox, you can either respond to msgStrListBoxProvideString or msgListBoxProvideEntry (or neither).
Notification

Unlike clsListBox, clsStringListBox does respond to gestures made over its entries. The only gesture it responds to is a single tap. clsStringListBox turns on the string the user tapped, and depending on its role, it turns off any other on entries.

The string list box maintains the "on" state of each string in the LIST_BOX_ENTRY information of its ancestor clsListBox.

A string list box doesn't send previewing or other messages to its client. This is true even if you set its role to be slbRoleChoice01 or slbRoleChoice1—these make it respond to gestures like a choice, but don't give it an API like a choice. If you want, you can change the buttons that it creates to send button messages to its client.

Value

The value of a slbRoleChoice01 or slbRoleChoice1 string list box is the data of the on string. You can get the value using msgStrListBoxGetValue. You provide each entry's data in response to msgStrListBoxProvideEntry and can maintain it by modifying the LIST_BOX_ENTRY for that position. If the string list box role is slbRoleChoice01 and no entry is on, this returns stsStrListBoxNoValue.

If the role of the string list box is slbRoleToggles, msgStrListBoxGetValue returns stsFailed. To find out those entries that the user has selected, send msgListBoxEnum to the string list box, setting flags to IbSelected.

If the role is slbRoleToggles, sending msgStrListBoxSetValue will return stsFailed. If the role is slbRoleChoice01 and you set the value to maxU32, this will deselect every entry.

Control Dirty

A string list box maintains a dirty status in STRLB_STYLE and displays itself differently depending on how it is set. This is similar to the dirty field of a control, which indicates whether or not the control has been altered. You can send msgStrListBoxGetDirty or msgStrListBoxGetStyle to a string list box to find out if the user has altered it. As with most controls, it is up to you to reset the control dirty state of the list box.

Destruction

clsStringListBox destroys the button window it creates for a string when it receives msgListBoxDestroyEntry. It doesn't have to free the string since it only passed it to clsButton to create the entry's window.
**Painting**

clsStringListBox doesn’t respond to repaint messages. It creates a left-aligned, edgeless button of the appropriate kind in response to msgListBoxProvideEntry, and the button paints itself.

**Font List Boxes**

clsFontListBox is a descendant of clsStringListBox. The strings it supplies are the names of the currently installed fonts. You don’t have to respond to any messages to use it.

**Creating a Font List Box**

You send msgNew to clsFontListBox to create a new font list box. This takes a pointer to a FONTLB_NEW structure for its message arguments. The message arguments are the same as those for clsStringListBox, with the addition of a FONTLB_NEW_ONLY structure. Currently, the only additional field is:

- style fields (fontListBox.style).

Currently, the only field in STRLB_STYLE is:

- prune the pruning control for the font list display. This is a set of flags. It is defined to be a U16 in FONTLB.H, but is actually a FIM_PRUNE_CONTROL structure defined by the installed font manager in FONTMGR.H. This lets you reduce the number of fonts displayed. You can set prune to fimNoPruning to get a full list, or OR in the following flags to reduce the font list:

  - fimPruneDupFamilies remove fonts in the same family, such as Swiss Bold, Swiss Italic.
  - fimPruneSymbolFonts remove symbol fonts.

At msgNew time, clsFontListBox gets the list of installed fonts from the font install manager. It stores their short ID’s in an array and sets the number of entries in the list box (listBox.nEntries) to the number of ID’s. When asked to provide a string, it converts the font ID into a name.

Note that clsFontListBox does not regenerate the list of fonts when the set of installed fonts changes.

**Notification**

You do *not* need to respond to any message from clsFontListBox.
Chapter 42 / Fields

clsField is a subclass of clsLabel representing an editable text field that the user can fill in by handwriting with the pen. Since it inherits from clsControl and ultimately from clsWin, a field has all the usual properties of windows and controls. In particular, it has a client object that receives notification messages for important user actions and events affecting the field.

Field Style Flags

The actions available to the user for entering or editing the contents of a field depend on the style flags you specify when creating it:

- **In-line fields** provide full handwriting and gesture recognition, allowing the user to write with the pen directly into the field itself. Alternatively, the user can draw a circle gesture on the field to invoke a pop-up insertion pad with more specialized editing capabilities.

- **Overwrite fields** have combed segments, with each character displayed in a separate, outlined box of its own. Writing into a box that already contains a character replaces that character with a new one.

- **Pop-up fields** perform no handwriting or gesture recognition of any kind. Any pen stroke in such a field invokes an insertion pad for editing the field’s contents.
Notice that in-line and pop-up fields allow the user the option of popping up an insertion pad for editing the field’s contents. An insertion pad has an area for writing, and button controls for accepting, clearing, and canceling the contents of the field.

**Messages**

Table 42-1 summarizes `clsField` messages. All of the object messages listed here are defined in the header file `FIELD.H` and are discussed in the relevant sections later in this chapter.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_FIELD_NEW</td>
<td>Creates and initializes a new instance of <code>clsField</code>.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_FIELD_NEW</td>
<td>Initializes the FIELD_NEW structure to default values.</td>
</tr>
<tr>
<td>msgFieldGetStyle</td>
<td>P_FIELD_STYLE</td>
<td>Passes back the style value held by the field.</td>
</tr>
<tr>
<td>msgFieldSetStyle</td>
<td>P_FIELD_STYLE</td>
<td>Sets the style of the field.</td>
</tr>
<tr>
<td>msgFieldGetXlate</td>
<td>P_UNKNOWN</td>
<td>Passes back the translator information for the field.</td>
</tr>
<tr>
<td>msgFieldSetXlate</td>
<td>P_UNKNOWN</td>
<td>Specifies the translator information for the field.</td>
</tr>
<tr>
<td>msgFieldGetMaxLen</td>
<td>P_U16</td>
<td>Returns the maximum length allowed for input in the field.</td>
</tr>
<tr>
<td>msgFieldSetMaxLen</td>
<td>P_U16</td>
<td>Sets the maximum length for input in the field.</td>
</tr>
<tr>
<td>msgFieldGetCursorPosition</td>
<td>P_U16</td>
<td>Sets the position of the keyboard insertion point in the field.</td>
</tr>
<tr>
<td>msgFieldGetCursorPosition</td>
<td>P_U16</td>
<td>Returns the position of the keyboard insertion point in the field.</td>
</tr>
<tr>
<td>msgFieldCreatePopUp</td>
<td>P_FIELD_CREATE_POPUP</td>
<td>Creates and returns the insertion pad when the pop-up is invoked.</td>
</tr>
<tr>
<td>msgFieldCreateTranslator</td>
<td>P_OBJECT</td>
<td>Self-sent to create a translator. Returns the translator.</td>
</tr>
<tr>
<td>msgFieldActivate</td>
<td>void</td>
<td>Forces activation of the field.</td>
</tr>
<tr>
<td>msgFieldDeactivate</td>
<td>void</td>
<td>Forces deactivation of the field.</td>
</tr>
<tr>
<td>msgFieldActivatePopUp</td>
<td>P_FIELD_ACTIVATE_POPUP</td>
<td>Sent to cause an insertion pad to be brought up for the field.</td>
</tr>
<tr>
<td>msgFieldAcceptPopUp</td>
<td>void</td>
<td>Causes the insertion pad to be accepted. Sent when the user collapses the insertion pad by hitting the OK button or accepts the P.</td>
</tr>
<tr>
<td>msgFieldCancelPopUp</td>
<td>void</td>
<td>Cancels the edit in the pop-up insertion pad. The previous value is unchanged. Sent when the user hits the cancel button or cancels the P.</td>
</tr>
<tr>
<td>msgFieldKeyboardActivate</td>
<td>void</td>
<td>Activates field for keyboard use.</td>
</tr>
</tbody>
</table>

*continued*
Creating a Field

The FIELD_NEW_ONLY structure, which holds the class-specific arguments to msgNew for creating a new text field, is defined as follows:

```c
typedef struct FIELD_NEW_ONLY {
    FIELD_STYLE style; // field style, see below
    FIELD_XLATE xlate; // translator or template
                 // for handwriting translation
    U16        maxLen; // maximum text length in characters
    U16        reserved; // reserved for future use, must be 0
} FIELD_NEW_ONLY, FAR *P_FIELD_NEW_ONLY;
```

The form and meaning of the style and xlate arguments are discussed below.

maxLen specifies the maximum number of text characters the field can contain; all text entered in the field will be truncated to this maximum length. The default value set by msgNewDefaults is 64 characters.

Style Flags

FIELD_STYLE contains flag settings to control the field's appearance and behavior, as summarized in Table 42-2. The remaining flags are all concerned with input processing and validation, and are discussed later in this chapter.
Specific values for some of these style settings are further summarized in Table 42-2. Except for the editType, popUpType, and focusStyle settings, all of these settings are concerned with input processing and validation and are discussed later in this chapter.

### The editType Settings

The value of editType determines the style of text entry and editing the field supports: in-line, overwrite, or pop-up.

### The focusStyle Setting

The focusStyle setting determines whether the field takes control of both the global selection and input focus when activated, the input focus only, or neither. Ordinarily, a field will want to take both, but it may occasionally be more convenient to take only the input focus, leaving the existing selection unchanged. (You might use this option, for example, to avoid disturbing the current selection when placing a temporary pop-up box on the screen. In fact, the clsField code itself uses it when displaying a field's insertion pad.) The fstNone option is included for completeness, but it's hard to imagine a realistic situation in which it would be useful.

<table>
<thead>
<tr>
<th>Styles/StyleValues</th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>editType</td>
<td>Determines the style of text entry and editing supported by the field.</td>
</tr>
<tr>
<td>fstInLine</td>
<td>Supports direct in-line editing.</td>
</tr>
<tr>
<td>fstOverWrite</td>
<td>Supports overwrite-style editing.</td>
</tr>
<tr>
<td>fstPopUp</td>
<td>Supports pop-up insertion pad only.</td>
</tr>
<tr>
<td>popUpType</td>
<td>For backward compatibility, insertion pad type.</td>
</tr>
<tr>
<td>fstCharBox</td>
<td>Character box.</td>
</tr>
<tr>
<td>fstCharBoxButton</td>
<td>Character box.</td>
</tr>
<tr>
<td>xlate</td>
<td>Whether the field uses a translator object or a translation template.</td>
</tr>
<tr>
<td>true</td>
<td>Use a translation template.</td>
</tr>
<tr>
<td>false</td>
<td>Use a translator object.</td>
</tr>
<tr>
<td>focusStyle</td>
<td>Determines whether field takes control of global selection and input focus.</td>
</tr>
<tr>
<td>fstInput</td>
<td>Take input focus only.</td>
</tr>
<tr>
<td>fstInputSelection</td>
<td>Take both selection and input focus.</td>
</tr>
<tr>
<td>fstInputNone</td>
<td>Take neither selection nor input focus.</td>
</tr>
<tr>
<td>noSpace</td>
<td>Defines the properties of the fields translator: instructs the translator to suppress all space characters during input translation.</td>
</tr>
<tr>
<td>capOutput</td>
<td>Defines properties of the field’s translator: determines capitalization style for entered text.</td>
</tr>
<tr>
<td>fstCapAll</td>
<td>Capitalize all letters.</td>
</tr>
<tr>
<td>fstCapAllWords</td>
<td>Capitalize first letter of each word.</td>
</tr>
<tr>
<td>bsDragAsIs</td>
<td>Capitilize exactly as entered.</td>
</tr>
<tr>
<td>fstCapFirstWord</td>
<td>Capitalize first letter of first word.</td>
</tr>
</tbody>
</table>

continued
<table>
<thead>
<tr>
<th>Styles/StyleValues</th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>upperCase</td>
<td>Forces all input to uppercase from any source. Not limited to pen input (not a translator-only property).</td>
</tr>
<tr>
<td>delayed</td>
<td>Delay translation until explicitly requested.</td>
</tr>
<tr>
<td>clientNotifyModified</td>
<td>Notify client when field's contents are modified.</td>
</tr>
<tr>
<td>clientNotifyReadOnly</td>
<td>Notify client on attempt to modify read-only fields.</td>
</tr>
<tr>
<td>validatePending</td>
<td>Indicates field's contents have been modified and not yet validated.</td>
</tr>
<tr>
<td>clientPreValidate</td>
<td>Notify client before beginning validation.</td>
</tr>
<tr>
<td>clientValidate</td>
<td>Notify client to perform validation.</td>
</tr>
<tr>
<td>clientPostValidate</td>
<td>Notify client after completing validation.</td>
</tr>
<tr>
<td>clientNotifyInvalid</td>
<td>Notify client if validation fails.</td>
</tr>
<tr>
<td>dataMoveable</td>
<td>Determines whether the field window or its string gets moved.</td>
</tr>
<tr>
<td>dataCopyable</td>
<td>Determines whether the field window or its string gets copied.</td>
</tr>
</tbody>
</table>

### Custom Handwriting Translation

Associated with every field is a **translator** for converting the user's input scribbles into meaningful text characters. (Translators are discussed at length in Part 5: *Input and Handwriting Translation.*) When you create a new field, the `xlate` argument in `FIELD_NEW` identifies the type of translator the field is to use. A null value for this argument calls for a standard translator of `clsXWord`. However, if you have more specific knowledge of the kind of input the field will contain, you can improve the accuracy of the handwriting translation by supplying a custom translator of your own.

The `xlate` argument is defined to be of the following union type:

```c
typedef union FIELD_XLATE {
    OBJECT translator;
    P_UNKNOWN pTemplate;
} FIELD_XLATE, FAR *P_FIELD_XLATE;
```

This allows you to supply either an existing translator (`translator`) or a template from which to build one (`pTemplate`), as described in Part 5: *Input and Handwriting Translation*. The `xlateType` flag in the `FIELD_STYLE` structure distinguishes the two options: 0 for the translator itself, 1 for a template.

The `FIELD_STYLE` flags `noSpace` and `capOutput` define properties of the field's translator. The field code uses these values when creating a translator on your behalf, whether by default (`xlate = false`) or from a template you supply (`xlate != false, xlateType = true`). (If you supply your own translator directly by setting `xlate != false` and `xlateType = false`, these flags are ignored. It is then your responsibility to set the corresponding properties of the translator yourself.)
Access to Field Properties

The messages `msgFieldSetStyle` and `msgFieldGetStyle` allow you to access a field's style flags. Both take a pointer to a `FIELD_STYLE` structure and either copy it to the field's own style flags or fill it with their current values. `msgFieldSetStyle` will fail (return `stsFailed`) if you attempt to send it to a field that is currently active on the screen.

`msgFieldSetXlate` and `msgFieldGetXlate` set and retrieve the field's translator information. The argument to both messages is of type `P_UNKNOWN`. If the field's `xlateType` flag is 0, this is taken to be a `P_OBJECT` pointing to the translator object itself; if `xlateType` is 1, it is a `P_XTEMPLATE` pointing to a translator template. `msgFieldSetXlate` accepts a raw (uncompiled) template, while `msgFieldGetXlate` passes it back in compiled form. (Be careful not to send `msgFieldSetXlate` to a field that is currently active or has an insertion pad present on the screen. The results are undefined and probably undesirable.)

`msgFieldSetMaxLen` and `msgFieldGetMaxLen` manipulate the field's maximum text length via an argument of type `P_UI6`. If `maxLen` is changed to less than the number of characters the field currently contains, the text will be truncated to the new length at the next editing operation. To access or change the current contents of the field itself, use the inherited messages `msgLabelGetString` and `msgLabelSetString`.

`msgFieldSetCursorPosition` and `msgFieldGetCursorPosition` set and retrieve the cursor position marking the insertion point for keyboard input. Both messages take an argument of type `P_UI6`. The cursor appears on the screen only when the field is activated for keyboard input with `msgFieldKeyboardActivate`. If no cursor position has been set explicitly, `msgFieldSetCursorPosition` reports it as 0.

Component Creation

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>msgFieldCreatePopUp</code></td>
<td><code>P_FIELD_CREATE_POPUP</code></td>
<td>Creates and returns the insertion pad when the pop-up is invoked.</td>
</tr>
<tr>
<td><code>msgFieldCreateTranslator</code></td>
<td><code>P_OBJECT</code></td>
<td>Self-sent to create a translator. Returns the translator.</td>
</tr>
<tr>
<td><code>msgFieldFieldGetSecondTrans</code></td>
<td><code>P_OBJECT</code></td>
<td>Returns a second translator for editBox IP's.</td>
</tr>
</tbody>
</table>

When the user invokes a field's pop-up insertion pad, the field creates the pad by sending itself `msgFieldCreatePopUp`. Ordinarily, this creates a standard insertion pad of the type called for by the field's `popupType` setting. If necessary, however, you can customize the appearance or behavior of the insertion pad by creating a specialized subclass of `dsField` and reimplementing `msgFieldCreatePopUp` to suit your needs. This message takes an argument of the form:
typedef struct {
    U16    type;
    OBJECT ip;
    U32    reserved;
} FIELD_CREATE_POPUP, FAR * P_FIELD_CREATE_POPUP;

where type is one of the values:

#define fipReplaceAll 0
#define fiplnsert 1
#define fipReplaceRange 2;

fipReplaceAll calls for an insertion pad large enough to hold the field's full maximum text length, as specified by its maxLen property. fiplnsert is used in response to an insertion gesture by the user, and calls for only the number of additional characters that the field could actually accommodate (that is, the maximum text length minus the length of the current contents).

If you write your own method for msgFieldCreatePopUp, it should create an insertion pad of the requested size and pass back its UID in the ip field of the FIELD_CREATE_POPUP structure.

Subclasses redefine the standard method for msgFieldCreateTranslator, which clsField sends itself to create a field's translator object. If the field's xlateType flag is 1, the translator is created from the template supplied by the client program via the FIELD_NEW parameter xlate. If xlateType is 0 and xlate is null, msgFieldCreateTranslator must create a standard clsXWord translator. In either case, the newly created translator is passed back through the method argument, which is of type P_OBJECT.

### Activation and Deactivation

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgFieldActivate</td>
<td>void</td>
<td>Forces activation of the field.</td>
</tr>
<tr>
<td>msgFieldDeactivate</td>
<td>void</td>
<td>Forces deactivation of the field.</td>
</tr>
<tr>
<td>msgFieldActivatePopUp</td>
<td>P_FIELD_ACTIVATE_POPUP</td>
<td>Sent to cause an insertion pad to be brought up for the field.</td>
</tr>
<tr>
<td>msgFieldAcceptPopUp</td>
<td>void</td>
<td>Causes the insertion pad to be accepted. Sent when the user collapses the insertion pad by hitting the OK button or accepts the IP.</td>
</tr>
<tr>
<td>msgFieldCancelPopUp</td>
<td>void</td>
<td>Cancels the edit in the pop-up insertion pad. The previous value is unchanged. Sent when the user hits the cancel button or cancels the IP.</td>
</tr>
<tr>
<td>msgFieldKeyboardActivate</td>
<td>void</td>
<td>Activates field for keyboard use.</td>
</tr>
</tbody>
</table>

The user normally activates a field by writing into it with the pen, causing the field to send itself msgFieldActivate. This allocates some temporary memory and prepares the field to receive the user's input. When the pen input is complete, the field sends itself msgFieldDeactivate to deallocate the temporary memory.
msgFieldActivatePopUp is sent when the user invokes a field's pop-up insertion pad. This, in turn, self-sends msgFieldActivate to activate the field if necessary, then displays the pop-up and prepares it for editing. The pop-up is normally centered directly over the field, but you can supply a P_RECT32 (in root window coordinates) to reposition it elsewhere.

msgFieldAcceptPopUp signals that the user has completed and confirmed an editing operation in the insertion pad. This deactivates the field by self-sending msgFieldDeactivate, removes the insertion pad from the screen, and sends msgFieldModified to process the new value. Dismissing the insertion pad without confirmation sends msgFieldCancelPopUp instead, deactivating the field but leaving its previous contents unchanged.

msgFieldKeyboardActivate activates a field and prepares it to accept keyboard input. This message is intended to be sent by client programs, and is useful mainly for navigation between fields with the keyboard (for example, when the user presses the z key).

### Input Processing

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgFieldModified</td>
<td>self</td>
<td>Self-sent when a field is modified.</td>
</tr>
<tr>
<td>msgFieldFormat</td>
<td>void</td>
<td>Self-sent to perform formatting.</td>
</tr>
<tr>
<td>msgFieldClear</td>
<td>NULL</td>
<td>Clears the value of the field.</td>
</tr>
<tr>
<td>msgFieldReadOnly</td>
<td>self</td>
<td>Self-sent when an attempt is made to modify a read-only field.</td>
</tr>
</tbody>
</table>

Whenever the contents of a field are modified, it sends itself msgFieldModified to process the new value. If the FIELD_STYLE flag clientNotifyModified is set, the same message is also sent to the field's client, allowing it to take any actions that may be required to respond to the change. msgFieldModified also sets the field's validatePending flag, triggering an input validation sequence. The flag will be cleared after successful validation.

When the user enters or edits text in a pop-up insertion pad, msgFieldModified is not sent to the client until the user dismisses the pad and confirms its edited contents. When text is entered directly into an in-line or overwrite field, this message may potentially be sent each time the handwriting translator recognizes a character. However, the message is not sent if the field's dirty flag (part of the CONTROL_STYLE structure inherited from clsControl) is set. That is, it is sent only when the field's contents are modified for the first time from a known, clean state. The standard method for msgFieldModified sets the dirty flag to suppress the message at the next modification; it is the client's responsibility to re-enable the message by clearing this flag when appropriate. The client can clear the dirty flag immediately if it wishes to be notified each time a new character is recognized (for example, to format the character on the screen.
as soon it is received). Alternatively, the client can choose to leave the flag set, deferring its input processing until some later time (such as when the field is deactivated).

After successfully validating any input, the field sends itself msgFieldFormat to format its new contents for display on the screen. Subclasses of clsField can redefine this message to perform any special formatting they may require. msgFieldClear clears the field’s contents, leaving it empty. (This message also has another purpose in connection with delayed pen input; see below for details.)

The enable flag in a field’s CONTROL_STYLE structure determines whether the field is able to respond to user input. If this flag is not set, the field is considered read-only: any attempt to enter input data into such a field is an error. If the FIELD_STYLE flag clientNotifyReadOnly is also set, the field will send msgFieldReadOnly to its client, allowing the client to take corrective action such as sounding a beep or displaying an error message.

### Delayed Input

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgFieldGetDelayScribble</td>
<td>P_OBJECT</td>
<td>Returns the delayed scribble for delayed fields.</td>
</tr>
<tr>
<td>msgFieldSetDelayScribble</td>
<td>P_OBJECT</td>
<td>Puts the field in delayed mode with the given scribble.</td>
</tr>
<tr>
<td>msgFieldTranslateDelayed</td>
<td>NULL</td>
<td>Translates a field with delayed captured strokes.</td>
</tr>
</tbody>
</table>

By setting the delayed flag in FIELD_STYLE, you can specify that the user’s pen input in a field is to be saved for later processing, rather than translated immediately on entry. In this case, incoming pen strokes are simply buffered as they are received, with no attempt at handwriting translation. You must then signal explicitly when to translate the accumulated input by sending msgFieldTranslateDelayed. (For example, you might display an OK button on the screen and send this message when the user taps the button.)

If necessary, you can manipulate a field’s pending pen input explicitly. msgFieldGetDelayScribble passes back a pointer to the accumulated scribble awaiting translation, and msgFieldSetDelayScribble takes a pointer to a new scribble and substitutes it for the old one. In addition, msgFieldClear, already mentioned earlier, serves a two-fold purpose for delayed fields. If the field currently contains a value, msgFieldClear simply clears it to empty, as before. If the field is already empty, this message clears its pending pen input instead. (You might use this message, for example, to implement a Clear button for canceling all previous pen input in the field and starting over.)
## Input Validation

### Table 42-7

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgFieldValidate</td>
<td>void</td>
<td>Performs the validation protocol for a field.</td>
</tr>
<tr>
<td>msgFieldPreValidate</td>
<td>self</td>
<td>Allows clients to pre-process the value of a field before validation occurs.</td>
</tr>
<tr>
<td>msgFieldValidateEdit</td>
<td>P_FIELD_NOTIFY</td>
<td>Self-sent to perform validation on the field.</td>
</tr>
<tr>
<td>msgFieldPostValidate</td>
<td>self</td>
<td>Self-sent to perform post-validation processing.</td>
</tr>
<tr>
<td>msgFieldNotifyInvalid</td>
<td>P_FIELD_NOTIFY</td>
<td>Sent to notify a client that a field was invalid.</td>
</tr>
</tbody>
</table>

When a field receives a new value, you can send it `msgFieldValidate` to verify the validity of the new value. For example, a field representing a day of the month might want to verify that its value is an integer between 1 and 31, or a field containing a two-letter state abbreviation might check it against a list of known abbreviations.

Input validation occurs automatically whenever a field loses the input focus after its value has changed. You can also force validation by explicitly sending `msgFieldValidate` at other appropriate times, such as the following:

- When the user writes directly into an in-line or overwrite field.
- When a field’s insertion pad is accepted and dismissed.
- When a field is given a new value directly with `msgLabelSetString`.

The basic validation message `msgFieldValidate`, in turn, triggers a sequence of subsidiary messages that perform the actual validation processing. Each stage of the validation sequence is controlled by a flag bit in `FIELD_STYLE`, and can be disabled by clearing the corresponding flag.

If the `clientPreValidate` flag is set, validation begins by sending `msgFieldPreValidate` to the field’s client. This allows the client to do any needed initialization to prepare for validation processing.

Next, the field performs the actual validation by sending `msgFieldValidateEdit`. If the `clientValidate` flag is set, this message is directed to the field’s client; otherwise it is sent to the field itself. The latter option is particularly useful for subclassing. For example, a field containing a two-letter state abbreviation might be implemented as a subclass of `clsField`, redefining the method for `msgFieldValidateEdit` to check the value against a list of known abbreviations. You would then want the validation message sent to the field itself, to be intercepted by the newly subclassed method. If, on the other hand, you wanted to verify a postal code field by comparing it for validity with the corresponding state field, a subclass wouldn’t work because one field’s validation method couldn’t access the contents of the other field. In this case, you would set the
clientValidate flag to send the validation message to the client, and perform the validation at the application level instead.

msgFieldValidateEdit takes an argument of the form:

```c
typedef struct {
    MESSAGE failureMessage;
    OBJECT field;
} FIELD_NOTIFY, FAR * P_FIELD_NOTIFY;
```

where field is the UID of the field requesting validation; the input value to be validated can be obtained from the field with msgLabelGetString. If the validation is successful (that is, if the input value is valid), msgFieldValidateEdit returns stsOK; otherwise it returns an error status and passes back a failure message in failureMessage.

Assuming validation was successful (and provided that the field’s clientPostValidate flag is set), the client is next notified via msgFieldPostValidate to perform any final housekeeping it may require to complete the validation process. Then the field clears its validatePending flag and sends itself msgFieldFormat to format the new value for display on the screen.

If the clientNotifyInvalid bit is set, the client will be notified of any unsuccessful validation with msgFieldNotifyInvalid. This message takes an argument of type FIELD_NOTIFY, whose failureMessage describes the reason for the failure. The client can then respond in whatever way is appropriate, such as by displaying an error message on the screen.

**Layout**

You can set the width of a field by specifying its label width. For example, in FIELD_NEW, you can set .label.cols to 3 and .label.style.numCols to lsNumAbsolute. This will set the label wide enough for the largest characters in the font—the em-width of the font. If the field is, for example, only for numerals, the width is often more than is really necessary.

If the field has a pop-up input pad, the user can resize the pop-up as he or she would any other floating window. By setting .field.maxLen, you can restrict the number of characters the user can enter.

**User Interface**

Even if a field does not allow direct handwritten entry, the user can still make common editing gestures on it. See Chapter 7, Editing and Formatting Text, of the Using PenPoint manual for information on the user interface of fields.
**Data-Specific Fields**

`clsDateField`, `clsIntegerField`, `clsFixedField`, and `clsTextField` are descendants of `clsField` tailored to the handwritten entry of dates, integers, fixed-format numbers, and text.

As well as setting and getting the actual text in these fields using `msgLabelSetString` and `msgLabelGetString`, you can also set and retrieve the values of these fields in a canonical format using specific messages:

- `msgDateFieldSetValue` and `msgDateFieldGetValue` take `P.TIME_DESC`.
- `msgControlSetValue` and `msgControlGetValue` take `P.U32`, passes value in `YYYYMMDD` format.
- `msgDateFieldSetValue` and `msgDatFieldGetVal`ue take `P.TIME_DESC`.

For `clsFixedField`:

- `msgControlSetValue` and `msgControlGetValue` take `P.U32`, passes value in hundredths format.
Chapter 43 / Notes

Notes are windows that appear, present information to the user, and encourage the user to make some response. In other user interfaces, similar functionality is provided by dialog boxes and alerts.

Standardized Messages

There are many standard messages to present to the user, such as “Low memory,” “Disk full,” and so on. You can put up standardized notifications, warnings, and requests to the user using a procedural standard message interface. Standard messages look exactly like notes, but use a procedural interface that isolates you from the text of the message and its presentation. Notes, on the other hand, are proper objects, controlled via a message interface. The standard message interface is documented later in this chapter.

Kinds of Notes

There are two broad kinds of notes: system and application. System-modal notes consume all input and prevent the user from continuing until he or she has dismissed the note. Application-modal notes only consume input to their application and so don’t lock up the whole user interface.
clsNote inherits from clsFrame. A note has an optional title, an optional command bar, and contents. To provide a lot of flexibility, you specify the contents as an array of toolkit table entries. You can also tell clsNote to get the contents of the note from a resource file.

The input-locking functionality of system-modal notes is provided in part by an instance of clsModalFilter, which is described in **Part 5: Input and Handwriting Translation**.

Notes aren't restricted to applications. They are a reasonable way for any piece of code, such as a DLL, to inform the user and interact with the user. For example, the volume support code in the PenPoint file system prompts the user to insert another floppy disk using a note.

### clsNote Messages

Table 43-1 summarizes the messages defined by clsNote.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_NOTE_NEW</td>
<td>Creates a note.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_NOTE_NEW</td>
<td>Initializes the NOTE_NEW structure to default values.</td>
</tr>
<tr>
<td>msgNoteGetMetrics</td>
<td>P_NOTE_METRICS</td>
<td>Get the metrics of a note.</td>
</tr>
<tr>
<td>msgNoteSetMetrics</td>
<td>P_NOTE_METRICS</td>
<td>Set the metrics of a note.</td>
</tr>
<tr>
<td>msgNoteShow</td>
<td>P_MESSAGE</td>
<td>Displays a note.</td>
</tr>
<tr>
<td>msgNoteCancel</td>
<td>P_MESSAGE</td>
<td>Informs a note that it should take itself down.</td>
</tr>
<tr>
<td>msgNoteDone</td>
<td>MESSAGE</td>
<td>Sent to clients when a note is dismissed.</td>
</tr>
</tbody>
</table>

### Creating a Note

You send msgNew to clsNote to create a new note. This takes a pointer to a NOTE_NEW structure for its message arguments. The NOTE_NEW structure includes the following values:

- `metrics` a NOTE_METRICS structure specifying the metrics of the note.
- `pTitle` an optional string for the title. The title will be preceded with the words “Note from” if metrics.flags includes nfUnformattedTitle. Otherwise it will appear exactly as specified.
- `pContentsEntries` an array of TK_TABLE_ENTRYs specifying the content of the note.
- `pCmdBarEntries` an optional array of TK_TABLE_ENTRYs specifying the command bar of the note.
clsNote creates labels from the toolkit table entries in note.pContentEntries, unless you specify some other class in the class field of the TK_TABLE_ENTRYs.

If note.pCmdBarEntries is not objNull, clsNote creates a command bar from it. You should only specify a string and a message for each button in the command bar.

NOTE_METRICS includes:

- **flags** various flags. The flags are described in the following section.
- **autoDismissMsg** the message the note returns or sends if and when it is dismissed.
- **modalFilter** the filter the note uses.
- **timeout** the timeout, in milliseconds, before the note auto-dismisses.
- **client** the client of the note.

Usually, you set the filter to objNull and clsNote creates the right modal filter, depending on the type of the note.

**Flags**

metrics.flags can contain any combination of the following flags (the descriptions summarize behavior when the flags are set):

- **nfSystemModal** the note is system-modal.
- **nfAutoDestroy** the note is automatically destroyed after it is dismissed.
- **nfSystemTitle** use system title, ignoring pTitle.
- **nfAppTitle** use application name, ignoring pTitle.
- **nfUnformattedTitle** use pTitle as specified, without prepending the words “Note from.”
- **nfTimeout** dismiss on timeout as well as input.
- **nfNoWordWrap** don’t word-wrap content labels.
- **nfResContent** pContentEntries is specified as a resource ID (a P_NOTE_RES_ID structure).
- **nfNoBeep** disable preferences-controlled beeping.
- **nfExplicitCancel** note will ignore cmdBar buttons.

The default flag setting for a system modal note is nIDefaultSysFlags, a combination of nfSystemModal, nfAutoDestroy, and nfSystemTitle. The default flags setting for an application-modal note is nIDefaultAppFlags, a synonym for nfAppTitle. nIDefaultFlags is a synonym for nIDefaultSysFlags.

**Contents from Resource Files**

If nfResContent is set, then pContentsEntries should not be an array of TK_TABLE_ENTRYs. Instead, it should be a pointer to a NOTE_RES_ID structure.

The intent is to store strings, separate from the note and its toolkit table, as string tables in resource files. This facility is used by the standard message procedural...
The standard message interface is described later in this chapter.

**System Modal vs. Application Modal**

The behavior of a note is profoundly affected by whether it is system-modal or application-modal. A system-modal note filters all input, preventing the user from activating any other part of the user interface until it is dismissed.

An application-modal note filters all input for its process. It allows input processing in other applications to continue. Hence the user can do other things while the note is up, and these other operations may affect the application.

For example, if you display an application-modal “Do you want to abandon changes?” note, the user can still access the Notebook tabs and tap them to turn to another page, and your application will still receive PenPoint Application Framework messages, even though it is dead to input. If you use application-modal notes, you'll probably have to remember in your application’s state that it has a note up so you know what parts of the user interface to disable. Note that embedded documents are separate processes and will still be live, although embedded components are in the same process.

If the user doesn’t have to respond to a note and can do other things, it’s probably better to use an option sheet instead of a note.

The nfSystemModal flag you specify when creating a note is passed on by clsNote to the modal filter it creates. If the note has no buttons, then clsNote sets the mfAutoDismiss flag in the modal filter. If you specify a modal filter to use, clsNote leaves its flags up to you.

**Using a Note**

Having created a note, you display it using msgNoteShow. This displays the note on the screen.

msgNoteShow uses msgWinInsert, not msgAppAddFloatingWin, to display the note—the note might not be displayed by an application. If the note is application-modal, you probably should not permit the user to turn the page while the note is displayed.

If nfSystemModal is on, then sending this message will block until the note is dismissed. At that time, msgNoteShow will set pArgs to point to the message sent by the button that was hit (or autoDismissMsg if the win was dismissed by its modal filter). Be aware that the entire input system, and therefore the window system, will be blocked while msgNoteShow is waiting for completion with nfSystemModal on.

**Filter**

If you supplied a modal filter in metrics.modalFilter, clsNote uses it to filter input. Otherwise, it creates an instance of clsModalFilter.
The filter that clsNote creates does not restrict input to its subtree, although that is the perceived effect. It allows input to any children of the root that are menus or insertion pads, on the assumption that these are part of the note. This lets you put pop-up choices and handwriting fields in a note.

## Notification

### What Comes Back

If the note is system-modal, then msgNoteShow does not return until the note is dismissed. When msgNoteShow does return, clsNote passes back (not by sending another message) a pointer to a value which indicates what happened. clsNote never uses the client in the note's metrics.

If the note is application-modal, then msgNoteShow returns immediately. Your application and the rest of PenPoint continue to run while the note is displayed. When the note is dismissed, the note's client later receives msgNoteDone (as a separate message). The message argument of msgNoteDone is a pointer to a value that indicates what happened.

For convenience, you usually use MESSAGE values to indicate the result of displaying the note.

### Note Dismissal

A note may be dismissed programmatically or by the user. The message you get back indicates what happened.

If the note has a command bar, the user can only dismiss it by tapping one of the buttons. The message that comes back is the button message of the button that the user activated.

If the note does not have a command bar, the user can dismiss the note by tapping anywhere on it. The message that comes back is the message you specified in note.metrics.autoDismissMessage.

If nffimeout is set in note.metrics.flags, then clsNote will dismiss the note after the timeout in note.metrics.timeout expires. Again, the message that comes back is the message you specified in note.metrics.autoDismissMessage. This means that you can't tell the difference between the user tapping a note without a command bar and it getting timed-out by the system.

If clsNote created its filter, it destroys it when the note is dismissed.

### More Detailed Button Notification

You should not set the client of the buttons in the command bar or you will get a double message from the button. First the note gets msgButtonDown and sends its client msgNoteDone, then the button's own notification sequence will begin.

Similarly, don't bother setting data for the buttons in the command bar. msgNoteDone only passes back the button's message, not its data.
Painting

clsNote alters the word-wrap and shrink-wrap styles of any label items in its contents toolkit table to get a pleasing appearance depending on whether the note has a title and command bar or not. You can force the note not to word-wrap by setting the flags.nfNoWordWrap flag in the note's metrics.

Layout

A note lays itself out at msgNew time, so that it appears on-screen more quickly when it is shown.

If you supply an origin at new time in win.bounds.origin, clsNote will put the note at that location. Otherwise, it centers the note on the screen. If you turn off window shrink-wrap for the note, it will use the size you supply in win.bounds.size. Otherwise, the note sizes to fit its contents.

Destruction

If the nfAutoDestroy flag in the note's metrics is set, when the note is dismissed, clsNote will destroy it. Otherwise, the note still exists.

When the note is destroyed, it removes its modal filter from the filter list, if necessary, and destroys it, if necessary.

If you want to take down a note yourself after creating it, you can use msgNoteCancel.

If you supplied your own modal filter, you must destroy it yourself.

Standard Message Interface

The UI Toolkit provides a standard message interface for displaying standard messages to the user. This uses the same UI components as a note, but is a purely procedural interface. You specify a STATUS or TAG to identify a resource containing the message text, and the procedure creates an instance of clsNote with the text and optional command buttons.

There are five kinds of standard messages (associated procedure names are shown in parentheses):

- **Dialogs** (StdMsgO) system and application dialog boxes, including command buttons for interaction.
- **Progress notes** (StdProgressUpO) notes informing the user that a long operation is taking place. There is currently no support for cancelling a progress note, so there are no command buttons in the note.
- **Application errors** (StdErrorO) notes informing the user that an application error has occurred.
- **System errors** (StdSystemErrorO) notes informing the user that a PenPoint error has occurred.
**Unknown errors** (StdUnknownError()) notes informing the user that an error has occurred which your application cannot identify.

The standard message procedure for each of these types of standard messages is discussed in its own section below. Following these discussions is an explanation of how you can customize the text of each message using the format codes defined in CMPSTEXT.H and specify command buttons for some notes using button definitions.

### System and Application Dialogs

A **dialog** is a mechanism for allowing the user to choose between several courses of action. To create a dialog, execute the StdMsg() procedure. Like most of the standard message procedures, StdMsg() has a variable number of arguments. The first argument is a TAG identifying a dialog message string in the process resource file. The remaining arguments, if any, are parameters interpreted for formatted text. Format codes are described in more detail below.

By default, StdMsg() creates a note whose message is the text specified in the resource, with a single OK button. When the user dismisses the dialog by tapping the OK button, StdMsg() returns 0. If the text of the message string defines one or more buttons, then those buttons appear instead of the default OK button, and StdMsg() returns the button number of the button the user taps to dismiss the dialog. Button definitions are described in more detail below.

StdMsg() returns a negative error status if it had a problem bringing up the note. If StdMsg() cannot find a resource matching the specified TAG, it returns stsResResourceNotFound.

### Progress Notes

A **progress note** is a mechanism for informing the user that a long procedure is taking place. To create a progress note, execute the StdProgressUp() procedure. Like most of the standard message procedures, StdProgressUp() has a variable number of arguments. The first argument is a TAG identifying a string resource in the process resource file. The second argument is a P_SP_TOKEN, a pointer to an SP_TOKEN that StdProgressUp() will fill in. The remaining arguments, if any, are parameters interpreted for formatted text. Format codes are described in more detail below.

StdProgressUp() creates a note with the text specified in the resource as its message, and no command buttons. If the text of the message string defines one or more buttons, the button definitions are ignored. Button definitions are described in more detail below. StdProgressUp() also assigns a token to the SP_TOKEN pointed to by its second argument. This token is used later to take the progress note down, so do not alter it in any way.

StdProgressUp() returns a negative error status if it had a problem bringing up the note. If StdProgressUp() cannot find a resource matching the specified TAG, it returns stsResResourceNotFound.

---

Currently, there is no way for the user to dismiss or "cancel" a progress note.
When the procedure that the progress note describes is complete, execute `StdProgressDown()`. `StdProgressDown` has only one argument, the `P_SP_TOKEN` filled in when you executed `StdProgressUp()`. It uses this token to identify and take down the correct progress note, so don’t alter the token.

### Application Errors

An **application error** note is a mechanism for informing the user that an error has occurred within the application. To create an application error note, execute the `StdError()` procedure. Like most of the standard message procedures, `StdError()` has a variable number of arguments. The first argument is a `STATUS` identifying an error message string in the process resource file. The remaining arguments, if any, are parameters interpreted for formatted text. Format codes are described in more detail below.

By default, `StdError()` creates a note whose message is the text specified in the resource, with a single `OK` button. When the user dismisses the dialog by tapping the `OK` button, `StdError()` returns 0. If the text of the message string defines one or more buttons, then those buttons appear instead of the default `OK` button, and `StdError()` returns the button number of the button the user taps to dismiss the dialog. Button definitions are described in "Message String Format Codes."

`StdError()` returns a negative error status if it had a problem bringing up the note. If `StdError()` cannot find a resource matching the specified `STATUS`, it returns `stsResResourceNotFound`.

### System Errors

A **system error** note is a mechanism for informing the user that one of the standard PenPoint errors has occurred. To create a system error note, execute the `StdSystemError()` procedure. Like most of the standard message procedures, `StdSystemError()` has a variable number of arguments. The first argument is a `STATUS` identifying an error message string in the system resource file (`PENPOINT.RES`). The remaining arguments, if any, are parameters interpreted for formatted text. Format codes are described in "Message String Format Codes."

By default, `StdSystemError()` creates a note whose message is the text specified in the resource, with a single `OK` button. When the user dismisses the dialog by tapping the `OK` button, `StdSystemError()` returns 0. If the text of the message string defines one or more buttons, then those buttons appear instead of the default `OK` button, and `StdSystemError()` returns the button number of the button the user taps to dismiss the dialog. Button definitions are described in "Specifying Command Buttons."

`StdSystemError()` returns a negative error status if it had a problem bringing up the note. If `StdSystemError()` cannot find a resource matching the specified `STATUS`, it returns `stsResResourceNotFound`. 
**Unknown Errors**

An unknown error note is a mechanism for informing the user that your application has encountered an error it does not know how to handle. To create an unknown error note, execute the StdUnknownError() procedure. Unlike most of the standard messages, StdUnknownError() has just one argument, the STATUS of the error encountered.

StdUnknownError() creates a note whose message is the text specified in the resource, with a single OK button. When the user dismisses the dialog by tapping the OK button, StdUnknownError() returns 0. If the text of the message string defines one or more buttons, the button definitions are ignored. Any format code included in the message string is replaced with “???”.

Button definitions and format codes are described in “Specifying Commands Buttons” and “Message String Format Codes.”

If StdUnknownError() cannot find a resource matching the specified STATUS, it creates a standard “Error not found” message.

**Formatted Message Text**

You specify the text of standard messages in resource files. For dialogs and progress notes, you specify the TAG associated with the resource for the message text. For error messages, you specify the STATUS identifying the resource for the message text. The message string resources may specify the text of one or more command buttons to appear in the note. In addition, the text of the message string may contain format codes, as specified in CMPSTEXT.H.

**Specifying Command Buttons**

Message strings may contain optional button definitions. A **button definition** is a series of characters enclosed in square brackets ([ ]) before the text of the message. The text between the square brackets appears in a command button on the displayed note. You may define any number of buttons, but all buttons must be defined at the beginning of the string, before the text of the message. Multiple button definitions appear in the same left-to-right order on the displayed note as they do in the message string. If you do not define any buttons, the standard message procedures will display a single button with the text OK.

Button numbers increase from left to right, starting with 0 for the left-most button. StdMsg(), StdError(), and StdSystemError() return the number of the button pressed to dismiss the note. StdUnknownError() and StdProgressUp() ignore button definitions. StdUnknownError() always displays a single OK button. StdProgressUp() does not display any command buttons.

**Message String Format Codes**

Following any button definitions, the text of a message string resource may contain any combination of literal text and the format codes defined in CMPSTEXT.H. A **format code** is a caret (^), followed by one or more digits, followed by a single character. The digits identifies one of the procedure
arguments following the format string (recall that the standard message procedures have a variable number of arguments), and the single character specifies how to interpret the argument.

A typical example of a format code is \texttt{^2s}, which indicates that the second argument (identified by the 2) should be interpreted as a string (s) and the format code replaced with the result. As a further example, suppose a TAG called \texttt{tagMyAppDialog1} identified the string \texttt{"Could not find a ^1s called ^2s."} and you executed the following procedure:

\begin{verbatim}
buttonHit = StdMsg(tagMyAppDialog1, "document", "Policy");
\end{verbatim}

This would create a dialog box displaying the message, \texttt{"Could not find a document called Policy."} Since the resource string does not define any buttons (see above), \texttt{StdMsg()} would create a single OK button and return its button number, 0, when the user tapped it to dismiss the note.

To include a literal caret (\texttt{^}) in the format string, use two consecutive carets (\texttt{^^}). Otherwise, format code types include the following:

- \texttt{s} the argument is a string.
- \texttt{r} the argument is the resource ID of a string resource.
- \texttt{l} the argument is the group number and indexed list resource ID for a string list.
- \texttt{d} the argument is a U32 which should be printed as a decimal number.
- \texttt{x} the argument is a U32 which should be printed as a hexadecimal number.
- \texttt{1} the argument is a number which selects between a singular and a plural term.

The left brace (\texttt{I}) character indicates that a singular and a plural term are to follow, separated with a vertical bar (\texttt{I}) and terminated with a right brace (\texttt{)}), as in \texttt{^3[singular|plural]}. If the specified argument is 1, the format code is replaced with the singular term; otherwise, it is replaced with the plural term.

For example, suppose the string resource tagged with \texttt{tagMyAppDialog1} is \texttt{"There ^1(islae) ^1d ^1(slot|slots) remaining."} and you executed the following procedure:

\begin{verbatim}
buttonHit = StdMsg(tagMyAppDialog1, 1);
\end{verbatim}

The displayed note would read: \texttt{"There is 1 slot remaining."} On the other hand, if the second argument were 2 instead of 1, the displayed note would read: \texttt{"There are 2 slots remaining."}
Chapter 44 / Frames

Nearly every application in PenPoint has a frame. By default, the PenPoint Application Framework automatically creates and files a frame for an application. Even the simplest sample application (Empty Application, described in the Application Writing Guide) has a fancy frame.

A frame maintains a host of child windows. The most important one is the client window, which you supply. This is where the application or component displays its user interface. This is surrounded by several decoration windows that give frames their familiar appearance. The decoration windows may include any or all of the following components:

- Close box (the triangular corner at the left of the title bar).
- Title bar at the top.
- Page number.
- Menu bar.
- Tab bar.
- Cork margin.
- Command bar.

For document frames, some of these are controllable by the user. The user can turn some of these on and off, either by setting options on the Options Controls... option sheet or by making gestures in the frame title bar. Other decorations are under the control of the PenPoint Application Framework. For example, pages in the regular Notebook don't have close corners if the system Preference for zooming documents is turned off, and the PenPoint Application Framework decides which documents have page numbers.

There are other components of a frame which it picks up from its ancestors. Frames can have shadows from clsBorder (and clsShadow, described below), and resize handles from clsBorder.
Note that scrollbars aren't part of the frame. If you want scroll bars, you must put your window in a custom layout with some scrollbars, or use clsScrollWin.
# Using clsFrame Messages

Table 44-1 summarizes the messages defined by clsFrame. The following sections provide a more detailed discussion.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_FRAME_NEW</td>
<td>Creates a new frame window, setting pArgs-&gt;frame.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_FRAME_NEW</td>
<td>Initializes the FRAME_NEW structure to default values.</td>
</tr>
</tbody>
</table>

### Property Manipulation Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgFrameGetMetrics</td>
<td>P_FRAME_METRICS</td>
<td>Passes back the metrics.</td>
</tr>
<tr>
<td>msgFrameSetMetrics</td>
<td>P_FRAME_METRICS</td>
<td>Sets the metrics.</td>
</tr>
<tr>
<td>msgFrameGetStyle</td>
<td>P_FRAME_STYLE</td>
<td>Passes back the current style values.</td>
</tr>
<tr>
<td>msgFrameSetStyle</td>
<td>P_FRAME_STYLE</td>
<td>Sets the style.</td>
</tr>
<tr>
<td>msgFrameGetClientWin</td>
<td>P_WIN</td>
<td>Passes back metrics.clientWin.</td>
</tr>
<tr>
<td>msgFrameSetClientWin</td>
<td>WIN</td>
<td>Sets metrics.clientWin.</td>
</tr>
<tr>
<td>msgFrameGetMenuBar</td>
<td>P_WIN</td>
<td>Passes back metrics.menuBar.</td>
</tr>
<tr>
<td>msgFrameSetMenuBar</td>
<td>WIN</td>
<td>Sets metrics.menuBar; sets style.menuBar as appropriate.</td>
</tr>
<tr>
<td>msgFrameDestroyMenuBar</td>
<td>VOID</td>
<td>Sets style.menuBar to false and destroys the menu bar, if any.</td>
</tr>
<tr>
<td>msgFrameSetTitle</td>
<td>P_CHAR</td>
<td>Sets the string in the metrics.titleBar.</td>
</tr>
<tr>
<td>msgFrameGetClient</td>
<td>P_OBJECT</td>
<td>Passes back metrics.client.</td>
</tr>
<tr>
<td>msgFrameSetClient</td>
<td>OBJECT</td>
<td>Sets metrics.client.</td>
</tr>
<tr>
<td>msgFrameGetAltVisuals</td>
<td>P_BORDER_STYLE</td>
<td>Passes back the alternate border visuals.</td>
</tr>
<tr>
<td>msgFrameSetAltVisuals</td>
<td>P_BORDER_STYLE</td>
<td>Sets the alternate border visuals.</td>
</tr>
<tr>
<td>msgFrameGetNormalVisuals</td>
<td>P_BORDER_STYLE</td>
<td>Passes back the normal border visuals.</td>
</tr>
<tr>
<td>msgFrameSetNormalVisuals</td>
<td>P_BORDER_STYLE</td>
<td>Sets the normal border visuals.</td>
</tr>
<tr>
<td>msgFrameSelect</td>
<td>VOID</td>
<td>Selects the frame.</td>
</tr>
<tr>
<td>msgFrameShowSelected</td>
<td>BOOLEAN</td>
<td>Makes the frame look selected or not.</td>
</tr>
<tr>
<td>msgFrameMoveEnable</td>
<td>BOOLEAN</td>
<td>Enables or disables UI for moving.</td>
</tr>
<tr>
<td>msgFrameResizeEnable</td>
<td>BOOLEAN</td>
<td>Enables or disables UI for resizing.</td>
</tr>
<tr>
<td>msgFrameZoom</td>
<td>BOOLEAN</td>
<td>Zooms the frame up or down.</td>
</tr>
<tr>
<td>msgFrameIsZoomed</td>
<td>P_BOOLEAN</td>
<td>Passes true if the frame is currently zoomed.</td>
</tr>
</tbody>
</table>

### Client Responsibility Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgFrameDelete</td>
<td>pNull</td>
<td>Asks the frame's client to delete the frame.</td>
</tr>
<tr>
<td>msgFrameClose</td>
<td>pNull</td>
<td>Asks the frame's client to close the frame.</td>
</tr>
<tr>
<td>msgFrameFloat</td>
<td>VOID</td>
<td>Asks the frame's client to float the frame.</td>
</tr>
<tr>
<td>msgFrameZoomOK</td>
<td>P_FRAME_ZOOM</td>
<td>Sent to the client when msgFrameZoom is received.</td>
</tr>
<tr>
<td>msgFrameSelectOK</td>
<td>FRAME</td>
<td>Sent to the client when msgFrameSelect is received.</td>
</tr>
<tr>
<td>msgFrameZoomed</td>
<td>P_FRAME_ZOOM</td>
<td>Sent to client and observers after frame is zoomed.</td>
</tr>
</tbody>
</table>
Creating a Frame

The FRAME_NEW structure has fields for the UID's of all these child windows. In addition, it has a FRAME_STYLE structure. This contains bit fields indicating whether each child window is present. You can plug your own decoration windows into a frame, but clsFrame assumes that its windows have certain behavior. For example, it assumes that its title bar is a label.

If you set the style bit for the appearance of a frame component, and do not supply one to msgNew, clsFrame creates a default component of that type for you.

Usually, clsApp creates a frame for your application in response to msgAppProvideMainWin, which it self-sends during msgAppInit processing. Application frames are explained in more “PenPoint Application Framework and Frames.”

Multiple Windows, Multiple Frames

Frames only support a single client window. If you want to have more than one window in your application, you have two choices:

- Use a layout window to position your windows. The layout window is the frame's client window, and its children are your windows.

- Create multiple frames. The PenPoint Application Framework supports a single application main window, and it is this frame that it puts on-screen, assigns a page number to, and so forth. However, your application is free to create other frames outside this frame.

If you use the second approach, you can either insert your extra windows in the window hierarchy yourself, or you can use msgAppAddFloatingWindow to manage them. This is the method used by PenPoint to keep track of system windows such as option sheets and pop-up insertion pads.

Modifying a Frame

If you later turn off the style bit for one of the frame decorations, clsFrame hides the decoration window. When you turn it on again, clsFrame reinserts the decoration window. If you want to save memory, you should destroy it. You must lay out the frame again at some point.

You can set and get the frame metrics using msgFrameSetMetrics and msgFrameGetMetrics, set the frame title, set and get the frame style fields with msgFrameSetStyle and msgFrameGetStyle, and set the client window with msgFrameSetClientWin and msgFrameGetClientWin.

Frame Layout

clsFrame is a descendant of clsCustomLayout. During layout, the frame looks at its FRAME_STYLE bits and specifies constraints for each of its child windows present.
If `wsLayoutResize` is set in the message arguments to `msgWinLayoutSelf` and you have turned on shrink-wrap, the frame will fit around its components and client window, leaving the latter as is.

`clsFrame` aligns the client window so that it is inset from the surrounding decorations (it uses `clBefore` and `clAfter` rather than `clSameAs` in its custom layout child alignment specifications). This prevents the client window from overlapping the borders of some controls.

**Notification**

A frame does not know what to do with most messages from controls, so you should never make it the client for controls in your application; instead, your application object should be the client for your controls. Frames receive messages, self-send messages, and notify their clients of messages.

Frames receive messages from their decorations. One way this occurs is through simple button notification. The only example of this is the default frame close corner. `clsFrame` sets the button message of the close corner to be `msgFrameClose` and so it receives this message when the user taps on the close triangle.

Another way a frame responds to user actions in it is through gestures. A frame *receives* forwarded gestures from all its child windows. However, the frame *accepts* forwarded gestures only from its title bar. It forwards forwarded gestures from other child windows to its client. Thus, if you make an `xgsCross` gesture on the title bar, the frame receives this and acts on it, but the frame ignores the same gesture made in other windows, merely passing them on to its client.

Another way a frame responds to user actions in it is through subclass handling of messages. Frames are borders and turn on resizing and dragging, so the user can drag and resize the frame using the functionality provided by `clsBorder`.

Usually the frame doesn’t know how to respond to a message, so it forwards the message onto its client. For example, frames don’t know what to do with `msgFrameDelete`, so `clsFrame` sends this message to its client.

Like a control, a frame has a client to which it sends some messages. The `FRAME_METRICS` structure includes a `client` field. If `clsApp` creates your frame, the client is your application object, or you can set the frame’s client to some other object during `msgNew` or with `msgFrameSetClient`.

Frame notifications are not the same as control notifications. The frame client gets predefined messages with fixed arguments. Also, if a frame has observers, it sends them some of the same messages that it sends its client.

Table 44-2 summarizes the frame action messages. Remember that resizing and dragging a frame and bringing a frame to the front are handled by `clsFrame`’s ancestor `clsBorder`. Frames respond to recognized gestures in their title bars by sending themselves these frame messages. If `clsFrame` doesn’t recognize a gesture in the title bar, it forwards the gesture to the frame’s client.
<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgFrameDelete</td>
<td>Sent to self and forwarded to client in response to delete gestures (xgsCross) in the title bar.</td>
</tr>
<tr>
<td>msgFrameClose</td>
<td>Sent to frame by its close box.</td>
</tr>
<tr>
<td>msgFrameFloat</td>
<td>Sent to self and forwarded to client in response to float gestures (xgs2Tap) in the title bar.</td>
</tr>
<tr>
<td>msgFrameZoom</td>
<td>Sent to self and forwarded to client in response to zoom gestures (xgsFlickUp, xgsFlickDown) in the title bar.</td>
</tr>
<tr>
<td>msgFrameSelect</td>
<td>Sent to self and forwarded to client in response to selection gestures (xgsPlus) in the title bar.</td>
</tr>
<tr>
<td>triple-tap</td>
<td>Upon receiving a triple tap (xgs3Tap), frames currently turn off MaskWrapWidth/Height, and lay out again. This allows windows that the user has resized to resize themselves to their desired size.</td>
</tr>
<tr>
<td>triple-flick-up</td>
<td>In the debugging version of the toolkit DLL, frames send msgWinDumpTree to self when you triple-flick upwards on the title bar. In debugging versions of the windows DLL, this prints out an informative dump of the window tree.</td>
</tr>
</tbody>
</table>

`clsFrame` can perform the bring-to-front operation itself because it inherits from `clsBorder`. But it has to ask its client to do the other actions. For most, it just sends the message to the client and the client must implement the action. For zooming and selecting, `clsFrame` has special protocols.

### Selection

The user can select a frame. When a frame receives a forwarded `xgs1Plus` gesture from its title bar, it sends itself `msgFrameSelect`. `clsFrame` responds to this by sending `msgFrameSelectOK` to its client.

When the client receives `msgFrameSelectOK`, the client should determine if it is appropriate to acquire the selection, and acquire the selection if possible by interacting with `theSelectionManager`. If the client is able to acquire the selection, it should send its frame `msgFrameShowSelected` with `true` as the message argument. Conversely, if the frame client detects that the frame has lost the selection, it should send `msgFrameShowSelected` with `false` as the message argument. Usually the frame's client is an application object, and `clsApp` does this for you. By default, you cannot select the frame itself.

When a frame receives `msgFrameShowSelected` it should indicate that it is selected or not, depending on the message argument. `clsFrame` does this by drawing the frame's title bar (if the instance has one) with a double underline.

### Zoom

The user can expand frames to fill the screen. This is called `zooming`. When a frame receives a forwarded `xgsFlickUp` or `xgsFlickDown` gesture from its title bar, it checks its `zoomable` style field and the Allow Zooming system preference. If
zooming is allowed, it sends itself `msgFrameZoom`. `clsFrame` responds to this by sending `msgFrameZoomOK` to its client.

When the client receives `msgFrameZoomOK`, the client should determine if it is appropriate zoom or unzoom, and set the window to zoom/unzoom to accordingly. `msgFrameZoomOK` takes a `FRAME_ZOOM` structure. In this the frame specifies:

```plaintext
  frame  the frame's UID.
  up     whether it wants to zoom (up is true) or unzoom (up is false).
```

The frame client can either veto the operation (by returning `stsRequestDenied`), or allow it. If the client allows a zoom operation, it must pass back the window to which the frame should zoom to (`toWin`).

If the frame is zooming, then it determines the dimensions of the window to which it is zooming, then sizes and positions itself so that its border and shadow are not visible. To do this, it sends itself `msgBorderGetOuterOffsets`. If the frame is unzooming, it returns to its previous size. The frame also sets its tab bar window to be opaque.

Finally, the frame sends `msgFrameZoomed` to its client and its observers.

**Close, Float, Bring-to-Front, Delete**

The button message of the close box is `msgFrameClose`, so that when the user taps on it, it will tell the frame to close. Other gestures in title bars map to frame float, close, and delete messages.

`clsFrame` can't respond to these messages, so it sends them to its client in the hope that the client can implement them. Naturally, `clsApp` can and does.

**Filing**

Frames file their state and all their windows. They file their decoration windows if they exist and have `wsSendFile` set in their window style flags. There are some special tricks for menu bars: `clsFrame` checks to see if the frame's menu bar files. If it does not, `clsFrame` will not file the menu bar object, and will file the menu bar style flag as off. Because `clsFrame` inherits from `clsWin`, it will create a default menu bar when it restores. `clsFrame` uses the same trick with command bars and tab bars.

If the client window has `wsSendFile` set, `clsFrame` tells it to file (this behavior also is inherited from `clsWin`). If the frame's notification client is the application, then the frame will restore its client to be the new application UID (using `OSThisApp`). Otherwise, you have to specify the client after a restore.

**Frame Menus**

You send `msgFrameSetMenuBar` to a frame to change its menu bar, or you can get its metrics and change the `menuBar` UID and style flag.
PenPoint Application Framework and Frames

Application Main Window
The PenPoint Application Framework has a notion of an application’s main window. This is the window that the application framework fills a page with, or floats, or embeds in another application. Usually, this is a frame; clsApp will create a frame for your application by default. Your application instance can retrieve its frame by sending msgAppGetMetrics to self; in the APP_METRICS structure, mainWin is the UID of the frame.

Creating a Custom Main Window
While processing msgAppInit, clsApp normally creates a frame. It does this by sending itself msgAppProvideMainWin, to which it responds by telling clsFrame to create a frame with the standard defaults. If you want to create a different application main window, you have several choices:

- You can create your own frame when your application receives msgAppInit (before you ObjectCallAncestor), and use msgAppSetMainWin to inform clsApp about it. When you pass the message to clsApp, it will see that the application has a main window already and will leave it alone.
- You can also create your frame when your application receives msgAppProvideMainWin.
- You could create your own kind of window altogether instead of a frame, but that’s a risky maneuver since there are so many frame messages.
- You can change the frame metrics to have a different set of windows after the frame is created, for example, in msgAppOpen.

If your application’s frame is a page in the Notebook, the Notebook’s window is an ancestor of your frame in the window tree. It captures size changes of notebook pages, and vetoes them. Thus, pages (frames) in the Notebook don’t change size unless the user floats them or drags the Notebook’s resize handle.

Standard Application Menus
Standard application menus (SAMs) are Document, Edit, and the other menus that appear automatically for any instance of clsApp.

Applications should use this menu bar since it promotes visual consistency and ease of learning and ease of use. You can disable those menu items that do not apply to your application, and add others.

You can send msgAppCreateMenuBar to an application, and clsApp will pass back a pointer to a SAM menu bar. If you pass in a pointer to an existing menu, clsApp appends the supplied menu to the standard application menus.
clsApp defines unique tags (using `MakeTag(clsApp, nnn)`) for the submenus and menu buttons in the standard application menu, such as `tagAppMenuPrint` and `tagAppMenuSelectAll`. These distinguish them from all the other items that could be in an application’s menu bar. For the list of common tags, see `\PENPOINT\SDK\INC\APPTAG.H`.

If some of the items in the SAM are never applicable to your application’s current state, you should remove them. If a menu item is sometimes applicable, but is not currently, you can disable it. See the sections on “Control Enable” in the chapters on Controls, Toolkit Tables, and Menus, and Menu Buttons. `clsApp` implements default enabling behavior for some of the SAM items.

**Border Adjustment**

The standard application Document menu has lines in it to break it into sections. It does this by setting top and bottom margins in its menu items (`tkBorderEdgeTop` and `tkBorderEdgeBottom` in `clsTkTable`). If you delete items, the menu break lines may overlap or end the menu. After removing or adding items to the SAMs, you should send `msgTblLayoutAdjustSections` to the menu. This takes a `Boolean` as its message argument; if `true`, the menu will self-send `msgWinLayout` after adjusting its breaks.

**Subclasses of Frames**

Option sheets are a special kind of frame. You use them to display the properties, or attributes, of the selected object. If the selected object has several different sets of properties (for example, it is a span of characters, and also a paragraph), then the option sheet has multiple “cards,” one for each set of options.

Option sheets are covered in more detail in Chapter 46, Option Sheets.
Chapter 45 / Frame Decorations

Close Box

clsCloseBox is a descendant of clsMenuButton. It implements the small triangular area at the top left of frames. When the user taps on the close box, the frame closes.

Creating a Close Box

Usually you let clsFrame create this for you. clsCloseBox has no tailorable msgNew arguments of its own.

clsFrame sets the close box's message (maintained by clsButton) to msgFrameClose, so that when the user taps the box, it sends this to its parent.

Notification

Close boxes send button notifications and put up a menu just like a regular menu button. clsFrame does not supply a menu and sets the button message to msgFrameClose.

clsCloseBox determines whether the user tapped in the shaded part of the triangle; if not, it doesn't send a message. It disables gestures.

Title Bars

clsTitleBar is a simple subclass of button. It has no instance data or metrics. clsTitleBar centers its string (the document title) relative to the top edge of the frame, so that it includes the close box size in its centering calculation. To get this opportunity, clsTitleBar sets self's horizontal alignment to IsAlignCustom, and responds to msgLabelAlign.

Creating a Title Bar

Normally, you let the frame create a title bar for you. clsTitleBar sets its border fields so that when the user taps and holds the pen over the title bar, it goes into drag mode.

User Interface

If the user has used the Settings notebook to enable, zooming, the user zooms a frame, drawing an flick up \( \uparrow \) on its title bar, and unzooms it by drawing a flick down \( \downarrow \) on its title bar.

The user deletes a frame by writing the cross-out \( \times \) gesture on its title bar.
Tab Bars

Creating a Tab Bar

clsTabBar inherits from clsTkTable. You specify the buttons in the tab bar as a set of TK_TABLE_ENTRYs. Consequently, the most important field in TAB_BAR_NEW is tkTable.pEntries.

In TAB_BAR_NEW_ONLY, you specify:
- style various style fields.
- incrementalLayout how to re-ly out the tab bar when new tabs are added.

You can get and set the style fields alone using msgTabBarGetStyle and msgTabBarSetStyle.

clsFrame creates an empty tab bar for you if tabBar in the frame style fields is set and the frame doesn’t have a UID for a tab bar.

The tab bar window itself is usually invisible. Only the buttons in the tab bar are visible.

Adding Items

You send msgTkTableAdd... messages to add tabs to a tab bar. If you set incrementalLayout in TAB_BAR_STYLE, then the tab bar will alter its layout to get the new child window inserted correctly, without causing a full re-layout. The latter would “crunch” all the children at one end.

Layout

When there are too many child windows in a tab bar for them all to be fully displayed, clsTabBar crunches the children together, causing them to overlap. The user can flick on tabs to rearrange the overlapping of tabs.

clsFrame positions a frame’s tab bar carefully so that it is outside the frame’s apparent shadow if the frame is not zoomed.

Command Bar

clsCommandBar is another class inheriting from clsTkTable. It creates the command bars at the bottom of option sheets and other floating frames. It turns off gestures in its nested buttons, and sets their button feedback style to bsFeedback3D. clsCommandBar is also used to implement the document cork margin.

There are (currently) no fields in COMMAND_BAR_NEW_ONLY that are interesting for application development. As usual, you specify the buttons in the command bar as a set of TK_TABLE_ENTRYs.

You can use clsOption directly to get an option sheet with a close corner and a command bar with Apply and Apply & Close buttons.
**Page Number**

The PenPoint Application Framework tells clsFrame to create a page number for Notebook pages.

If the page is floating ("torn out of the Notebook"), the page number is an instance of clsPageNum. Page numbers are labels that display a number. The user can flick sideways on the button to increase or decrease the number.

The PenPoint Application Framework changes the page number by getting the frame's metrics, and then sending messages to the page number.

If a page is in the Notebook, the user can navigate to the next page. So the PenPoint Application Framework creates an instance of clsCounter. A counter is a variation on page number. It has arrows on either side that the user can tap on to increment or decrement the number. clsCounter inherits from clsTableLayout.

**Shadow**

The shadow drawn by clsBorder is inside the window's edge, so if the notched areas are damaged, they may not be painted properly. clsShadow displays a true shadow if you set the trueShadow style bit. Frames, a subclass of clsShadow, use this feature to draw a true shadow—a transparent painted effect.

clsFrame inherits from clsShadow. The frame window itself includes the shadow. It has to do special things to handle its tab bar if it has a shadow.

A shadow is a transparent window and never draws anything. A shadow contains two children and the children draw. The window being shadowed is any kind of border window, which draws in the regular fashion. The shadow window positions it at the top left of itself. The transparent shadow window places a second window behind the border window, slightly below and to the right, to provide a drop-shadow effect. The shadow window sets the second window's shadow border style and shadowGap border style.

clsShadow inherits from clsBorder. clsBorder actually draws the shadow, and clsShadow just sets some specialized options in BORDER_STYLE to get the right area drawn.

**Creating a Shadow**

You create a shadow by sending it msgNew. In SHADOW_NEWONLY, you specify:

- **borderWin** the window to be shadowed.
- **style** various style fields. The only one is:
  - **trueShadow** whether or not to create a separate window for a true shadow effect.

The window being shadowed must be a border window so that clsShadow can intercept its setting of its shadow bits and do the right thing.
Chapter 46 / Option Sheets

Much of the PenPoint user interface uses a noun-verb, selection-oriented, property sheet interaction model. The user selects something, then makes a gesture (usually a check mark) to see the properties of the selection. clsOption and clsOptionTable support the standard user interface for displaying the properties of the selected object.

\textit{clsOption}

\textit{clsOption} inherits from clsFrame; option sheets are a special kind of frame. You use them to display the properties of the selected object. If the selected object has several different sets of properties (for example, it is a span of characters, and also a paragraph), the option sheet has several windows stacked in it like a deck of cards, each with a menu button in the pop-up choice at the top that allows the user to go to a different set of options. Each of these windows is called an \textit{option card}, although it can be any kind of window.

The frame client window of an option sheet is a scrollwin. This means that option cards you display in the option sheet can be of any size. \textit{clsOption} also uses \textit{clsScrollWin}'s ability to maintain several windows at once while only displaying one.

In this screen shot, note that both \texttt{Apply} buttons are gray. The card is not applicable because it is in an option sheet for Gadgets, but a Widget is selected. The \texttt{Apply} won't be enabled until the user selects a Gadget. The code to create the option cards shown in this screen shot is from \texttt{\PENPOINT\SDK\SAMPLE\TKDEMO\OPTIONS.C}. 

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{option_sheets.png}
\caption{Option Sheets and Option Cards}
\end{figure}
## clsOption Messages

Table 46-1 summarizes the messages defined by clsOption.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_OPTION_NEW</td>
<td>Creates an option or command sheet.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_OPTION_NEW</td>
<td>Initializes the OPTION_NEW structure to default values.</td>
</tr>
<tr>
<td>msgSave</td>
<td>P_OBJ_SAVE</td>
<td>Causes an object to file itself in an object file.</td>
</tr>
<tr>
<td>msgRestore</td>
<td>P_OBJ_RESTORE</td>
<td>Creates and restores an object from an object file.</td>
</tr>
<tr>
<td>msgOptionGetStyle</td>
<td>P_OPTION_STYLE</td>
<td>Passes back the style of the receiver.</td>
</tr>
<tr>
<td>msgOptionSetStyle</td>
<td>P_OPTION_STYLE</td>
<td>Sets the style of the receiver.</td>
</tr>
<tr>
<td>msgOptionGetNeedCards</td>
<td>BOOLEAN</td>
<td>Passes back the value of style.needCards.</td>
</tr>
<tr>
<td>msgOptionSetNeedCards</td>
<td>BOOLEAN</td>
<td>Sets style.needCards.</td>
</tr>
<tr>
<td>msgOptionGetCard</td>
<td>P_OPTION_CARD</td>
<td>Passes back some information about a card in the receiver.</td>
</tr>
<tr>
<td>msgOptionGetTopCard</td>
<td>P_OPTION_CARD</td>
<td>Passes back some information about the top card in the receiver.</td>
</tr>
<tr>
<td>msgOptionGetCardAndName</td>
<td>P_OPTION_CARD</td>
<td>Passes back some information about a card in the receiver.</td>
</tr>
<tr>
<td>msgOptionEnumCards</td>
<td>P_OPTION_ENUM</td>
<td>Enumerates the tags of the cards in the option sheet.</td>
</tr>
<tr>
<td>msgOptionSetCard</td>
<td>P_OPTION_CARD</td>
<td>Changes an existing card in the receiver.</td>
</tr>
<tr>
<td>msgOptionAddCard</td>
<td>P_OPTION_CARD</td>
<td>Adds a card to the receiver.</td>
</tr>
<tr>
<td>msgOptionAddLastCard</td>
<td>P_OPTION_CARD</td>
<td>Adds the last card of a group to the receiver.</td>
</tr>
<tr>
<td>msgOptionAddFirstCard</td>
<td>P_OPTION_CARD</td>
<td>Adds the first card of a group to the receiver.</td>
</tr>
<tr>
<td>msgOptionAddAndInsertCard</td>
<td>P_OPTION_CARD</td>
<td>Adds a card to the receiver and inserts it into the sheet.</td>
</tr>
<tr>
<td>msgOptionRemoveCard</td>
<td>P_OPTION_CARD</td>
<td>Removes a card from an option sheet and destroys that card.</td>
</tr>
<tr>
<td>msgOptionExtractCard</td>
<td>P_OPTION_CARD</td>
<td>Extracts a card's window from an option sheet.</td>
</tr>
<tr>
<td>msgOptionShowCard</td>
<td>P_OPTION_CARD</td>
<td>Causes the given card to be displayed as the current card in the receiver option sheet. Will cause msgOptionRefreshCard to be sent to that card.</td>
</tr>
<tr>
<td>msgOptionShowTopCard</td>
<td>VOID</td>
<td>Shows the client-defined top card.</td>
</tr>
<tr>
<td>msgOptionGetCards</td>
<td>VOID</td>
<td>Results in msgOptionAddCards to the option sheet's client.</td>
</tr>
<tr>
<td>msgOptionApply</td>
<td>nothing</td>
<td>Tells the receiver to initiate the Apply protocol.</td>
</tr>
<tr>
<td>msgOptionApplyAndClose</td>
<td>nothing</td>
<td>Tells an option sheet to run the Apply protocol and then close itself.</td>
</tr>
<tr>
<td>msgOptionRefresh</td>
<td>nothing</td>
<td>Tells an option sheet to refresh its card settings.</td>
</tr>
<tr>
<td>msgOptionApplicable</td>
<td>nothing</td>
<td>Tells an option sheet to ask the top card if it is applicable.</td>
</tr>
</tbody>
</table>

### Class Messages

### Instance Messages
Table 46-1 (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgOptionDirty</td>
<td>nothing</td>
<td>Tells an option sheet to ask the top card to dirty its controls.</td>
</tr>
<tr>
<td>msgOptionClean</td>
<td>nothing</td>
<td>Tells an option sheet to ask the top card to clean its controls.</td>
</tr>
<tr>
<td>msgOptionToggleDirty</td>
<td>nothing</td>
<td>Tells an option sheet to toggle the dirty state of the cards.</td>
</tr>
<tr>
<td>msgOptionClose</td>
<td>nothing</td>
<td>Tells an option sheet to close itself.</td>
</tr>
<tr>
<td>msgOptionGetCardMenu</td>
<td>P_MENU</td>
<td>Passes back the card navigation menu.</td>
</tr>
<tr>
<td>msgOptionCardMenuDone</td>
<td>MENU</td>
<td>Indicates the caller is finished with the card menu.</td>
</tr>
</tbody>
</table>

**Client Responsibility Messages**

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgOptionShowSheet</td>
<td>P_OPTION_TAG</td>
<td>Asks the client of the option sheet to show the option sheet.</td>
</tr>
<tr>
<td>msgOptionProvideCardWin</td>
<td>P_OPTION_CARD</td>
<td>Asks the client of the card to provide the window for the card.</td>
</tr>
<tr>
<td>msgOptionProvideTopCard</td>
<td>P_OPTION_CARD</td>
<td>Asks the client of the option sheet to provide the tag for the top card.</td>
</tr>
<tr>
<td>msgOptionProvideCardDirty</td>
<td>P_OPTION_CARD</td>
<td>Asks the client of the card to provide the dirtiness of the card window.</td>
</tr>
<tr>
<td>msgOptionApplyCard</td>
<td>P_OPTION_CARD</td>
<td>This is sent to a card client when the card should apply its settings.</td>
</tr>
<tr>
<td>msgOptionRefreshCard</td>
<td>P_OPTION_CARD</td>
<td>Tells a card client to refresh its settings from the current selection.</td>
</tr>
<tr>
<td>msgOptionApplicableCard</td>
<td>P_OPTION_CARD</td>
<td>Asks a card whether it is applicable to the current selection.</td>
</tr>
<tr>
<td>msgOptionDirtyCard</td>
<td>P_OPTION_CARD</td>
<td>Sent to a card client when the card should dirty all its controls.</td>
</tr>
<tr>
<td>msgOptionCleanCard</td>
<td>P_OPTION_CARD</td>
<td>Sent to a card client when the card should clean all its controls.</td>
</tr>
<tr>
<td>msgOptionUpdateCard</td>
<td>P_OPTION_CARD</td>
<td>Sent to a card client every time the card is about to be shown.</td>
</tr>
<tr>
<td>msgOptionRetireCard</td>
<td>P_OPTION_CARD</td>
<td>Sent to a card client every time the current shown card is about to be un-shown.</td>
</tr>
<tr>
<td>msgOptionClosed</td>
<td>OPTION</td>
<td>Sent to an option sheet's frame client when the sheet is closed.</td>
</tr>
<tr>
<td>msgOptionCreateSheet</td>
<td>P_OPTION_TAG</td>
<td>A message to be sent by convention by clients creating option sheets.</td>
</tr>
<tr>
<td>msgOptionAddCards</td>
<td>OPTION_TAG</td>
<td>Supplies the second part of the conventional way of creating option sheets. Can be self-sent by an application to fill in a sheet with some cards, and allow subclasses of the sheet creator to modify subclass cards or add different ones.</td>
</tr>
</tbody>
</table>
Creating an Option Sheet

You send msgNew to clsOption to create an option sheet. This takes a pointer to an OPTION_NEW structure. This includes all the fields in FRAME_NEW, together with OPTION_NEW_ONLY fields. The fields in OPTION_NEW_ONLY include:

- **style**: various style fields, described below.
- **pCmdBarEntries**: an optional pointer to a set of TK_TABLE_ENTRYs for the buttons in the command bar, if you want to override the default set.

Option Sheet Styles

In OPTION_STYLE you specify:

- **senseSelection**: whether the option sheet should observe theSelectionManager.
- **modality**: whether the sheet is modal, and if so, whether system- or application-modal.
- **cardNav**: whether to use a tab bar along the side of the option sheet frame or the standard pop-up choice to navigate from one card to another.
- **getCards**: whether cards are static or dynamic. If dynamic, option sheet will send msgOptionAddCards to client when cards are needed.
- **needCards**: whether the current list of cards is invalid.
- **needTopCard**: whether the top card is invalid.

Manipulating Cards

clsOption defines messages that support its user interface, including msgOptionShowCard, msgOptionApply, msgOptionClose, and msgOptionApplyAndClose.

The API also allows you to add a card to an option sheet, get a card, set a card, and remove a card. All these messages take a pointer to an OPTION_CARD structure as their message arguments. In this, you specify:

- **option**: when a card receives an OPTION_CARD structure as an argument, this is the option sheet sending the message.
- **tag**: a tag to use for the option card's menu button in the option sheet's pop-up choice.
- **win**: the UID of the option card window.
- **pName**: a pointer to the name of the card, which also appears in the menu or tab bar button.
- **nameLen**: maximum length permitted for pName.
- **client**: the client of the card that receives apply messages.
- **clientData**: arbitrary client data.
You add cards to an option sheet using `msgOptionAddCard`. You specify the UID of the card window, a tag for its tab, a client for the card, and a name for the window (and its tab). Usually, option cards are instances of `clsOptionTable`, which is a kind of toolkit table that implements the standard two-column appearance of option cards. `clsOptionTable` is described below.

You don't have to supply a UID to `msgOptionAddCard`. If the option sheet is about to display a card, and the UID of that card is `objNull`, then `clsOption` will send the card's client `msgOptionProvideCard`.

You can remove a card by sending `msgOptionRemoveCard`. This searches by tag for an option card to remove.

To display a particular card, send `msgOptionShowCard`. Again, this searches by tag for the card to display, then it retrieves the card's window.

You can set card information with `msgOptionSetCard`, and get it with `msgOptionGetCardWin`.

**Current Card**

The displayed card in the option sheet is the client window of the option sheet's scrollwin. Thus, you could determine the current card by sending `msgFrameGetClientWin` to the option sheet to retrieve its scrollwin, then sending `msgScrollWinGetClientWin` to the scrollwin. However, `clsOption` defines `msgOptionGetTopCard` as a more convenient way to get information about the displayed card.

**Uniqueness**

It's very important that the card tag you specify for a card is unique. You can use `MakeTag` specifying your application class or some other well-known class in order to guarantee uniqueness.

You can only reuse the same window for two option cards if you are sure that they cannot both be asked to be on-screen at the same time—one window cannot be inserted at two places!

**Layout**

When you add a card to an option sheet, it internally converts this to a `msgScrollWinSetClientWin` to set the top window in its scroll window. `clsScrollWin` checks to see if the window is already one of its children. If not, `clsScrollWin` inserts the window in itself and sends `msgWinLayout`.

**Painting**

The option sheet displays a pop-up choice in its title bar, with a button for each option card. The title bar displays the name of the currently displayed card.
The option sheet displays three buttons in its command bar: Apply, Apply & Close, and Close. You can override these by supplying your own TK_TABLE_ENTRY array in pCmdBarEntries.

**Notification**

Whenever a card is shown (by the user flicking the pop-up choice button, by tapping on its menu, by a client sending msgOptionShowCard, or by the card being the visible card), clsOption sends the client of that card the sequence:

- msgOptionApplicableCard
- msgOptionProvideCard, if needed
- msgOptionRefreshCard, if needed
- msgOptionUpdateCard

msgOptionRefreshCard takes the familiar OPTION_CARD message arguments. The client should modify the card as necessary to indicate the properties of the selection. Note that this message is sent each time the card is shown. If win is objNull, when the option sheet tries to show a card, clsOption sends msgOptionProvideCard to that card's client.

Usually, the user closes the option sheet by tapping one of the three buttons. However, you can programmatically drive this by sending msgOptionApply, msgOptionApplyAndClose, or msgOptionClose.

When clsOption receives msgOptionApply or msgOptionApplyAndClose, it gets the top option card and sends that card's client msgOptionApplyCard. Several cards may have the same client, so the message argument for msgOptionApplyCard is a OPTION_CARD structure identifying the option card. The client should figure out the settings of the option card and do whatever it takes to apply them to the selection.

When clsOption receives msgOptionClose, it sends the frame client of the card msgOptionClosed. The client should extract or destroy the card.

**Destruction**

When you destroy an option sheet, clsOption destroys each option card that you defined.

If you want to preserve a card window after the option sheet is destroyed, you can either set its tab's window to objNull, or remove the card and its tab with msgOptionExtractCard.

**Performance**

Creating an option sheet and its option cards is fairly slow. You can increase performance by creating option cards and option sheets in advance. When a sheet comes down, you can choose to destroy the sheet or file it, or maybe only save certain cards within the sheet. When your process is terminated, you should either destroy or file any saved sheets or cards.
Another approach is to create option sheets and option cards in process 0 of your application code, then file them as resources. You can then use `msgResFindResource` to read in the option sheet or cards. If the page orientation and default font haven't changed, the windows will not need to be relaid out.

This is a general technique for improving performance; it is also covered in Chapter 31, Concepts and Terminology.

One possible trade-off between memory, start-up time, and display time is to create the option sheet, install each card, but only provide the window for the first card. Tell the option sheet to lay out, then file it with a well-known resource ID. Create and lay out the other cards separately, and save them under separate resource IDs. This way, you can read the option sheet in quickly when needed, since it doesn't have many child windows.

**Option Sheet Protocol**

Option sheets must interact with the selection holder so that the currently displayed card shows the correct properties for the selection. Also, the option sheet must notify the correct client if the user chooses to Apply changes in properties. Moreover, option sheets are (usually) modeless—while an option sheet is displayed the user may select something else. The option sheet must figure out whether it still applies to the new selection.

The user can display an option sheet by choosing Options... from the Edit menu, or by making a check ✚ gesture. The latter changes the selection and refreshes the option sheet if it's already displayed.

**Check Gesture Processing**

In order for your option sheets to have the correct behavior, you must follow a complex protocol when using option sheets. To illustrate how this protocol works, here's a run-through of what happens when the user makes a check gesture in an application window. In this example, the gesture is received by clsMyView, an application-specific subclass of clsTextView. The protocol is the same for any application- or component-level subclass of a built-in window class which attaches meaning to the check gesture.

1. The instance of clsMyView receives a check gesture (msgGWinGesture with gesture ID of xgsCheck) and calls its ancestor clsTextView.
2. clsTextView determines whether to change the selection. If it does not, the processing resumes with step 4, below.
3. clsTextView could go about changing the selection in two ways: either by directly changing the selection, or by self-sending a public message and changing the selection in response to that message. The latter method would allow subclasses like clsMyView to get into the game and perhaps do something different.
Suppose clsTextView just changes the selection. By default, option sheets have senseSelection set in their style fields, and hence, observe theSelectionManager. Thus, any displayed option sheets receive msgSelChangedOwner when the selection owner changes. Each does step 3a, and processing resumes with step 4.

3a When a displayed option sheet receives msgSelChangedOwner, it determines whether the state of its Apply and Apply & Close buttons should be enabled and active (black) or disabled and inactive (gray). It figures out the new state by:

- Testing whether the new selection owner is in the same process as the sheet. If it is not, the new button state is gray/inactive—option sheets only apply to their own process.
- If the selection owner is in the same process, then the option sheet sends msgOptionApplicableCard to the current card. The status value returned determines the new button state: stsOK means that Applying that option card makes sense for the current selection; stsFailed means that it does not.

The card changes its buttons as appropriate. If the selection is in the same process, and the sheet remains applicable, it self-sends msgOptionDirtyCard. This indicates to the user that every setting on the card will be applied if the user chooses Apply.

4 clsTextView now calls its ancestor with the original msgGWinGesture. Assuming gestureForward is set in the gesture window style fields, clsGWin uses msgGWinForwardedGesture to pass the check gesture to the parent window of the text view. Most classes should not need to handle the check gesture specially, so they should call ancestor. The gesture message eventually winds up in clsEmbeddedWin.

5 The process stops at clsEmbeddedWin, because it does not call ancestor when it receives msgGWinForwardedGesture. If it’s a check gesture, as in this case, then clsEmbeddedWin sends the application msgOptionRefresh.

6 Application classes usually don’t need to respond to msgOptionRefresh, because clsApp responds by enumerating all the floating windows that inherit from clsOption and sending each msgOptionRefresh. At this point, each displayed option sheet in the same application has been told to refresh.

7 When an option sheet receives msgOptionRefresh, the sheet checks the state of its Apply buttons, and will send the current card msgOptionRefreshCard if the Apply buttons are active (and do nothing if they’re inactive).

8 clsEmbeddedWin then gets the current selection owner (the instance of clsTextView/clsMyView) and sends it msgSelOptions.

After all this, the processing unwinds back to clsEmbeddedWin and finally clsTextView (which called ancestor with msgGWinGesture).
clsMyView receives msgSelOptions and calls ancestor (clsTextView). clsTextView checks to see if an option sheet of the correct type is already up (probably by checking a UID in some instance data). If one is, then nothing more needs to be done. If not, then we go to step 12.

clsTextView self-sends msgOptionCreateSheet. It fills in the OPTION_TAG structure with the tag it wants for the option sheet and objNull for the option sheet UID. A descendant class like clsMyView could choose to intercept msgOptionCreateSheet to create its own sheet and plug its UID into option. The logic in each class that responds to msgOptionCreateSheet should be: if the option UID in OPTION_TAG is not objNull, then do nothing assuming some descendant created the sheet, otherwise maybe create a sheet.

Assuming that clsMyView does not create a custom sheet, then msgOptionCreateSheet reaches clsTextView with the UID in option still objNull. clsTextView creates a sheet by sending msgNew to clsOption. We now have an option sheet for the selection, but no cards in it.

clsTextView self-sends msgOptionAddCards, passing the option sheet tag and UID. Again, clsMyView could add cards to the sheet. The usual handler for msgOptionAddCards would first call ancestor, then add/change the cards in the sheet and return.

When the message reaches clsTextView, it adds its cards for the paragraph and character options, then clsMyView adds its cards, and so forth.

The last step clsTextView takes in response to msgSelOptions is to display the new option sheet by sending msgAppAddFloatingWin.

The process now unwinds to clsEmbeddedWin (which sent msgSelOptions), to clsGWin (which forwarded the gesture with msgGWinForwardedGesture, to clsTextView (which called ancestor with the original msgGWinGesture), and finally to clsMyView (which received the initial check gesture).

When an option sheet comes down, it sends msgOptionClosed to its frame client. When the client receives this message, it should use msgAppRemoveFloatingWin. It can then send msgDestroy to the sheet if it does not want to cache the sheet.

What the Card Client Does

Applicability to the Selection

The card's client receives msgOptionApplicableCard. It must figure out whether or not the card is applicable to the selection.

A card client can find out whether it can apply protocol to see whether the selection holder recognizes its tag. The card client can send msgSelOptionTagOK to the selection holder, passing it its tag. If the selection holder recognizes the tag, and judges that it is applicable to the selection, it should return stsOK. This technique relies on you (as the developer of the option sheet) publishing the tags
of your option cards as part of your API. In practice, few option cards will be applicable to anything other than their own “native” selection type.

**Applying**

When a card client receives `msgOptionApplyCard`, it should apply all those settings that are dirty (in the **control dirty** sense). To find all its settings, it can either enumerate its child windows or find them by tag, using `msgWinFindTag`. It can send each control `msgControlGetDirty`, and if the argument passed back is **true**, it should apply that setting. The card client must know how to apply the settings to the selection, since it acknowledged `msgOptionApplicableCard`.

The card client will respond with one of the following actions:

- Send a message to the selected object to get its current attributes.
- Change some attributes to the new values in dirty settings on the option card.
- Send a message to the object to set the new attributes.

**Multiple Card Types per Object Type**

Since the selection owner responding to `msgSelOptions` is at complete liberty to put any cards in the sheet, anything goes. This API does not imply any correspondence between the type of selection and the number of card types.

**Indicating Mixed Attributes**

Often, the selection may extend to have mixed attributes. For example, the user may select several sentences in which some characters are bold, some are underlined, and some are plain. Should the **Bold** control in the character format option card be on or off?

The option sheet user interface does not indicate whether objects have non-homogeneous attributes. Thus, **Bold** may or may not be on, depending on how `clsText` figures out what the character format is. For example, `clsTextView` might look only at the first character of the selection.

`clsOption` does keep track of which controls on the card would be applied, and gives the user some visual feedback about this. In this example, the **Bold** control would not be dirty, indicating to the user that it would not be applied to the selection. The user would have to touch that control in order to dirty it and make it apply to the selection. Controls in cards in option sheets should have the **showDirty** control style bit set in order for the visual distinction to show up.

When the card in an option sheet changes from inactive to active (its **Apply** button becomes active), `clsOption` sends `msgOptionDirtyCard` to the current card's client. If the card's client doesn't respond and the card is a toolkit table, `clsOption` sends `msgBorderSetDirty` to the card window. `clsBorder` responds to this by self-sending `msgBorderSetDirty`. 
Nested Components

Applying an option sheet will work across nested components that are in the same process, so long as every object associated with a sheet is in the process that created the sheet.

Multiple Option Sheets Up at Once

It is up to the components (such as clsTextView) that receive msgSelOptions to figure out whether to use an existing displayed option sheet or to create a new one.

Dimmed Controls

Card clients can mark their controls as dirty to indicate whether or not they apply to the current selection.

Selection Interaction

While an option sheet is up, the user may select items in the option sheet. The option sheet preserves the selection using msgSelPromote and msgSelSetOwnerPreserve.

Option Tables

You are free to put any kind of window in an option sheet. However, the PenPoint UI Guidelines encourage you to use a standard style for the appearance of an option card. Typically, each of the "cards" in an option sheet is a table of option settings. These adopt a standard form with a label on the left and the component to the right, baseline-aligned.

clsOptionTable supports this layout. It inherits from clsTkTable. It organizes toolkit components on the card as a table of label-item pairs. It sets the default toolkit table style to be a bold label, and sets the table layout to be two column.

All clsOptionTable does is set up the appropriate table layout defaults in msgNewDefaults. It has no msgNew arguments of its own. Use clsTkTable messages to add and remove items in an option table.

Code to create a complex nested option table is in \PENPOINT\SDK\SAMPLE\TKDEMO\OPTIONS.C and is excerpted in Chapter 32, Toolkit Tables.

Command Sheets

Command sheets are just like option sheets except that they have different buttons in the command bar at the bottom of the frame. The format of an option sheet is useful even if you don’t need the entire API for Apply, Apply & Close, Close, and so on. Consequently, there is no separate class for a command sheet.

The Export, Import, and Search and Replace dialogs are examples of command sheets.
Creating a Command Sheet

All you need to do is supply a TK_TABLE_ENTRY array in option.pCmdBarEntries which specifies the buttons in the command bar. By default, this is pNull, and clsOption creates the standard Apply/Apply & Close command bar. clsOption sets the client for the buttons in the command bar to be the frame's client.

If you don't want to play with all the selection protocols, you can turn off senseSelection in OPTION_STYLE. If the frame doesn't have multiple tabs and client windows, doesn't need to scroll its client window, and doesn't interact with the selection, then you're better off getting the same visual appearance by modifying a standard frame.
Chapter 47 / Icons

cslIcon draws a picture and some text in a label. PenPoint uses icons extensively in the icon bookshelf below the Notebook, and in the browser windows of the Disk Viewer and tables of contents.

Icons inherit from cslMenuButton. An icon displays a picture as well as a string. It can have a pop-up menu associated with it.

Messages

Table 47-1 summarizes the messages defined by cslIcon.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_ICON_NEW</td>
<td>Creates an icon window.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_ICON_NEW</td>
<td>Initializes the ICON_NEW structure to default values.</td>
</tr>
<tr>
<td>msgIconGetStyle</td>
<td>P_ICON_STYLE</td>
<td>Passes back the current style values.</td>
</tr>
<tr>
<td>msgIconSetStyle</td>
<td>P_ICON_STYLE</td>
<td>Sets the style values.</td>
</tr>
<tr>
<td>msgIconGetPictureSize</td>
<td>P_SIZE16</td>
<td>Passes back the picture size.</td>
</tr>
<tr>
<td>msgIconSetPictureSize</td>
<td>P_SIZE16</td>
<td>Sets the picture size.</td>
</tr>
<tr>
<td>msgIconGetActualPictureSize</td>
<td>P_SIZE16</td>
<td>Computes and passes back the actual picture size.</td>
</tr>
<tr>
<td>msgIconFreeCache</td>
<td>pNull</td>
<td>Frees the cached picture, if any.</td>
</tr>
<tr>
<td>msgIconGetRects</td>
<td>P_RECT32</td>
<td>Passes back the bounds for the picture in pArgs[0] and the label in pArgs[1].</td>
</tr>
<tr>
<td>msgIconProvideBitmap</td>
<td>P_ICON_PROVIDE_BITMAP</td>
<td>Sent to control client when icon needs the picture bitmap.</td>
</tr>
<tr>
<td>msgIconCopyPixels</td>
<td>P_ICON_COPY_PIXELS</td>
<td>Causes the icon to copy pixels from pArgs-&gt;srcWin to a pixelmap.</td>
</tr>
<tr>
<td>msgIconSampleBias</td>
<td>P_ICON_SAMPLE_BIAS</td>
<td>Computes the sample-biased size for a given picture size.</td>
</tr>
</tbody>
</table>
Creating

The icon’s string is maintained by clsLabel. You specify it in the LABEL_NEW_ONLY arguments. You don’t specify the picture at creation; instead, you specify the picture style.

In ICON_NEW_ONLY, you specify:

- style  the style of the icon.
- pictureSize  the picture size in device units.

The fields in ICON_STYLE include:

- transparent  whether the icon background should be transparent.
- picture  the type of picture in the icon.
- freeBitmap  whether to send msgDestroy to the bitmap after providing the icon.
- open  whether to modify the picture to make the icon look as though the user has opened the icon by tapping it.
- sizeUnits  the units in which pictureSize is specified. This can be bsUnitsDevice or bsUnitsLayout, as defined in BORDER.H.
- sampleBias  whether to round the icon size to a multiple of the bitmap size.
- aspect  how to determine the aspect ratio of the picture. Possible values are isAspectRatioWidthFromHeight (compute width based on sample size and height), isAspectRatioHeightFromWidth (compute Height based on sample size and width), and isAspectRatioAsIs (use the sample height and width as is).

The picture field in ICON_NEW_ONLY’s style field indicates the style of picture for the icon. The possible picture styles are:

- isPictureBitmap  the picture is in a bitmap.
- isPictureNone  there is no picture.
- isPicturePixelmap  the picture is a pixelmap.

Bitmap Picture

If the picture style is isPictureBitmap, then clsIcon sends the icon’s client msgIconProvideBitmap when it needs the bitmap for the icon. clsIcon fills in a ICON_PROVIDE_BITMAP structure for the message arguments, specifying:

- icon  the UID of the icon requiring a bitmap.
- tag  the tag of the icon requiring a bitmap.
- device  the device on which the bitmap will be rendered.
- pictureSize  the size of the picture.

You should pass back a bitmap optimized for device and for size. In practice, the two common icon sizes are ten layout units and 21 layout units. The constants iconSizeSmall and iconSizeNormal define these two sizes. clsIcon uses the sampled image operator to render it into a cached image, so your image will be scaled and dithered if your bitmap isn’t the right size or depth.
If freeBitmap is true, clsIcon will free the bitmap you pass back. This is useful if you read the bitmap in from a resource file.

clsBitmap is a useful class for storing sampled images as objects. It and the sampled image operator are described in Part 3, Windows and Graphics.

**Pixelmap Picture**

If the picture style is isPicturePixelmap, then you can get clsIcon to grab its picture from the pixels of some window. You do this by sending msgIconCopyPixels to the icon. This takes an ICON_COPY_PXELS structure, in which you specify:

- **srcWin** the source window to copy pixels from.
- **srcXY** the origin of the area to copy, in LWC of the source window.

clsIcon gets the icon’s picture from the pixels in srcWin. It pulls in the same number of pixels as you specified for the icon size.

Now, the pixels in the area of the source window to be copied may not all be visible. clsIcon forces any hidden areas of the source window to repaint, even if in front of other windows, in order to get the pixels for its image.

**Painting**

 clsIcon uses the bitmap caching facilities of the Windows and Graphics system to get good performance.

**Invalidation**

When it is about to paint an icon’s picture, clsIcon checks to see if it has a cached picture. If the picture is not cached, it sends msgIconProvideBitmap to its client if the picture style is isPictureBitmap. If clsIcon does have a cached image, it checks the orientation and resolution of the output device. If these have changed, the cached image is no good any more, so clsIcon invalidates it.

You can programmatically invalidate the cached picture by sending msgIconFreeCache.

**Notification**

Icons can have an associated pop-up menu. The icons below the Notebook do not have a menu, so they only send their button notification messages, but you can supply a menu.

**Layout**

The various layouts of icons are controlled by clsLabel’s alignment style flags. If the label alignment is lsAlignCenter, the string appears below the icon. If the alignment is lsAlignRight or lsAlignLeft, the string appears to the side of the icon.
Chapter 48 / Trackers and Grab Boxes

Trackers provide transient drawing feedback while the user is dragging the pen, such as when resizing or moving objects. clsTrack handles some simple rectangular drawing styles directly, but it sends client notifications so that you can draw any figure in response to pen movements.

Note that trackers draw, but aren't windows.

To track the pen, trackers grab input.

clsTrack has several style flags specifically for drawing frames. If you drag or resize a floating document or option sheet, notice how the outline includes the tab bar and command bar.

Table 48-1 summarizes the messages defined by clsTrack.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>PTRACK_NEW</td>
<td>Creates a tracker.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>PTRACK_NEW</td>
<td>Initializes the TRACK_NEW structure to default values.</td>
</tr>
<tr>
<td>msgTrackGetStyle</td>
<td>PTRACK_STYLE</td>
<td>Passes back current style values.</td>
</tr>
<tr>
<td>msgTrackSetStyle</td>
<td>PTRACK_STYLE</td>
<td>Sets style values.</td>
</tr>
<tr>
<td>msgTrackGetMetrics</td>
<td>PTRACK_METRICS</td>
<td>Passes back the current metrics.</td>
</tr>
<tr>
<td>msgTrackSetMetrics</td>
<td>PTRACK_METRICS</td>
<td>Sets the metrics.</td>
</tr>
<tr>
<td>msgTrackStart</td>
<td>PXY32</td>
<td>Starts the tracker.</td>
</tr>
<tr>
<td>msgTrackDone</td>
<td>PTRACK_METRICS</td>
<td>Sent to metrics.client when the track is done.</td>
</tr>
<tr>
<td>msgTrackUpdate</td>
<td>PTRACK_METRICS</td>
<td>Sent to metrics.client when the pen moves.</td>
</tr>
<tr>
<td>msgTrackProvideMetrics</td>
<td>PTRACK_METRICS</td>
<td>Sent to a tracker client before tracker is created.</td>
</tr>
<tr>
<td>msgTrackConstrain</td>
<td>PXY32</td>
<td>Constrains a point.</td>
</tr>
<tr>
<td>msgTrackShow</td>
<td>PTRACK_METRICS</td>
<td>Displays the tracker visuals at pArgs-&gt;rect.</td>
</tr>
<tr>
<td>msgTrackHide</td>
<td>PTRACK_METRICS</td>
<td>Removes the tracker visuals at pArgs-&gt;rect.</td>
</tr>
<tr>
<td>msgInputEvent</td>
<td>P_INPUT_EVENT</td>
<td>Notification of an input event.</td>
</tr>
</tbody>
</table>
**Drawing**

If you set `tsDrawViaMessages`, the tracker forwards `msgTrackShow` and `msgTrackHide` to its client. These take `TRACK_METRICS` as their message arguments. From these the client can figure out what to draw.

**Notification**

If you set `tsDrawViaMessages`, you receive `msgTrackUpdate`. The tracker passes you a pointer to a `TRACK_METRICS` structure that gives you enough information to paint appropriate feedback.

Between each `msgTrackUpdate`, you receive `msgTrackHide`. You must remove the old tracker visuals. The most common way to do this is to draw dynamically using XOR or `sysDcDrawDynamic`.

At pen-up, you receive `msgTrackDone`.

**Destruction**

If `autoDestroy` is set in the tracker’s style field, it will destroy itself when it’s done (on pen up).

You can create a tracker on the fly, or keep the object around for reuse. A tracker maintains a drawing object in order to draw on the screen, which is a large object.

**Grab Boxes**

Grab boxes are objects that implement the pop-up handles on borders. `clsGrabBox` uses `clsTrack` internally. Border windows use grab boxes to handle resizing. Grab boxes draw, but aren’t windows.

`clsBorder` watches for pen proximity events and checks the location of the pen to see if it falls in certain “hot” regions. If so, `clsBorder` creates a grab box to track the pen.

It’s unlikely that you would ever want to use `clsGrabBox` directly, but Table 48-2 summarizes the messages and procedures defined by `clsGrabBox`. 
### clsGrabBox Messages and Procedures

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_GRAB_BOX_NEW</td>
<td>Creates a grab box object.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_GRAB_BOX_NEW</td>
<td>Initializes the GRAB_BOX_NEW structure to default values.</td>
</tr>
<tr>
<td>msgGrabBoxGetStyle</td>
<td>P_GRAB_BOX_STYLE</td>
<td>Passes back current style values.</td>
</tr>
<tr>
<td>msgGrabBoxSetStyle</td>
<td>P_GRAB_BOX_STYLE</td>
<td>Sets style values.</td>
</tr>
<tr>
<td>msgGrabBoxGetMetrics</td>
<td>P_GRAB_BOX_METRICS</td>
<td>Passes back current metrics.</td>
</tr>
<tr>
<td>msgGrabBoxSetMetrics</td>
<td>P_GRAB_BOX_METRICS</td>
<td>Sets metrics.</td>
</tr>
<tr>
<td>msgGrabBoxShow</td>
<td>P_GRAB_BOX_INFO</td>
<td>Puts up or takes down the grab box.</td>
</tr>
</tbody>
</table>

**Class Messages**

**Instance Messages**

**Procedures**

- **GrabBoxIntersect()**
  - Determines where pRect is in win. Returns a grab box location, for example, gbLocLRCorner.

- **GrabBoxLocToRect()**
  - Computes the rectangle of the grab box at the given location.

- **GrabBoxPaint()**
  - Paints the grab box at the specified location.

**Messages from Other Classes**

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgInputEvent()</td>
<td>P_INPUT_EVENT</td>
<td>Notification of an input event.</td>
</tr>
<tr>
<td>msgTrackDone()</td>
<td>P_TRACK_METRICS</td>
<td>Sent by a tracker when it's done.</td>
</tr>
</tbody>
</table>
Chapter 49 / Progress Bars

clsProgressBar is a descendant of clsControl that implements a progress indicator in the form of a dynamic bar graph. A progress bar's major axis can be horizontal or vertical, with or without tick marks. Progress bars can also include numeric or custom labels indicating the maximum, minimum, and current values represented by the progress bar. Additionally, clsProgressBar supports flexible layout for the dimensions and units of the bar itself and its associated labels, if any.

The following topics are covered in this chapter:

- Progress bar concepts.
- Progress bar style and metrics structures.
- How to use clsProgressBar messages.
- Some messages clsProgressBar inherits from ancestor classes.

Progress Bar Concepts

A progress bar is a visual indicator of a value relative to its minimal and maximal possible values. It represents this information in the form of a bar graph. The major axis of the progress bar is a line representing a range of possible values, from zero to an arbitrary maximum value. A gray bar called the filled region extends from zero to a point on the axis corresponding to the represented value. That is, if the represented value is one third of the maximum value, then the filled region is one third the length of the major axis. The minor axis of the progress bar is a line extending from the zero end of the major axis through the thickness of the filled region.

You can include decoration including tick marks and labels. Tick marks mark regular intervals along the major axis of a progress bar. Labels are strings identifying the minimum and maximum values represented. You can specify numeric labels or you can have the progress bar query its client for custom labels, arbitrary strings to identify the minimum and maximum values. You can control the color of the filled region as well as the remainder of the progress bar rectangle, the unfilled region. Also, you can specify what combination of top, bottom, left, and right edges of the rectangle the progress bar should draw.

Figure 49-1 shows a progress bar indicating the amount of disk space and memory available in the system. The major axis represents the range of possible values, from zero to the total amount of memory installed. The gray filled region represents the number of megabytes still available in the system. If more memory is consumed, the filled region will shrink to indicate the reduction in available memory.
The minimum internal value of a progress bar is zero. It is up to the client of the progress bar to map actual values to a scale that begins at zero, but the labels may indicate the actual values represented by that scale. If numeric values are not sufficient (for example, if you wish to display units as in the figure), you can instruct the progress bar to ask its client to specify custom labels, arbitrary strings that the progress bar will display.

**Progress Bar Style and Metrics**

Like many toolkit classes, progress bars have associated style and metrics structures that define or modify its behavior and appearance. Each instance of clsProgressBar maintains its own set of metrics in a PROGRESS_METRICS structure. The first field in a PROGRESS_METRICS structure is an inline PROGRESS_STYLE structure called style. This section describes the meaning of the PROGRESS_STYLE fields, then the remaining PROGRESS_METRICS fields.

**The PROGRESS_STYLE Structure**

The style field of a PROGRESS_METRICS structure describes the overall visual style of a progress bar in an inline PROGRESS_STYLE structure. A PROGRESS_STYLE structure describes only the overall visual style of a progress bar, not the values and measurements that make it meaningful.

A PROGRESS_STYLE structure includes the following bit fields:
### Styles/Style Values

<table>
<thead>
<tr>
<th><strong>labels</strong></th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>psLabelsNumeric</td>
<td>Describes what kind of label to use for a progress bar. Use numeric labels. The default.</td>
</tr>
<tr>
<td>psLabelsNone</td>
<td>Do not display labels.</td>
</tr>
<tr>
<td>psLabelsCustom</td>
<td>Query client with msgProgressProvideLabel to determine text of labels.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>ticks</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>psTicksSmall</td>
<td>Determines kind of tick marks to paint along the major axis. Use small tick marks. The default.</td>
</tr>
<tr>
<td>psTicksFull</td>
<td>Use full-height tick marks.</td>
</tr>
<tr>
<td>psTicksNone</td>
<td>No tick marks.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>direction</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>psDirectionHorizontal</td>
<td>Determines the orientation of the major axis, relative to the system. Horizontal (left-to-right axis). The default.</td>
</tr>
<tr>
<td>psDirectionVertical</td>
<td>Vertical (bottom-to-top axis).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>units</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bsUnitsPoints</td>
<td>Determines the units with which to scale progress bar components other than the labels. The default. But any of the clsBorder unit specifications can be used. Note that progress bar labels have a scale value, labelScaleUnits, which is independent of units.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>thickness</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>psThicknessRelFont</td>
<td>Determines the tick mark thickness style. Thickness varies with size of system font.</td>
</tr>
<tr>
<td>psThicknessFixed</td>
<td>Thickness is fixed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>labelRotation</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IsRotationNone</td>
<td>Determines the angle at which to draw the label text. The default. But any of the clsLabel rotation specifications are available.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>labelScaleUnits</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bsUnitsLayout</td>
<td>Determines the units with which to scale progress bar labels. The default. But any of the clsBorder unit specifications can be used. Note that value is independent of units, which control the scale of progress bar components other than the labels.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>edge</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>psEdgeNone</td>
<td>Determines which edges of the progress bar to draw. Edges are pixels next to the inner rectangle occupied by the filled and unfilled region regions (the inner rectangle does not intersect with the edge). Can be default, or any combination of the other flags. No edges drawn.</td>
</tr>
<tr>
<td>psEdgeMinLat</td>
<td>Draw edge perpendicular to the major axis at the minimum value.</td>
</tr>
<tr>
<td>psEdgeMaxLat</td>
<td>Draw edge perpendicular to the major axis at the maximum value.</td>
</tr>
<tr>
<td>psEdgeMinLong</td>
<td>Draw the bottom edge on horizontal progress bars, or the right edge on vertical progress bars.</td>
</tr>
<tr>
<td>psEdgeMaxLong</td>
<td>Draw the top edge on horizontal progress bars, or the left edge on vertical progress bars.</td>
</tr>
<tr>
<td>psEdgeAll</td>
<td>All edges drawn. The default.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>labelFontType</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IsFontSystem</td>
<td>Determines what font to use for displaying the labels. The default. But any of the clsLabel font specifications can be used.</td>
</tr>
</tbody>
</table>
Progress Bar Metrics

In addition to the overall visual style a PROGRESS_STYLE structure describes, a progress bar is further defined by a number of measurements and values. The PROGRESS_METRICS structure holds this information. The first field of a PROGRESS_METRICS structure is an in-line PROGRESS_STYLE structure called style. Thus, a PROGRESS_METRICS structure contains all of the information in a PROGRESS_STYLE structure, as well as additional information that defines the remaining metrics.

A PROGRESS_METRICS structure includes the following fields (data types are in parentheses):

- **style** (an in-line PROGRESS_STYLE structure) PROGRESS_STYLE is described above.
- **numIntervals** (S32) how many tick marks to include along the major axis.
- **ticksPerLabel** (S32) how frequently to attach a label to tick marks. For example, if you set ticksPerLabel to 5, then every fifth tick mark will have a numeric label.
- **minNumericLabel** (S32) the value of the numeric label to identify the minimum value. This value is ignored unless style.labels is psLabelsNumeric.
- **maxNumericLabel** (S32) the value of the numeric label to identify the maximum value. This value is ignored unless style.labels is psLabelsNumeric.
- **thicknessBase** (U16) the thickness of the progress bar, perpendicular to the major axis. If style.thickness is psThicknessRelFont, thicknessBase is a multiplier against the system font size. If style.thickness is psThicknessFixed, thicknessBase is a fixed number of style.units.
- **latitude** (U16) when the progress bar is shrink-to-fit along the minor axis, the minimum size of the minor axis in style.units.
- **longitude** (U16) when the progress bar is shrink-to-fit along the major axis, the minimum size of the major axis in style.units.
- **maxvalue** (S32) the maximum internal value of the progress bar. Although progress bar labels may display any range of values, the internal value is always between zero and maxvalue, inclusive. It is the client’s responsibility to map the represented value to a zero-based scale.
- **value** (S32) the current value of the progress bar. See the note above regarding mapping the represented value to a zero-based scale.
**Progress Bar Messages**

Table 49-2 summarizes the messages defined by clsProgressBar. The following text describes these messages in more detail.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_PROGRESS_NEW</td>
<td>Creates a progress indicator.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_PROGRESS_NEW</td>
<td>Initializes the PROGRESS_NEW structure to default values.</td>
</tr>
<tr>
<td>msgProgressGetStyle</td>
<td>P_PROGRESS_STYLE</td>
<td>Passes back the current style.</td>
</tr>
<tr>
<td>msgProgressSetStyle</td>
<td>P_PROGRESS_STYLE</td>
<td>Sets the style.</td>
</tr>
<tr>
<td>msgProgressGetMetrics</td>
<td>P_PROGRESS_METRICS</td>
<td>Passes back the current metrics.</td>
</tr>
<tr>
<td>msgProgressSetMetrics</td>
<td>P_PROGRESS_METRICS</td>
<td>Sets the metrics.</td>
</tr>
<tr>
<td>msgProgressGetFilled</td>
<td>P_PROGRESS_REGION</td>
<td>Passes back the current filled region color and pattern.</td>
</tr>
<tr>
<td>msgProgressSetFilled</td>
<td>P_PROGRESS_REGION</td>
<td>Sets the current filled region color and pattern.</td>
</tr>
<tr>
<td>msgProgressGetUnfilled</td>
<td>P_PROGRESS_REGION</td>
<td>Passes back the current unfilled region color and pattern.</td>
</tr>
<tr>
<td>msgProgressSetUnfilled</td>
<td>P_PROGRESS_REGION</td>
<td>Sets the current unfilled region color and pattern.</td>
</tr>
<tr>
<td>msgProgressGetVisInfo</td>
<td>P_PROGRESS_VIS_INFO</td>
<td>Passes back information about the current size of the filled and unfilled regions.</td>
</tr>
</tbody>
</table>

**Creating a Progress Bar**

You create a progress bar just as you create any other PenPoint object:

1. Send msgNewDefaults to clsProgressBar to fill a PROGRESS_NEW structure with default values.
2. Modify the default metrics and style fields, if necessary.
3. Send msgNew to clsProgressBar, with a pointer to the modified PROGRESS_NEW structure as its argument.

When msgNew completes successfully, the PROGRESS_NEW structure contains the UID of the new progress bar. When you send a message to an object, you identify the object by its UID.
Manipulating Style and Metrics Fields

The purpose of the style and metrics fields is described earlier in this chapter. This section explains how to manipulate the style and metrics fields to read and modify characteristics of the progress bar.

Determining Progress Bar Style

To determine the state of the progress bar metrics.style:

1. Declare a PROGRESS_STYLE structure.
2. Send msgProgressGetStyle to the progress bar, with a pointer to the PROGRESS_STYLE structure as its argument.

When msgProgressGetStyle completes successfully, the PROGRESS_STYLE structure will reflect the state of the progress bar style at the time you sent msgProgressGetStyle.

Modifying Progress Bar Style

To modify the state of the progress bar metrics.style:

1. Declare a PROGRESS_STYLE structure.
2. Set the fields of the PROGRESS_STYLE structure to reflect the desired progress bar style.
3. Send msgProgressSetStyle to the progress bar, with a pointer to the PROGRESS_STYLE structure as its argument.

When the progress bar receives msgProgressSetStyle, it sets all of its style fields equal to those of the referenced PROGRESS_STYLE structure. If necessary, the progress bar will set its layout bit dirty (as if with msgWinSetLayoutDirty) or send itself msgWinDirtyRect after you modify its style. However, it is the client's responsibility to send msgWinLayout to the progress bar when the style change might affect the layout.

If you wish to set just a subset of the style fields, you must first initialize the PROGRESS_STYLE structure with msgProgressGetStyle, make the necessary changes to the structure, then use the structure with msgProgressSetStyle to set the progress bar style.

Determining Progress Bar Metrics

To determine the state of the progress bar metrics:

1. Declare a PROGRESS_METRICS structure.
2. Send msgProgressGetMetrics to the progress bar, with a pointer to the PROGRESS_METRICS structure as its argument.

When msgProgressGetMetrics completes successfully, the PROGRESS_METRICS structure will reflect the state of the progress bar metrics at the time you sent msgProgressGetStyle.
Modifying Progress Bar Metrics

To modify the state of the progress bar metrics:

1. Declare a `PROGRESS_METRICS`.
2. Set the fields of the `PROGRESS_METRICS` structure to reflect the desired progress bar metrics.
3. Send `msgProgressSetMetrics` to the progress bar, with a pointer to the `PROGRESS_METRICS` structure as its argument.

When the progress bar receives `msgProgressSetMetrics`, it sets its `metrics` fields equal to those of the referenced `PROGRESS_METRICS` structure. If necessary, the progress bar will set its layout bit dirty (as if with `msgWinSetLayoutDirty`) or send itself `msgWinDirtyRect` after you modify its `metrics`. However, it is the client's responsibility to send `msgWinLayout` to the progress bar when the style change might affect the layout.

If you wish to set just a subset of the `metrics` fields, you must first initialize the `PROGRESS_METRICS` structure with `msgProgressGetMetrics`, make the necessary changes to the structure, then use the structure with `msgProgressSetMetrics` to set the progress bar `metrics`.

Manipulating Region Appearance

The main body of the progress bar indicator is composed of the filled region and the unfilled region. The filled region represents the value of the progress bar; its length in relation to the major axis is proportional to the progress bar's current value in relation to its maximum value. The unfilled region fills the rest of the progress bar's inner rectangle. By default, the filled region color is set to `sysDcRGBGray66`, with pattern `sysDcPatForeground`. The unfilled region defaults are `sysDcRGBTransparent` with pattern `sysDcPatNil`.

Structures for Manipulating Regions

The following sections explain how to manipulate the region colors and patterns, and how to determine the size of both regions. The messages for reading and modifying these data do not let you reach into the internals of the progress bar. Instead, the arguments are passed in structures designed for passing the information into and out of the progress bar. PenPoint provides two data structures for this purpose: `PROGRESS_REGION` for color and pattern data, and `PROGRESS_VIS_INFO` for region size information.

Region Color and Pattern

The `PROGRESS_REGION` structure has two fields (field data types are shown in parentheses):

- `rgb` (U32) the RGB color of the region.
- `pattern` (SYSDC_PATTERN) the pattern of the region.
Region Size

The PROGRESS_VIS_INFO structure also has two fields:

- filledRect (RECT32) the rectangle of the filled region.
- unfilledRect (RECT32) the rectangle of the unfilled region.

Determining Region Color and Pattern

To determine the color and pattern of the filled region:

1. Declare a PROGRESS_REGION structure.
2. Send msgProgressGetFilled to the progress bar, with a pointer to the PROGRESS_REGION structure as its argument.

When msgProgressGetFilled completes successfully, the PROGRESS_REGION structure will reflect the color and pattern of the filled region at the time you sent msgProgressGetFilled.

To determine the color and pattern of the unfilled region, follow the same steps, but use msgProgressGetUnfilled instead of msgProgressGetFilled.

Modifying Region Color and Pattern

To modify the color and pattern of the filled region:

1. Declare a PROGRESS_REGION structure.
2. Set the fields of the PROGRESS_REGION structure to reflect the desired color and pattern.
3. Send msgProgressSetFilled to the progress bar, with a pointer to the PROGRESS_REGION structure as its argument.

When the progress bar receives msgProgressSetFilled, it sets its filled region color and pattern equal to those in the referenced PROGRESS_REGION structure. The progress bar will send itself msgWinDirtyRect as necessary. However, it is the client's responsibility to send msgWinLayoutSelf to the progress bar.

If you wish to modify just the color or the pattern, you must first initialize the PROGRESS_REGION structure with msgProgressGetFilled, make the necessary changes to the structure, then use the structure with msgProgressSetFilled to set the filled region color and pattern.

To modify the color and pattern of the unfilled region, follow the same steps, but use msgProgressGetUnfilled instead of msgProgressGetFilled.

Determining Region Bounds

It is possible to determine the sizes of the filled and unfilled regions, using msgProgressGetVisInfo. To determine the sizes of the filled and unfilled regions:

1. Declare a PROGRESS_VIS_INFO structure.
2. Send msgProgressGetVisInfo to the progress bar, with a pointer to the PROGRESS_VIS_INFO structure as its argument.
When `msgProgressGetFilled` completes successfully, the two rectangles (RECT32s) within the `PROGRESS_VIS_INFO` structure will reflect the origins and sizes of the filled and unfilled regions at the time you sent `msgProgressGetVisInfo`.

### Responsibilities of Progress Bar Clients

The client of a progress bar has two primary responsibilities:

- To send `msgWinLayoutSelf` to the progress bar when appropriate (the progress bar properly maintains its layout and dirty bits).
- When using custom labels (style.labels is `psLabelsCustom`), to provide label strings to the progress bar when the progress bar requests them.

*Part 3: Windows and Graphics* provides more information on layout and window repainting. The remainder of this section explains the protocol for providing custom labels.

### Providing Custom Labels

If you write code that uses a progress bar, your code is a client of the progress bar. A progress bar provides a lot of service on its own. For example, if you request it to display numeric labels (style.labels is set to `psLabelsNumeric`), you can specify numbers to label the minimum and maximum values. The progress bar will handle the work of converting the numbers to labels and rendering them in the label font. If you ask it to, it will even create a series of numeric labels at intervals along the major axis.

However, if you tell the progress bar that you want custom labels (style.labels is set to `psLabelsCustom`), it cannot guess at what the labels should be. The progress bar does as much as it can, but it must call on the client for the actual text of the labels. The client must respond with this information on demand, since the progress bar may ask for it at any time.

PenPoint provides the `PROGRESS_PROVIDE_LABEL` structure to facilitate the custom label protocol between the progress bar and its client. The `PROGRESS_PROVIDE_LABEL` structure includes the following fields (field data types are shown in parentheses):

- `progressBar` (CONTROL) the progress bar requesting the custom label string.
- `position` (UI6) the major axis position of the label (zero at a minimum).
- `pString` (P_CHAR) a pointer to a 256-byte buffer which the progress bar provides to hold the label string.

When a progress bar needs a custom label, it creates a pointer to a `PROGRESS_PROVIDE_LABEL` structure. The progress bar sets the `progressBar` field to its own UID, the `position` field to the major axis value for which the label is required, and the `pString` field to point to a 256-byte buffer. The progress bar then sends `msgProgressProvideLabel` to the client, with a pointer to the `PROGRESS_PROVIDE_LABEL` pointer as its argument.
When a progress bar client receives `msgProgressProvideLabel`, it should check the `progressBar` and `position` fields of the `PROGRESS_PROVIDE_LABEL` structure to determine what the label string should be. The client should then copy the label string into the buffer to which `pString` points. At that point, the progress bar can take over, converting the string to a label and drawing it in the label font, just as it does with numeric labels.

**Useful Inherited Messages**

Like most UI Toolkit classes, `clsProgressBar` inherits from a chain of ancestor classes. For `clsProgressBar`, this chain begins at `clsObject` and continues through `clsWin`, `clsGWin`, `clsEmbeddedWin`, `clsBorder`, and `clsControl`. You can read about these ancestors of `clsProgressBar` in more detail in other chapters, but Table 49-3 summarizes a few useful messages `clsProgressBar` inherits from these classes.

<table>
<thead>
<tr>
<th>Message</th>
<th>pArgs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>msgControlGetVal</code></td>
<td>P_S32</td>
<td>Passes back the progress bar's value.</td>
</tr>
<tr>
<td><code>msgControlSetValue</code></td>
<td>S32</td>
<td>Sets the progress bar's value. The progress bar will self send <code>msgWinDirtyRect</code> as necessary.</td>
</tr>
<tr>
<td><code>msgSave</code></td>
<td>P_OBJ_SAVE</td>
<td>Causes the progress bar to file itself in an object file.</td>
</tr>
<tr>
<td><code>msgRestore</code></td>
<td>P_OBJ_RESTORE</td>
<td>Creates and restores a progress bar from an object file.</td>
</tr>
<tr>
<td><code>msgWinLayoutSelf</code></td>
<td>P_WIN_METRICS</td>
<td>Tells the progress bar to recompute its layout parameters.</td>
</tr>
</tbody>
</table>

If the progress bar that is shrink-to-fit in either dimension receives `msgWinLayoutSelf`, it will use the `latitude` or `longitude` metric, as appropriate, to determine the interior dimension of the progress bar (this does not include the inked edges of the bar). If the progress bar is not shrink-to-fit in a dimension, it ignores the corresponding `latitude` or `longitude` metric.
Part 5 / Input and Handwriting Translation
Chapter 50 / Introduction

The PenPoint™ operating system is distinguished by its unique pen-driven user interface. PenPoint provides user interface tools such as gesture windows and insertion pads that your applications incorporate to recognize pen-based commands and to capture handwritten text. You use three PenPoint subsystems to implement this technology in your applications: input, windows and graphics, and handwriting translation. PenPoint is designed to handle device driver input from the pen and from other types of hardware that may be connected to the PenPoint computer. PenPoint fully supports keyboard and hardware timer events, and it has the flexibility to interface to third-party device drivers. Figure 50-1 shows the relationship of these subsystems to the overall PenPoint software architecture.

![Figure 50-1 PenPoint Subsystem Hierarchy](image)

The input subsystem converts device driver function calls into messages that are handled by the class manager. The input subsystem determines which device driver is sending an event, and then routes the event to the appropriate object. If the event originated from the pen driver, the input subsystem routes the event through the window tree. If the event originated from an attached keyboard (or other device), the input subsystem routes the event to the object that is currently assigned input focus. The input and the windows and graphics subsystems provide...
all of low-level coordinate system and dispatch loop handling that most applications need.

The handwriting translation subsystem operates at the class level. Your applications use these classes to handle accumulated pen input and do shape recognition, gesture (command) interpretation, or handwriting translation. A set of special window classes is used to capture handwriting input, and another set of translator classes is used to translate the captured pen stroke data.

The window and graphics subsystem handles conversion from hardware coordinates to local window coordinates. It maintains a window tree that represents which windows are “contained” in parent windows, and it handles the routing of input events based on that hierarchy. All window objects have a set of flags that optionally enable or disable sensitivity to different classes of pen events, giving your applications a great deal of flexibility in defining the visual behavior of the user interface. Many of the input and handwriting translation classes take advantage of windows and graphics features other than window input flags. See Part 3: Windows and Graphics for details.

The Input Subsystem

The input subsystem is the central mechanism for handling user input events. An input event is generated by a hardware device driver to indicate the occurrence of an action by the user such as the pen touching the screen. Device drivers make function calls to the input subsystem to communicate state changes. The input subsystem translates the calls into msgInputEvent messages. The input event messages are routed through the input registry, a set of objects that have expressed interest in receiving input. Any object in the input registry can process any event message it receives, and it can either pass the message along the chain for further processing or terminate processing by returning stsInputTerminate to the input subsystem.

msgInputEvent has an argument structure that describes the type of event, the time it occurred, and x-y coordinate information if the event was generated by the pen driver. Device drivers use this same data structure when making a function call to the Input subsystem. Objects receiving the event message use this information to do the appropriate processing such as rendering in a window, accumulating strokes for the translation classes, or changing the application’s mode due to a key event.

Figure 50-2 shows how events are sent from device drivers to the input subsystem, and how the input subsystem routes them to your application as event messages.
Input Registry

After a call from a device driver has been translated into a `msgInputEvent`, the message is routed through the **input registry**, the set of objects that have expressed interest in receiving input messages. These are the objects shown in a shaded rectangle on the right side of Figure 50-2. Event messages are always routed through the input registry recipients in the following order (each of these recipients is described in more detail later):

1. **Filter objects.** You register an object as a filter object with the `InputFilterAdd()` function. The Input subsystem maintains a list of filter objects, and always gives them the first crack at processing an event message.

2. **Grabber object.** At any time, a single object may be registered as the grabber object. The grabber is similar to a filter in that it receives all input messages that haven't been terminated by the filters ahead of it. The grabber is different from filters in that there is only one active grabber at a time. Grabbing the input stream is meant to be a temporary, dynamic operation.
3 **Listener object.** If the listener field in the event data structure is filled in by the device driver when it calls the input subsystem (with a valid object UID), the outgoing event message is sent directly to that object.

4 **Window tree objects.** If the x-y coordinate field in the event data structure is filled in by the device driver when it calls the input subsystem, the outgoing event message is routed through the hierarchy of window objects. You can optionally limit what types of pen events a particular window is interested in by setting the window input flags field. This technique is demonstrated in Chapter 51. The window input flags are described in Chapter 53.

5 **Target object.** If the x-y coordinate field in the event data structure is not filled in by the device driver, the event message is routed to the target object. At any time a single object can be registered as the target object. The target is a special object that has been singled out to receive non-x-y events such as keyboard or modem input (this is similar to the **input focus** in some other systems).

**Routing**

The input subsystem expects a status code reply from each of the objects which receives the event message. The reply can specify that the input event should be terminated (all processing completed), or continued (passed on to objects further down the input registry). The status return is also used to set and release the grabber object.

The input subsystem maintains an internal queue for events coming in from the device drivers (and applications), and it parcels out event messages utilizing the class manager **ObjectSend** mechanism. Events are serialized within the object hierarchy because the input subsystem doesn’t start processing another event until the current event is terminated somewhere in the registry, or falls through and is terminated by the input subsystem itself (the input subsystem ignores such events).

**Filters**

You use filters to provide behavior which persists over multiple input events. Filters get the first opportunity to process each input event that enters the system. Filters have priorities that determine the order in which multiple filters receive an event. Filters return a status to terminate an event, or a status to allow the event to continue to the next filter (or to the remainder of the input registry if there are no more filters).

The Quick Help system and UI Toolkit menus provide two examples of the use of input filters. When the user starts the Quick Help system, the Quick Help system creates a high-priority filter to trap input events. The Quick Help system determines which window falls under the user’s tap, and gets and displays the Quick Help message
for that window. The filter prevents taps from activating the window the user tapped on, and this behavior persists until the user quits Quick Help.

UI Toolkit menus use filters in a similar fashion. When the user brings up a menu by tapping on a menu button, the menu inserts a filter to trap input events. If the user taps within the menu, the menu processes the tap. If the user taps outside of the menu, the menu takes itself down and terminates the input event. Again, the filter prevents taps outside the menu from activating the window the user tapped on, and this behavior persists until the menu comes down.

Writing input filters can be complicated, because the filters can modify the input stream to the rest of the objects in the input registry. These other objects may depend on particular sequences of input events, and may exhibit unexpected side effects if new filters modify the input stream in unexpected ways.

The Grabber

An object can establish itself as the grabber by returning stslInputGrab as the result of receiving an event message (synchronously) or by calling the input subsystem directly (with the InputGrab() function). Until the grab is terminated, the grabber receives all input event messages that have passed through the filters and must return status indicating whether an event should be passed on to the rest of the input registry or if an event should be terminated. Grab termination is usually controlled by the grabber object returning status to the input subsystem without the stslInputGrab flag set.

A grab can also be terminated when one object requests a grab while another object is already grabbing input events. The second grab request overrides the first. The input subsystem maintains a stack of grab requests. Whenever the current grabber relinquishes control, the previous grabber object on the stack is handed back the grab. To prevent thrashing (two applications repeatedly overriding one another’s grabs), each client can override the grab only once.

Whenever a grab is terminated, the input subsystem sends msglnputGrabTerminated to the previous grab object. This allows the new client to determine if it really wants to continue looking at the input stream.

Listener Objects

If a driver has information about the current state of the object hierarchy in PenPoint, it can request that input events be sent directly to a particular object. It does this by setting the listener field of the input event structure to the UID of a valid object when it places an input event in the input subsystem with InputEventGen. The event is still passed to the filter objects and the grabber. If none of these objects terminates the event, it is next passed to the listener object as designated by the event data structure itself. If the input system routes an event to a listener object, the event terminates there.
**Window Tree**

If an input event has made it past the filters and grabber without being terminated, it is next passed through the window tree to see if any of the open windows on the PenPoint display want to react to the event.

The **window tree** is extended whenever you create a window object and fill in its **parent** field. The hierarchy of windows and parent windows should not be confused with the inheritance hierarchy of classes and superclasses. The first window object in the display screen window tree is called **theRootWindow**. It defines the total area of the computer's display. See Part 3: Windows and Graphics to learn more about the window tree and its implementation.

A window can be **extracted** from the tree, in which case the window disappears from the screen. It still exists in memory, but it does not receive input event messages. When a window is **inserted** into the tree it appears on the screen and may again receive input event messages as part of the input registry.

Any window in the window tree can selectively express interest in or ignore different types of pen events. Windows express their interest in different types of pen event through the **win.flags.input** field. This field includes flags to express interest in events such as pen-out-of-proximity, pen-moved-while-down, pen-moved-out-of-window, and so on. Chapter 53 describes the available input flags. Chapter 51 demonstrates their use.

**Target Object**

The **target object** is an object that is designated to receive input events that are not associated with a specific area of the display (for example, a keyboard event). The input subsystem routes an event to the target object when all of the following circumstances are true:

- The x-y field is **null**.
- The filters and grabber did not terminate the event.
- The listener field is **null**.

You use the **InputSetTarget()** function to make an object the target object. Typically, the target is an object that is designated to receive and process keyboard events. Only one object at a time can be designated as the target.

**Inserting Events into the Input Stream**

The input subsystem also allows events to be inserted into the input stream by applications through a procedure-call interface. Events may be added at the end of the input queue (processed after all of the events currently in the input queue) or at the head of the input queue (processed before the events currently in the input queue). Inserted events are treated the same as events generated by devices.
The Handwriting Translation Subsystem

Several PenPoint classes handle the capture and processing of handwriting. Some specialized subclasses of clsWin capture pen input. clsXtract and its subclasses translate the captured input.

Window Subclasses

Several subclasses of clsWin capture pen input. Figure 50-3 shows the hierarchical relationship of the window subclasses that handle handwriting capture.

The figure shows the complete inheritance ancestry of the handwriting capture classes. Not all of these classes are important to the discussion of handwriting capture and translation. The principal classes that your application will interact with are introduced in the following paragraphs.
• clsGWin is a special lightweight subclass of clsWin that automatically creates and attaches a gesture translator from clsXGesture. The application decides what kind of data is to be displayed under the gesture capture region, and what (if any) effects the gestures will have on the data. clsGWin instances also maintain a reference to a quick help resource file, so that a help gesture drawn over a clsGWin instance will cause a contextually appropriate help text to be displayed.

• clsView is used to allow user interaction with different data objects in PenPoint; it gives you a view on your data by serving as the connection between a window and the actual data. This allows your application to perform operations on data without having to worry about updating windows. The view object manages all window interaction, both input and display, while your application is concerned primarily with the algorithms for processing data.

• clsSPaper is a subclass of clsView. clsSPaper (scratch paper) objects have the capability of accumulating “scribbles.” An clsSPaper object is always attached to a window, and it maintains a scribble data object.

• clsScribble objects hold pen stroke data. It is a direct descendant of clsObject. A scribble is a list of pen vectors strokes. It is the scribble that is passed to a translation object to be converted into a meaningful gesture or ASCII character.

• clsIP provides a class of powerful user-interface components called insertion pads. An insertion pad is a window that accepts handwriting input, performs on-the-spot translation, and displays the result. Insertion pads are used for various kinds of text data entry such as text-insertion during word processing or mini-graphics capture within the context of a text document.

Chapter 56, Using clsIP, Chapter 57, Using clsSPaper, and Chapter 12, Using clsGWin, show how to use these classes to capture pen input.

Translation Classes

clsXtract and its subclasses are used to execute the translation of captured scribbles. Each specialized subclass provides one type of translation object which is attached to a capture object (described above). Figure 50-4 shows the hierarchical relationship of the handwriting translation classes in PenPoint.
dsXtract is the parent of all the translator classes. It contains the functional interface into the feature extraction engine. It observes a given scribble and handles strokes as they are added to the scribble, as well as the determination of when a scribble has been completed by the user. When it completes the extraction of a scribble from the user interface, it sends a message to itself. The specialized subclasses respond to the message differently to process the stroke information in different ways.

dsXGesture interprets the information from a scribble to determine if it is one of a set of defined gesture shapes. A dsXGesture translator is automatically created whenever a clsGWin object receives input.

dsXText provides the basic interface for processing ASCII characters written in English-form. clsXText can process horizontally split, left-to-right sequences of words. clsXText chains lists of characters together and has a number of heuristic analysis modes available: trigram, vertical placement, overlapping segments, and common words.

dsXWord provides translator objects that are used to recognize words.

Chapter 58, Using the Translation Classes, shows how to use these classes in your application to translate captured pen input.
Chapter 51 / Developer's Quick Start

This chapter shows you how to add handwriting recognition to the user interface for your application. The example uses UI Toolkit components that are standard in the PenPoint™ SDK. In a second example, you are shown how to add direct pen handling to the user interface of your application. This allows you to track the pen in a window for rendering, placement, or control tasks.

Capturing and Translating Handwriting

If you want to capture handwriting as input for your application, the application sends msgNew to one of the handwriting capture classes to create a handwriting capture object, which is inserted into the window tree as a child of your application's window. You must then connect a translator object to receive the captured scribbles and translate them into ASCII characters which are then delivered to your application's data manipulation algorithms.

This section uses an example application to demonstrate a simple use of the handwriting capture subsystem. The application is a simple arithmetic sum expression calculator. The user writes an expression such as 5-2 into an insertion pad. After translation, the application parses the expression, does the required calculations, and returns a value for display.

The following steps highlight the actions an application must perform to setup and execute handwriting input:

- Initialize a structure defining a new insertion pad by sending msgNewDefaults to clsIP.
- Initialize a structure defining a new translator by sending msgNewDefaults to clsXText.
- Specify and compile a template that allows only the desired characters to be accepted for input. The template is then added as a part of the translator msgNew arguments.
- Create the translator by sending msgNew to clsXText.
- Add the UID for the new translator as a part of the insertion pad msgNew arguments.
- Create the insertion pad by sending msgNew to clsIP.
- Make the insertion pad visible on the screen by inserting it into the application's main window.
Example 51-1 shows how these steps are implemented in the Adder application. You can find the source code for the Adder application in the SDK distribution directory \PENPOINT\SDK\SAMPLES\ADDER. Example 51-1 is from the CreateInsertWindow() function defined in ADDERAPP.C.

### Example 51-1

Creating and Inserting an IP Window

Adder sets itself up as an observer of the insertion pad (by setting the ipNew.ip.client field to self in the msgNew arguments as shown in the example. Notice also that the physical appearance of the insertion pad is determined by setting style fields in the msgNew arguments.

```c
STATUS LOCAL CreateInsertWindow ( 
    P_ADDER_APP_INST pInst, 
    OBJECT self) 
{
    STATUS s;
    WIN_METRICS wm;
    IP_NEW ipNew;
    P_UNKNOWN pNewTemplate;
    XLATE_NEW xNewTrans;
    U16 xlateFlags;
    XTM_ARGS xtmArgs;
    //
    // Create an insertion pad, which is the standard mechanism
    // for accepting handwritten input.
    //
    ObjectCall(msgNewDefaults, clsIP, &ipNew);
    // Do not show the resize button.
    ipNew.border.style.resize = bsResizeNone;
    // Translate when the user lifts the pen out of proximity
    ipNew.ip.style.buttonType = ipsProxButton;
    ipNew.ip.style.embeddedLook = true;
    // set the listener field for notifications
    ipNew.ip.client = self;
    //
    // Create a translator for the insertion pad
    //
    ObjectCall(msgNewDefaults, clsXText, &xNewTrans);
    //
    // Create a template for the insertion pad
    //
    xtmArgs.xtmType = xtmTypeCharList;
    xtmArgs.xtmMode = 0; // no special modes
    xtmArgs.pXtmData = "0123456789+-."; // ascii template
    StsRet(XTemplateCompile(&xtmArgs, osProcessHeapId, &pNewTemplate), s);
    xNewTrans.xlate.pTemplate = pNewTemplate;
    //
    // The handwriting engine is geared primarily towards text
    // translation. We can improve numeric translation by
    // disabling these context assumptions:
    //
    // alphaNumericEnable - enables character recognition that is
    // geared towards text-processing (e.g., most characters
    // are assumed to be letters, and numbers are separated from
    // letters by spaces.)
    // punctuationEnable - enables punctuation rules that are geared
    // succeeded
```
CHAPTER 51 / DEVELOPER’S QUICK START
Capturing and Translating Handwriting

Example 51-1 (continued)

    // towards text-processing (e.g., periods are always followed
    // by a space).
    // verticalEnable - enables rules that help recognize a character
    // based on its vertical orientation (e.g., "t" versus "+").
    //
    // By disabling these flags, we get far fewer unrecognized characters
    // in our numeric expressions.
    //
    xNewTrans.xlate.hwxFlags &=
        ~(alphaNumericEnable | punctuationEnable | verticalEnable);
    ObjCallRet(msgNew, clsXText, &xNewTrans, s);
    ipNew.ip.xlate.translator = xNewTrans.object.uid;
    // give our template veto power during the translation
    ObjCallRet(msgXlateGetFlags, xNewTrans.object.uid, &xlateFlags, s);
    xlateFlags |= xTemplateVeto | spaceDisable;
    ObjCallRet(msgXlateSetFlags, xNewTrans.object.uid, (P_ARGS)xlateFlags, s);
    ObjCallRet(msgNew, clsIP, &ipNew, s);
    pInst->iPad = ipNew.object.uid;
    //
    // Insert the insertion pad into our adderWin window
    //
    wm.parent = pInst->adderWin;
    wm.options = wsPosTop;
    ObjCallRet(msgWinInsert, ipNew.object.uid, &wm, s);
    return stsOK;
} /* CreateInsertWindow */

Note that the IP notifies the application with msgIPDataAvailable whenever the
IP receives and translates input. In Adder, the method to handle this message is
GetInsertPadData(). Example 51-2 shows its code.

Example 51-2

Translating and Displaying Captured Scribbles

GetInsertPadData() gets a list of translated characters from the insertion pad by sending msgIPGetXlateString to the
insertion pad. GetInsertPadData() next tries to clean up some common handwriting translation errors. After that it sends
the string to the expression analyzer engine with msgAdderEvaluatorEval. Finally, it displays the result in a label window
with msgLabelSetString.

MSG_HANDLER GetInsertPadData {
    const MESSAGE msg,
    const OBJECT self,
    const OBJECT pArgs,
    const CONTEXT ctx,
    const PP_ADDER_APP_INST pData)
{
    IP_STRING ipStr;
    STATUS s;
    char display[MAX_DISP];
    EVAL_FRAME ex;
    P_ADDER_APP_INST pInst;
    pInst = *pData;
    continued
// Get the text from our instance's insertion pad into ex.expression...
// ipStr.len = SizeOf(ex.expression);
ipStr.pString = ex.expression;
ObjCallRet(msgIPGetXlateString, pInst->iPad, &ipStr, s);
StsRet(RemoveNewLines (ex.expression), s);

// ...and evaluate the expression.
// ObjCallRet (msgAdderEvaluatorEval, pInst->evalObj, &ex, s);
_Debugf("GetlnsertPadData () returned from msgAdderEvaluatorEval: ex. badExpr = \%s,"
   " value = \%g valueStr = <\%s>\", ex.badExpr? "true":"false",
ex.value, ex.valueStr);

// Construct the string to display
// sprintf (display, "\%s = \%s", ex.expression, ex.valueStr);

// Display the string
// ObjCallRet (msgLabelSetString, pInst->label, display, s);
// return stsoK;
//MsgHandlerParametersNoWarning;
} /* GetInsertPadData */

## Handling Low-Level Pen Input

In some cases, you may want to handle low-level pen input events directly. You may want to provide visual user feedback, or you may want to control your own rendering processes. To do this, you subclass clsWin (the window class), and not one of the more specialized subclasses such as clsIP or clsSPaper which automatically provide pen handling and input interpretation.

This example shows how the **input flags** are set for a window to track particular pen input events. When one of these events occurs, a method is activated that renders a simple cursor in the window to give visual feedback for the location of the pen.

The following steps highlight the actions an application must perform to setup and respond to low-level pen input.

- **Subclass clsWin** and add the functionality for the responses to pen input that you need in your application.
- **Add a drawing context to the window** that will render the user feedback.
- **Add the methods that will call the drawing context appropriately according to the type of pen events that are detected in the window.**
- **Install this new clsWin subclass.**
- **Create and install an application class that uses an instance of the clsWin subclass as its main window.**
When you launch the application, it comes up with the window ready to receive input, and responds according to the methods in your special window class.

Example 51-3 shows how a specialized subclass of clsWin handles msgInputEvent. The code comes from ININPUT.C, part of a demonstration application in the SDK distribution directory \PENPOINT\SDK\SAMPLE\ININPUT.

Example 51-3
Handling Low-Level Pen Events

INPUTAPPC defines clsInWin as a local subclass of clsWin. clsInWin handles msgInputEvent with the InWinInput() method. msgInputEvent always arrives with an input event data structure pointed to by pArgs (Chapter 53 describes the data structure in detail). The structure fields indicate what kind of device initiated the event, spatial data, modal data, and so on.

pArgs->devCode is a standard message token defined with the MakeMsg() macro. MakeMsg() binds a class UID and a message number to create a unique token. In the case of input events, however, the second value represents an event type rather than a message number. The case statement below produces different on-screen behavior depending on the message number (the type of pen event).

InWinInput() handles msgInputEvent by drawing a small square at the location of the input event.

```c
MsgHandlerArgType(InWinInput, P_INPUT_EVENT)
{
    INWIN_INST inst;
    RECT32 box;
    inst = *(P_INWIN_INST)pData;
    box.size.w = 14;
    box.size.h = 14;
    ObjCallWarn(msgWinBeginPaint, inst.dc, Nil(P_ARGS));
    switch (pArgs->devCode) {
        case msgPenDown:
            /* draw a rectangle with the origin at the point */
            box.origin = pArgs->xy;
            ObjectCall(msgDcDrawRectangle, inst.dc, &box);
            inst.prevLoc = pArgs->xy;
            ObjectWrite(self, ctx, &inst);
            break;
        case msgPenExitDown:
            case msgPenOutProxDown:
            case msgPenUp:
                /* redraw the rect at the last location to clear it */
                box.origin = inst.prevLoc;
                ObjectCall(msgDcDrawRectangle, inst.dc, &box);
                inst.prevLoc.x = minS32;
                ObjectWrite(self, ctx, &inst);
                break;
        case msgPenMoveDown:
            if (inst.prevLoc.x != minS32) {
                /* redraw the rect at the last location */
                box.origin = inst.prevLoc;
                ObjectCall(msgDcDrawRectangle, inst.dc, &box);
            }
        continued
```
/* draw a rectangle with the origin at the point */
box.origin = pArgs->xy;
ObjectCall(msgDcDrawRectangle, inst.dc, &box);
inst.prevLoc = pArgs->xy;
ObjectWrite(self, ctx, &inst);
break;
default:
    break;
}
ObjectCall(msgWinEndPaint, inst.dc, Nil(P_ARGS));
return (stslnputTerminate); // indicates completion of the event
MsgHandlerParametersNoWarning;
} /* InWinInput */

In the case where the user has placed the pen down on the screen (msgPenDown) a square is drawn at the location of the event. Note how easily the units are handled. The input subsystem provides the x-y location of the event in local window coordinates (pixels), and the drawing context which is bound to this window (shown in a later fragment) also knows how to handle these coordinates without any scaling.

It is useful to see how rendering capability was added to this otherwise unembellished subclass of clsWin. Example 51-4 shows how clsInWin initializes its instances when it receives msgNew or msgRestore.

### Example 51-4

**Initializing a Window with Rendering Capability**

clsSysDrwCtx is a standard Window and Graphics class that is used to create a drawing context (DC) object. A DC manages rendering in a window. InWinInput(), shown in Example 51-3, sends msgDcDrawRectangle to the DC to render a small rectangle in the window. You can render in a window under programmatic control or, using the technique shown here, you can allow input events to drive the rendering process.

InWinInit(), shown here, creates a DC and sets the DC to operate in raster XOR mode (sysDcRopXOR). In this mode, drawing the rectangle a second time in the same position restores the pixels to their original state. Using this feature, InWinInput() can animate the rectangle by first drawing at the old location to XOR it out, and then drawing it at the new location.

```c
MsgHandler(InWinInit)
{
    INWIN_INST inst;
    SYSDC_NEW dcNew;
    STATUS s;
    // Ancestor called in method table
    // Clear the local instance data
    memset(&inst, 0, sizeof(inst));
    // Create a drawing context to perform the drawing
    ObjectCall(msgNewDefaults, clsSysDrwCtx, &dcNew);
    ObjCallRet(msgNew, clsSysDrwCtx, &dcNew, s);
    // Tie the window and the drawing context together
    ObjectCall(msgDcSetWindow, dcNew.object.uid, self);
    // we'll only deal in screen coordinates in this example
    ObjectCall(msgDcUnitsDevice, dcNew.object.uid, Nil(P_ARGS));
    // Tell the DC how the drawing will be done
    ObjectCall(msgDcSetRop, dcNew.object.uid, (P_ARGS)sysDcRopXOR);
    continued
```
That’s how a drawing context is bound to a window, and how the window handles input events when they are handed to it. To see how the window object is told what input events to be interested in, we must look at how it is inserted into the window tree, and thus entered in the input registry.

The application is responsible for inserting the window. Example 51-5 shows how clsInputApp inserts an instance of clsInWin as its main application window.

Example 51-5

Inserting a Custom Window as the Main Application Window

clsInputApp calls the initialization method InputMsgAppOpen() whenever it receives msgAppOpen (for example, when you launch a new instance of clsInputApp by tapping on its entry in the stationery menu).

The application uses msgNewDefaults to initialize the msgNew argument structure inNew, then modifies the window input flags field, inNew.win.flags.input. This field determines to which pen input events the window responds (Chapter 53 describes window input flags in detail). In this case, the input system will send only the msgPenUp, msgPenDown, msgPenMoveDown, msgPenMoveUp, msgPenOutProxUp, msgPenExitDown, msgPenExitUp, and msgInProxUp pen events to the window. msgNew creates the new clsInWin object. The window input flags filter pen events only. They do not filter out events from other input sources.

The application sends msgFrameGetMetrics to the main window of the application to copy the application frame metrics to frameMetrics, sets the client window (frameMetrics.clientWin) to the new instance of clsInWin, then sends msgFrameSetMetrics to establish these changed metrics as the metrics of the application frame. As soon as the application manager displays your application’s UI on the screen, the window can receive input event messages.

```c
MsgHandler(InputMsgAppOpen)
{
  STATUS s;
  WIN_NEW inNew;
  FRAME_METRICS frameMetrics;
  APP_METRICS appMetrics;
  WIN_METRICS winMetrics;
  // Ancestor called in method table
  // Create an instance of the inWin class
  ObjCallRet(msgNewDefaults, clsInWin, &inNew, s);
  // set the input flags to get tip & move events
  inNew.win.flags.input = inputTip | inputMoveDown | inputMoveUp |
    inputOutProx | inputExit | inputInProx | inputMoveDelta | inputAutoTerm;
  ObjCallRet(msgNew, clsInWin, &inNew, s);
}
```
// Tell the frame of the application about the client window
ObjCallJmp(msgAppGetMetrics, self, &appMetrics, s, Error1);
ObjCallJmp(msgFrameGetMetrics, appMetrics.mainWin, &frameMetrics, \
  s, Error1);
frameMetrics.clientWin = inNew.object.uid;
ObjCallJmp(msgFrameSetMetrics, appMetrics.mainWin, &frameMetrics, \
  s, Error1);

// Relayout the window
winMetrics.options = wsLayoutDefault;
ObjCallJmp(msgWinLayout, appMetrics.mainWin, &winMetrics, \
  s, Error1);
return stsOK;

// Recover from errors here. In particular, free resources.
Error1:
  (void) ObjCallWarn(msgFree, inNew.object.uid, pNull);
  (void) ObjCallAncestorWarn(msgDestroy, self, pNull, ctx);
  return s;
MsgHandlerParametersNoWarning;
} /* InputMsgAppOpen */
Chapter 52 / Event Processing

The input system translates the event data provided by device drivers into event messages and distributes them according to standard mechanisms. The input system uses `ObjectSend()` to send `msgInputEvent` to the first object in the input registry. This object processes the event, then either terminates the event or allows the input system to send the input event to the next object in the input registry.

**Event Generation**

Events normally enter the system through the `InputEventGen()` interface. Device drivers use this interface to deposit event information in the input system’s event queue. The event information includes, as a minimum, the `devCode`, a message that identifies the device driver and the type of event. Optionally, the event information may include an x-y coordinate in local window units, a specific listener object, and up to 24 bytes of other event-specific data.

For example, when the pen touches the screen in a window, the event data might include `msgPenDown` as the devCode, and the local window coordinates of the pixel the pen touched. Depending on the design of the device driver and the capabilities of the hardware, the event-specific data might include such information as the hardware device coordinates of the touched pixel, pen pressure, and so on.

**Event Handling**

When the input system sends an input event message to an object, the object processes the message by switching control to the message handler for `msgInputEvent`. The method does the necessary processing, typically based on the `devCode` value of the input event, then returns an input event status return value. Table 52-1 summarizes the possible return values and their meanings.
### Table 52-1

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>stsInputContinue</td>
<td>The input system processes the event normally, passing it to the next object in the input registry.</td>
</tr>
<tr>
<td>stsInputTerminate</td>
<td>The input system does not pass the event to other objects in the input registry.</td>
</tr>
<tr>
<td>stsInputGrabContinue</td>
<td>The input system gives the grab to the object returning this status and continues to pass the event to other objects in the input registry.</td>
</tr>
<tr>
<td>stsInputGrabTerminate</td>
<td>The input system gives the grab to the object returning this status and does not pass the event to other objects in the input registry.</td>
</tr>
<tr>
<td>stsInputGrab</td>
<td>Identical to stsInputGrabTerminate. The input system gives the grab to the object returning this status and does not pass the event to other objects in the input registry.</td>
</tr>
<tr>
<td>stsInputIgnored</td>
<td>The input system interprets this status identically to stsInputContinue. An ancestor class can use stsInputIgnored instead of stsInputContinue to inform subclasses that the ancestor was not interested in the input event.</td>
</tr>
<tr>
<td>stsInputGrabIgnored</td>
<td>The input system interprets this status identically to stsInputGrab. An ancestor class can use stsInputGrabIgnored instead of stsInputGrab to inform subclasses that the ancestor was not interested in the input event.</td>
</tr>
<tr>
<td>For Filters Only</td>
<td></td>
</tr>
<tr>
<td>stsInputSkip</td>
<td>The input system passes the event to the next object in the input registry that is not filter.</td>
</tr>
<tr>
<td>stsInputSkipTo2</td>
<td>The input system passes the event to the next object in the input registry that is not a filter in the first range (zero through 63) of filter priorities.</td>
</tr>
<tr>
<td>stsInputSkipTo3</td>
<td>The input system passes the event to the next object in the input registry that is not a filter in the first range (zero through 63) or second range (64 through 127) of filter priorities.</td>
</tr>
<tr>
<td>stsInputSkipTo4</td>
<td>The input system passes the event to the next object in the input registry that is not a filter in the first range (zero through 63), second range (64 through 127), or third range (128 through 191) of filter priorities.</td>
</tr>
<tr>
<td>stsInputTerminateRemoveStroke</td>
<td>The input system does not pass the event to other objects in the input registry. Furthermore, the input system removes any remaining input events from the same stroke.</td>
</tr>
<tr>
<td>stsInputGrabTerminateRemoveStroke</td>
<td>The input system gives the grab to the object returning this status and does not pass the event to other objects in the input registry. Furthermore, the input system removes any remaining input events from the same stroke.</td>
</tr>
</tbody>
</table>

An object receiving an input event message must return one of these status codes in response to every input event it receives. An object that is not the grab object should send stsInputContinue to continue normal processing of the event through the input registry, stsInputTerminate to stop the processing of the event, or stsInputGrab to become the grab object. A grab object should return stsInputGrabTerminate to keep the grab and terminate the input event, or stsInputGrabContinue to both keep the grab and let the event continue through the input registry.

The input system attempts to return the status and event data to the device driver that created the event. This feature allows device drivers to determine when the processing of the event is complete. For example, the pen driver uses this feature to free up the memory it allocates for stroke events.
X-Y Distribution Mechanism

Due to the spatial nature of window-based input, a sophisticated distribution mechanism for processing events is provided. This mechanism allows child windows to pass events to their parent windows for processing. The mechanism is activated for any event with a x-y coordinate other than (minS32, minS32), whether the event is the result of an InputEventGen() call from a device driver, or an InputEventInsert() call from an application.

The input system uses a leaf-to-root model of x-y distribution. This means that when an event occurs in a window, that window receives the event message first. The window processes the message, then terminates the event or allows it to continue. If the window does not terminate the event, the input system sends the message to the window's parent, which in turn processes the message, then either terminates it or allows it to continue. Thus, if no window terminates the event, the input system will send the message to each window from the window in which the input occurred (the leaf window) up the chain to the root window.

The input system uses ObjectSend() to send the input event message to each window in the chain, and it always transforms the event coordinates into the window's local coordinates before sending the message. Thus, the input event appears to penetrate any window that does not terminate it, and is finally processed as if the event had occurred in the same screen position in a window behind the child window.

For more details on the PenPoint™ window system, see Part 3: Windows and Graphics.

Pen Input Sampling

The input system uses the window system to locate the frontmost window which contains the digitizer x-y coordinates for an event, and to access the window tree to determine the window ancestor chain of that frontmost window. In addition, each window has a win.input.flags field to optimize event processing by filtering pen events. The win.input.flags field provides a means for each window to limit what types of pen events it receives.

Chapter 53 describes the possible input flag values. The input flags flags are used for filtering only pen events generated by the standard PenPoint pen driver. Other input events with x-y coordinates will always find their way to the window tree if not sent to a listened or terminated by a filter or grabber.

In addition to the pen event input flags that control which pen events the input system sends to a window, the input system defines a number of special flags that control how the input system samples pen input events. The more interesting of these special flags follow (see PENPOINT\SDK\INC\INPUT.H for the others):

- **inputMoveDelta** controls when msgPenMoveUp and msgPenMoveDown events are sent. With this flag set, the pen will send out move events only when the previous move event has been fully processed. This helps to prevent a backlog of unprocessed move events.
**inputInk** controls inking in the acetate plane. Acetate inking provides immediate feedback when writing with the pen. Because the acetate exists in a different drawing plane than normal windows, it can be erased without affecting the underlying display. When the pen goes out of proximity (or after a brief timeout on hardware that does not support proximity detection), the system erases the acetate plane.

**inputInkThrough** controls inking in the window receiving the input event. With this flag set, the input system deposits ink directly onto the window. Unlike ink in the acetate plane, this ink is not erased on out-of-proximity. The window need not draw the strokes until it becomes dirty and must repaint.

**inputNoBusy** controls whether the system displays the busy clock if the message recipient does not return a status before a certain period of time.

**inputResolution** controls the resolution at which the points are reported. With this flag set, the input system sends pen events when the pen moves at least one digitizer point. With this flag clear, the input system sends pen events are sent when the pen moves at least one screen pixel.

**inputTransparent** forces the window system to ignore the client window when determining the location of an event.
dsInput is a descendant of dsObject that provides the interface between low-level input device drivers and the object-oriented PenPoint™ operating system input registry. Recall from Chapter 50, Introduction, that the input registry is the set of objects that can process input events.

dsInput uses procedural APIs for interacting with device drivers, and message-based APIs for interacting with the input registry.

**Event Data Structure**

When a device driver sends an input event to the input subsystem (with InputEventGen()), it uses an INPUT_EVENT structure to describe the input event. Similarly, when the input subsystem sends msgInputEvent to the objects in the input registry, it also uses an INPUT_EVENT structure. An INPUT_EVENT structure includes the following elements:

- **length** the length of the event packet.
- **flags** various flags affecting the distribution of the input event.
- **devCode** the event type, encoded as a message corresponding to the event. For example, the devCode for a pen-down event is msgPenDown. The Input Subsystem Constants section, below, further discussed the creation of devCode messages.
- **timestamp** the time at which the event entered the input queue, measured in milliseconds since system startup.
- **:x:y** the location of the event local window coordinates (pixels) from the lower-left corner of the window in which the input event occurred. For keyboard events and other events with no specific screen location, the device driver sets both the x and y coordinates to minS32.
- **listener** the listener object. After the input system passes the input event through any input filters or grabbers in the input registry, it routes the input event directly to the listener object.
- **destination** the frontmost window enclosing the input event :x:y location. Currently, the pen driver is the only input device driver to set the destination.
- **originator** the object that generated the event. For example, the pen device driver sets originator to thePen.
- **eventData** an array containing information device-specific event information. For keyboard events, eventData may contain key codes and key translations. Each device driver specifies the contents of the eventData array as appropriate for the particular device.
Device drivers calling InputEventGen() or objects sending msgInputEvent fill in the structure's fields according to the event type. For example, a keyboard event sets the xy field to (minS32, minS32) and uses different devCode values than a pen event.

### Input Subsystem Constants

This section describes the various constants that the input system uses to communicate event types to the input registry.

### Input Subsystem UIDs

PenPoint uses well-known UIDs to refer to the standard pen and keyboard device drivers, and for references to the input subsystem itself.

In each case, the device and class UIDs are synonymous and refer to the same class manager object. The device is instantiated during the boot process—only one instance of it exists in the system.

- theInputManager and clsInput represent the Input subsystem.
- thePen and clsPen represent the pen device.
- theKeyboard and clsKey represent the keyboard device.

The class UIDs are never subclassed. However, they are used to define messages in the class manager macro MakeMsg(), as described in the next section.

### Standard Pen Event Codes

Table 53-1 lists the standard event codes generated by the PenPoint pen device driver.

<table>
<thead>
<tr>
<th>Event</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventTipUp</td>
<td>Pen tip in proximity</td>
</tr>
<tr>
<td>eventTipDown</td>
<td>Pen tip touches the screen.</td>
</tr>
<tr>
<td>eventMoveUp</td>
<td>Pen moved while in proximity.</td>
</tr>
<tr>
<td>eventMoveDown</td>
<td>Pen moved while touching the screen.</td>
</tr>
<tr>
<td>eventEnterUp</td>
<td>Pen entered a window in proximity.</td>
</tr>
<tr>
<td>eventEnterDown</td>
<td>Pen entered a window touching the screen.</td>
</tr>
<tr>
<td>eventExitUp</td>
<td>Pen exited a window in proximity.</td>
</tr>
<tr>
<td>eventExitDown</td>
<td>Pen exited a window touching the screen.</td>
</tr>
<tr>
<td>eventInProxUp</td>
<td>Pen entered proximity while not touching the screen.</td>
</tr>
<tr>
<td>eventOutProxUp</td>
<td>Pen exited proximity while not touching the screen.</td>
</tr>
<tr>
<td>eventStroke</td>
<td>Pen made a stroke.</td>
</tr>
<tr>
<td>eventTap</td>
<td>Pen made a tap.</td>
</tr>
<tr>
<td>eventHoldTimeout</td>
<td>Pen down and hold timed out.</td>
</tr>
</tbody>
</table>
These event codes are not used to set the INPUT_EVENT devCode for pen events. The devCode is a device message derived from the UID of the device object and the event code. For example, when the user touches the pen to the screen, the pen device driver generates an INPUT_EVENT with devCode set to msgPenDown. The PEN.H header file defines msgPenDown with the MakeMsg() macro, using the UID of the pen device and the unique event code:

```c
#define msgPenDown MakeMsg(clsPen, eventTipDown)
```

### Input Event Status Codes

A method for handling an input event must return an event status. For example, suppose your client code receives msgInputEvent and determines that an eventOutProxUp event has occurred. The client switches to a method that does whatever processing is appropriate for that event. That method could return stsInputGrabContinue when it has processed the event to indicate that the client wants to grab subsequent events until further notice and that the input system should continue passing the event through the input registry. Chapter 52, Event Processing, discusses input event status values in more detail.

### Window Input Flags

The creator of a window must set the window input flags to indicate what types of pen input events the input system should send to the window. After creating a window, you can read and modify these flags with msgWinGetFlags and msgWinSetFlags. Some of these flags correspond to a type of pen event; when a window has such a flag set, it receives pen events of the type corresponding to the flag. Other flags indicate a particular type of input system behavior. Table 53-2 summarizes the available pen input flags for windows.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputTip</td>
<td>Enables TipUp and TipDown events.</td>
</tr>
<tr>
<td>inputMoveUp</td>
<td>Enables MoveUp events.</td>
</tr>
<tr>
<td>inputEnter</td>
<td>Enables EnterUp and EnterDown.</td>
</tr>
<tr>
<td>inputExit</td>
<td>Enables ExitUp and ExitDown.</td>
</tr>
<tr>
<td>inputInProx</td>
<td>Enables InProxUp and InProxDown.</td>
</tr>
<tr>
<td>inputStroke</td>
<td>Enables Stroke events.</td>
</tr>
<tr>
<td>inputMoveDown</td>
<td>Enables MoveDown events.</td>
</tr>
<tr>
<td>inputHoldTimeout</td>
<td>Enables holdtimeout events.</td>
</tr>
<tr>
<td>inputTap</td>
<td>Enables tap events.</td>
</tr>
<tr>
<td>inputNoBusy</td>
<td>Disables default busy UI processing.</td>
</tr>
<tr>
<td>inputChar</td>
<td>Enables character events.</td>
</tr>
<tr>
<td>inputMultiChar</td>
<td>Enables multiple character events.</td>
</tr>
<tr>
<td>inputMakeBreak</td>
<td>Enables receipt of make/break events.</td>
</tr>
</tbody>
</table>

These flags enable the dispatch only of pen events.
Flags

Flag Meaning
inputMoveDelta Enables compression of move events.
inputDestOnly Send if destination == self.
inputLRContinue Allows LR dist to continue.
inputDisable Disables input to the window.
inputResolution 0 pixel, 1 digitizer.
inputInk 1 ink on, 0 ink off.
inputInkThrough 1 to ink in the window instead.
inputEnableAll 1 to allow ALL event to be sent to grabbers.
inputAutoTerm 1 to automatically terminate events if Grab doesn't have the event flag enabled.
inputGrabTracker 1 to disable hit detect during grab.
inputInkEnableDisable 1 to disable all pen inking.
inputTransparent 1 invisible to hit detect (WIN).
inputSigVerify 1 to switch to high-speed sampling.

Messages

The input subsystem is primarily procedural, but there are a few messages that it sends to objects in the input registry. If you create a class of object that is intended to handle input, it will need to handle msgInputEvent, and may need to handle some of the other messages summarized in Table 53-3.

Table 53-3 Input System Messages

Message pArgs Description
msgInputEvent P_INPUT_EVENT Requests that the input registry object process the specified input event.
msgInputFilterTerminated NULL Notifies that an input filter is terminated.
msgInputGrabPushed OBJECT Notifies the input registry object that it is losing the input grab to the specified object.
msgInputGrabPopped OBJECT Notifies the input registry object that the specified object is giving up the input grab.
msgInputTargetActivated OBJECT Notifies the input registry object that it is the target object.
msgInputTargetDeactivated OBJECT Notifies the input registry object that it is no longer the target object.
Input Subsystem Procedures

The remainder of the input system API is based on a procedural interface. This section describes all of the input system procedures.

Inserting a Message into the Event Queue

To insert a message into the event queue, call the function `InputEventInsert()`. Applications use `InputEventInsert()` to insert event messages into the input stream and utilize the input system’s distribution mechanisms. You can add an event at the end of the input queue (to be processed after all events already in the queue) or at the head of the queue (to be processed before other events in the queue). The prototype for the function is:

```c
STATUS EXPORTED InputEventInsert(
    P_INPUT_EVENT pEvent, // pointer to the event structure
    BOOLEAN stamp       // true for end of queue, false for head
);
```

`pEvent` is a pointer to an event data structure with at least the `devCode` and `originator` fields filled in.

If you want to add the event to the end of the queue, set `stamp` to `TRUE`. This causes `InputEventInsert()` to stamp the event with the current system clock value.

If you set `stamp` to `FALSE`, `InputEventInsert()` will not stamp the event with the current system clock value. This will insert the input event at the head of the input queue.

Adding a Filter

To add an event to the input filter list, call the function `InputFilterAdd()`. The prototype for the function is:

```c
STATUS EXPORTED InputFilterAdd(
    OBJECT listener,       // object to add to the list
    U8 priority           // priority within the list, 0 high, 255 low
);
```

`listener` is the UID of the filter object to be placed on the filter list.

`priority` is a value from 0 to 255 that indicates the relative priority of the filter. This value specifies the position of the filter in the list.

Filters are inserted into the list in priority order, the higher the priority, the closer to the head of the list. More than one filter can have the same priority; within a single priority, filters are sorted in the order in which they are added.

Removing a Filter

To remove an object from the filter list, call the function `InputFilterRemove()`. The function prototype is:

```c
STATUS EXPORTED InputFilterRemove(
    OBJECT listener       // object to be removed
);
```
listener is the UID of the filter object that you want to remove from the list.

Setting the Input Grabber

To set an object that grabs all input events as they are popped from the event queue, call the function InputSetGrab(). This function grabs input events rather than returning the stInputGrab code from an event. The input system will send msgInputGrabPushed to the prior grab object, if any.

This function may be called during the processing of an input message even if another object currently is grabbing input. The system will correctly create the new grab object after the response for the input message is returned.

The function prototype is:

```c
STATUS EXPORTED InputSetGrab(
    OBJECT grabber,              // object Id for new grabber
    U32 flags                    // flags for use during grab (ORd with windowflags)
);                            //
```

grabber is the object to send input events.

flags are flags specific to the grab object.

If the grab was successful, the function returns stsOK.

Getting Grab Information

To get information about the grab object, call the function InputGetGrab(). The function prototype is:

```c
void EXPORTED InputGetGrab(
    P_OBJECT pGrabber,         // object Id for graber
    P_U32 pFlags               // current Grab flags
);                            //
```

InputGetGrab() sets grabber to the UID of the current grab object, and flags to the current grab flags. If there is no grabber, the function sets grabber to NULL.

Setting the Input Target

To make an object the target of input system messages that are not redirected by filters and grabs, call the function InputSetTarget(). The function has the prototype:

```c
STATUS EXPORTED InputSetTarget(
    OBJECT target,              // new target object
    U32 flags                   // new target flags
);                            //
```

target is the UID of the object that is the new target for input event messages.

flags specifies filter flags for the target object.

Applications do not normally call this function. It is used by the application framework's selection model to redirect input.
Getting the Target

To get the UID of the current Input target, call the function `InputTargetGet()`. The function prototype is:

```
OBJECT EXPORTED InputTargetGet(void);
```

The function has no parameters. It returns the object UID of the current target object.
Chapter 54 / Pen Event

The pen and screen act in concert to produce a series of events based on the windows presently displayed on the screen. When the pen is held out of proximity of the screen, no events are generated. A typical sequence of events generated by the pen include:

1. An in-proximity (msgPenInProxUp) event is generated when the pen first comes within the sensing range of the screen.
2. As the pen moves over the window, move events (msgPenMoveUp) are sent to the window.
3. When the pen touches down in the window, a pen-down event (msgPenDown) is sent to the window.
4. As the pen moves around the window, more move events (msgPenMoveDown) are sent to the window.
5. As the pen crosses the window boundary, a pen exit event (msgPenExitUp or msgPenExitDown) is sent to the exited window, and a pen enter event (msgPenEnterUp or msgPenEnterDown) message is sent to the entered window.
6. When the pen is lifted from the screen, a pen-up event (msgPenUp) is sent to the window.
7. When the pen leaves the sensing range of the screen, a out-proximity (msgOutProxUp) event is sent to the window.

Pen Event Data

This section describes how the fields in the input event data structure are filled in for the pen event types. These definitions are for pen events generated by the standard PenPoint™ pen driver.

The input event data structure for pen events includes the following fields:

- **devCode**: a message representing the type of input event.
- **timestamp**: the time at which the event entered the input queue, measured in milliseconds from system startup.
- **xy**: where on the screen the event occurred, in local window coordinates.
- **listener**: not used for pen events.
- **destination**: the UID of the window to receive the input event. Normally, this is the frontmost window that does not have inputTransparent set.
- **originator**: the object that generated the event. For pen events, the originator is always thePen.
eventData for pen events, a PEN_DATA structure (defined in PEN.H) containing the location of the pen in hardware digitizer coordinates (penXY) and a pointer to any stroke data associated with the event (pStroke).

The following sections describe each of the pen input event types and summarize the event data for each type.

**msgPenDown**

The input subsystem sends msgPenDown when the pen tip changes touched the screen. The event data structure contains the following information:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>devCode</td>
<td>msgPenDown</td>
</tr>
<tr>
<td>timestamp</td>
<td>the time at which the event entered the input queue, measured in milliseconds from system startup.</td>
</tr>
<tr>
<td>xy</td>
<td>where the event occurred, in LWC</td>
</tr>
<tr>
<td>listener</td>
<td>not used</td>
</tr>
<tr>
<td>destination</td>
<td>frontmost window without inputTransparent</td>
</tr>
<tr>
<td>originator</td>
<td>the object that generated the event. For pen events, the originator is always thePen.</td>
</tr>
<tr>
<td>eventData</td>
<td></td>
</tr>
<tr>
<td>penXY</td>
<td>digitizer location of the pen</td>
</tr>
<tr>
<td>pStroke</td>
<td>pNull</td>
</tr>
</tbody>
</table>

**msgPenUp**

The input subsystem sends msgPenUp when the pen tip is lifted from the screen. The event data structure contains the following information:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>devCode</td>
<td>msgPenUp</td>
</tr>
<tr>
<td>timestamp</td>
<td>the time at which the event entered the input queue, measured in milliseconds from system startup.</td>
</tr>
<tr>
<td>xy</td>
<td>where the event occurred, in LWC</td>
</tr>
<tr>
<td>listener</td>
<td>not used</td>
</tr>
<tr>
<td>destination</td>
<td>frontmost window without inputTransparent</td>
</tr>
<tr>
<td>originator</td>
<td>the object that generated the event. For pen events, the originator is always thePen.</td>
</tr>
<tr>
<td>eventData</td>
<td></td>
</tr>
<tr>
<td>penXY</td>
<td>digitizer location of the pen</td>
</tr>
<tr>
<td>pStroke</td>
<td>pNull</td>
</tr>
</tbody>
</table>
msgPenMoveUp and msgPenMoveDown

The input subsystem sends msgPenMoveDown when the pen tip is moved over the surface of the screen. The pen must move at least one pixel to generate these events (see inputResolution). Depending on how quickly the pen moves, the reported points will not necessarily be adjacent pixels. The event data structure contains the following information:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>devCode</td>
<td>msgPenMoveUp or msgPenMoveDown</td>
</tr>
<tr>
<td>timestamp</td>
<td>the time at which the event entered the input queue, measured in milliseconds from system startup.</td>
</tr>
<tr>
<td>xy</td>
<td>where the event occurred, in LWC</td>
</tr>
<tr>
<td>listener</td>
<td>not used</td>
</tr>
<tr>
<td>destination</td>
<td>frontmost window without inputTransparent</td>
</tr>
<tr>
<td>originator</td>
<td>the object that generated the event. For pen events, the originator is always thePen.</td>
</tr>
<tr>
<td>eventData</td>
<td></td>
</tr>
<tr>
<td>penXY</td>
<td>digitizer location of the pen</td>
</tr>
<tr>
<td>pStroke</td>
<td>pNull</td>
</tr>
</tbody>
</table>

msgPenEnterUp and msgPenEnterDown

The input subsystem sends when msgPenEnterUp and msgPenEnterDown when the pen tip enters a window displayed on the screen which has inputEnabled and is not inputTransparent. The event data structure contains the following information:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>devCode</td>
<td>msgPenMoveUp or msgPenMoveDown</td>
</tr>
<tr>
<td>timestamp</td>
<td>the time at which the event entered the input queue, measured in milliseconds from system startup.</td>
</tr>
<tr>
<td>xy</td>
<td>where the event occurred, in LWC</td>
</tr>
<tr>
<td>listener</td>
<td>not used</td>
</tr>
<tr>
<td>destination</td>
<td>frontmost window without inputTransparent</td>
</tr>
<tr>
<td>originator</td>
<td>the object that generated the event. For pen events, the originator is always thePen.</td>
</tr>
<tr>
<td>eventData</td>
<td></td>
</tr>
<tr>
<td>penXY</td>
<td>digitizer location of the pen</td>
</tr>
<tr>
<td>pStroke</td>
<td>pNull</td>
</tr>
</tbody>
</table>
**msgPenExitUp and msgPenExitDown**

The input subsystem sends `msgPenExitUp` and `msgPenExitDown` to the last window that was entered when the pen tip enters another window. The event data structure contains the following information:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>devCode</td>
<td><code>msgPenExitUp</code> or <code>msgPenExitDown</code></td>
</tr>
<tr>
<td>timestamp</td>
<td>the time at which the event entered the input queue, measured in milliseconds from system startup.</td>
</tr>
<tr>
<td>xy</td>
<td>where the event occurred, in LWC</td>
</tr>
<tr>
<td>listener</td>
<td>not used</td>
</tr>
<tr>
<td>destination</td>
<td>frontmost window without <code>inputTransparent</code></td>
</tr>
<tr>
<td>originator</td>
<td>the object that generated the event. For pen events, the originator is always <code>thePen</code>.</td>
</tr>
<tr>
<td>eventData</td>
<td></td>
</tr>
<tr>
<td>penXY</td>
<td>digitizer location of the pen</td>
</tr>
<tr>
<td>pNull</td>
<td></td>
</tr>
</tbody>
</table>

**msgPenInProxUp**

The input subsystem sends `msgPenInProxUp` when the pen tip moves into the sensing range of the screen. The event data structure contains the following information:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>devCode</td>
<td><code>msgPenInProxUp</code></td>
</tr>
<tr>
<td>timestamp</td>
<td>the time at which the event entered the input queue, measured in milliseconds from system startup.</td>
</tr>
<tr>
<td>xy</td>
<td>where the event occurred, in LWC</td>
</tr>
<tr>
<td>listener</td>
<td>not used</td>
</tr>
<tr>
<td>destination</td>
<td>frontmost window without <code>inputTransparent</code></td>
</tr>
<tr>
<td>originator</td>
<td>the object that generated the event. For pen events, the originator is always <code>thePen</code>.</td>
</tr>
<tr>
<td>eventData</td>
<td></td>
</tr>
<tr>
<td>penXY</td>
<td>digitizer location of the pen</td>
</tr>
<tr>
<td>pNull</td>
<td></td>
</tr>
</tbody>
</table>

**msgPenOutProxUp**

The input subsystem sends `msgPenOutProxUp` when the pen leaves the sensing range of the screen. The event data structure contains the following information:
**Table 54-7**

**msgPenOutProxUp Event Data**

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>devCode</td>
<td>msgPenOutProxUp</td>
</tr>
<tr>
<td>timestamp</td>
<td>the time at which the event entered the input queue, measured in milliseconds from system startup.</td>
</tr>
<tr>
<td>xy</td>
<td>where the event occurred, in LWC</td>
</tr>
<tr>
<td>listener</td>
<td>not used</td>
</tr>
<tr>
<td>destination</td>
<td>frontmost window without inputTransparent</td>
</tr>
<tr>
<td>originator</td>
<td>the object that generated the event. For pen events, the originator is always thePen.</td>
</tr>
<tr>
<td>eventData</td>
<td>digitizer location of the pen</td>
</tr>
<tr>
<td>penXY</td>
<td>pNull</td>
</tr>
<tr>
<td>pStroke</td>
<td>pointer to the stroke data, a PEN_STROKE structure (see PEN.H)</td>
</tr>
</tbody>
</table>

**msgPenStroke**

The input subsystem sends `msgPenStroke` between the `msgPenDown` and `msgPenUp` events and passes the collected stroke points (at digitizer resolution). The event data structure contains the following information:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>devCode</td>
<td>msgPenStroke</td>
</tr>
<tr>
<td>timestamp</td>
<td>the time at which the event entered the input queue, measured in milliseconds from system startup.</td>
</tr>
<tr>
<td>xy</td>
<td>where the event occurred, in LWC</td>
</tr>
<tr>
<td>listener</td>
<td>not used</td>
</tr>
<tr>
<td>destination</td>
<td>frontmost window without inputTransparent</td>
</tr>
<tr>
<td>originator</td>
<td>the object that generated the event. For pen events, the originator is always thePen.</td>
</tr>
<tr>
<td>eventData</td>
<td>digitizer location of the pen</td>
</tr>
<tr>
<td>penXY</td>
<td>pNull</td>
</tr>
<tr>
<td>pStroke</td>
<td>pointer to the stroke data, a PEN_STROKE structure (see PEN.H)</td>
</tr>
</tbody>
</table>

**msgPenTap**

The input subsystem sends `msgPenTap` when the pen tip goes Down-Up within a fixed amount of time. Multiple taps are counted and the count is passed in a subfield of `eventData`. This event is generated in addition to the normal `msgPenDown/msgPenUp` sequence. The event data structure will have the following information:
### msgPenTap Event Data

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>devCode</td>
<td>msgPenTap</td>
</tr>
<tr>
<td>timestamp</td>
<td>the time at which the event entered the input queue, measured in milliseconds from system startup.</td>
</tr>
<tr>
<td>xy</td>
<td>where the event occurred, in LWC</td>
</tr>
<tr>
<td>listener</td>
<td>not used</td>
</tr>
<tr>
<td>destination</td>
<td>frontmost window without inputTransparent</td>
</tr>
<tr>
<td>originator</td>
<td>the object that generated the event. For pen events, the originator is always thePen.</td>
</tr>
<tr>
<td>eventData</td>
<td></td>
</tr>
<tr>
<td>penXY</td>
<td>digitizer location of the pen</td>
</tr>
<tr>
<td>taps</td>
<td>penTapCount, the number of taps before timeout</td>
</tr>
</tbody>
</table>
Chapter 55 / Keyboard Events

Unlike pen events, keyboard events are not associated with a particular screen location. When keyboard events are routed through the input subsystem, they typically are sent to the input target object maintained by the selection manager.

**Keyboard Event Data**

This section describes how the fields in the event data structure are filled in for the keyboard event types. The input event data structure for keyboard events includes the following fields:

- **devCode**: a message representing the type of input event.
- **timestamp**: the time at which the event entered the input queue, measured in milliseconds from system startup.
- **xy**: Keyboard events have no associated screen location. To indicate this, the x and y coordinates of xy are both set to minS32 for keyboard events.
- **listener**: not used for keyboard events.
- **destination**: the object which is to receive the event. For keyboard events, **destination** is always null. The selection manager keeps track of which object is to receive keyboard events.
- **originator**: the object which generated the event.
- **eventData**: a KEY_DATA structure (defined in KEY.H) containing information about the keyboard event, such as the keycode of the pressed key.

For keyboard events, the **eventData** field of the event data structure is a KEY_DATA structure (defined in KEY.H). The KEY_DATA structure, which is specific to keyboard events, contains the following fields:

- **keyCode**: the ASCII value associated with the key.
- **scanCode**: the scan code that the keyboard associates with the key. This value is specific to the type of keyboard, and therefore any code which relies on it is hardware-dependent.
- **shiftState**: the state of the shift, control, alt, and other keyboard modifiers at the time the key was pressed. The various shift states are defined in KEY.H.
- **repeatCount**: the number of times this ASCII character character has been repeated. In the case of msgKeyMulti events, the number of entries in the multi array.
- **multi**: an array of KEY_MULTI structures, representing a number of keypresses queued for processing in a single event. A KEY_MULTI structure includes every field that a KEY_DATA structure includes, except for the multi field (see KEY.H for more details).
The following sections describe each of the pen input event types and summarize the event data for each type.

**msgKeyDown**

The input subsystem sends **msgKeyDown** when a key is depressed. The **msgKeyDown** event data structure contains the following information:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>devCode</td>
<td>msgKeyDown</td>
</tr>
<tr>
<td>timestamp</td>
<td>time the event entered the input queue, in milliseconds since system startup</td>
</tr>
<tr>
<td>xy</td>
<td>(minS32, minS32)</td>
</tr>
<tr>
<td>listener</td>
<td>not used</td>
</tr>
<tr>
<td>destination</td>
<td>null</td>
</tr>
<tr>
<td>originator</td>
<td>object which generated the event</td>
</tr>
<tr>
<td>eventData</td>
<td></td>
</tr>
<tr>
<td>keyCode</td>
<td>null</td>
</tr>
<tr>
<td>scancode</td>
<td>scan code for the key</td>
</tr>
<tr>
<td>shiftState</td>
<td>shift state when the key was pressed</td>
</tr>
<tr>
<td>repeatCount</td>
<td>1</td>
</tr>
<tr>
<td>multi</td>
<td>null</td>
</tr>
</tbody>
</table>

**msgKeyUp**

The input subsystem sends **msgKeyUp** when a key is released. The **msgKeyUp** event data structure contains the following information:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>devCode</td>
<td>msgKeyUp</td>
</tr>
<tr>
<td>timestamp</td>
<td>time the event entered the input queue, in milliseconds since system startup</td>
</tr>
<tr>
<td>xy</td>
<td>(minS32, minS32)</td>
</tr>
<tr>
<td>listener</td>
<td>not used</td>
</tr>
<tr>
<td>destination</td>
<td>null</td>
</tr>
<tr>
<td>originator</td>
<td>object which generated the event</td>
</tr>
<tr>
<td>eventData</td>
<td></td>
</tr>
<tr>
<td>keyCode</td>
<td>null</td>
</tr>
<tr>
<td>scancode</td>
<td>scan code for the key</td>
</tr>
<tr>
<td>shiftState</td>
<td>shift state when the key was pressed</td>
</tr>
<tr>
<td>repeatCount</td>
<td>1</td>
</tr>
<tr>
<td>multi</td>
<td>null</td>
</tr>
</tbody>
</table>
Keyboard Event Data

**msgKeyChar**

The input subsystem sends `msgKeyChar` after the `msgKeyDown` event; the message yields the translation of the key scan code and the shift state into an ASCII key code. This event is sent whenever a new key is depressed and whenever the processing for the same key has completed. If the ASCII key code is the same as the last one reported, the `repeatCount` value is incremented. The `msgKeyChar` event data structure contains the following information:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>devCode</td>
<td><code>msgKeyChar</code></td>
</tr>
<tr>
<td>timestamp</td>
<td>time the event entered the input queue, in milliseconds since system startup</td>
</tr>
<tr>
<td>xy</td>
<td><code>(minS32, minS32)</code></td>
</tr>
<tr>
<td>listener</td>
<td>not used</td>
</tr>
<tr>
<td>destination</td>
<td>null</td>
</tr>
<tr>
<td>originator</td>
<td>object which generated the event</td>
</tr>
<tr>
<td>eventData</td>
<td></td>
</tr>
<tr>
<td>keyCode</td>
<td>ASCII character associated with the key</td>
</tr>
<tr>
<td>scancode</td>
<td>scan code for the key</td>
</tr>
<tr>
<td>shiftState</td>
<td>shift state when the key was pressed</td>
</tr>
<tr>
<td>repeatCount</td>
<td>number of repeats since last <code>msgKeyChar</code></td>
</tr>
<tr>
<td>multi</td>
<td><code>null</code></td>
</tr>
</tbody>
</table>

**msgKeyMulti**

The input subsystem sends `msgKeyMulti` after the `msgKeyChar`; the message is used to group sequences of characters into a single event. This single event summarizes all `msgKeyChar` events since the last time that a `msgKeyMulti` event was sent. Because the `msgKeyMulti` event duplicates the information in the `msgKeyChar` events, process either `msgKeyChar` or `msgKeyMulti` messages, not both. The `msgKeyMulti` event data structure contains the following information:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>devCode</td>
<td><code>msgKeyMulti</code></td>
</tr>
<tr>
<td>timestamp</td>
<td>time the event entered the input queue, in milliseconds since system startup</td>
</tr>
<tr>
<td>xy</td>
<td><code>(minS32, minS32)</code></td>
</tr>
<tr>
<td>listener</td>
<td>not used</td>
</tr>
<tr>
<td>destination</td>
<td>null</td>
</tr>
<tr>
<td>originator</td>
<td>object which generated the event</td>
</tr>
<tr>
<td>eventData</td>
<td></td>
</tr>
<tr>
<td>keyCode</td>
<td><code>null</code></td>
</tr>
<tr>
<td>scancode</td>
<td><code>null</code></td>
</tr>
<tr>
<td>shiftState</td>
<td><code>null</code></td>
</tr>
<tr>
<td>repeatCount</td>
<td><code>null</code></td>
</tr>
<tr>
<td>multi</td>
<td>array of <code>KEY_MULTI</code> structures</td>
</tr>
</tbody>
</table>
Chapter 56 / Using clsIP

clsIP is called to create insertion pads for use as UI components in applications. Insertion pads are windows that accept pen input, accumulating the strokes in a scribble object and handing the accumulated stroke information to a translation object when the user has finished entering input.

Topics covered in this chapter are:
- Creating and destroying insertion pads.
- Inserting an insertion pad into a parent window.
- Deleting an insertion pad.
- Handling the translation data.

### clsIP Messages

clsIP inherits from clsBorder. Table 56-1 summarizes the messages clsIP defines.

<table>
<thead>
<tr>
<th>Message</th>
<th>pArgs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>IP_NEW</td>
<td>Creates a new insertion pad object.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_IP_NEW</td>
<td>Initializes the IP_NEW structure to default values.</td>
</tr>
<tr>
<td>msgIPGetStyle</td>
<td>P_IP_STYLE</td>
<td>Passes back the style flags for the IP.</td>
</tr>
<tr>
<td>msgIPSetStyle</td>
<td>P_IP_STYLE</td>
<td>Changes the style flags for the IP.</td>
</tr>
<tr>
<td>msgIPGetTranslator</td>
<td>P_OBJECT</td>
<td>Passes back the translator for the IP.</td>
</tr>
<tr>
<td>msgIPSetTranslator</td>
<td>P_OBJECT</td>
<td>Sets the translator for the IP.</td>
</tr>
<tr>
<td>msgIPGetClient</td>
<td>P_OBJECT</td>
<td>Passes back the client of the IP.</td>
</tr>
<tr>
<td>msgIPSetClient</td>
<td>P_OBJECT</td>
<td>Sets the client of the IP.</td>
</tr>
<tr>
<td>msgIPSetString</td>
<td>P_CHAR</td>
<td>Stores a string into the IP.</td>
</tr>
<tr>
<td>msgIPTranslate</td>
<td>BOOLEAN</td>
<td>Forces the translation of scribbles in the IP.</td>
</tr>
<tr>
<td>msgIPCancelled</td>
<td>OBJECT</td>
<td>Indicates cancellation of the IP.</td>
</tr>
<tr>
<td>msgIPClear</td>
<td>OBJECT</td>
<td>Clears the value in an IP.</td>
</tr>
</tbody>
</table>

Table 56-1 clsIP Messages
Creating an Insertion Pad

Normally, you create an insertion pad as you build up and initialize the user interface for your application. You send msgNew to clsIP to get a new insertion pad object, and msgWinInsert to insert the insertion pad as a child window of one of the windows in the UI for your application. Typically, you will insert the new insertion pad into a custom layout.

If the insertion pad is created in response to user activity, it will be inserted into the window tree as a child of the window where the activity took place, causing it to "pop up" on the display.

To create a new insertion pad, you must initialize an IP_NEW typed structure variable by sending it along with msgNewDefaults to clsIP. When the filled-in structure is returned you may modify the fields to get the insertion pad behavior that is appropriate for your application.
The most interesting fields to modify for an insertion pad are the ones that control the style of the insertion pad. Of these, the ip.style.displayType field creates the most common visible aspect of the insertion pad. Table 56-2 lists the possible values of the ip.style.displayType field.

<table>
<thead>
<tr>
<th>Style Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ipsRuledLines</td>
<td>Standard ruled lines.</td>
</tr>
<tr>
<td>ipsCharBox</td>
<td>Character boxes.</td>
</tr>
<tr>
<td>ipsCharBoxButtons</td>
<td>Same as ipsCharBox.</td>
</tr>
<tr>
<td>ipsSignature</td>
<td>Signature pad.</td>
</tr>
<tr>
<td>ipsBlank</td>
<td>Same as ipsSignature.</td>
</tr>
</tbody>
</table>

### Displaying the Insertion Pad

After you create the insertion pad, you must insert it into a window tree to make it visible. To insert the IP into a window tree, set its wm.parent to the intended parent window and send it msgWinInsert. The following code fragment shows how to insert a new insertion pad object into the main window of an application:

```c
// Insert the pad into the window.
//
// pInst->iPad   = ipNew.object.uid;
// wm.parent     = pInst->parentWin;
// wm.options    = wsPosTop;
// ObjCallRet(msgWinInsert, ipNew.object.uid, &wm, s);
```

When `exprWin` is displayed, the insertion pad will be displayed in front of it (`wsPosTop`). This is how the parent/child window relationship works: the status of the parent window (displayed or not) determines the status of the child window.

### Deleting an Insertion Pad

To extract an insertion pad from the display, send `msgWinExtract` to the IP object. To make the insertion pad appear on the screen again, send `msgWinInsert` to the object. These `clsWin` messages are described in Part 3: Windows and Graphics. To destroy the object completely and free the memory used by the instance, send `msgDestroy` to the object. The consequences of destroying an object are discussed in Part 1: Class Manager.

### Setting the Translator Object

When you are creating an insertion pad, you should have at hand an already created translation object so that you can pass its UID in the `msgNew` structure for the new insertion pad.
You can also dynamically change the translator object for an insertion pad. For example, you might want to change from alphabetic character translation to numeric translation based on some mode in your form. To do this you must first create the new translator object, and then send its UID in the pArgs (which is a simple P_OBJECT structure) for msgSetTranslator to the insertion pad Instance.

### Handling Xlist Data

Your application must be ready to handle msgIPDataAvailable which is a clsIP message. The insertion pad and translation objects are designed to integrate well together, so you don’t need to oversee the input control and data handling that occurs between them during a translation. When you create an insertion pad you register the client in the ip.client field. Typically, you enter self (your application) in this field. This registers the client in a notification list for the insertion pad object when it is created. The insertion pad is in turn registered in the notification list for the translator object that is attached to it. When the translator is done it notifies the insertion pad which in turn notifies the client that the translated input is ready.

The data is returned in the form of a dynamic array called an Xlist. The IP_XLATE_DATA data type is used to handle the specific type of Xlist handed back by an insertion pad.

After being notified the data is ready with msgIPDataAvailable, the client must send msgIPGetXlateData to the insertion pad. msgIPGetXlateData takes an IP_XLATE_DATA structure as its pArgs.

Once you have received the data back in the IP_XLATE_DATA structure, Xlist.H defines several functions that you can use to decompose the raw Xlist. Table 56-3 lists the more important functions for extracting Xlist information:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XlistTraverse()</td>
<td>Iterate across the list of elements.</td>
</tr>
<tr>
<td>XlistGetPtr()</td>
<td>Return a pointer to a specific element.</td>
</tr>
<tr>
<td>XlistGet()</td>
<td>Return a copy of a specific element.</td>
</tr>
<tr>
<td>Xlist2Gesture()</td>
<td>Extracts the gestures from an Xlist.</td>
</tr>
<tr>
<td>Xlist2String()</td>
<td>Extracts the text from an Xlist.</td>
</tr>
</tbody>
</table>

Most Xlist functions are described in Chapter 60, Using Xlists. SAMPLEADDER, a sample program included with the PenPoint™ Software Developer's Kit, provides a good example of the use of insertion pads and translators.
Chapter 57 / Using clsSPaper

clsSPaper (scratch paper) is called to create a view object that will both display a user interface window that accepts pen input and manage the collected input in a data object called a scribble. Additionally, instances of clsSPaper have the built-in facility to attach and manage a translator object from the handwriting translation classes.

Topics covered in this chapter are:

- clsSPaper messages.
- Subclassing clsSPaper in your application.
- Rendering feedback with a drawing context.
- Handling the translation message cycle.

clsSPaper Messages

clsSPaper inherits from clsView. Table 57-1 summarizes the messages defined by clsSPaper.

<table>
<thead>
<tr>
<th>Message</th>
<th>Takes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNew</td>
<td>P_SPAPER_NEW</td>
<td>Creates a new clsSPaper object.</td>
</tr>
<tr>
<td>msgNewDefaults</td>
<td>P_SPAPER_NEW</td>
<td>Initializes the SPAPER_NEW structure to default values.</td>
</tr>
<tr>
<td>msgSPaperGetFlags</td>
<td>P_U16</td>
<td>Passes back the flags.</td>
</tr>
<tr>
<td>msgSPaperSetFlags</td>
<td>P_U16</td>
<td>Sets the flags.</td>
</tr>
<tr>
<td>msgSPaperGetCellMetrics</td>
<td>P_SPAPER_CELL_METRICS</td>
<td>Passes back the metrics for the internal divisions.</td>
</tr>
<tr>
<td>msgSPaperGetCellMetrics</td>
<td>P_SPAPER_CELL_METRICS</td>
<td>Changes the cell metrics and resizes window.</td>
</tr>
<tr>
<td>msgSPaperGetSizes</td>
<td>P_SIZE16</td>
<td>Passes back the line height and character width sizes, in points.</td>
</tr>
<tr>
<td>msgSPaperSetSizes</td>
<td>P_SIZE16</td>
<td>Sets the line height and character width sizes, in points.</td>
</tr>
<tr>
<td>msgSPaperGetTranslator</td>
<td>P_OBJECT</td>
<td>Passes back the translator object uid.</td>
</tr>
<tr>
<td>msgSPaperSetTranslator</td>
<td>P_OBJECT</td>
<td>Replaces the translation object, passes back the old translator.</td>
</tr>
<tr>
<td>msgSPaperGetScribble</td>
<td>P_OBJECT</td>
<td>Passes back the scribble object (may be NULL).</td>
</tr>
</tbody>
</table>

57.1

Table 57-1
clsSPaper Messages
**clsSPaper Facilities**

**clsSPaper** provides general facilities for accepting pen input on the screen, handling the collected data, performing a translation on the pen data, and rendering user feedback. Other more specialized classes such as clsIP use **clsSPaper**’s general functionality, and then wrap a useful component or toolkit interface around it. With clsIP you only need to set a style flag to get various forms of pen input capture and translation. Using these **clsSPaper**-based components makes it easier for you to build advanced input capabilities in the user interface for your application.

**clsSPaper** maintains a scribble object and a translator object. As a descendent of clsView, it displays a window and the data object that it maintains is the scribble. It then orchestrates interaction with the input subsystem so that pen strokes are accumulated in the scribble data object.

**clsSPaper** manages the window display with options that will display rule lines or character boxes to aid handwriting input. Because **clsSPaper** maintains the scribble data object, you can ask **clsSPaper** to redisplay the user’s original digitized input.
Whenever the user terminates input at the screen (different styles of termination are options that are settable) clsSPaper passes the accumulated scribble data to a translator object. When the translator object has completed its pass on the data, clsSPaper notifies the client that the translation data is available for processing.

**Examples**

To use clsSPaper directly in your application UI, you should create a subclass of clsSPaper that adds the particulars necessary for the capture and translation of pen input. Your subclass will need to add:

- Rendering for user feedback in the window.
- clsSPaper options for input and translation styles.

This section provides examples showing the steps to do this using the sample program WriterApp. The application source files are part of the PenPoint™ SDK distribution, in the directory `\PENPOINT\SDK\SAMPLE\WRITERAP`.

---

### Using a Drawing Context to Render Visual Feedback

Your clsSPaper-based window will need to show some sort of feedback as the user writes input in it. This example shows how WriterApp sets up a drawing context (DC) so that it will print text characters in the window after the translation is done.

**WriterCompleted()** is the message handler for **msgSPaperXLateCompleted**, which clsSPaper self-sends when the user has completed input and the translation object has finished the translation. You will see in a moment how to set up a window as a subclass of clsSPaper, so that self (the window object) inherits the capability to self-send msgSPaperXLateCompleted.

```c
/****************************************************************************
** WriterCompleted                                                          **
** Called when the translation is complete. Will get xlist from the spaper **
** (self), set up a dc to draw the text, and traverse the xlist to draw the **
** text.                                                                  **
****************************************************************************/

MsgHandler(WriterCompleted){
    XLATE_DATA xData;
    SHOW_DATA show;
    SYSDC_NEW dcNew;
    STATUS s;
    // Get the translated data from the spaper (self)
    xData.heap = osProcessHeapId;
    ObjCallRet(msgSPaperGetXlateData, self, &xData, s);
    // Convert the xlist form bounds/word/bounds/word pairs to
    // bounds/text/bounds/text
    XList2Text(xData.pXList);
    // Create a drawing context to paint the text and initialize
    ObjectCall(msgNewDefaults, clsSysDrwCtx, &dcNew);
    ObjCallRet(msgNew, clsSysDrwCtx, &dcNew, s);
    ObjectCall(msgDcSetWindow, dcNew.object.uid, self);
    ObjectCall(msgDcUnitsDevice, dcNew.object.uid, Nil(P_ARGS));

```

---

(Continued)
Parsing the Xlist Data

The first thing that `WriterCompleted()` does is ask for the translated data in the form of an Xlist. Because `clsSPaper` automatically manages a scribble data object and a translator object, you don’t need to worry about these. The method simply sends `msgSPaperGetXLateData` to the client object `self`. The message will find its way up to the ancestor `clsSPaper` and be processed correctly. The data is returned and further formatted by being passed through the `Xlist2Text` filter function.

Rendering the Translated Text

Then `WriterCompleted()` sends a series of messages to `clsSysDrwCtx` to create and initialize a DC. The local `SetUpFont()` function opens and sets the characteristics of the font for the DC. Finally, the DC is activated when the Xlist is traversed with the `XlistTraverse` function. This call contains a pointer to the local `ShowText()` function which actually pulls the characters from the Xlist and plugs them into the DC. The window repaints whenever data is released to the DC, because `msgWinBeginRepaint` was sent to the DC giving it the right to repaint the window with new rendering.

Subclassing `clsSPaper`

`WriterApp`’s main window is a view object that incorporates both the inherited functionality from `clsSPaper` as well as the customization in `WriterCompleted()`. To do this requires a subclass of `clsSPaper` with additional behavior. The following example from `WriterApp`’s `WriterInit()` function shows how to create a subclass of `clsSPaper` called `clsWriter`. 
**Creating a Subclass of clsSPaper**

Now that you have defined this special view class, you need to build the application shell that will run it. The application provides hooks into the PenPoint application environment—it registers the application as an entry in the stationery menu, provides a frame with a title bar and close corner, and in this example it provides a local menu bar that allows the user to select the type of translation object to use.

### Example 57-2
**Creating a Subclass of clsSPaper**

This example shows how to create a subclass of clsSPaper. It starts by sending msgNewDefaults to clsClass, assigning the CLASS_NEW variable c to contain the defaults for all new classes. The following statements set the values specific to the new class, such as its UID (clsWriter) and ancestor in the class hierarchy (clsSPaper). Finally, it uses this completed CLASS_NEW structure as the argument to msgNew, to which clsClass responds with the new class.

```c
/*******************************
WriterInit
Installs the clsWriter class. This is a subclass of sPaper.
*******************************/
STATUS GLOBAL WriterInit(void)
{
    CLASS_NEW c;
    ObjectCall(msgNewDefaults, clsClass, &c);
    c.object.uid = clsWriter;
    c.object.cap |= objCapCall;
    c.object.key = (OBJ_KEY)clsWriter;
    c.cls.pMsg = clsWriterTable;
    c.cls.ancestor = clsSPaper;
    c.cls.size = Nil(SIZEOF); // no instance data
    c.cls.newArgsSize = SizeOf(SPAPER_NEW);
    return ObjectCall(msgNew, clsClass, &c);
}
```

### Example 57-3
**Initializing a clsSPaper-Based Main Window**

This code fragment shows how the application initializes its user interface. It creates an instance of clsWriter to be used as the main window in the application. It also sets up a menu for selecting translation styles, and creates a first default translator object to be attached to the clsWriter instance when the application is opened. WriterAppInit() is called whenever the user launches an instance of the application.

```c
/*******************************
WriterAppInit
Initialize the display for the first time.
*******************************/
MSG_HANDLER WriterAppInit(
    const MESSAGE msg,
    const OBJECT self,
    const P_ARGS pArgs,
    const CONTEXT ctx,
        continued
```
Setting the clsSPaper Input Flags

One of the first things WriterAppInit() does is send msgNew to clsWriter to get a new instance of your special view class. As a part of the pArg for this message, the application sets several flags that control the input termination style and pen inking behavior. The flags are defined in SPAPER.H. Table 57-2 lists the clsSPaper input flags briefly to show what is available.
### Table 57-2

<table>
<thead>
<tr>
<th>Input Flag</th>
<th>Meaning when set</th>
</tr>
</thead>
<tbody>
<tr>
<td>spCapture</td>
<td>save scribbles between translations</td>
</tr>
<tr>
<td>spProx</td>
<td>out of proximity calls msgSPaperComplete</td>
</tr>
<tr>
<td>spFixedPos</td>
<td>top left is fixed during resize</td>
</tr>
<tr>
<td>spPenCoords</td>
<td>window-relative pen coordinates from Xlist (for xtPolyline and xtSpline only)</td>
</tr>
<tr>
<td>spGrab</td>
<td>grab input on penDown, release after msgPaperAbort or msgPaperComplete</td>
</tr>
<tr>
<td>spScribbleEdit</td>
<td>enable scribble editing</td>
</tr>
<tr>
<td>spRedisplay</td>
<td>redraw and display strokes</td>
</tr>
<tr>
<td>spSuppressMarks</td>
<td>disable spRuling, spVRuling, spTick and spGrid display</td>
</tr>
<tr>
<td>spRuling</td>
<td>enable horizontal ruling</td>
</tr>
<tr>
<td>spVRuling</td>
<td>enable vertical ruling</td>
</tr>
<tr>
<td>spGrid</td>
<td>enable grid lines (with spRuling)</td>
</tr>
<tr>
<td>spBaseLine</td>
<td>make horizontal rules a baseline</td>
</tr>
<tr>
<td>spTick</td>
<td>enable tick marks (with spRuling)</td>
</tr>
</tbody>
</table>

After building the application menu (the menu table entries are defined at the top of WRITERAP.C), WriterAppInit() plugs all of the UI elements into the frame metrics for the application. The clsWriter instance is inserted as the application’s main window with the line:

```c
fm.clientWin = sNew.object.uid;
```

The metrics are set when the method sends `msgFrameSetMetrics`.

### Dynamically Setting the clsPaper Translator

The default translator object is set when `WriterAppInit()` sends `msgWriterAppTranslator` to self. This invokes a method `WriterAppTranslator()` that dynamically sets the translation object in the clsWriter instance.

**Example 57-4**

**Setting the Translator for a clsPaper Object**

This code fragment shows the message handler `WriterAppTranslator()`. This method is called whenever the user selects a translation style from WriterApp’s Translator menu. It is also called by the initialization method `WriterAppInit()` when the application is opened so that the view object will have a default translator when the UI is presented to the user.

Depending on the selection that the user has made in the Translator menu, this method will create a new clsXWord or clsXTxt object and pass the new object’s UID to the view object with `msgPaperSetTranslator`.

Because clsWriter is a subclass of clsPaper, and has its spRedisplay flag set, the view object automatically keeps the scribble data around between translations. This makes it possible for the user to toggle between translators and retranslate the current input. This is done with the line:

```c
ObjectCall(msgPaperComplete, fm.clientWin, null);
```
This in effect simulates an input termination to the view object which then begins the translation using the old scribble data.

/* ********************************************
* WriterAppTranslator
* Called when the translator is changed from the menu in the application.
* Will create a new translator based on which menu item was chosen and
* and give to the spaper. Will cause translation to re-occur.
* *********************************************/

MsgHandler(WriterAppTranslator)
{
    APP_METRICS am;
    FRAME_METRICS fm;
    XLATE_NEW xNew;
    STATUS s;
    OBJECT oldTranslator;
    ObjCallRet(msgAppGetMetrics, self, &am, s);
    ObjCallRet(msgFrameGetMetrics, am.mainWin, &fm, s);
    switch ((U16)(U32)pArgs) {
        case 0:
            // clsXWord
            ObjectCall(msgNewDefaults, clsXWord, &xNew);
            ObjCallRet(msgNew, clsXWord, &xNew, s);
            break;
        case 1:
            // clsXText with only numbers
            ObjectCall(msgNewDefaults, clsXText, &xNew);
            xNew.xlate.charConstraints =
                disableUpperCase | disableLowerCase
                | disableCommonPunct | disableOtherPunct;
            ObjectCall(msgNew, clsXText, &xNew);
            break;
        case 2:
            // clsXText
            ObjectCall(msgNewDefaults, clsXText, &xNew);
            ObjectCall(msgNew, clsXText, &xNew);
            break;
        default:
            return stsOK;
    }
    // dirty everything for the repaint
    ObjectCall(msgWinDirtyRect, fm.clientWin, Nil(P_ARGS));
    // force the repaint
    ObjectCall(msgWinUpdate, fm.clientWin, Nil(P_ARGS));
    ObjCallRet(msgSPaperGetTranslator, fm.clientWin, &oldTranslator, s);
    ObjCallRet(msgSPaperSetTranslator, fm.clientWin, &xNew.object.uid, s);
    // Free the old translator
    if (oldTranslator != objNull)
        ObjectCall(msgDestroy, oldTranslator, Nil(P_ARGS));
    // force retranslation
    ObjectCall(msgSPaperComplete, fm.clientWin, Nil(P_ARGS));
    return stsOK;
    MsgHandlerParametersNoWarning;
}
Chapter 58 / Using the Translation Classes

The translation classes segment the strokes of a scribble object into sets of one or more strokes and translate the sets into computer-readable characters or other meaningful content. Different translation classes focus on various types of expected input such as gestures, letters, numbers, or whole words. The appropriate translator object can be created by the client according to the context of the input session.

A translation object can accept stroke data from a scribble as the user enters strokes, or it can wait until the user has completed the input and then translate the accumulated stroke data. Translator objects return gesture IDs or ASCII character data to the client in a dynamic array called an Xlist.

The client object that manages the input window, scribble data object, and translator object is typically an instance of clsView or one of its descendants, such as clsSPaper. For example, an input pad manages an input window for the purpose of gathering scribble data from the user. The input pad also manages a translator object, to which it passes the scribble data for translation.

Topics covered in this chapter are:

- Translation class hierarchy.
- Translation class data structures.
- Handwriting translation flags.
- Translation templates.
- Translation class messages.

In the current version of PenPoint™, the translation classes are designed for American English input. Future versions of PenPoint with international language support will use an improved approach to handwriting translation.
Hierarchy of the Translation Classes

The translation classes all descend from clsXtract, which contains the functional interface to the feature extraction and recognition engine. Figure 58-1 shows the inheritance hierarchy of the translation classes.

Translation Data Structures

clsXtract defines several data structures that serve as arguments to its messages. Some of the data structures, such as the XLATE_METRICS data structure, are composed of a collection of smaller, lower level data structures. This section describes each of these data structures.

XLATE_METRICS Structure

The XLATE_METRICS structure contains information about the insertion pad to which the translator is attached. The insertion pad sets the values in this structure when the translator is attached with msgIPSetTranslator. The XLATE_METRICS structure contains the following fields (the types of each field appear in parentheses following the name):

- lineCount (U16) number of lines in the insertion pad. If this value is zero, the number of lines is not fixed.
charCount (U16) number of character columns in the insertion pad. If this value is zero, the number of columns is not fixed.

charBox (SIZE32) height and width of character box.

baselineOffset (S32) baseline offset to bottom of char box (if charCount is not zero).

**XLate_CASE_METRICS Structure**

Translators can use “smart case” heuristics for handling letter case. If the smartCaseDisable flag is not set (see “Handwriting Translation Flags,” below), then the translator uses the information in its XLate_CASE_METRICS structure to determine how to capitalize words.

type (XLate_CASE_TYPE) whether to translate as a field or sentence.

Possible values for type include:
- xcmNone force everything to lower case.
- xcmSentence capitalize the first letter of each sentence. Use the context.sentence value to determine letter case heuristics.
- xcmField use the context.field value to determine letter case heuristics.

writer (XLate_CASE_WRITER) whether to expect all uppercase or mixed-case text. This normally reflects the user preference set in the Settings notebook. Possible values for writer include:
- xcmMixedCaseWriter writer uses mixed upper and lower case.
- xcmAllCapsWriter writer uses all upper case letters.

context (union of SPELL_CASE_CONTEXT and XLate_CASE_FIELD) context guidelines for translating text. If type is xcmSentence, use context.sentence, a SPELL_CASE_CONTEXT structure as defined in SPELL.H. If type is xcmField, use context.field, an XLate_CASE_FIELD structure with the following possible values:
- xcmOneInitialCapField capitalize first letter in the field.
- xcmAllInitialCapsField capitalize first letter in each word.
- xcmAllCapsField capitalize all letters in the field.

**XLate_NEW Structure**

The XLate_NEW structure includes an OBJECT_NEW_ONLY structure called object, through which clsXtract inherits from clsObject. For information specific to clsXtract, the XLate_NEW structure includes an XLate.NEW_ONLY structure called xlate. The following section describes the XLate.NEW_ONLY structure.

**XLate.NEW_ONLY Structure**

The XLate.NEW structure includes an XLate.NEW_ONLY structure called xlate. xlate includes the following fields:
hwxFlags (U32) a set of flags representing translation rules (see msgXlateFlagsSet). "Handwriting Translation Flags," below, discusses the flags in more detail.

ccharConstraints (U16) flags constraining the characters the writer is expected to write. This may improve translation accuracy because the shape matcher will know that it does not need to consider certain characters as possibilities. The possible constraints (which you may combine) include:

- xltDisableUpperCase disallow A through Z.
- xltDisableLowerCase disallow a through z.
- xltDisableNumerals disallow 0 through 9.
- xltDisableCommonPunct disallow ,?%;$#+-/*().
- xltDisableOtherPunct disallow all other punctuation.

metrics (XLATE_METRICS) information about the insertion pad to which the translator is attached. See “The XLATE_METRICS Structure,” above, for details.

pTemplate (p_UNKNOWN) a pointer to a compiled translation template, or pNull if none. See “Translation Templates,” below, for details.

xlateCaseMetrics (XLATE_CASE_METRICS) case post-processing controls. See “The XLATE_CASE_METRICS Structure,” above, for details.

Handwriting Translation Flags

A translator receives a scribble from an input pad, then segments the scribble into sets of one or more strokes. The translator translates the sets of strokes and associates a score, a measure of confidence in the translation, with the translation. Translators use a number of rules in scoring a translation, giving higher scores to translations that obey the rules. One of the fields in an XLATE_NEW structure is a U32 called xlate.hwxFlags, which contains a variety of flags that govern which rules to use in this process. You can think of these rules as falling into three categories:

- **Built-In Rules** rules involving translation abilities built into the translator.
- **Knowledge-Source Rules** rules involving external sources of information such as a spelling checker or template.
- **Post-Processing Rules** rules involving changes to the text after it is translated.

The following sections discuss each of these types of rules and the xlate.hwxFlags flags associated with them.

Built-In Rules

The built-in rule flags direct the translation object use various default language rules to assist recognition. When a flag is turned on, the translator will show a preference for translations which obey the rule associated with that flag. For
example if caseEnable is on, the translator will show a preference for words that are all lower case, all upper case, or all lower case except the first letter.

The flags governing the built-in rules include:

- xltSegmentVeto disallow translations that yield more than one character per character box.
- xltCaseEnable weigh in favor of translations that fit standard rules of capitalization (the first word of each sentence is capitalized, for example).
- xltAlphaNumericEnable weigh in favor of translations that fit standard groupings of letters and digits (for example, a word that begins with a digit is a number).
- xltPunctuationEnable weigh in favor of translations that fit standard rules of punctuation (for example, a comma does not follow a space).

Knowledge Source Rules

PenPoint supports two knowledge sources which a translator can use to improve its translation accuracy: the spelling dictionary and translation templates. The spelling dictionary is a list of correctly spelled words and rules for generating other correct words (by adding word endings, for example). A translation template improves translation accuracy by constraining the way in which the translator interprets handwritten input. “Translation Templates,” later in this chapter, discusses translation templates in more detail.

Certain translation flags direct the translator to use these knowledge sources as an aid to handwriting recognition. There are three types of flag for each knowledge source:

- Enable enables the use of a particular knowledge source, giving preference to translations that conform to the source.
- Veto when the knowledge source is enabled, rejects any translation that does not conform to the source.
- Propose when the knowledge source is enabled, if the translator can't generate a translation with a high enough score, the knowledge source proposes translations that conform to the source.

Therefore, the valid knowledge-source flags include:

- xltSpellingEnable use spelling dictionary and give preference to words in the dictionary.
- xltSpellingVeto reject words not in the spelling dictionary.
- xltSpellingPropose propose words from the dictionary when the translator doesn’t find a best guess.
- xltTemplateEnable use the translation template and give preference to words that match the template.
- xltTemplateVeto reject words that do not match the template.
- xltTemplatePropose propose translations that match the template when the translator doesn’t find a best guess.
The Veto and Propose flags may be used individually or combined, but they have no effect unless the corresponding Enable flag is set.

Post-Processing Rules

The translation object can apply postprocessing rules to assist errorchecking and proofing, or spelling correction. The only post-processing currently implemented is the smart case behavior. This capability calls for the translator to use linguistic rules to correct the capitalization of the translation. The translator applies smart case corrections unless you disable this behavior by setting the xltSmartCaseDisable flag.

Translation Templates

A translation template is a data structure that defines constraints on the way a translator interprets handwriting input. For example, using a translation template, you can force the translator to restrict its output to a certain set of words or characters. By limiting the translator output to a smaller set, translation templates can improve translation accuracy.

A translation template is a single, allocated block of memory containing no internal pointers. PenPoint provides a function, XTemplateCompile(), that lets you “compile” an XTM_ARGS data structure into a translation template. An XTM_ARGS structure includes the following fields:

- **xtmType** (XTEMPLATE_TYPE) the type of translation template. Each of the types is discussed further in “Template Types,” below.
- **xtmMode** (XTEMPLATE_MODE) the way in which the translator should use the template. Each of the modes is discussed in “Template Modes,” below.
- **pXtmData** a pointer to the data set defining the template, according to template type. For example, pXtmData might point to a list of acceptable characters, or a list of acceptable words.

Template Types

The xtmType value specifies how to interpret the data to which pXtmData points. Possible values are:

- **xtmTypeNone** The template does not restrict translations in any way. pXtmData is ignored.
- **xtmTypeGesture** the template restricts translations to a limited set of gestures. pXtmData points to an XTEMPLATE_GESTURE structure, which includes the following fields:
  - **count** (U32) the number of gestures in the list of acceptable translations.
  - **pGestures** (p_MESSAGE) a pointer to an array of count acceptable gestures (32-bit codes defined in XGESTURE.H).
xtmTypeCharList the template restricts translations to a limited set of characters. pXtmData is a P_STRING pointing to a null-terminated array of acceptable characters.

xtmTypeWordList the template restricts translations to a limited set of words. pXtmData is a P_STRING pointing to an array of null-terminated strings denoting the acceptable words. To identify the end of the array of acceptable words, the last pointer in the list must be Nil(P_STRING).

xtmTypePicture the template restricts translations to strings matching a specific pattern, or picture. pXtmData points to a string representing the picture. Certain characters in a picture string have special meanings:

- 9 the corresponding character must be numeric (a digit from 0 to 9).
- a the corresponding character position must be alphabetic.
- A the corresponding character position must be upper-case alphabetic.
- n the corresponding character position must be alphabetic or numeric.
- N the corresponding character position must be upper-case alphabetic or numeric.
- x the corresponding character position may be any character.
- [ the following characters up to but not including the next right square bracket (]) define a set of characters of which the corresponding character position must match one. The 9, a, A, n, N, and x characters do not have special meanings within square brackets. A hyphen within square brackets indicates a range of characters. For example, [abe] matches a, b, or e, while [ab-e] matches a, b, c, d, or e.
- \ ignore any special meaning for the following character literally. For example, 9 matches any digit, while \9 matches only the digit 9. Similarly, [a-c] matches a, b, or c, while [a-c] matches a, -, or c.

Template Modes

The xtmType values described in the previous section specify how to interpret the template data (for example, as a list of words). The xtmMode value specifies how to apply the data set to the translations. Possible values for xtmMode include:

- xtmModeDefault reject any translation that does not match the template exactly.
- xtmModePrefixOK consider translations that match any prefix of the template to match the template.
- xtmModeLoopBackOK consider translations that match any number of repetitions of the template to match the template.
- xtmModeCoerced coerce the translation to match the template, even if it doesn't match exactly. This is meaningful only for templates of type xtmTypeWordList.
Translation Messages

For historical reasons, the two header files that declare clsXtract declare some private messages which you should not use. A future release of PenPoint will correct this problem. For now, just don’t use the private messages.

Table 58-1 summarizes the public clsXtract messages and functions.

<table>
<thead>
<tr>
<th>Message</th>
<th>pArgs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_XLATE_NEW</td>
<td>Initializes the XLATE_NEW structure to default values.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_XLATE_NEW</td>
<td>Creates a new translation object.</td>
</tr>
<tr>
<td>msgXlateModeSet</td>
<td>HW_MODE</td>
<td>Sets the mode of the translation object.</td>
</tr>
<tr>
<td>msgXlateModeGet</td>
<td>P_HW_MODE</td>
<td>Gets the mode of the translation object.</td>
</tr>
<tr>
<td>msgXlateMetricsSet</td>
<td>P_XLATE_METRICS</td>
<td>Set the translation object’s XLATE_METRICS to communicate UI-based information that may assist in segmenting the incoming strokes into characters.</td>
</tr>
<tr>
<td>msgXlateMetricsGet</td>
<td>P_XLATE_METRICS</td>
<td>Reports the translation object’s XLATE_METRICS values.</td>
</tr>
<tr>
<td>msgXlateStringSet</td>
<td>P_XLATE_STRING</td>
<td>Sets the current textual context for the translation object.</td>
</tr>
<tr>
<td>msgXlateSetFlags</td>
<td>U32</td>
<td>Sets the translation flags of the translation object.</td>
</tr>
<tr>
<td>msgXlateGetFlags</td>
<td>P_U32</td>
<td>Gets the translation flags of the translation object.</td>
</tr>
<tr>
<td>msgXlateFlagsClear</td>
<td>U32</td>
<td>Clears the translation flags of the translation object.</td>
</tr>
<tr>
<td>msgXlateCharConstrainsSet</td>
<td>P_U16</td>
<td>Sets the character constraints of the translation object.</td>
</tr>
<tr>
<td>msgXlateCharConstrainsGet</td>
<td>P_U16</td>
<td>Gets the character constraints of the translation object.</td>
</tr>
<tr>
<td>msgXlateTemplateGet</td>
<td>PP_UNKNOWN</td>
<td>Gets the template for the translation object.</td>
</tr>
<tr>
<td>msgXlateTemplateSet</td>
<td>P_UNKNOWN</td>
<td>Sets the template for the translation object.</td>
</tr>
<tr>
<td>msgXlateCharMemorySet</td>
<td>P_CHARACTER_MEMORY</td>
<td>Sets the current character memory for character box mode.</td>
</tr>
<tr>
<td>msgXlateCharMemoryGet</td>
<td>P_CHARACTER_MEMORY</td>
<td>Gets the current character memory for character box mode.</td>
</tr>
<tr>
<td>msgXlateSetXlateCaseMetrics</td>
<td>P_XLATE_CASE_METRICS</td>
<td>Sets the “smart case” metrics.</td>
</tr>
<tr>
<td>msgXlateGetXlateCaseMetrics</td>
<td>P_XLATE_CASE_METRICS</td>
<td>Gets the “smart case” metrics.</td>
</tr>
<tr>
<td>msgXlateSetHistoryTemplate</td>
<td>P_UNKNOWN</td>
<td>Subclass hook for implementing a translation history for the translator.</td>
</tr>
<tr>
<td>msgXlateGetHistoryTemplate</td>
<td>PP_UNKNOWN</td>
<td>Gets the current alternate translation template, if the subclass implements such behavior.</td>
</tr>
<tr>
<td>msgXtractGetScribble</td>
<td>P_OBJECT</td>
<td>Reports the UID of the scribble object to which the translator is attached.</td>
</tr>
</tbody>
</table>

Do not use private messages.
CHAPTER 58 / USING THE TRANSLATION CLASSES

Translation Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>pArgs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgScrAddedStroke</td>
<td>P_SCR_ADDED_STROKE</td>
<td>The scribble object sends this message to the translator when it gets a new stroke.</td>
</tr>
<tr>
<td>msgScrRemovedStroke</td>
<td>P_SCR_REMOVED_STROKE</td>
<td>The scribble object sends this message to the translator when an existing stroke is deleted.</td>
</tr>
<tr>
<td>msgScrCompleted</td>
<td>pNull</td>
<td>When the scribble knows there will be no more strokes added, it sends this message to the translator. This causes the translator to self-send msgXtractComplete.</td>
</tr>
<tr>
<td>msgXtractComplete</td>
<td>pNull</td>
<td>Self-sent in response to msgScrCompleted. Provides hook for subclasses to complete translation of scribble.</td>
</tr>
<tr>
<td>msgXlateData</td>
<td>P_XLATE_DATA</td>
<td>Reports translated data from a translation object.</td>
</tr>
<tr>
<td>msgXlateCompleted</td>
<td>pNull</td>
<td>Notifies client that translation is complete. Client normally will respond by sending msgXlateData to discover the translation result.</td>
</tr>
</tbody>
</table>

Creating a Translator

You create instances of subclasses of clsXtract, not of clsXtract itself.

You create a translator in the usual way. Send msgNewDefaults, with a P_XLATE_NEW as its argument, to a clsXtract subclass (clsXText, clsXWord, or clsXGesture). clsXtract sets the referenced XLATE_NEW structure to default values. You then send msgNew to the class, with the same P_XLATE_NEW as its argument. This creates a new instance of the class, and the class sets the uid field of the referenced XLATE_NEW to the UID of the new instance.

If you think you need to change any of the default values, you can do so after sending msgNewDefaults, by directly assigning fields of the XLATE_NEW structure, or after creating the translator object with msgNew. After creating the object, you can change these values by sending msgXlateMetricsSet, msgXlateStringSet, and msgXlateSetFlags.

Initialization Messages

Translator initialization messages are messages sent before the translator interprets the user's input. For example, msgXlateSetFlags sets the translator flags described in "Handwriting Translation Flags," above. Another initialization message, msgAdded, establishes the translator as an observer of a particular scribble object. After the translator is attached to a scribble in this way, the scribble object will send messages to the translator to notify it of changes in the state of the scribble.
Control Messages

Control messages are messages that control how the translator gathers stroke data from the scribble it is observing. The scribble sends msgScrAddedStroke to the translator every time the user adds a stroke to the input, and msgScrRemovedStroke every time the user removes a stroke. The translator can use these notifications to implement dynamic translation.

When the scribble determines that the user input is complete (for example, when the user presses the OK button on the input pad), it sends msgScrCompleted to the translator. The translator object responds to msgScrCompleted by self-sending msgXtractComplete. clsXtract subclasses should respond to msgXtractComplete by translating the stroke data that the clsXtract object has gathered.

Notification Messages

The translator object sends msgXlateCompleted to its client (usually an input pad) to notify the client that the translation is complete.

The client sends msgXlateData to get the result of the translation. The translation result is an Xlist whose specific type depends on the specific subclass of clsXtract. See Chapter 60, Using Xlists, for more information on Xlists. The client can send msgXlateData only once, after which the translator frees all resources related to the translated data.
Chapter 59 / Using Scribbles

A stroke is a mark defined by the path of the pen from the point at which the pen tip touches the display to the point at which it comes up from the display. A scribble is simply a collection of strokes. clsScribble, which inherits directly from clsObject, provides a storage mechanism for scribbles, enables scribbles to render themselves, and establishes a protocol for communicating with translators (subclasses of clsXtract).

**Scribble Concepts**

This section describes some of the basic concepts of scribbles.

**Stroke Indexing**

A scribble object maintains a collection of strokes in an indexed, internal list. The strokes are numbered from zero, and the index increases for each added stroke. Strokes may be removed from a scribble, as well, but this does not affect the indexing. When a stroke is removed, it is simply marked as removed; its structure remains intact.

**Scribble Base and Bounds**

Each scribble has a base, a point in digitizer coordinates that specifies the scribble's offset from the origin of the window in which it exists. The coordinates of strokes in the scribble are stored relative to the scribble base.

This arrangement—the strokes relative to the scribble base, the scribble base relative to the window origin—lets you position the scribble as a unit within the window. An input pad, for example, resets its scribble's base when the user changes the window origin by resizing the input pad. By changing the scribble base, the input pad keeps the scribble a constant distance from the top of the input pad even though the base is specified relative to the window origin.

A scribble's bounding rectangle, or simply bounds, is the smallest rectangle that encloses all of the scribble's strokes (except those strokes marked as removed).

**Rendering**

If you provide a drawing context, a scribble can render itself. Input pads take advantage of this behavior whenever the input pad window becomes dirty. See Part 3: Windows and Graphics for more information about windows and drawing contexts.
Translator Notification

A scribble implements part of the handwriting translation protocol by defining messages to send to a translator observing the scribble. A scribble notifies its observing translator when the scribble adds or removes a stroke, and when the scribble determines that the user is not going to add more strokes.

clsScribble Messages

Table 59-1 summarizes the messages clsScribble defines. The following sections discuss these messages in more detail.

<table>
<thead>
<tr>
<th>Message</th>
<th>pArgs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgNewDefaults</td>
<td>P_SCR_NEW</td>
<td>Sets the default values for the msgNew arguments.</td>
</tr>
<tr>
<td>msgNew</td>
<td>P_SCR_NEW</td>
<td>Creates and initializes a new scribble object.</td>
</tr>
<tr>
<td>msgScrSetBase</td>
<td>P_XY32</td>
<td>Sets the stroke coordinate base for the scribble.</td>
</tr>
<tr>
<td>msgScrGetBase</td>
<td>P_XY32</td>
<td>Passes back the scribble base.</td>
</tr>
<tr>
<td>msgScrGetBounds</td>
<td>P_RECT32</td>
<td>Passes back the scribble bounds.</td>
</tr>
<tr>
<td>msgScrCount</td>
<td>P_U16</td>
<td>Passes back the total number of strokes in the scribble, including strokes marked as removed.</td>
</tr>
<tr>
<td>msgScrAddStroke</td>
<td>P_SCR_ADD_STROKE</td>
<td>Adds a stroke to the scribble.</td>
</tr>
<tr>
<td>msgScrCat</td>
<td>SCRIBBLE</td>
<td>Concatenates the strokes from another scribble.</td>
</tr>
<tr>
<td>msgScrDeleteStroke</td>
<td>U16</td>
<td>Deletes (marks as removed) the stroke associated with a specified index.</td>
</tr>
<tr>
<td>msgScrDeleteStrokeArea</td>
<td>P_SCR_DELETE_STROKE_AREA</td>
<td>Deletes all the strokes that touch the specified area.</td>
</tr>
<tr>
<td>msgScrClear</td>
<td>void</td>
<td>Deletes and frees all of the scribble's stroke data.</td>
</tr>
<tr>
<td>msgScrHit</td>
<td>P_SCR_HIT</td>
<td>Searches for the next stroke which intersects the specified rectangle.</td>
</tr>
<tr>
<td>msgScrRender</td>
<td>P_SCR_RENDER</td>
<td>Given a drawing context, renders the scribble in a window.</td>
</tr>
<tr>
<td>msgScrStrokePtr</td>
<td>P_SCR_STROKE_PTR</td>
<td>Passes back a pointer to a specific stroke.</td>
</tr>
<tr>
<td>msgScrComplete</td>
<td>void</td>
<td>Clients send this message to the scribble to notify the scribble that stroke entry is complete.</td>
</tr>
<tr>
<td>msgScrCompleted</td>
<td>NULL</td>
<td>Sent to observers to indicate that the stroke entry is completed.</td>
</tr>
<tr>
<td>msgScrAddedStroke</td>
<td>P_SCR_ADDED_STROKE</td>
<td>Sent to observers to notify them of the addition of a stroke to the scribble.</td>
</tr>
<tr>
<td>msgScrRemovedStroke</td>
<td>P_SCR_REMOVED_STROKE</td>
<td>Sent to observers to notify them of the removal of a stroke from the scribble.</td>
</tr>
</tbody>
</table>
Creating a New Scribble Object

You create a scribble in the usual way, by sending `msgNewDefaults` to `clsScribble` to set a SCRIBBLE_NEW structure to default values, then `msgNew` to create the scribble object. The only value in the scribble part of a SCRIBBLE_NEW is `scribble.base`, an XY32 giving the scribble base offset from the lower left corner of the window.

Scribble Attribute Messages

A scribble object has a base offset which determines where it will begin to render itself in the window. The scribble base represents digitizer coordinates relative to the lower left corner of the window. You can change the scribble base with `msgScrSetBase`, and learn its current value with `msgScrGetBase`.

A scribble maintains a bounding rectangle, which is the smallest rectangle that contains all of its strokes. You can determine the origin and size of the bounding rectangle, relative to the scribble base, by sending `msgScrGetBounds` to a scribble object.

You can determine the number of strokes a scribble object has recorded by sending `msgScrCount` to the scribble. The number passed back is the total number of strokes, including strokes marked as removed. You normally use `msgScrCount` to set an index for a loop that iterates over every stroke in the scribble.

Stroke Messages

The primary purpose of a scribble object is to maintain a collection of stroke objects, so it is not surprising that most of the messages `clsScribble` defines are related to the management and manipulation of the strokes.

You can add a stroke to the end of a scribble's list of strokes by sending `msgScrAddStroke` to the scribble, passing the stroke as an argument. You can also copy all of the strokes from one scribble object to another by sending `msgScrCat` to the destination scribble object, passing the source scribble as an argument.

Deleting a stroke does not destroy or free the stroke data, but simply marks the stroke as removed by setting a bit in the stroke data structure. You send `msgScrDeleteStroke` to a scribble object to delete a stroke whose index you specify. You can also delete all strokes within a specified rectangle by sending `msgScrDeleteStrokeArea` to the scribble object. `msgScrDeleteStrokeArea` interprets the rectangle origin and size relative to the scribble base, and marks as removed any stroke whose bounding rectangle intersects the area rectangle.

You can send `msgScrClear` to delete all strokes in the scribble. Unlike `msgScrDelete` and `msgScrDeleteArea`, `msgScrClear` does not simply mark the strokes as removed, but destroys the stroke data and frees the memory space they used.
You can find strokes that fall within a specified rectangle by sending `msgScrHit` to a scribble, passing a pointer to a SCR_HIT structure as an argument. You pass in a rectangle (relative to the scribble base) and stroke index in the SCR_HIT structure. The scribble passes back the index of the first stroke whose bounding rectangle intersects the rectangle you passed in.

You can instruct a scribble to render all of its scribble by sending `msgScrRender` to the scribble, passing a pointer to a SCR_RENDER data structure as an argument. The SCR_RENDER structure specifies a drawing context, a rectangle, and a range of stroke indices. `msgScrRender` renders the strokes in the window to which the drawing context is attached, rendering only those strokes which fall within the specified rectangle (interpreted in LWC) and stroke range. It does not render strokes marked as removed.

If you wish to manipulate an individual stroke, you can send `msgScrStrokePtr` to a scribble, passing a pointer to a SCR_STROKE_PTR structure as an argument. You pass in a stroke index in the SCR_STROKE_PTR argument. The scribble passes back a pointer to the stroke whose index you passed in. Be careful when you use this `msgScrStrokePtr`, as it generates a pointer to the stroke data, not a copy of the stroke.

### Notification Messages

Clients of scribble objects typically establish a protocol for interacting with scribbles, so clsScribble defines a few notification messages to facilitate such protocols. For example, clients can send `msgScrComplete` to a scribble object to inform it that there are no more strokes to process.

Scribble objects send three notification messages to objects that observe the scribble. A scribble sends `msgScrAddedStroke` to observers when the scribble adds a stroke to its stroke list. A scribble sends `msgScrRemovedStroke` to its observers when the scribble marks a stroke as removed. Finally, a scribble sends `msgScrCompleted` to observers when the scribble receives `msgScrComplete`. 
Chapter 60 / Using Xlists

An Xlist is a dynamic array of elements, stored in a heap you associate with the Xlist when you create it. Xlists of various types are used throughout the system. The primary purpose of Xlists is to pass translated information between translator objects and their clients (such as input pads), but it is possible to use Xlists for other purposes.

Xlist.H defines functions that you use to:

- Create, modify, and destroy Xlists.
- Traverse lists, access, and set list element.
- Filter data from Xlists.

An Xlist filter converts an Xlist into an Xlist of a different format (see XLFILTER.H for an example).

**Concepts**

An Xlist is a data structure that contains a list of pointers to Xlist elements, a set of Xlist flags, and other internal data. The actual organization of an Xlist is private; you use the Xlist functions to create and modify Xlists.

When a client creates an Xlist, the list and its elements are allocated from a heap specified by the client. A client can use this heap to allocate space for the list elements (with XlistAlloc; the element flag must be xfHeapAlloc). The data allocated from the heap is deallocated when the client frees the Xlist or calls XlistFreeData. Clients can allocate other data from the Xlist heap, but it is not recommended (it is the client's responsibility to free this data).

**Xlist Flags**

Each Xlist has a 32-bit value used for flags. These flags store data about the Xlist. Do not confuse these Xlist flags with the element flags that are stored in each Xlist element.

The flag values flag0 through flag15 are used by PenPoint™ (and are described below). The flag values flag16 through flag31 can be used by clients.

The only flag defined by PenPoint is xfXlist2Text, which indicates that the Xlist has already been processed through the function Xlist2Text.

**Xlist Elements**

Each Xlist element is defined by the structure Xlist_ELEMENT. The structure contains the following fields:

- flags (U16) element flags.
- type (XTYPE) the type of the data in pData.
pData (pUNKNOWN) a pointer to element data.

These Xlist element values are explained in detail below.

**Xlist Element Flags**

Each Xlist has a set of flags that specifies how the element's data should be treated when the list is freed or duplicated. The flags are:

- **xfHeapAlloc** the element's data was allocated from the Xlist heap.
- **xfObject** the pData is the UID for an object. Do not duplicate this element.
- **xfXlist** the element's data is a pointer to another Xlist.
- **xfExtracted** the element's data is used somewhere else and should not be freed when freeing the Xlist. If you set this flag, it is your responsibility to free the data.

The first three flags (xfHeapAlloc, xfObject, and xfXlist) are mutually exclusive. Setting more than one of these flags will have unpredictable results.

**Xlist Element Data**

Each Xlist element contains a pointer to data (pData). The type of data is specified in the type indicator. Table 60-1 lists the Xlist element types and the corresponding pData interpretations.

<table>
<thead>
<tr>
<th>Type</th>
<th>pData</th>
<th>pData refers to</th>
</tr>
</thead>
<tbody>
<tr>
<td>xtNull</td>
<td>pNull</td>
<td>Null element. Used for placeholders and dummy elements.</td>
</tr>
<tr>
<td>xtBounds</td>
<td>P_BDATA</td>
<td>Bounds data used by clsXGesture and clsText.</td>
</tr>
<tr>
<td>xtGesture</td>
<td>P_GDATA</td>
<td>Gesture data used by clsXGesture.</td>
</tr>
<tr>
<td>xtText</td>
<td>P_STRING</td>
<td>Text data used by clsText and the function Xlist2Text().</td>
</tr>
<tr>
<td>xtObject</td>
<td>OBJECT</td>
<td>Object data.</td>
</tr>
<tr>
<td>xtBoundsX</td>
<td>P_BDATA</td>
<td>Screen relative bounds data.</td>
</tr>
<tr>
<td>xtCharAttrs</td>
<td>P_Xlist_CHAR_ATTRS</td>
<td>Character attributes data used by clsText (TXTXlist.H).</td>
</tr>
<tr>
<td>xtParaAttrs</td>
<td>P_Xlist_PARA_ATTRS</td>
<td>Paragraph attributes data used by clsText (TXTXlist.H).</td>
</tr>
<tr>
<td>xtTabs</td>
<td>P_Xlist_TABS</td>
<td>Tabs data used by clsText (TXTXlist.H).</td>
</tr>
<tr>
<td>xtCharPos</td>
<td>TEXT_INDEX</td>
<td>Character position data.</td>
</tr>
<tr>
<td>xtTextList</td>
<td>P_WORD_LIST</td>
<td>Word list data used by handwriting translation.</td>
</tr>
<tr>
<td>xtTextListEnd</td>
<td>NULL</td>
<td>End of word list indicator used by clsSPaper.</td>
</tr>
<tr>
<td>xtTextWord</td>
<td>P_XTEXT_WORD</td>
<td>Text data used by clsText and clsSPaper.</td>
</tr>
<tr>
<td>xtStroke16</td>
<td>P_SPAPER_STROKE_DATA</td>
<td>Stroke data used by clsText and clsSPaper.</td>
</tr>
<tr>
<td>xtTeachData</td>
<td>P_XTEACH_DATA</td>
<td>Teaching data used by clsTeach.</td>
</tr>
<tr>
<td>xtUID</td>
<td>UID of a gesture object</td>
<td>A gesture object.</td>
</tr>
<tr>
<td>xtEmbedObject</td>
<td>P_TEXT_EMBED_OBJECT</td>
<td>Embedded text object data used by clsText.</td>
</tr>
<tr>
<td>xtExtended</td>
<td>UID of data object</td>
<td>Extended data.</td>
</tr>
</tbody>
</table>
**Xlist Functions**

Table 60-2 lists the Xlist functions and provides a brief description for each one.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XlistNew()</td>
<td>Creates a new Xlist.</td>
</tr>
<tr>
<td>XlistFree()</td>
<td>Frees an Xlist and all its data.</td>
</tr>
<tr>
<td>XlistGetFlags()</td>
<td>Passes back the Xlist flags for the Xlist.</td>
</tr>
<tr>
<td>XlistSetFlags()</td>
<td>Sets the Xlist Flags.</td>
</tr>
<tr>
<td>XlistMetrics()</td>
<td>Passes back the number of entries and heap Id.</td>
</tr>
<tr>
<td>XlistInsert()</td>
<td>Create a new element at the index'th location.</td>
</tr>
<tr>
<td>XlistDelete()</td>
<td>Delete the element at the index'th location.</td>
</tr>
<tr>
<td>XlistTraverse()</td>
<td>Iterates across the list of elements.</td>
</tr>
<tr>
<td>XlistIndex()</td>
<td>Passes back the current traversal index.</td>
</tr>
<tr>
<td>XlistSet()</td>
<td>Stores the copy of the index'th element.</td>
</tr>
<tr>
<td>XlistGet()</td>
<td>Passes back a copy of the index'th element.</td>
</tr>
<tr>
<td>XlistGetPtr()</td>
<td>Passes back a pointer to the index'th element.</td>
</tr>
<tr>
<td>XlistAlloc()</td>
<td>Allocates some memory from the Xlist heap.</td>
</tr>
<tr>
<td>XlistFreeData()</td>
<td>Releases the data with the given entry.</td>
</tr>
<tr>
<td>XlistDup()</td>
<td>Duplicates the contents of one Xlist into another.</td>
</tr>
<tr>
<td>XlistDupElement()</td>
<td>Duplicates the source element, append to the destination.</td>
</tr>
<tr>
<td>Xlist2Gesture()</td>
<td>Extracts the gestural information from an Xlist.</td>
</tr>
<tr>
<td>Xlist2StringLength()</td>
<td>Passes back the length of the string that Xlist2String will need.</td>
</tr>
<tr>
<td>Xlist2String()</td>
<td>Extracts the text information from an Xlist.</td>
</tr>
<tr>
<td>XlistDump()</td>
<td>Debugging interface for displaying an Xlist in the debug log.</td>
</tr>
<tr>
<td>XlistDumpSetup()</td>
<td>Sets the Xlist debug log display routine by type.</td>
</tr>
</tbody>
</table>

**Using the Xlist Functions**

Typical clients create Xlists with XlistNew(), add and delete elements with XlistInsert() and XlistDelete(), access the value of elements through filters or with XlistGet(), traverse the elements with XlistTraverse(), and destroy and free all the elements with XlistFree().

The other functions, while useful in certain circumstances, are rarely used. These functions are described in the PenPoint API Reference and in the file Xlist.H.
Creating a New Xlist
To create a new Xlist, call the function XlistNew(). The function has two parameters:

- `heap` (OS_HEAP_ID) the heap from which to allocate the Xlist and its elements.
- `ppXlist` (P_Xlist) a pointer to the Xlist the function creates.

Inserting an Xlist Element
To insert an element in an Xlist, call XlistInsert(). The function has three parameters:

- `pXlist` (P_Xlist) a pointer to the Xlist in which to insert the element.
- `Index` (U16) an index to the location where the new element is to be inserted.
- `pElem` (P_Xlist_ELEMENT) a pointer to the Xlist_ELEMENT to insert.

If the index is greater than the current number of entries, the element is added to the end of the Xlist.

Deleting an Xlist Element
To delete an Xlist element, call XlistDelete(). The function has two parameters:

- `pXlist` (P_Xlist) a pointer to the Xlist from which to delete the element.
- `index` (U16) the index of the element to delete.

XlistDelete() calls XlistFreeData() to free the memory of the Xlist element structure as well as of the data to which the element points.

Freeing All Elements of an Xlist
To destroy all of an Xlist’s elements and free their memory, call XlistFree(). The message has only one parameter: a P_Xlist pointing to the Xlist whose elements it is to free.

This function traverses the Xlist, frees the data for each element (unless the element has xfExtracted set) and frees the elements data structures themselves.

Traversing an Xlist
The function XlistTraverse() allows you to iterate across the elements in an Xlist. For each element in the Xlist, XlistTraverse() calls a callback function which you provide. If the callback routine returns anything but stsOK, XlistTraverse() terminates and returns the same status that the callback function returns. You can use XlistTraverse() to perform nested traversals.

The XlistTraverse takes three parameters:

- `pXlist` (P_Xlist) a pointer to the Xlist to traverse.
- `pProc` (P_XPROC) a pointer to the callback function.
pUserData (P_UNKNOWN) a pointer to user data that will be passed to the callback function.

The prototype for the the callback function is:

```
STATUS (PASCAL *P_XPROC) (P_Xlist pXlist, P_Xlist_ELEMENT pElem, P_UNKNOWN pUserData);
```

The parameters are:

- `pXlist` a pointer to the Xlist being traversed
- `pElem` a pointer to the current element in the list
- `pUserData` a pointer to the user data passed to `XlistTraverse()`.

#### Getting and Setting Xlist Elements

To get an element from an Xlist, call `XlistGet()`; to set an element in an Xlist, call `XlistSet()`.

Both messages take three parameters:

- `pXlist` (P_Xlist) a pointer to the Xlist whose element the function should get or set.
- `index` (U16) the index of the element to get or set.
- `pPtr` (P_Xlist_ELEMENT) a pointer to an Xlist_ELEMENT structure.

`XlistGet()` copies the element from the Xlist to `pPtr`, while `XlistSet()` copies `pPtr` to the Xlist. Remember that an Xlist element is a pointer to some data. `XlistGet()` and `XlistSet()` copy pointers to data, but they do not make copies of the data itself.

If the index is greater than the number of elements in the list, `XlistGet()` will get the last element in the list and `XlistSet()` will store the last element in the list.

`XlistSet()` replaces the existing element specified by the index. If you want to add a new element, use `XlistInsert()`.
Chapter 61 / Using Gesture Windows

Introduction

clsGWin (Gesture Window) inherits from clsWin. clsGWin adds some special functionality to the standard window input handling capabilities described in Chapter 53 (Input Subsystem API). The various clsGWin subclasses use this functionality to interpret pen input as gestures. Many of the PenPoint™ Windows and Graphics and UI Toolkit classes inherit from clsGWin.

You will very rarely need to use a clsGWin instance directly, or subclass it. In most cases where you want to capture gestural input from the screen, you can use a standard UI Toolkit class.

clsGWin also implements the Quick Help interface. This interface lets you specify a resource ID for each of the UI Toolkit components the user interface for your application. Whenever the Quick Help gesture is drawn over an element that inherits from clsGWin, the Quick Help manager uses the resource ID that is stored in the clsGWin object’s helpID field to locate the help text resource to display.

A clsGWin object is able to receive pen input, test to see if it is a gesture, and if a gesture is recognized send msgGWinGesture to clients. msgGWinGesture has a pArgs data structure GWIN_GESTURE that describes what gesture has been entered by the user, as well as some state information about the size (bounds) of the glyph that the user has made, and the x-y coordinate of the gesture hotspot. The frontmost window containing the hotspot is typically the target of the gesture.

A clsGWin object automatically creates a clsScribble object whenever it receives pen input.

clsGWin implements a lightweight window that automatically interprets input as gestures. When it receives input, a clsGWin object collects the strokes in a scribble. When the input event is terminated, the gesture window creates an clsXGesture translator and passes the scribble data to it. The gesture window then reports the translator results to the relevant client code.

Gesture Window Messages

Table 61-1 summarizes the messages clsGWin defines.
## clsGWin Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>pArgs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>msgNewDefaults</strong></td>
<td>P_GWIN_NEW</td>
<td>Initializes the GWIN_NEW structure to default values.</td>
</tr>
<tr>
<td><strong>msgNew</strong></td>
<td>P_GWIN_NEW</td>
<td>Creates and initializes a new instance.</td>
</tr>
<tr>
<td><strong>msgGWinGetStyle</strong></td>
<td>P_GWIN_STYLE</td>
<td>Returns the current style.</td>
</tr>
<tr>
<td><strong>msgGWinSetStyle</strong></td>
<td>P_GWIN_STYLE</td>
<td>Sets the style settings.</td>
</tr>
<tr>
<td><strong>msgGWinSetHelpId</strong></td>
<td>U32</td>
<td>Sets the gesture window’s helpId for quick help.</td>
</tr>
<tr>
<td><strong>msgGWinGetHelpId</strong></td>
<td>P_U32</td>
<td>Returns the gesture window’s helpId.</td>
</tr>
<tr>
<td><strong>msgGWinGetTranslator</strong></td>
<td>P_OBJECT</td>
<td>Returns the gesture window’s translator object.</td>
</tr>
<tr>
<td><strong>msgGWinSetTranslator</strong></td>
<td>P_OBJECT</td>
<td>Sets the translator object and returns the previous one.</td>
</tr>
<tr>
<td><strong>msgGWinTransformGesture</strong></td>
<td>P_GWIN_GESTURE</td>
<td>Transforms gesture information into local window coordinates.</td>
</tr>
<tr>
<td><strong>msgGWinTransformXList</strong></td>
<td>P_XLIST</td>
<td>Transforms xlist information to local window coordinates.</td>
</tr>
<tr>
<td><strong>msgGWinAbort</strong></td>
<td>pNull</td>
<td>Stop processing a gesture.</td>
</tr>
<tr>
<td><strong>msgGWinBadGesture</strong></td>
<td>P_GWIN_GESTURE</td>
<td>Displays feedback for unrecognized and unknown gestures.</td>
</tr>
<tr>
<td><strong>msgGWinHelp</strong></td>
<td>pNull</td>
<td>Displays quick help for the gesture window.</td>
</tr>
<tr>
<td><strong>msgGWinForwardGesture</strong></td>
<td>P_GWIN_GESTURE</td>
<td>Instructs the gesture window to forward a gesture to parent windows.</td>
</tr>
<tr>
<td><strong>msgGWinForwardedGesture</strong></td>
<td>P_GWIN_GESTURE</td>
<td>Gesture window sends this message to parent window in response to msgGWinForwardGesture.</td>
</tr>
<tr>
<td><strong>msgGWinForwardKey</strong></td>
<td>P_INPUT_EVENT</td>
<td>Instructs the gesture window to forward a keyboard event to parent windows.</td>
</tr>
<tr>
<td><strong>msgGWinForwardedKey</strong></td>
<td>P_INPUT_EVENT</td>
<td>Gesture window sends this message to parent window in response to msgGWinForwardedKey.</td>
</tr>
<tr>
<td><strong>msgGWinIsComplete</strong></td>
<td>P_GWIN_GESTURE</td>
<td>Determine whether there are more gestures to process.</td>
</tr>
<tr>
<td><strong>msgGWinGestureDone</strong></td>
<td>P_GWIN_GESTURE</td>
<td>Sent to indicate the end of a gesture.</td>
</tr>
<tr>
<td><strong>msgGWinStroke</strong></td>
<td>P_INPUT_EVENT</td>
<td>Self-sent to process a pen stroke received from the input system.</td>
</tr>
<tr>
<td><strong>msgGWinTranslator</strong></td>
<td>P_OBJECT</td>
<td>Self-sent to retrieve the translator used to gather and translate strokes.</td>
</tr>
<tr>
<td><strong>msgGWinComplete</strong></td>
<td>pNull</td>
<td>Self-sent to process a gesture, typically in response to msgGWinForwardedGesture.</td>
</tr>
<tr>
<td><strong>msgGWinXList</strong></td>
<td>P_XLIST</td>
<td>Self-sent to process an xlist.</td>
</tr>
<tr>
<td><strong>msgGWinGesture</strong></td>
<td>P_GWIN_GESTURE</td>
<td>Self-sent to process a gesture.</td>
</tr>
<tr>
<td><strong>msgGWinKey</strong></td>
<td>P_INPUT_EVENT</td>
<td>Self-sent to process a key input event, typically in response to msgGWinForwardedKey.</td>
</tr>
<tr>
<td><strong>msgGWinBadKey</strong></td>
<td>P_INPUT_EVENT</td>
<td>Self-sent to allow a subclass to handle bad keys.</td>
</tr>
</tbody>
</table>
Application window class, 195–197
  see also clsAppWin
AppMain() function, 72
document activation and, 102, 105
AppMain routine, 97
APP_METRICS structure, 161–162
APP_MGR_ACTIVATE structure, 148
APP_MGR_CREATE structure, 148
APP_MGR_DELETE structure, 149
APP_MGR_METRICS structure, 146–147, 149
APP_MGR_MOVE_COPY structure, 148–149
APP_MGR_NEW structure, 97
AppMonitorMainO function, 72, 97, 151
  in installation, 99
appProcess parameter, 96
APP_SET_PRIORITY structure, 171
APP_WIN_METRICS structure, 196–197
APP_WIN_NEW structure, 196
Arc, 271
argc parameter, 96
Argument data, 14
  modifying, 17
argv parameter, 96
Background colors, 274
  graphic state element, 278
  painting, 376
  palette colors, 295
  RGB color values, 295
backgroundlnk, 376
Baseline alignment, 387–388
Bezier curve, 271
Binding, 210
Bitmap, 329–331
  creating, 330
  images, 263
    allocating, 258
    fonts and, 314
    messages, 329–330
    modifying, 330
    notifications, 331
    pictures, 524–525
tag, 330
  using, 330
BITMAP_NEW structure, 330
BORDER_BACKGROUND structure, 377
BORDER_NEW structure, 373
  styles, 373–375
Border rectangle region, 379
Borders, 361
  adjusting, 505
  geometry, 379–380
  outer offsets, 380
  subclassing clsBorder and, 380
  toolkit ancestors and, 370
  see also Border windows
BORDER_STYLE structure, 373
Border windows, 371–380
  bsUnitsLayout measurement, 375
  creating, 373–375
  geometry, 379–380
  layout, 378
  messages for, 371–373
  painting, 375–377
  propagating/notify visuals and, 378
  regions of, 379
  resizing, dragging, topping, 377
  sample, 371
  visuals of, 376
  see also clsBorder
Bounding box, 397
Bounds, 232
  accumulation, 270–271
  defined, 269
  client interface to layout, 249
  DC, 256–257
  scribble, 607
  setting, 234
bsUnitsLayout units, 375
Buf field, 14
Built-in rules, 600–601
  defined, 600
Button definition, 495
BUTTON_METRICS structure, 423
BUTTON_NEW structure, 419
Buttons, 417–424
  choice component, 362
  command, 495
  creating, 354, 419–420, 421
  many, 422
  defaults, 419
  kinds of, 418
  manager, 423
  messages, 417–418
  notification, 420–421
  advanced techniques, 423–424
  unwelcome, 421
  painting, 422
  preview messages, 362
  sample, 417
  showDirty control, 422
  styles of, 418, 419–420
  value, 422
  see also clsButton; Icons; Men
  buttons; Pop-up choices;
  Reference buttons
BUTTON_STYLE, 419
  styles, 419–420
  Button table, 236

Cached image, 273–274, 299–301
  creating, 300
  defined, 299
  drawing, 300–301
  hot spot, 273, 300
  image devices vs., 301
  invalidating, 301
  mask, 300, 301
  related classes, 301
Callback function, 273
  sampled images and, 299
Canvas, 215
Capabilities, object, 25–29
  changing capability, 29
  checking capability, 29
  creation capabilities, 28
  creation notification, 28
  flags, 25
  freeing capability, 26
  inheritance capability, 27
  mutation capability, 28
  ObjectCallO capability, 26
  ObjectSendO capability, 26
  observable capability, 27
  owner capability, 26
  scavenging capability, 27
Cap argument, 24
  changing capabilities and, 29
  OBJECT_NEW structure, 47
Changing capability, 29
Character
  metrics, 310
  positions, 416
  widths, 309–310
Check gesture
  handling, 169–170
  processing, 517–521
  card client activity, 519–520
  dimmed controls, 521
  mixed attributes, 520
  multiple card types, 520
  multiple option sheets, 521
  nested components, 521
  run-through, 517–519
  selection interaction, 521
Checking capability, 29
Child windows, 216
  altering, 246
  labels and, 415–416
  layout, 415–416
  painting, 416
  toolkit tables, creating, 434
ChkO macros, 24
Choice, 427, 442–444
  component buttons, 362
  creating, 352–353, 443
  management, 439–440
Selection, 440
Classes, 5
application, 82, 86–87
class's, getting, 55
compiling, 47
components, 76
creating, 6, 31–48
ancestor calls and, 36–37
design considerations, 33
header file and, 36
installing and, 47–48
instance data and, 34–35
methods for, 37–41
method table and, 41–46
objects and, 34
overview, 31–33
reasons for, 31
defined, 5
going information about, 54–56
handwriting
capture, 551
translation, 552–553
implementation of, 32
installing, 47–48
new, 146–147
summary of, 33
keys and, 24
layout, 381–398
linking, 47
manager, 423, 439–440
metaclasses, 84–85
method tables, 119
object
confirming, 54
getting, 55
parts of, 31–32
predefined, 83
with clsAppMgr, 85
return values and, 14
simple, 82–84
UI Toolkit, 357–360
inheriting from clsControl, 359
kinds of, 361
not inheriting from clsControl, 358
outline of, 360
CLASS_INFO array, 44–45
defined, 42
entry fields, 45
Class Manager
capabilities and, 25–29
concepts, 9–29
creating new objects and, 15–18
document process and, 90
handling message status and, 18–19
identifiers, 9–13
instance data, 34
memory protection and, 34
message-sending macros, 23–24
method entry points and, 37
method invocation, 35
method table and, 41–42
observable objects and, 78
overview, 6
passing messages with, 33
repaint process and, 242–243
scavenging and, 60
sending messages, 13–15
other ways for, 19–22
using keys and, 24–25
CLASS_NEW_ONLY structure, 47–48
contents, 47–48
CLASS_NEW structure, 47
CLASS_ONLY_NEW structure, 146
Class value, 55
ClExtend() macro, 393
Client window, 92
closing document and, 109
creating, for application frame, 212–213
document termination and, 112
frame layout and, 501
positioning scroll window, 461
Clipping, 219–221, 235
children, 221
unclipped, 236
disadvantage of, 220
local, 269, 270
messages, 283
on image, 219
overriding, 220
rectangles, 338
region, 219–220
defined, 235
sharing parent's, 221
window, 220
siblings, 221
window, 236
Close box, 507
Closed figures, 271–272
drawing, 294
messages, 283–284
Closing
application window, 196
documents, 109–110
class, 6, 31, 69
Application Framework messages, 71
default behavior, 166
document activation and, 105, 106
document object and, 92
header file, 160–161
menu button, 163–164
tags and, 144
messages, 157–161
advanced, 171
Application Framework, 161
class, 157
document attributes, 158
document hierarchy, 158, 161–163
document life cycle, 157–158
document window, 159, 163
observer, 160
printing, 160
standard application menu, 159–160, 163–170
msgAppAbout, 168
msgAppActivate, 113
msgAppClose, 73
msgAppCopySel, 168
msgAppInit, 72
msgAppMoveSel, 168
msgAppPrint, 167
msgAppPrintSetup, 167
msgAppRevert, 168
msgAppSave, 111
msgAppSearch, 168
msgAppSend, 167
msgAppSpell, 167–168
msgFree and, 110–111
msgHandle and, 72, 104
msgInit and, 113–114
msgRestore and, 73, 113
msgSave and, 73, 112
option sheets for printing and, 138
overview, 86
in removing frame decorations, 140–141
in requesting move or copy, 120
standard application menus and, 504–505
see also Application class
class, 177
messages, 178
using, 177–178
see also Application directory handle class
class, 70
see also Application directory handle class
class, 69
application classes and, 87
function of, 86
initialization routine and, 97–99
instance of, 82
messages, 145, 146
metrics and, 145
msgFree routine, 99
predefined classes with, 85
see also Application manager class
clsAppMonitor, 69, 151
for loading and unloading help, 154
for loading and unloading
stationary, 153–154
messages, 152–153
descendent modified, 153
instance, 152–153
using, 153–154
misc directory and, 154
monitor installation and, 151–152
overriding default behavior, 152
subclassing, 155
see also Application monitor class
clsApp Win, 118
messages, 195
using, 195–197
see also Application window class
clsBitmap, 301, 329, 525
messages, 329–330
notification, 331
clsBorder, 361, 370
custom backgrounds and, 377
descendants and colors and, 376
flags, 373–375
frames and, 501–502
label classes and, 413
messages, 371–373
attribute, 371–372
border geometry, 372
class, 371
rendering, 372
subclass responsibility, 373
painting
background, 376
border, 375
foreground, 376
subclassing, 380
see also Borders; Border windows
csButton, 352, 354
controls support, 419
messages, 417–418
msgBorderGetForegroundRGB
and, 422
msgControlAcceptPreview and, 400
previewing messages, 423
response to, 424
see also Buttons
clsCalcApp, 86
clsChoice, 352–353, 442
choice management and, 439, 443
messages, 443
notification, 443
value, 444
clsChoiceMgr, 439
choice value and, 444
messages, 440
clsClass, 5, 6
application classes and, 87
initialization routine, 98–99
classes relationship to, 82–84
class installation and, 33, 47–48
metaclass and, 85
purpose of, 82
in system process, 82–84
clsCloseBox, 507
clsCommandBar, 508
clsContainerApp, 185, 186
clsControl, 399
button notification and, 420
conrol enable and, 404
dirty controls and, 403
in filing controls, 399
gesture handling and, 369
label notification and, 414
message dispatching and, 399–400
messages, 401–402
msgGWInGesture and, 408
previewing and, 406–407
subclasses values of, 403
UI Toolkit classes inheriting from, 359
UI Toolkit classes not inheriting from, 358
xgs1Tap gesture and, 408
see also Controls
clsCounter, 509
clsCustomLayout, 196, 353–354
aligning edges and, 393
csFrame and, 389
function, 361, 381
layout constraints, 392
specifying, 390–391
layout loop and, 397
layout of adjacent windows by, 394
messages, 390
shrink-wrap and, 395
window layout and, 382
see also Custom layout
clsDateField, 486
clsDrwCtx, 211
storing graphic states and, 277
clsEmbeddedWin, 69, 117
clsAppWin comparison, 118
in creating embedded window, 190
default behavior, 118
descendants of, 118
getting exact pen location and, 123
identifying selection and, 120
inheriting from, 118
message intercepts, 127
messages, 118–119, 189
in move and copy operations, 191–192
protocol, 121
move and copy, 119
Tic-Tac-Toe application and, 123
translations, 119
UI Toolkit and, 370
using, 189–193
see also Embedded window class
clsField, 356, 475
messages, 476–477
clsFileSystem, 10
clsFixedField, 486
clsFontInstallMgr, 441
clsFontListBox, 463, 473
clsFrame, 112
creating close box and, 507
filing state and, 503
frame layout and, 501
messages, 499
modifying frame and, 500
page number creation and, 509
shrink-wrap and, 395
tab bars and, 508
see also Frames
clsGotoButton, 118
clsGrabButton, 528
messages, 529
clsGWIn, 368–370, 552, 617
clsScribble object, 617
Help gesture and, 370
ehlp ID field, 617
message previewing and, 407
messages, 617–618
xgsQuestion gesture and, 408
see also Gesture windows
clsHelloWin, 238
clsIcon
bitmap picture and, 524–525
messages, 523
painting and, 525
pixelmap picture and, 525
see also Icons
clsImgDev, 210, 255
creating imaging device and, 256
landscape and portrait mode and, 262
memory allocation and, 258
clsInput, 567
clsInputApp, 561
clsIntegerField, 486
clsINWin, 560
clsIP, 552
clsSPaper
general facility, 590
messages, 585–586
using, 585–588
creating insertion pad, 586–587
deleting insertion pad, 587
displaying insertion pad, 587
translator object, 587–588
Xlist data, 588
clsLabel, 400
child windows and, 415
decoration drawing and, 413
dirty controls and, 403
field support and, 416
function, 409
icon layout and, 525
messages, 410
notification, 414
painting and, 415
see also Labels
clsList, 83–84
clsListBox, 463
entries and, 465–466
gestures and, 468
messages, 449, 463–464
painting and, 468–469
scrolling and, 466
see also List boxes
clsManager, 439
clsMark, 69, 129
embedded window marks and, 118
instance (mark), 129
link messages, 134
messages
class, 199
holders send to marks, 199
marks sent to
components, 199–200
sent internally, 200
protocol, 133
see also Marks class
clsMenu, 5, 445
creating menus and, 447–448
displaying menu and, 448
messages, 447
see also Menus
clsMenuButton, 363, 445
messages, 445
clsModalFilter, 488
clsMoveCopyIcon, 125
clsMyView, 517–519
clsNBApp, 186–187
clsNote, 488
argument structure, 15–16
creating new instance of, 18
filter, 490–491
label creation, 489
message handling status and, 18
messages, 488
painting and, 492
see also Notes
clsObject, 5–6
application classes and, 87
clsClass and, 82
clsView and, 174
method tables and, 42
msgCopy and, 49–50
msgFree and, 59
msgFreeOK and, 58
object/class information messages and, 54
saving/restoring data and, 77
saving/restoring instance data and, 35
in system process, 82–84
see also Objects
clsOption, 508, 511
card destruction and, 516
command sheets and, 522
indicating mixed attributes and, 520
messages, 512–513
clsOptionTable, 521
clsPageNum, 509
clsPicSeg, 270, 279, 317
class, 319–320
graphics applications and, 323–326
hit testing and, 325
messages, 317–318
drawing, 319
text attributes, 321
see also Picture segment class
clsPixelDev, 210, 266
clsPopupChoice, 450
messages, 450, 451
clsPrint, 140
clsPrLayout, 139
clsProgressBar, 531
messages, 535
inherited, 540
metrics, 532
see also Progress bar
clsOfMonth, 70
msgNew and, 111, 113
saving and restoring data, 77
clsRootContainerApp, 185, 186
messages, 187
concepts, 187
clsScribble, 552, 607
messages, 608
clsScrollbar, 453
clsScrollbarWin, 457
creating scrollwin and, 458
messages, 458
option card layout and, 515
scrollwin layout and, 460–461
clsSelChoiceMgr, 439
messages, 440
clsSelection, 128
clsShadow, 509
clsSpaper, 552, 589–596
creating subclass, 593
instance of, 593–594
facilities, 590–592
examples, 591–592
functions, 589
input flags, 594
messages, 589–590
parsing Xlist data, 592
rendering translated text, 592
subclassing, 592–593
subclass instance, 593–594
translator, 595–596
clsScrollWin, 243
clsStringListBox, 463, 470
destroying button window and, 472
messages, 470
painting and, 473
providing entries and, 471
clsSysDC, 285
clsSysDwCtx, 210–211, 235, 267
colors and, 295
in coordinate system
transformations, 270
drawing context features and, 269
fill areas and, 292
hit detection and, 270
messages, 281–284
drawing, 319
picture segments and, 319
raster operation and, 292–293
sampled images and, 272
in sending window messages, 289
see also Drawing context (DC)
clsTabBar, 223, 508
subclasses of clsTkTable and, 442
clsTabButton, 220, 221
clsTableLayout, 353, 381
Calc’s positioning of child window using, 388
control enable and, 450
function, 361, 383
layout loop and, 397
messages, 384
table window constraints, 386–387
toolkit table buttons and, 425
window layout and, 382
see also Table layout
clsText, 105
clsTextField, 486
clsTextView, 517–519
clsTkTable, 352, 425
changing defaults and, 434
creating buttons with, 422
creating child windows and, 434
creating toolkit tables and, 428, 430
descendants, 362
flags, 435
function, 425
manager field, 438
messages, 427
**INDEX**

notification, 438
painting and, 436–437
response to button previewing messages, 424
specifying item class and, 435
subclasses of, 442
table layout and, 437
see also Toolkit tables
clsToggleTable, 442
clsTrack, 527
clsGrabBox and, 528
messages, 527
clsTransfer, 119
clsTrView, 125
clsView, 69, 118, 552
  concepts, 173
  creating new view and, 174
  subclassing, 175
  view filing and, 174
  see also View class
clsWin, 5
  behavior, 211
  default repainting behavior, 222
  input and, 228
  inserting window and, 233
  layout policies and, 361
  low-level pen input and, 558
  messages, 230–231
  drawing contexts and, 284
  moving/resizing windows and, 247
msgSave and, 112
  procedures for instances of, 217–218
self-sending messages, 216, 226
structures, 227–229
subclasses, 209
  custom, 215
  subclassing, 326
  see also Window class
clsWinDev, 209, 210, 217
  messages, 255
  printers, 301
  root window, 255
  see also Window device classes
clsXfer protocol, 121, 122
  Tic-Tac-Toe application and, 126
clsXGesture, 553
clsXTranslate messages, 604–605
clsXTest, 553
clsXtract, 551, 553
  function, 552
  translation classes and, 598
clsXWord, 553
Colors, 269, 274, 295–296
  border window, 376
  compatibility of, 295–296
  inverting, 295
palette, 295
  hardware-dependent, 274, 295
  RBG values, 274, 295
  rendering, 299
Command bars, 428
  notes with, 491
Command sheets, 521–522
  creating, 522
Compiling, method tables, 45–46
Component classes, 76
  saving/restoring data and, 77
Component (mark), 130
  ancestor, 201
  parent, 201
  UUID and UID, 203
  validating, 201
Component objects, 92
Component (mark), 130
  ancestor, 201
  parent, 201
  UUID and UID, 203
  validating, 201
Component (mark), 130
  ancestor, 201
  parent, 201
  UUID and UID, 203
  validating, 201
Component objects, 92
Compiling, method tables, 45–46
Component classes, 76
  saving/restoring data and, 77
Connections notebook, 96
Constraints
  custom layout, 392
  alignment, 393
  flags, 395
  four child window, 391
  specifying, 390–391
  types of, 392
  shrink-wrap and, 395–396
  relative window, 396
  value, 396
  table layout, 386–387
Container application classes, 185–188
  contents page and, 188
  hierarchy, 186
  reference document and, 188
  root, 186–188
  see also Embedded document
CONTROL_ENABLE structure, 404–405
CONTROL_METRICS structure, 402
CONTROL_PROVIDE_ENABLE structure, 405, 449–450
Controls, 399–408
  creating, 402–403
  default, 403
dirty, 403
enable, 404–405
dynamic, 405
evaluating, 404–405
protocol, 449
filing, 399
gesture notification and, 408
  how to use, 399
inactive, 404
internal notification and, 405–408
  message dispatching and, 399–400
  messages, 401–402
  previewing stage and, 400
  sent in response to events, 407
nesting of, 363
presentation and interaction behavior of, 400
  sent in response to events, 407
  style of, 402
  fields, 402
  values of, 403
  see also clsControl
CONTROL_STYLE structure, 402
Coordinates
  drawing, 286–289
  defaults, 287
  resetting LUC, 288
  rotation, 288
  scale, 288
  transformation matrices, 289
  translation, 288
  units, 287
  world coordinates, 288–289
graphics, 320
integral, 270
  rounding error, 269
  unit size of, 270
Coordinate system, 265–266, 478
  conversion messages, 283
  layout classes, 382
  logical unit coordinates (LUC), 267–268
  logical window coordinates (LWC), 267–268
  transformations, 268
Copy icon, 120, 125
  presenting, 124–125
Copying
  data, steps for, 119–120
  between embedded windows, 118–119
  embedded windows, 191–193
  mark, 203
  objects, 49–50
  picture segments, 326–327
  pixels, 259–260
Copy protocol, 119–123
  copying data, 122–123
  data type determination, 121
  destination by user, 120
  destination in file system, 122
  destination to copy, 121
open, 67
  process, 109
opening, 107–109
paginating, 137
parts of, 92–93
printing, 135–136
  repainting and, 222–223
  as processes, 89–91
reactivating, 113–115
renaming, 162
saved, 78–79
screen information and, 87–88
sequence number, 178
state diagram, 100
states of, 100
transitions, 101
terminating, 110–113

Document window messages, 163
Dragging, 377
Drag handle, 453
Drawing
  box, 287
    cached images, 300–301
    coordinates, 286–289
    with a drawing context, 286
    dynamic, 293
    line, 337–338
    on image device, 259
    operations, 293–294
    closed figures, 294
    filling window, 294
    open figures, 293
    in picture segments, 322
    other objects, 323
    sampled image, 298
    text, 302, 311–312
    trackers, 528
    in window, 236
  context (DC), 209, 211, 266
    binding, 266
    bound to window, 210
    class, 281–290
    clsHelloWin, 238
    color, 295–296
    creating, 285
    default state, 285
    defined, 266
    destination rectangle, 273
    drawing coordinates, 286–289
    drawing operations, 266, 293–294
    drawing with, 286
    encoding attribute, 277
    features, 269–270
      bounds accumulation, 269, 270–271
      color, 269, 274
      figure drawing operations, 269, 271–272
      graphic state, 270, 277–278
      hit detection, 269, 270
      local clipping, 269, 270
    picture segments, 279
    sampled images, 269, 272–274
    text operations, 269, 275–276
    graphic state, 266, 290–293
    in handling low-level pen input, 558
    image mask, 273
    ImagePoint font support, 302–315
    messages, 281–284
      associating DCs with windows, 281
      class, 281
      clipping and hit detection, 283
      clsWin messages and, 284
      coordinate conversion, 283
      drawing operation, 283–284
      graphic state, 281
      hardware-dependent color, 282
      matrix manipulation, 282
      RGB color, 282
      text interface, 284
      msgWinDevBindPixelmap, 257
      printing, 301–302
      for rendering visual feedback, 591–592
      sampled images, 297–301
      state, 320–321
      not stored, 321
      paint, 320–321
      picture segment storage, 321
      system, 210, 267
      text and, 302
      unit size, 270
      when to create, 286
      window messages to, 289–290
      window repaint and, 237–238
      see also clsDrwCtx; clsSysDrwCtx
    DrawingPaper application, 317
    Dynamic drawing, 293
    Dynamic UIDs, 9, 10

Editing
  graphics, 324
  picture segment, 324
Edit menu, 165
  Copy command, 120
  Move command, 120
Ellipse figure, 272
  drawing, 341
Embedded application, 77, 195
  creating, 196
  Embedded document, 67, 117–128
    activating, 161
    counting, 183
    directory, 89
    embedded window concepts, 117–118
    moving and copying between, 118–119
    gestures and, 128
    getting and setting number of, 181–182
    intercepted messages and, 127
    managing, 161
    move/copy protocol and, 119–123
    moving in Tic-Tac-Toe and, 123–127
    printing, 136–137
    situations for, 136
    selection and, 128
    traversal and, 129
    see also Container application classes
Embedded Printing card, 136
Embedded window, 117
  child, 193
  concepts, 117–118
    clsEmbeddedWin descendants, 118
    marking support, 118
    copying, 191–193
    between, 118–119
    creating, 190
    destroying, 190
    examples, 117
    gestures, 127
    metrics, 190
    moving, 191–193
    between, 118–119
    protocol, 122
    style of, 190–191
    toolkit ancestors and, 370
    UUID, 193
Embedded window class, 69, 189–193
  see also clsEmbeddedWin
EMBEDDED_WIN_EXTRACT_CHILD structure, 193
EMBEDDED_WIN_GET_DEST structure, 193
EMBEDDED_WIN_INSERT_CHILD structure, 193
EMBEDDED_WIN_METRICS structure, 190
EMBEDDED_WIN_MOVE_COPY_OK structure, 192
EMBEDDED_WIN_MOVE_COPY structure, 191–192
EMBEDDED_WIN_NEW_ONLY structure, 190
EMBEDDED_WIN_STYLE structure, 190–191
Embedding documents, 77
Ems, 308, 309
Encoding
  character, 277
  font, 307
  field, 312
  run-length, 298
  universal standard, 312
Entries
  list box, 465
  defined, 464
inserting and removing, 467
supplying, 465–467

Entry points
for application process, 96–97
for document activation, 102
for methods, 37

Episodic layout, window, 225

Error-checking macros, 23–24
Errors
application, 494
system, 494
unknown, 495

Event
generation, 563
handling, 563–564
inserting, 550
keyboard, 581–584
data, 581–584
pen, 575–580
processing, 563–566
pen input sampling and, 565–566
x-y distribution and, 565
queue, inserting messages in, 571
routing, 547
status codes, 564

Event data, 575–580
msgKeyChar, 583
msgKeyDown, 582
msgKeyUp, 582
msgPenDown, 576
msgPenEnterDown, 577
msgPenEnterUp, 577
msgPenExitDown, 578
msgPenExitUp, 578
msgPenMoveDown, 577
msgPenMoveUp, 577
msgPenOutProxUp, 579
msgPenStroke, 579
msgPenTap, 580
msgPenUp, 576

Extracting
window, 218, 234
from window tree, 550

Facilities
clsSPaper, 590–592
examples, 591–592

Failed() macros, 24
Fax Viewer application, 317
Field
activation and deactivation, 481–482
messages, 481
component creation, 480–481
creating, 477–479
custom handwriting translation, 479
style flags, 477–479
data-specific, 486
delayed input, 483
input processing, 482–483
input validation, 484–485
layout, 485
messages, 476–477
properties access, 480
sample, 475
style flags, 475–476
tEditType settings, 478
focusStyle setting, 478
translator, 479
user interface and, 485
see also clsField
FIELD_NEW_ONLY structure, 477
Fields, 475–486
support, 416
FIELD_STYLE, 477
flags, 479
values, 478–479
Figure drawing operations, 271–272
closed figures, 271–272, 294
defined, 269
open figures, 271, 293
File system
documents and, 88–89
embedded windows and, 122
organization, 88–89
Filing
controls, 399
frames, 503
list boxes, 469
UI Toolkit components, 365
windows, 253
Filled region (progress bar), 531
defaults, 537
manipulating, 537–539
Fill patterns, 291
alignment, 292
determining, 292
graphic state element, 278
styles, 292
windows and, 294
Filter, 548–549
adding, 571
objects, 547
removing, 571
FIM_PRUNE_CONTROL structure, 441
FIM_SHORT_IDS, 441
Fixed-point numbers, 288
Flags
built-in rule, 600–601
clsBorder style, 373–375
constraint, 395
described by APP_MGR_FLAGS, 147
field style, 475–476
file mode, 180
filter, 298
generating and setting, 180
handwriting translation, 600–602
input, 558, 565–566
clsSPaper, 594–595
window, 569–570
knowledge source, 601–602
note, 489
object capability, 25
scrollwin, 459
table layout, 384–385
toolkit table, 430–432
modifying items with, 435
values, 431
window, 228–229
input, 228
setting, 234–235
style, 228, 229, 232
Xlist, 611
element, 612
Flow pagination, 137
FONTLB_NEW_ONLY structure, 473
Font list boxes, 473
creating, 473
notification, 473
Fonts, 275–276
aspect ratio of, 306
attributes, 275, 303, 305
bitmap, 275, 314
cache loading, 314
changing, 302
common, 306–307
default, 305–306
defined, 303
encoding, 307
fields, 312
enumeration of, 304
geometry, 308–309
graphic state element, 278
groups of, 305–307
ID, 275, 276, 303
opening fonts and, 303–304
installed, displaying, 441
metrics, 275, 307–310
names, 303, 308
opening, 275–276, 303–305
outline, 275
scaling, 276, 310–311
search path, 313–314
state, 315
storage, 304
strings, 304–305
switching between, 315
system, 306
transforming, 307
user choice of, 304
user-visible, 306
weight of, 306
Foreground colors, 274
graphic state element, 278
painting, 376
palette colors, 295
RGB color values, 295
Format codes, 495–496
Frame application, 212
Frame decorations, 141, 507–509
close box, 507
command bar, 508
page number, 509
removing, 140–141
shadow, 509
tab bars, 508
title bars, 507
user interface, 507
frameNewFields, 16
Frame_New structure, 500
Frames, 497–505
Application Framework and, 504–505
components of, 497
creating, 500
custom layout example, 389
filing, 503
layout, 500–501
messages, 499
modifying, 500
multiple, 500
notification, 501–503
close, float, bring-to-front, delete, 503
selection, 502
zoom, 502–503
resizing and dragging, 501
scrollbars and, 498
styles of, 498
subclasses of, 505
see also clsFrame
FRAME_STYLE structure, 500
FRAME_ZOOM structure, 500
Freeing capability, 26
FS_New structure, 500
 FS_New_only structure, 179
toolsNew structure, 179
fullEnvironment flag, 98
Geometric shapes, rendering with thick borders, 340–342
Gesture windows, 368–370
gesture propagation, 368–369
messages, 617–618
Quick Help IDs, 370
responding to gestures, 369
using, 617–618
see also clsGWIn
Getting and setting
application directory attributes, 180–182
all, 180
individual, 180–182
embedded window style, 190–191
pixel values, 259
Xlist elements, 615
Global sequence numbers, 178
getting, 179
Glyphs
finding, 313–314
metrics and encoding, 314
search path, 313–314
missing, 277
Goto buttons. see Reference buttons
Grabber, 549
input, 572
object, 547
Grab boxes, 528–529
Graphics, 319–320
drawing by adding, 325
drawing messages and, 319
editing, 324
index, 324
invisible, 326
opcode, 319
scaling, 327
Graphics classes, 210–211
hierarchy, 211
coordinate systems, 267–269
drawing context features, 270
models and implementation, 265–267
using picture segments in, 323–326
Graphic state, 270, 277–278
determining filled areas, 292
elements, 277–278
filling and stoking, 290
fill pattern alignment, 292
line and fill patterns, 291–292
line styles, 290–291
messages, 281
raster operations, 292–293
setting attributes of, 271
storage, 277
Group, font, 305–307
common, 306–307
default, 305–306
encoding, 307
names, 305
transforming, 307
growChildHeight flag, 385, 387
growChildWidth flag, 385, 387
GWIN_GESTURE structure, 369, 617
Handwriting capture
classes, 551
object, 555
subsystem, 555
Handwriting translation, 479
flags, 600–602
built-in rules, 600–601
knowledge source rules, 601–602
post-processing rules, 602
subsystem, 546, 551–553
translation classes, 552–553
window subclasses, 551–552
Header files, 31
for clsApp, 160–161
creating, 36
goto buttons, 117
labels, 14
MT output file, 46
UI Toolkit classes, 364
Help
loading and unloading, 154
notebook, 154
Hierarchy
Application Framework classes, 71
class messages, 158, 161–163
container application classes, 186
graphics classes, 211
software, 76
subsystem, 545
translation class, 598
UI Toolkit classes, 357
window, 251–252
Hit detection, 270
defined, 269
messages, 283
Holder, 202
messages sent by, 202–203
Hot mode
documents in, 79, 107
setting, 162
Hot spot
cached image, 273, 300
gesture, 617
Icons
creating, 524–525
layout, 525
messages, 523
ICON_COPY_PIXELS structure, 525
ICON_NEW_ONLY structure, 524
Icons, 418, 523–526
creating, 524–525
layout, 525
messages, 523
notification, 525
painting, 525
picture styles for, 524
see also clsIcon
ICON_STYLE structure, 524
Identifiers. see UID; UUIDs
Image device, 256–263
accessing, 256–257
pixels in image window and, 259–262
binding, 257–258
cached images vs., 301
comparison with windowing devices, 256
complex use of, 261–262
creating, 256–257
defined, 255
destroying, 262
dirty windows and, 259
drawing, 259
landscape and portrait mode, 262
multiple pixelmaps and, 262
performance tips, 263
target device, 257, 258
UID and, 256
Image mask, 273
cached, 273
ImagePoint, 209
text support, 302–315
amount of fitting text, 313
API use, 302
color metrics, 310
drawing text, 311–312
finding glyph, 313–314
font attributes, 305
cache, 314
defined, 303
font metrics, 307–310
group, 305–307
improving performance, 315
measuring text, 312
opening font, 303–305
scaling font, 310–311
spacing text, 312
text and drawing context, 302
overview, 209–210
rendering details, 335–342
earlier release differences, 342
geometric shapes with thick borders, 340–342
LDC, 336
draw, 337–338
line width and corner radius
scaling, 339–340
LUC, 335
polygons, 339
tiling and, 292
windows and, 210
Image window, 259–262
copying pixels in, 259–260
going and setting pixel values in, 259
overlapping windows in, 260
stenciling in, 260–262
Imaging models, 265–266
abstract coordinates of, 267
sampled images and, 299
simple, 266
Input arguments, 14
Input, 14
Index, table, 388
Inheritance, 82
of application class, 86
Inheritance capability, 27
In-line fields, 475
Inner rectangle region, 379
Input
delayed, 483
messages, 483
device drivers, 567
INPUT_EVENT structure and, 567–568
event status codes, 564, 569
see also Event
flags, 558, 565–566
clsSPaper, 594–595
window, 569–570
see also specific input flags
focus, 548
functions
InputEventInsert(), 565, 571
InputFilterAdd(), 547, 571
InputFilterRemove(), 571
InputGetGrab(), 572
InputGetTarget(), 573
InputSetGrab(), 572
InputSetTarget(), 572
InputTarget(), 550
grabber, 572
leaf-to-root model and, 565
low-level pen, 558–562
pen, sampling, 565–566
processing, 482–483
messages, 482
registry, 547–548, 567
defined, 546
system messages, 570
target, 572–573
object, 581
validation, 484–485
messages, 484
INPUTAPP.C, 559–560
INPUT_EVENT_DATA structure, 575–576
for keyboard events, 581
InputEventGen() interface, 549, 563
INPUT_EVENT structure, 567
inputLink flag, 566
inputLinkThrough flag, 566
inputMoveDelta flag, 565
inputNoBuy flag, 566
inputResolution flag, 566
Input subsystem, 545, 546–550
API, 567–573
event data structure, 567–568
constants, 568–570
event routing, 547
filters, 548–549
grabber, 549
input registry, 547–548
inserting events, 550
listener objects, 549
messages, 570
procedures, 571–573
adding filter, 571
getting grab information, 572
getting target, 573
inserting message into event queue, 571
removing filter, 571
setting input grabber, 572
setting input target, 572
routing, 548
target object, 550
UIDs, 568
window tree, 550
inputTransparent flag, 566
Inserting
custom window as main application window, 561–562
events in input stream, 550
IP window, 556–557
list box entries, 467
messages in event queue, 571
window in window tree, 550
windows, 217, 233–234
Xlist element, 614
Insertion pads, 585
creating, 586–587
deleting, 587
displaying, 587
UI styles, 587
Insertion point, 416
Installing
application monitor and, 151–152
applications, 96–99
classes, 33, 47–48
new, 146–147
Instance data, 33, 34–35
accessing, 35
allocating, 34
contents, 99
defining additional, 85
document activation and, 104
maintaining dynamic, 35
memory protection and, 34
saving and restoring, 35
size, 47–48
Instances, 5
  application, 82
  of simple class, 84
Integral coordinates, 268
IP_NEW typed structure, 586
IP_XLATE_DATA structure, 588
Jmp() macros, 24
Kerning, 310
Keyboard event data, 581–584
KEYBOARD.H, 581
KEY_DATA structure, 581–582
KEY.H, 581
Keys, 24
  changing capabilities and, 28
  OBJECT_NEW_structure, 47
  using, 24–25
Knowledge source, 601
  rules, 601–602
  defined, 600
  spelling dictionary, 601
  translation template, 601
LABEL_NEW_ONLY structure, 411
  label strings and, 413
LABEL_RECTs structure, 416
Labels, 409–416
  child windows and, 415–416
  creating, 411–413
  annotation, 413
  displays of, 409
  field support and, 416
  layout, 414
  messages, 410
  notification, 414
  painting, 414–415
  progress bar, 531
    custom, 531, 539–540
    sample, 409
  strings and special characters, 413
  styles of, 411–413
  fields, 411–412
  see also clsLabel
LABEL_STYLE structure, 411, 411–412
Layout, 361
  baseline, 387
  border window, 378
  calculator example, 388
  capturing vs., 398
  child windows, 415–416
  custom, 389–390
    creating, window, 390–395
    dimensions, 391–392
    initialization, 396
    sample, 389
    field, 485
    frames, 500–501
    icons, 525
    labels, 414
    lazy, 397
    loops, 397–398
    notes, 492
    option card, 515
    scrollbar, 453
    scrollwin, 460–461
    speedup, 365
    tab bars, 508
    table, 383
      constraints, 386–387
      flags, 384–385
      specifying, 385–386
      structure, 384–385
      using tAlignBaseline for, 388
    toolkit table, 437
    units, 375
    window, 216, 224–225, 247–248
      adding child windows to, 381
      classes, 381–382
      dirty, 249, 365
      episodic, 225
      parent-veto, 225
      processing, 249
      shrink-wrap and, 397
      unconstrained, 224
    Layout classes, 381–398
      capturing vs. layout, 398
      coordinate system, 382
      custom layout, 389–390
      creating, 390–395
      layout loops, 397–398
      lazy layout, 397
      shrink-wrap, 397
      constraints and, 395–396
      table layout, 383
      window layout, 381–382
    lbFreeDataByMessage flag, 466
Leaf-to-root model, 565
Life cycles, 95–115
  application class, 95–99
  deinstalling, 99
  installing, 96–99
  document, 100–115
  activating, 102–107
  closing, 109–110
  creating, 102
  deleting, 115
  opening, 107–109
  reactivating, 113–115
  terminating, 110–113
Line
  cap and join, 278
  drawing, 337–338
  end points in one-pixel, 337
  modes, 291
  patterns, 271, 291–292
    graphic state element, 278
    segments, 265
    stoking and, 290
    styles, 290–291
    thickness, 291
      graphic state element, 278
      in rectangle, 341
      width, 339–340
      physical, 340
Link files, 134
LIST_BOX_ENTRY structure, 465
  free mode, 466
  inserting and removing entries and, 467
  setting state with, 467
List boxes, 463–473
  contents of, 464–465
  creating, 464
  entries, 465
    inserting and removing, 467
    supplying, 465–467
  filing, 469
  font, 473
  gestures, 468
  messages, 463–464, 469
  modifying, 467
  notification, 467–468
  painting, 468–469
  pre-loading, 467
  scrolling, 466
  state, 467
  string, 470–473
  toolkit table vs., 470
  see also clsListbox
LIST_BOX_NEW structure, 464
LIST_BOX_STYLE structure, 464
Listener
  field, 549
  objects, 548, 549
Loader database, 96
Local clipping, 270
  defined, 269
  rectangle, 270
  graphic state element, 277
Logical device coordinates (LDC), 336
  rendering details, 336
  rounding to positive side of LUC, 337
Logical transformation matrix (LTM), 336
Logical unit coordinates (LUC), 267–268, 335
  coordinate systems
    transformations, 268
  drawing coordinates and, 286–287
  graphic state element, 277
  messages to set, 287

obtaining, 336
origin, 292
scaling fonts and, 276, 310
system, figure, 335
transforming, 268
units, 287
unit size, 268-269
Logical window coordinates (LWC), 232, 267-268
Macros
DEBUG warning, 23
declaration, 39–40
error-checking, 23–24
for extracting information from UIDs, 12
message sending, 23–24
for testing UIDs, 13
see also specific macros
main function, 72
in application installation, 96-97
in document activation, 102-103
initialization routine and, 97
mainO routine, 33
Main window, 92
application, 504
creating, 504
inserting custom window as, 361-362
in document activation, 105
initializing clsSPaper-based, 593-594
setting, 163
MakeMsgO macro, 11
class UIDs and, 568
MakeStatusO macro, 11
Make() macro, 12, 235
defining window tags and, 436
MakeWarning() macro, 12
MakeWKN() macro, 10
Manager
button, 438–439
choice, 443
classes, 423, 439–440
objects, 423
Managers, 438
for menu buttons, 166
for toolkit tables, 438
button manager notification details, 438–439
menu management, 439
Mapping, 130–131
stamp, 131–132
table, 131
Margin rectangle region, 379
Mark, 129
copying, 203
creating and holding, 133
delivery messages, 201–202
positioning, 202
sent to components that have children, 202
holder of, 129
messages, 132
parts, 130
sending message to, 202–203
setting, to component, 203
supporting, 132
Marks class, 69, 129-130, 199-203
embedded windows and, 118
messages, 199-200
see also clsMark
Matrix
manipulation messages, 282
transformation, 289
Memory
for application code, 92
instance data and, 34
protection, 34
Menu bar, 363
creating, 353
creation, 353
Menu buttons, 163–164, 418
adding, 166
creating, 446–447
displaying submenus with, 448–450
document and edit menus, 165–166
managers for, 166
messages, 445
painting, 447
pop-up menu, 445
style of, 446
tags, 164
see also clsMenuButton
MENU_BUTTON_NEW_ONLY structure, 446
Menu buttons, 163–164, 418
adding, 166
creating, 446–447
displaying submenus with, 448–450
document and edit menus, 165–166
managers for, 166
messages, 445
painting, 447
pop-up menu, 445
style of, 446
tags, 164
see also clsMenuButton
MESSAGE_BUTTON_STYLE structure, 446
MENU_NEW_ONLY structure, 447
Menus, 428, 447–448
creating, 447–448
submenus, 448
displaying, 448
frame, 503
management, 439
nesting of controls in, 363
standard application, 363
working of, 363
see also clsMenu
Message argument, 14
structure, 14–15
new object, 15–17
Message handler, 18, 31
code, 19
defined, 5
status values and, 12
see also Methods
Message parameters, 38–39
Messages, 5
activation and deactivation, 481
clsApp, 157–161
advanced, 171
Application Framework and, 161
class, 157
document attributes, 158
document hierarchy, 158, 161–163
document life cycle, 157–158
document window, 159, 163
observer, 160
printing, 160
standard application menu, 159–160, 163–170
clsAppDir, 178
clsAppMgr, 145, 146
clsAppMonitor, 152–153
handling, 155
using, 153–154
clsAppWin, 195
clsBitmap, 329–330
clsBorder, 371–373
clsButton, 417–418
clsChoice, 443
clsChoiceMgr, 440
clsControl, 401–402
clsCustomLayout, 390
clsEmbeddedWin, 118–119, 189
clsField, 476–477
clsFrame, 499
clsGrabBox, 529
clsGWindow, 617–618
clsIcon, 523
clsIP, 585–586
clsLabel, 410
clsListBox, 449, 463–464
clsMark, 129, 199–200
clsMenu, 447
clsMenuButton, 445
clsNote, 488
clsOption, 512–513
clsPicSeg, 317–318
attribute, 317–318
class, 317
drawing, 318
clsPopupChoice, 450, 451
clsProgressBar, 535
clsRootContainerApp, 187
clsScribble, 608
clsScrollWin, 458
clsSelChoiceMgr, 440
in clsSelection, 128
clsSPaper, 589–590
clsStringListBox, 470
clsTableLayout, 384
clsTable, 331
clsTiff, 331
clsTkTable, 427
clsTrack, 527
clsWin, 230–231
defined, 5
delayed input, 483
delivery, 201–202
INDEX

drawing context, 281–284
event, 546–547
field component creation, 480
frame action, 502
identifiers, 11
input
  processing, 482
  subsystem, 570
  system, 570
  validation, 484
intercepted, 127
method table specifications for, 41
move and copy protocol and, 119
notification, 399
object, 49
  and class information, 54
ObjectCall() and, 19
ObjectPost() and, 21–22
ObjectSend() and, 20
observer, 50, 150
overriding, 42
preview, 399, 405–406
message argument for, 407
preview button, 362
printing, 135, 136
protocol, 140
progress bar, 535–537
  inherited, 540
propagating, 225
sampled images, 297
sending, 13–15
  macros for, 23–24
  message argument structure
  and, 14–15
  to object, 15
return values and, 14
sent by holders, 202–203
sent to components, 201–202
superclass, 155
translation, 604–606
  control, 606
  initialization, 605
  notification, 606
window
  creation, 230
  display, 230–231, 235–244
  filing, 231
  layout, 231, 244–251
  management, 231, 251–253
  metrics, 230, 233–234
  sending to hierarchy, 251–252
see also specific messages
Message status, 18–19
Message text, 495–496
message string format codes, 495–496
specifying command buttons, 495
Metaclasses, 84–85
clsAppMgr and, 85
Methods, 31
  ancestor calls and, 36–37
code for, 34
creating, 37–41
  declaration macros, 39–40
  declaring entry points, 37
  message parameters, 38–39
  operations and, 41
for msgAppClose, 109
for msgAppInit, 105
for msgAppOpen, 108
for msgFree, 110
for msgRestore, 105, 114
for msgSave, 105, 112
see also Message handler
Method tables, 31
  class implementation and, 32
  compiling, 45–46
    two steps, 46
    with wildcards, 46
  declaring entry points, 37
  message parameters, 38–39
  operations and, 41
for msgAppClose, 109
for msgAppInit, 105
for msgAppOpen, 108
for msgFree, 110
for msgRestore, 105, 114
for msgSave, 105, 112
see also Message handler
Metrics, 145
  bitmap, 330
  character, 310
  font, 275, 307–310
  getting, application monitor, 153
  getting, application window, 196–197
  getting, embedded window, 190
  getting, for class, 149
  progress bar, 534, 536–537
  TIFF image, 332
  window messages, 233–235
MiniNote, document pagination, 137
MiniText
  Document menu buttons, 363
  document pagination, 137
  token implementation, 131
Move icon, 120, 125
  presenting, 124–125
Move protocol, 119–123
  data type determination, 121
  destination by user, 120
  destination in file system, 122
  destination to move, 121
  getting exact pen location, 123
  identifying selection, 120
  moving data, 122–123
  OK move, 121–122
  reasons for using, 119
  requesting move, 120
Moving
  data, embedded window, 126–127
  data, steps for, 119–120
  documents, 122
  embedded windows, 191–193
  between, 118–119
  picture segments, 326–327
  in Tic-Tac-Toe, 123–127
  windows, 246–247
msgAdded, 51
  translation and, 605
msgAddObserver, 27, 50–51
msgAddObserverAt, 50–51
msgAMGetMetrics, 153
msgAMLoadHelp, 154
msgAMLoadMisc, 154
msgAMLoadStationary, 153
msgAMRemoveHelp, 154
msgAMRemoveStationary, 154
msgAMTerminateOK, 155
msgAncestor, 55
msgAncestorsA, 55
msgAppAbout, 168
msgAppActivate, 105–107
  in document reactivation, 113
msgAppAddCards, 138, 168–169
  handling, 138–139
  responding to, 169
msgAppAddFloatingWin, 163
  managing multiple windows and, 500
msgAppClose, 73
  closing document and, 109–110
  for Tic-Tac-Toe application, 110
msgAppCopySel, 120, 168
msgAppCreateMenuBar, 163
msgAppDeInstalled, 150
msgAppDelete, 149
msgAppDeletedLink, 134
msgAppDirGetAttrs, 180
msgAppDirGetBookmark, 182
msgAppDirGetClass, 181
msgAppDirGetFlags, 180
msgAppDirGetGlobalSequence, 179
msgAppDirGetNext, 182–183
msgAppDirGetNextInit, 182
msgAppDirGetNumChildren, 181
  for counting embedded
documents, 183
msgAppDirGetSequence, 181
msgAppDirGetUID, 181
msgAppDirGetUUID, 181
msgAppDirGetSequenceToName, 183
msgAppDirGetAttr, 180
msgAppDirGetBookmark, 182, 183
msgAppDirSetClass, 181
msgAppDirSetFlags, 180
msgAppDirSetNumChildren, 181
msgAppDirSetSequence, 181
msgAppDirSetUID, 181
msgAppDirSetUUID, 181
msgAppDispatch, 109
msgAppExecute, 165
adding menu buttons and, 166
arguments to, 166
msgAppGetLink, 134
msgAppGetMetrics, 161
msgAppGetName, 163
msgAppGetOptionSheet, 137
msgAppGetRoot, 161, 162, 187
msgAppInit, 72
in application monitor installation, 151
creating frames and, 504
msgAppActivate and, 105-107
subclassing clsAppMonitor, 155
msgAppInstalled, 150
msgAppInvokeManager, 165
adding menu buttons and, 166
msgAppMgrActivate, 102
in activating application instance, 148
in activation, 103
in document creation, 148
in document reactivation, 113
msgAppMgrCopy, 102
in copying application instance, 148
msgAppMgrCreate, 102
creating new document and, 148
msgAppMgrDelete, 115
in deleting application instances, 149
msgAppMgrGetMetrics, 149
msgAppMgrMove, 148
msgAppMgrShutdown, 155
msgAppMoveSel, 120, 168
msgAppOpen, 72, 107-109
printing embedded documents and, 137
in removing frame decorations, 140-141
for Tic-Tac-Toe application, 108
msgAppPrint, 139, 167
msgAppPrintSetup, 138, 167
msgAppRemoveFloatingWin, 163
msgAppName, 162
msgAppRevert, 168
msgAppSave, 111
msgAppSaveChildren, 111
msgAppSearch, 168
msgAppSelectAll, 168
msgAppSend, 167
msgAppSetHotMode, 162
msgAppSetMainWin, 163
msgAppSetName, 163
msgAppSetParent, 171
msgAppSetPrintControls, 140-141
msgAppSetPriority, 171
msgAppShowOptionSheet, 138, 169
msgAppSpell, 167-168
msgAppTerminate, 155
msgAppTerminateProcess, 111
msgAppWinClose, 196
msgAppWinGetMetrics, 196
msgAppWinOpen, 196
msgAppWinSetStyle, 196
msgBitmapSetMetrics, 330
msgBitmapSetSize, 330
msgBorderGetBackgroundRGB, 376
msgBorderGetBorderRect, 379
msgBorderGetForegroundRGB, 376
clsButton and, 422
msgBorderGetInnerRect, 375, 379
subclassing clsBorder and, 380
msgBorderGetOuterOffsets, 380
custom layout constraints and, 392
msgBorderGetOuterSize, 379
msgBorderInkToRGB, 376
msgBorderPropagateVisuals, 378
msgBorderProvideBackground, 377
msgBorderProvideDeltaWin, 377
msgBorderRGBToInk, 376
msgBorderSetLook, 404
msgBorderSetStyle, 378
msgBorderSetVisuals, 378
msgButtonAcceptPreview, 438
msgButtonDone, 438-439
msgButtonGetSetMetrics, 419
msgButtonNotify, 400, 420
msgButtonNotifyManager, 424
msgButtonSetClient, 399
msgButtonSetDirty, 403
button value and, 422
toggle tables and, 442
toggle tables and, 442
msgControlEnable, 404-405
submenus and, 449-450
toolkit table notification and, 438
msgControlGetDirty, 403
toggle tables and, 442
msgControlGetMetrics, 402
msgControlGetStyle, 404
msgControlGetValue, 400, 403
button value and, 422
toggle tables and, 442
msgControlProvideEnable, 164, 405
msgControlRepeatPreview, 406, 408
msgControlSetClient, 399
msgControlSetDirty, 403
button value and, 422
toggle tables and, 442
msgControlSetEnable, 404
toggle tables and, 442
msgControlSetMetrics, 402
msgControlSetStyle, 404
msgControlSetValue, 400, 403
button notification and, 421
button value and, 422
choice value and, 444
toggle tables and, 442
msgControlUpdatePreview, 406
msgCopy, 49
using, 49-50
msgCopyRestore, 49-50
msgCreated, 28, 56
msgCstmLayoutGetChildSpec, 390-391, 396
msgCstmLayoutSetChildSpec, 390
msgDcAlignPattern, 292
msgDcCacheImage, 297, 299
creating cached images and, 300
drawing cached images and, 300
invalidating cached images and, 301
msgDcCharMetrics, 312
msgDcCopyImage, 297
cached images and, 299
copying cached images and, 300-301
msgDcCopyPixels, 259-260, 263
msgDcDrawArcRays, 293, 294, 340
msgDcDrawBezier, 293, 340
msgPicSegDrawSpline and, 323
msgDcDrawChordRays, 294, 340
msgDcDrawEllipse, 294, 313
msgDcDrawImage, 263, 297
cached images and, 299
copying cached images and, 300-301
msgDcCopyPixels, 259-260, 263
msgDcDrawArcRays, 293, 294, 340
msgDcDrawBezier, 293, 340
msgPicSegDrawSpline and, 323
msgDcDrawChordRays, 294, 340
msgDcDrawEllipse, 294, 340
msgDcDrawImage, 263, 297
cached images and, 299-300
call backs, 299
filtering and, 298
painting TIFF image with, 333
picture segments and, 320
run-length encoding and, 298
msgDcDrawImageMask, 297
  rendering colors and, 299
msgDcDrawPixels, 260–261
msgDcDrawPolygon, 294, 340
msgDcDrawPolyline, 293, 340
msgDcDrawRect, 375–376
msgDcDrawRectangle, 292, 340
msgDcDrawSectorRays, 294, 340
msgDcDrawText, 302, 311
  measuring text and, 312–313
msgDcDrawTextRun, 313
msgDcFillWindow, 290, 294
msgDcFontOpen, 305–306
msgDcGetCharMetrics, 310
msgDcGetFontMetrics, 307
msgDcGetFontWidths, 310, 312
msgDcGetLine, 291
msgDcGetMatrix, 289
msgDcGetMatrixLUC, 289
msgDcGetPixel, 259
msgDcHoldLine, 291
msgDcIdentity, 288, 310
msgDcIdentityFont, 310
clsPicSeg and, 321
msgDcInitialize, 285
msgDcInvertColors, 295
msgDcLineThickness, 291
msgDcMatchRGB, 296
msgDcMeasureText, 310, 312–313
msgDcMeasureTextRun, 313
msgDcOpenFont, 302, 304–305
msgDcPlaneNormal, 296
msgDcPlanePen, 296
msgDcPop, 302
msgDcPopFont, 315
msgDcPreloadText, 314
msgDcPush, 302
msgDcPushFont, 315
msgDcRotate, 288
msgDcScale, 288
msgDcScaleFont, 302, 310–311
clsPicSeg and, 321
msgDcScaleWorld, 288
msgDcSetBackgroundColor, 260, 295
msgDcSetBackgroundRGB, 295
msgDcSetForegroundColor, 260, 295
msgDcSetForegroundRGB, 295
msgDcSetLine, 291
msgDcSetLineWidth, 212
msgDcSetMode, 291, 292
msgDcSetPixel, 259
msgDcSetPlaneMask, 296
msgDcSetROP, 293
msgDcSetWindow, 212, 286
msgDcTranslate, 288
msgDcUnitsTwips, 308
msgDcUnitsWorld, 288
msgDestroy, 26
  dsMark and, 133
  in destroying image device, 262
  in destroying object, 57–58
key use and, 24
msgDisable, 29
msgDrawCtxSetWindow, 286
msgDump, 252
msgEmbeddedWinBeginCopy, 120, 191–192
msgEmbeddedWinBeginMove, 120, 124–125, 191–192
msgEmbeddedWinCopy, 121, 192
msgEmbeddedWinDestroy, 193
msgEmbeddedWinExtractChild, 122, 192
msgEmbeddedWinGetMetrics, 190
msgEmbeddedWinGetPenOffset, 123, 193
msgEmbeddedWinGotoChild, 127
msgEmbeddedWinInsertChild, 122–123, 193
msgFieldClear, 483
msgFieldCreatePopUp, 482
msgFieldCreateTranslator, 481
msgFieldDeactivate, 485
msgFieldGetCursorPosition, 480
msgFieldGetMaxLen, 480
msgFieldGetStyle, 480
msgFieldPreValidate, 484
msgFrameClose, 501, 503
  close boxes and, 507
msgFrameGetClientWin, 500
msgFrameGetMetrics, 500
msgFrameGetStyle, 500
msgFrameSelect, 502
msgFrameSetClientWin, 500
msgFrameSetMenuBar, 503
msgFrameSetMetrics, 500, 595
msgFrameSetStyle, 500
msgFrameShowSelected, 502
msgFrameZoom, 503
msgFree, 57, 73
  in deinstallation, 99
  in destroying application directory handle, 179
  in destroying document deletion, 115
  in document termination, 110–112
handling, 59
msgFreeing, 58
  handling, 58
msgFreeOK, 57–58
  handling, 58
msgFreePending, 58
  handling, 58–59
msgGetObserver, 52
msgGWInForwardedGesture, 369, 408
msgGWInGesture, 127
  application response to, 170
  gesture propagation and, 369
  gesture windows and, 368
  responding to gestures and, 369
msgGWInTransformGesture, 369
MSG_HANDLER, 37
MsgHandlerArgTypeO macro, 39
  using, 40
MsgHandlerO macro, 39
  using, 39
MsgHandlerParameterN WarningO macro, 39
  using, 40
MsgHandlerWithTypesO macro, 39
  using, 40
msgIconCopyPixels, 525
msgIconFreeCache, 525
msgNumObservers, 52
msgObjectClass, 55
msgObjectsA, 54
msgObjectNew, 28
msgObjectOwner, 55
msgObjectValid, 55
msgObjectVersion, 56
msgOptionAddCard, 138, 515
msgOptionAddLastCard, 139, 169
msgOptionApplicableCard, 516, 518-520
msgOptionApply, 516, 520
msgOptionApplyCard, 516, 520
msgOptionClose, 516
msgOptionCreateSheet, 519
msgOptionDirtyCard, 520
msgOptionExtractCard, 516
msgOptionProvideCard, 516
msgOptionRefreshCard, 516
msgOptionRemoveCard, 515
msgOptionSetCard, 515
msgOptionShowCard, 515
msgOutProxUp, 575
msgOwner, 55
msg parameter, 38
in OBJ_NOTIFY_OBSERVERS structure, 52
msgPenDown, 407, 563, 575
event data, 576
msgPenEnterDown, 575, 577
event data, 577
msgPenEnterUp, 575, 577
event data, 577
msgPenExitDown, 575, 578
event data, 578
msgPenExitUp, 575, 578
event data, 578
msgPenHoldTimeout, 120
msgPenInProxUp, 575, 578
event data, 578
msgPenMoveDown, 575, 577
event data, 577
msgPenMoveUp, 575, 577
event data, 577
msgPenOutProxUp, 578-579
event data, 579
msgPenStroke, 579
event data, 579
msgPenTap, 579-580
event data, 580
msgPenUp, 407, 575, 576
event data, 576
msgPicSegAddGrafic, 325
msgPicSegChangeOrder, 325
msgPicSegDelete, 324
msgPicSegDelta, 324
msgPicSegDrawGrafic, 324
msgPicSegDrawGraficIndex, 324
msgPicSegDrawObject, 320, 323
msgPicSegDrawSpline, 320, 323
msgPicSegErase, 322
msgPicSegGetCount, 324
msgPicSegGetGrafic, 324
msgPicSegHitTest, 325
msgPicSegMakeInvisible, 326
msgPicSegPaint, 319
msgPicSegPaintObject, 323
msgPicSegRedraw, 333
msgPicSegScaleUnits, 327
msgPicSegSetCurrent, 324
msgPicSegSetFlags, 323
msgPopupChoiceGetChoice, 451
msgPostObservers, 53
msgPrefsPreferenceChanged, 170
msgPrintEmbeddeeAction, 143
msgPrintGetProtocols, 137, 140
not understood, 143
msgPrintLayoutPage, 141, 142
not understood, 143
msgProgressGetFilled, 538
msgProgressGetMetrics, 536, 537
msgProgressGetStyle, 536
msgProgressGetUnfilled, 538
msgProgressGetVisInfo, 538-539
msgProgressProvideLabel, 539-540
msgProgressSetFilled, 538
msgProgressSetMetrics, 537
msgProp, 28, 57
msgQuickHelpShow, 370
msgRCAppGotoDoc, 184, 187
msgRCAppNextDoc, 187
msgRCAppPrevDoc, 187
msgRCAppReferenceContents, 188
msgRDAppCancelGotoDoc, 188
msgRemoveObserver, 27, 51
msgResFindResource, 517
msgResGetObject, 113
msgResPutObject, 112
msgResReadID, 306
msgResReadObject, 113-114
msgRestore, 35, 73
failures during, 59
msgSave, 35, 73
filing windows and, 253
method for, 105, 114
in reactivating document, 113-114
in Tic-Tac-Toe application, 114-115
msgResWriteObject, 111, 112
msgSave, 35, 73
filing windows and, 253
method for, 105
terminating document and, 111, 112-113
in Tic-Tac-Toe application, 113
msgSave, 27
msgScrAddedStroke, 606
msgScrCat, 609
msgScrClear, 609
msgScrComplete, 610
msgScrCompleted, 606
msgScrCount, 609
msgScrDeleteStroke, 609
msgScrGetBounds, 609
msgScrHit, 610
msgScrollbarHorizScroll, 454
normalizing scroll and, 456
scroll windows and, 457
msgScrollbarProvideHorizInfo, 454
scroll windows and, 457
msgScrollbarProvideVertInfo, 454
normalizing scroll and, 457
scroll windows and, 457
msgScrollbarUpdate, 456
msgScrollbarVertScroll, 454
normalizing scroll and, 456-457
scroll windows and, 457
msgScrollWinAddClientWin, 462
msgScrollWinCheckScrollbars, 460
msgScrollWinGetInnerWin, 457, 460
msgScrollWinGetStyle, 460
msgScrollWinProvideDelta, 461, 462
msgScrollWinShowClientWin, 462
msgScrRemovedStroke, 606
msgScrRender, 610
msgScrStrokePtr, 610
msgScrXtractComplete, 606
msgSelBeginCopy, 120
msgSelBeginMove, 120, 123-124
msgSelCopySelection, 121
handling, 128
picture segments and, 327
msgSelMarkSelection, 128
msgSelMoveSelection, 121, 125
handling, 128
picture segments and, 327
msgSelOptions, 520, 521
msgSelPromote, 521
msgSelSetOwnerPreserve, 521
msgSetOwner, 26
msgSetProp, 28, 56–57
msgSetTranslator, 588
msgShowMenu, 448
msgSPaperGetXlate, 592
msgSrollWinGetClientWin, 459
msgSrollWinSetClientWin, 459
msgStreamRead, 114
msgStreamWrite, 112
msgStrListBoxGetDirty, 473
msgStrListBoxGetStyle, 473
msgStrListBoxGetValue, 472
msgStrListBoxProvideString, 471
msgTabBarGetStyle, 508
msgTabBarSetStyle, 508
msgTableChildDefaults, 436
msgTblLayoutAdjustSections, 447
msgTblLayoutXYfoIndex, 388
msgTiffGetMetrics, 332
msgTimeRegister, 408
msgTkTableAddAsFirst, 436
msgTkTableAddAsLast, 436
msgTkTableAddAsSibling, 436
msgTkTableGetManager, 443
msgTkTableRemove, 441
msgTrackDone, 528
msgTrackHide, 528
msgTrackShow, 528
msgTrackUpdate, 528
msgVersion, 56
msgViewGetDataObject, 174
msgViewSetDataObject, 174
msgWin*OK messages, 247
msgWinBeginPaint, 240
msgWinBeginRepaint, 226, 237
msgWinCleanRect, 246, 247
msgWinCopyRect, 243
msgWinDevBindPixelmap, 257
msgWinDevGetRootWindow, 255, 256
msgWinDirtyRect, 239, 294
msgWinDumpTree, 252
msgWinEndPaint, 240
msgWinEndRepaint, 240
msgWinEnum, 367
msgWinExtract, 234
deleting insertion pad and, 587
msgWinFindTag, 235, 368
clsTkTable and, 435
modifying items in toolkit table and, 436
msgWinGetBaseline, 251, 387
msgWinGetDesiredSize, 249, 250
custom layout constraints and, 392
UI Toolkit and, 368
msgWinGetFlags, 234–235
msgWinGetLayout, 141, 247, 248
baseline alignment, 251
layout processing, 249
in layout speedup, 365
lazy layout and, 397
progress bar and, 536, 537
scrollwin layout and, 460
self's desired size, 250
shrink-to-fit, 250
UI Toolkit programming and, 366
when sent, 248
window border layout and, 378
msgWinLayoutSelf, 249, 250, 381
clsCustomLayout and, 382
clsTableLayout and, 382
function, 381
label child windows and, 415
label layout and, 414
scrollwin layout and, 460
table layout constraints and, 386
UI Toolkit and, 368
msgWinMoved, 246–247
msgWinRepaint, 226, 237
application printing and, 302
child windows and, 415, 416
explicitly painting and, 240
image devices dirty windows and, 259
painting window with, 239
repainting process and, 242–243
in writing paint/repaint code, 241
wsSynchRepaint flag and, 242
msgWinSend, 127, 368
in gesture propagation, 369
toolkit table manager notification and, 438–439
toolkit table notification messages and, 438
window hierarchy and, 251–252
msgWinSetFlags, 234–235
window input flags and, 569
msgWinSetLayoutDirty, 249
msgWinSetPaintable, 235, 237
msgWinSetTag, 235
msgWinSetVisible, 235
msgWinSize, 246–247
msgWinSort, 252
msgWinStartPage, 302
msgWinUpdate, 237
to tell window to repaint, 239
msgXferAppTranslator, 595
msgXferGet, 121, 126
picture segments and, 326–327
msgXferList, 121, 125
picture segments and, 326–327
msgXlateCompleted, 606
msgXlateData, 606
msgXlateMetricsSet, 605
msgXlateSetFlags, 605
msgXlateStringSet, 605
Mutation capability, 28

Nesting, 241
of controls, 363
newArgsSize, 48
.NEW structure, 17, 18
Nonflow pagination, 137
Notebook application, 68
activating document and, 102
Application Framework and, 68
creating document and, 102
defined, 67
document page number in, 178
file system and, 89
layout messages and, 244–245
sections, 89
INDEX

table of contents, 78, 89
terminating document and, 110
Notebook User Interface (NUl), 75, 349
embedding documents with, 77
NOTE_METRIC structure, 489
NOTE_NEW ONLY structure, 16, 17
NOTE_NEW structure, 16, 17, 488
Notes, 487-496
application error, 494
application-modal, 487
sample, 487
contents from resource files, 489-90
creating, 488-490
destroying, 492
dismissal of, 491
flags, 489
kinds of, 487-488
layout, 492
messages, 488
notification, 491
option sheets instead of, 490
painting, 492
progress, 493-494
standardized messages, 487
system error, 494
system-modal, 487-488
vs. application-modal, 490
unknown error, 495
using, 490
see also clsNote
Notification
button, 420-421
simple activation and, 420-421
unwelcome, 421
choice, 443
client, scrollbar, 454-456
close box, 507
font list boxes, 473
frames, 501-503
gesture, 408
icons, 525
internal, 405-408
list boxes, 467-468
menu button, 446
messages, 399
notes, 491
option card, 516
scribbles translator, 608
scrollbar, 453-454
scrollwin, 461-462
string list boxes, 472
submenu, 450
toolkit table, 438
trackers, 528
Notifying, observers, 52-53

ObjCallWarn() macro, 23-24
objcCapCall, 25, 26
objcCapCreate, 28
objcCapCreateNotify, 28, 56
objcCapFree, 26
objcCapInherit, 27
objcCapMutate, 28
objcCapObserver, 27, 29, 51
objcCapOwner, 26
objcCapProp, 28
objcCapScavenge, 27
objcCapSend, 26, 29
ObjectCallAncestor(), 24, 36
cxt parameter and, 39
msgSave and, 112
self parameter and, 38
ObjectCallAncestorCtx(), 36
using, 37
ObjectCall() function, 13, 19
argument data pointer and, 14
capability, 25, 26
macros, 24
for notifying observers, 52
in object destruction, 57
return values and, 14
self parameter and, 38
ObjectCallWarning() function, 23
OBJECT_NEW ONLY structure, 24
contents, 47
Object-oriented programming, 5-6
literature on, 7
ObjectPost() function, 19, 21-22
compared to ObjectCall(), 19
in object destruction, 57
prototype, 22
reasons for using, 21
ObjectPostU320() function, 22
ObjectRead, cxt parameter and, 39
Objects
argument structures, 15-17
capabilities, 25-29
component, 92
copying, 49-50
creating, 15-18, 34
with default values, 18
declared, 5
destruction of, 57-59
documents as, 92
filter, 547
grabber, 547, 549
handwriting capture, 555
identifiers for, 9-13
information about, 54-56
checking object's version
number, 56
checking object's validity, 55-56
confirming object's ancestor, 55
class's class, 55
getting object's class, 55
getting owner of object, 55
lightweight, 218-219
listener, 548, 549
manager, 423
mutating, 60
notifying observers and, 52
observing, 50-52, 78
properties of, 56-57
saving and restoring data, 77
scavenging, 60
target, 548, 550
toolkit components, 367
translator, 555
types of, 5
window tree, 548
ObjectSend() function, 19, 20-21
capability, 26
compared to ObjectCall(), 19
event processing, 563, 565
functions related to, 21
prototype, 20
reasons for using, 20
self parameter and, 38
ObjectSendU320() function, 21
ObjectSendUpdate() function, 21, 251
ObjectWrite() function, 35
cxt parameter and, 39
in document activation, 104
in document reactivation, 114
pData parameter and, 39
OBJ_ENUM_ENUM_OBSERVERS structure, 51-52
OBJ IS A structure, 54
OBJ MUTATE structure, 60
OBJ_NOTIFYObservers structure, 52-53
OBJ_PROP structure, 56-57
OBJ_SAVE structure, 112
objWildCard, 45
objWKNKey, 24
Observable capability, 27
Observed object class, 70
Observer, 50
adding an, 50-51
with position, 51
getting from list, 51-52
messages, 50, 150
notification, 50
example of, 53
notifying, 52-53
posting to, 53
removing, 51
OK() macros, 24
Opcode, grafic, 319
correct, 325
opCodeMaskInvisible flag, 326
Open document, 67
open figures, 271
drawing, 293
messages, 283
Opening
application window, 196
documents, 107–109
fonts, 275-276, 303-305
Option card, 511
current, 515
destroying, 516
illustrated, 511
layout, 515
manipulating, 514–515
notification, 516
painting, 515–516
performance, 516–517
tagging, 515
TK_TABLE_ENTRY array for, 432-433
OPTION_CARD structure, 169, 514
OPTION_NEW_ONLY structure, 514
Option sheets, 505, 511–522
check gesture processing, 517–521
run-through, 517–519
command sheets and, 521–522
creating, 514
displaying, 517
illustrated, 511
increasing performance of, 516–517
instead of notes, 490
manipulating cards and, 514–515
messages, 512–513
option tables and, 521
for printing, 138–139
protocol, 517
styles of, 514
see also clsOption
Options menu, protocol, 168–169
OPTION_STYLE structure, 514
Option tables, 428, 521
OPTION_TAG structure, 138, 167, 169
OSHeapBlockAlloc(), 35
OSProcessCreate, 151
OSProgramInstantiateO, 96
in activating documents, 102
OSThisApp, 399
Out arguments, 14
Out box, printing and, 135, 302
Outer rectangle region, 379
Overwrite fields, 475
Owner capability, 26

Page numbers, 509
Paginating
decisions, 137
flow documents, 137
nonflow documents, 137
paginationMethod, 141
PenPoint Class Diagram, 6
Physical line width, 340
PIC_SEG_GRAFIC structure, 319
PIC_SEG_HIT_LIST structure, 325
PIC_SEG_METRICS structure, 322
PIC_SEG_OBJECT structure, 323
PIC_SEG_PAINT structure, 320
picSegTopGrafic, 324
Picture segment, 279
building up, 323
changing contents of, 322
colors and, 321
converting, format, 326
copying, 326–327
creating, 318–319, 322
drawing in, 322
drawing messages, 322–323
drawing other objects in, 323
delaying, 324
hit testing, 325
moving, 326–327
painting, 322
storage, 321
TIFF images in, 333
Picture segment class, 317–327
DC state, 320–321
developer’s quick-start, 318–319
graphics, 319–320
messages, 317–318
moving and copying, 326–327
using, 322–323
in graphics applications, 323–326
see also clsPicSeg
Pixelmap, 256
allocating own, 258
binding image device to, 257–258
cached, 273
information, 257–258
memory, 258
multiple, 262
pictures, 525

Pixels
accessing, in image window, 259–262
alignment, 269
child windows and, 244
clsIcon and, 525
coordinate rounding errors and, 269
copying, 259–260
in windows, 243–244
damaged, 222
dirty, 222
going and setting values, 259
information, 257–258
LDC, 337
memory management, 258
metrics, 338
nesting and, 241
patterns and, 291
raster operations and, 292–293
removing frame decorations, 140–141
Print Setup dialog sheets, 136
Print Setup option sheet, 136
for changing margins, headers, footers, 138
processCount parameter, 72
Process, document, 92
Progress bar, 531–540
client responsibilities, 539–540
concepts, 531–532
creating, 535
custom labels, 532
providing, 539–540
defined, 531
illustrated, 532
messages, 535–537
inherited, 540
metrics, 534, 536–537
determining, 536
modifying, 537
region appearance, 537–539
shrink-to-fit, 540
style, 532–533
determining, 536
modifying, 536
tick marks, 531
value determination, 538
see also clsProgressBar
PROGRESS_METRICS structure, 532, 534
Progress note, 492–493
PROGRESS_PROVIDE_LABEL structure, 539–540
PROGRESS_REGION structure, 537
PROGRESS_STYLE structure, 532
styles, 533
PROGRESS_VIS_INFO structure, 538
Properties, 56–57
capability, 28
creating, 56–57
retrieving, 57
prPaginationFlow, 141
prPaginationTile, 141
Pruning, 441
P_TK_TABLE_ENTRY structure, 434
P_WIN_METRICS pointer, 227
window metric messages and, 233–235
Quick Help, 370
clsGWin and, 617
ID, 370
input filters and, 548–549
Quick start, developer’s capturing and translating handwriting, 555–558
creating a choice, 352–353
creating buttons, 354
creating custom layout window, 353–354
creating menu bar, 353
creating tabular layout window, 353
handling low-level pen input, 558–562
simple menu with nested buttons illustrated, 355
UI Toolkit, 352–355
rand() function, 25
Raster line, 33
Raster operation, 292–293
dynamic drawing and, 293
XOR, 293
Reactivating, documents, 113–115
Rectangle figure, 272
with borders, 338–339
illustrated, 338
bounding, 340–341
clipping, 277, 338
drawing, 294
line thickness in, 341
rounding problems, 294
without borders, 337–338
illustrated, 338
Reference buttons, 117
copying and, 122
marks and, 129
moving, 119, 122
Regions (progress bar), 531
bounds determination, 538–539
color and pattern, 538
manipulating, 537–539
structures for, 537–538
size of, 538
relWin value, 393
alignment edge and, 393
constraints and, 392
relative window and, 394
width and height dimensions and, 393–394
Rendering
capability, initializing windows with, 560–561
scribbles, 607
translated text, 592
for visual feedback, 591
Reprinting, 222–223
code, 238
writing, 241–242
dynamics, 237–243
avoiding repaints, 241
dirty windows, 237
explicit painting occasions, 240
nesting, 241
document, 100
  diagram of, 100
  transitions, 101
dormant, 100, 101
  list boxes, 467
Stationary, loading and unloading, 153–154
Status codes, input event, 564, 569
Status values, 11–12
  defined, 11
StdErrorO procedure, 494
StdMsgO procedure, 493
StdMsgResO procedure, 493
StdProgressDownO procedure, 494
StdProgressUpO procedure, 493–494
StdSystemErrorO procedure, 494
StdUnknownErrorO procedure, 495
Stenciling image device, 260–262
String list boxes, 470–473
  control dirty, 472
  creating, 470–471
  destroying, 472
  notification, 472
  painting, 473
  providing entries, 471
  value, 472
Strings, label, 410
STRLB_NEW_ONLY structure, 470
STRLB_PROVIDE structure, 471
STRLB_STYLE fields, 471
Stroke, 607
  indexing, 607
stsAppMgrLowMemNoActivate, 148
stsBadAncestor, 56, 60
stsBadObject, 56
stsEndOfDayData, 139
stsNotUnderstood, 42
stsOK, 12, 18
stsProtectionViolation, 24
stsScopeViolation, 56
stsSizeLimit, 60
stsTblLayoutBadConstraint, 386
stsTruncatedData, 12
stsVetoed, 58
Subclasses, of frames, 505
Subclassing
  clsAppMonitor, 155
  clsBorder, 380
  clsPaper, 392–393
  clsTkTable, 443
  clsView, 175
  clsWin, 226
Submenus
  creating, 448
dynamic, 449
  menu buttons displaying, 448–450
Subsystem, 545
  handwriting capture, 555
  handwriting translation, 546, 551–553
  hierarchy, 545
input, 545, 546–550
  API, 567–573
  constants, 568–570
  procedures, 571–573
  UIDs, 568
  window and graphics, 546
SYSDC_CACHE_IMAGE structure, 300
SYSDC_COPY_IMAGE structure, 300
sysDcDrawDynamic flag, 293
sysDcDrawFast flag, 293
sysDcDrawRectangle, 212
SYSDC_FONT_ATTR structure, 305
SysDcFontID, 303
SYSDC_FONT_METRICS structure, 307–308
SysDcFontString, 303
sysDcHoldDetail, 291
sysDcHoldLine, 291
SYSDC_IMAGE_INFO structure, 297–298
SYSDC_LINE structure, 291
SYSDC_MODE structure, 292–293
sysDcPatternNil, 294
SYSDC_PIXEL structure, 259
sysDcRopCopy, 292–293
SYSDC_TEXT_OUTPUT structure, 311
sysDcWindingFill flag, 292
System
  dialog, 493
  errors, 494
  System-modal note, 487–488
  application-modal note vs., 490
System preferences, 170
Tab
  clipping and, 230
  setting, 183–184
TAB_BAR_NEW_ONLY structure, 508
Tab bars, 428
  adding items to, 508
  creating, 508
  layout, 508
  windows, 223
Table layout, 383
  constraints, 386–387
  flags, 384–385
  sample, 383
  specifying, 385–386
  structure, 384–385
  using tAlignBaseline for, 388
see also clsTableLayout
table mapping, 131
Tabular layout window, 353
tagAppMgrDefaultDocName, 98
TagNum() macro, 12
Tags, 12
  window, 216, 225
Target, 129
  input, 572–573
  objects, 548, 550
  defined, 550
  record, 130
TBL_LAYOUT CONSTRAINT structure, 386
TBL_LAYOUT_COUNT structure, 385
  table layout constraints and, 386
TBL_LAYOUT_INDEX structure, 388–389
TBL_LAYOUT_SIZE structure, 386
  table layout constraints and, 386
TBL_LAYOUT STYLE structure, 384
Terminating, documents, 110–113
Text
  characters in, 312
  drawing, 311–312
  fitting, 313
  measuring, 312
  spacing, 312
  splicing, 313
  widths, 309
Text operations, 269, 277–279
  drawing, 276–277
  fonts, 275–276
  opening, 275–276
  scaling, 276
  messages, 284
thePrintManager, 139
theRootWindow, 211, 216, 223, 255
theScreen, 255
theSelectionManager, 168
theSpellManager, 168
theSystemPreferences, 170
  screen orientation changes and, 262
theWorkingDir, 10
Thumbing
  scrollbar, 454
  vs. line/page scrolling, 455–456
Tick marks, 531
Tic-Tac-Toe application
  application class creation, 86
  initialization routine, 97–98
  main routine, 97
  moving in, 123–127
  determining data type, 125–126
  move/copy icon, 125
  presenting, 124–125
moving data, 126–127
user move request, 123
msgAppClose method for, 110
msgAppInit method for, 105
msgAppOpen method for, 108
msgInit method for, 104
msgRestore method for, 114–115
msgSave method for, 113
notifying observers, 53
TIFF_NEW_ONLY structure, 332
TIFF (tagged Image File Format), 331
images, 331
metrics, 332
painting, 333
in picture segments, 333
object creating, 332
destroying, 333
filing, 333
repainting, 333
Tiling, 292
pagination method, 137
Title bar, 507
TkDemo application, 432–433
illustrated, 433
TK_TABLE_ENTRY, 429
array, 430
for menu contents, 448
for option card, 432–433
child window creation and, 434
defining statistically, 430
fields
interpretation, 429
menu button, 448
flags, 438
values in, 430–431
submenus and, 448
TkTableFillArrayWithFonts, 441
TKTABLE.H, 435
TK_TABLE_NEW_ONLY structure, 428, 434,
tlChildrenMax constraint, 387
tlGroupMax constraint, 386–387
tlMaxFit constraint, 386
T tแพge tables, 427, 442
modifying, 442
Tokens (mark), 130
comparing, 201
creating, 200
deleting, 201
implementing, 130–132
stamp mapping and, 131–132
storing, 131
table mapping and, 131
Toolkit
ancestors, 367–370
borders, 370
embedded windows, 370
gesture windows, 368–370
objects, 367
windows, 367–368
see also UI Toolkit
Toolkit tables, 364, 425–444
changing defaults, 434–435
flags to modify items, 435
low-level customization, 435
specifying item class, 435
using default item class, 434–435
choices and, 442–444
creating, 428–435
buttons and, 429
toing defaults and, 434–435
child windows, 434
class-dependent creation information, 429–433
common creation information, 428
displaying installed fonts, 441
flags, 430–432
modifying items with, 435
values, 431
installed fonts and, 441
kinds of, 427–428
layout, 437
list boxes vs., 470
managers, 438–440
messages, 427
modifying, 436
notification, 438
painting, 436–437
removing items from, 441
sample, 426
scroll windows and, 462
toggle tables, 427, 442
window tags, 436
see also clsTkTable; UI Toolkit
Topping, border windows, 377
Trackers, 527–528
destroying, 528
drawing, 528
notification, 528
see also clsTrack
Translating, captured scribbles, 557–558
Translation
data structures, 598–600
X LATE_CASE METRICS, 599
X LATE METRICS, 598–599
X LATE NEW, 599
X LATE NEW ONLY, 599
handwriting, flags, 600–602
messages, 604–606
control messages, 606
creating translator, 605
initialization messages, 605
notification messages, 606
score, 600
templates, 602–603
Translation classes, 597–606
data structures, 598–600
handwriting translation flags, 600–602
hierarchy of, 598
translation messages, 604–606
translation templates, 602–603
Translation template, 601
modes, 603
types, 602–603
Translator, 479
clsSPaper, 595–596
creating, 605
notification, 608
object, 555, 597
setting, 587–588
Transparency (color) values, 295
Traversal drivers, 129
TrtViewSelBeginMoveAndCopy() function, 124
TrtViewXferList() function, 125
UI components, 349
UID
component, 203
creating new objects and, 15
dynamic, 9
getting and setting, 181
input subsystem, 568
of root container application, 187
scope and type, 10
well-known, 9–11
UI Toolkit, 136
additional information, 356
button table class, 236
choices, 450
classes, 209, 357–360
clsGWin and, 617
creating, 15
dynamic, 9
getters and setters, 181
input subsystem, 568
of root container application, 187
scope and type, 10
well-known, 9–11
UI Toolkit
additional information, 356
button table class, 236
choices, 450
classes, 209, 357–360
clsGWin and, 617
creating, 15
dynamic, 9
getters and setters, 181
input subsystem, 568
of root container application, 187
scope and type, 10
well-known, 9–11
bounds, 232
setting, 234
child, 216
altering, 246
labels and, 415–416
client, 92
closing document and, 109
creating for application frame, 212–213
document termination and, 112
frame layout and, 501
positioning scroll window, 461
clipping, 216, 219–221
regions, 220
copying pixels in, 243–244
creating, 212, 217
size and position, 232
style flags, 232
damaged, 222
copied pixels and, 243
debugging, 252
decoration, 382
defined, 215
delta, 377
destination, 259–260
destroying, 218
devices, 210, 217
cached images for, 273–274
setting, 233
dirty, 222, 237
image devices and, 259
marking entire, 239
receiving msgWinRepaint and, 237
region, 238–239
drawing context bound to, 210
drawing in, 236
embedded, 117–118
child, 193
creating, 190
destroying, 190
metrics, 190
moving or copying, 191–193
between, 118–119
style of, 190–191
toolkit ancestors and, 370
UUID, 193
enumerating, 216, 225
environment information, 253
extracting, 218, 234
filing, 216, 225, 253
filling, 294
flags, 227, 228–229
input, 228, 569–570
setting, 234–235
style, 228, 229, 232
floating, 163
gesture, 368–370
messages, 617–618
using, 617–618
graphic classes and, 210–211
grouping, 223
image, 259–262
ImagePoint, 210
creating and inserting, 556–557
initializing, 560–561
inserting, 217, 233–234
layout, 216, 224–225, 247–248
adding child windows to, 381
classes, 381–382
dirty, 249, 365
episodic, 225
parent-veto, 225
processing, 249
shrink-wrap and, 397
unconstrained, 224
leaf, 565
life cycle, 217
lightweight, 216, 218–219
main, 92
application, 504
initializing
clsPaper-based, 593–594
setting, 163
management, 225–226
messages
creation, 230
display, 230–231, 235–244
filing, 231
layout, 231, 244–251
management, 231, 251–253
metrics, 230, 233–234
sending to DCs, 289–290
sending to hierarchy, 251–252
metrics, 227–228
moving, 246–247
off-screen, 297
orphan, 217
overview, 209
painting, 239
parent, 216
setting, 233
printer, 301
repainting, 212, 216, 222–223
dynamics, 237–243
painting and, 236
resizing, 246–247
root, 216
scroll, 457–462
scrollwin, 459
sibling relationship of, 216
size and position, 223–224
sorting, 216, 225
subclasses, 551–552
using, 219
table layout, 383
tagging, 216, 225
tags
setting, 235
tag field for, 228
toolkit components, 367–368
transparency, 223
tree, 215, 216, 550
extracting window from, 550
illustrated, 217
inserting window into, 550
objects, 548

see also Window system

Window and graphics subsystem, 546

Window class, 227–234
clsWin messages, 230–231
clsWin structures, 227
creating new window and, 232
layout messages, 244–251
management messages, 251–253
summary, 254
window display messages, 235–244
window metrics messages, 233–235

see also clsWin

Window device classes, 255–263
image devices, 256–263
windowing devices, 255

see also clsWinDev

Windowing device, 255
defined, 255

Window layout, 381–382
classes, 381–382

Window system, 215–226
caching desired sizes and, 250
concept overview, 215–226
elements, 225
imaging devices and, 256
layout and geometry capture, 251
layout episode, 249
in repaint process, 242–243
smart repainting and, 238–239
subclassing clsWin and, 226
windowing devices and, 255

see also Window

win.input.flags field, 565

WIN_METRICS structure, 227–228, 254
caching desired sizes and, 250
child windows and, 246
laying out self and, 250
window metric messages and, 233–235

WIN_NEW structure, 232
WIN_SEND structure, 251
WIN_SORT structure, 252

WKNAdmin() macro, 12
WKNScope() macro, 12
WKNValue() macro, 12

Wrapper. see Printing, wrapper
WriterAppInit(), 595
WriterAppTranslator() method, 595
WriterCompleted(), 592
wsCaptureGeometry flag, 247, 251
wsChildrenStay flag, 244
wsClipChildren flag, 236
wsClipParent flag, 236

wsClipSiblings flag, 236
wsDntNotDirty flag, 243
wsFileInLine flag, 253
wsFileLayoutDirty flag, 365
wsFileNoBounds flag, 253
wsGrow flags, 246, 289
wsLayoutDirty flag, 253
wsLayoutMinPaint flag, 248
wsLayoutNoCache flag, 250
wsLayoutResize flag, 249

laying out self and, 250
wsPlaneMask flag, 243
wsPlanePen flag, 243
wsSaveUnder flag, 241
wsSendFile flag, 366
wsSendGeometry flag, 246, 251
wsSendIntraProcess flag, 252
wsShrinkWrapHeight flag, 250
label layout and, 414
wsShrinkWrapWidth flag, 250
label layout and, 414
wsSrcNotDirty flag, 243
wsSynchPaint flag, 237, 242
wsSynchRepaint flag, 237, 242

XferMatch, 327
xferPicSegObject, 326
xferString, 126
xgsITap gesture, 408
xgsQuestion gesture, 408

xlate argument, 479
XLIST_ELEMENT structure, 599
XLIST.H, 598

XTemplateCompile() function, 602
XTM_ARGS structure, 602
xtmMode value, 603
xtmType value, 602–603

XTRIAL_MODE value, 603
xtmType value, 602–603
XTRACT.H, 598
X-Y distribution, 565

Zooming, 502–503
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:

<table>
<thead>
<tr>
<th>Question</th>
<th>Poor</th>
<th>Average</th>
<th>Excellent</th>
</tr>
</thead>
<tbody>
<tr>
<td>How useful was this book?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Was information easy to find?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Was the organization clear?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Was the book technically accurate?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Were topics covered in enough detail?</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
PenPoint™ Architectural Reference, Volume I

PenPoint Architectural Reference, Volume I, together with Volume II, provides a comprehensive description of the Application Programmatic Interface (API) for the PenPoint operating system. This volume describes the functions and messages that you use to manipulate classes, and the fundamental classes used by almost all PenPoint applications. The topics discussed in this volume include the:

PenPoint class manager
PenPoint Application Framework™
ImagePoint™ graphics imaging model
PenPoint user interface toolkit
PenPoint input subsystem and handwriting translation.

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 II presents the concepts of the supplemental PenPoint classes.
PenPoint API Reference, Volume I provides a complete reference to the fundamental PenPoint classes, messages, and data structures.
PenPoint API Reference, Volume II provides a complete reference to the supplemental PenPoint classes, messages, and data structures.

GO Corporation was founded in September 1987 and is a leader in pen computing technology for mobile professionals. The company's mission is to expand the accessibility and utility of computers by establishing its pen-based operating system as a standard.