I N S I D E   M A C I N T O S H

QuickDraw GX Environment and Utilities

Addison-Wesley Publishing Company

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

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

ISBN 0-201-40676-4
1 2 3 4 5 6 7 8 9-CRW-9897969594
First Printing, June 1994

The paper used in this book meets the EPA standards for recycled fiber.

Contents

Figures, Tables, and Listings

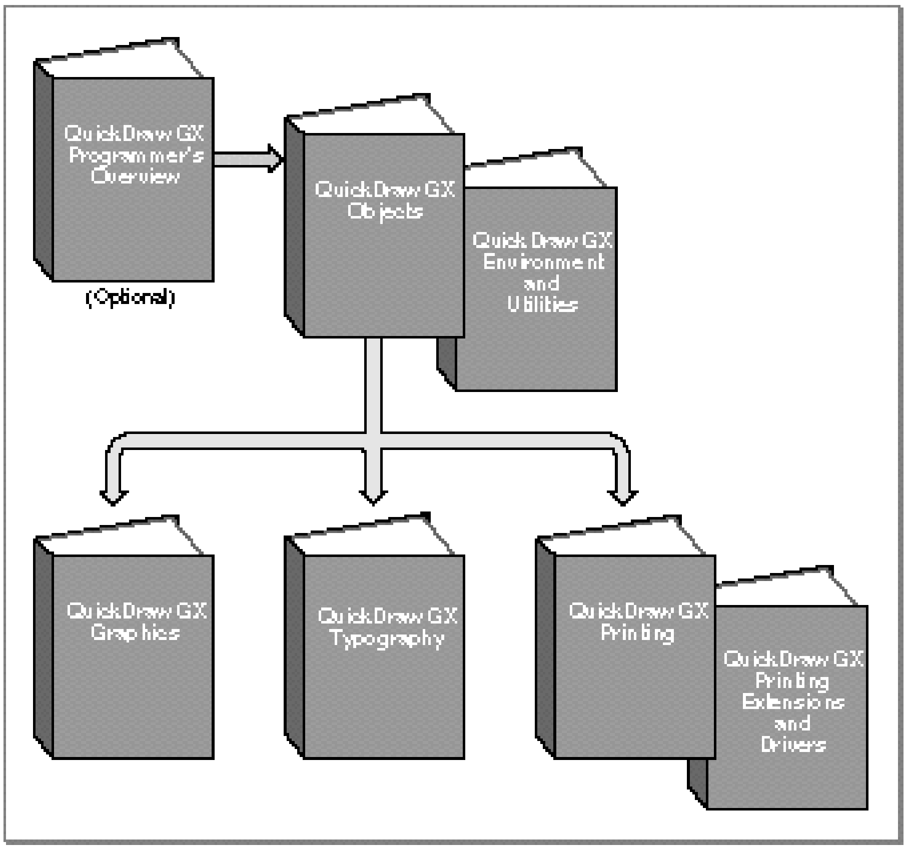
About This Book

QuickDraw GX is an integrated, object-based approach to graphics programming on Macintosh computers. This book, *Inside Macintosh: QuickDraw GX Environment and Utilities,* describes a wide variety of QuickDraw GX application-development topics.

For application programming purposes, QuickDraw GX augments or replaces the capabilities of some of the Macintosh system software managers documented in other parts of *Inside Macintosh.* In particular, The Memory Management capabilities, as described in the chapter “QuickDraw GX Memory Management,” augment the Macintosh Memory Manager described in *Inside Macintosh: Memory.* In addition, some of the mathematical functions described in the chapter “QuickDraw GX Mathematics” replace functions described in the “Mathematical Utilities” chapter of *Inside Macintosh:Operating System Utilities.* However, QuickDraw GX and other parts of Macintosh system software coexist without conflict and you can use both in the same program. Furthermore, the functions described in the chapter “QuickDraw GX and the Macintosh Environment” provide you with an interface to certain Macintosh system software capabilities outside the scope of QuickDraw GX.

Before you read this book, you should already be familiar with *Inside Macintosh: QuickDraw GX Objects.* Figure P-1 shows the suggested reading order for the QuickDraw GX books. A pictorial overview of *Inside Macintosh,* including the QuickDraw GX suite of books, appears on the inside back cover.

Figure P-1 Roadmap to the QuickDraw GX suite of books


QuickDraw GX Programmer's Overview
(Optional)
QuickDraw GX Objects
QuickDraw GX Environment and Utilities
QuickDraw GX Graphics
QuickDraw GX Typography
QuickDraw GX Printing
QuickDraw GX Printing Extensions and Drivers

What to Read

Each chapter in this book describes a separate topic that does not rely on any other chapter for its understanding. As a result, the chapters of this book may be read in any order.

- **QuickDraw GX and the Macintosh Environment.** This chapter describes those aspects of QuickDraw GX that relate specifically to the Macintosh Toolbox, Macintosh programming environment, and Macintosh image data format.
- **QuickDraw GX Memory Management.** This chapter describes the aspects of QuickDraw GX memory management that your application can control. Read this chapter if you want to understand how QuickDraw GX memory works or to supplement QuickDraw GX memory management operations.

- **Errors, Warnings, and Notices.** This chapter describes the errors, warnings, and notices that can be posted by QuickDraw GX functions and how you can manipulate them. In addition, this chapter describes how you can use application-defined handllers to provide alternative or complementary processing of errors, warnings, and notices.
- **QuickDraw GX Debugging.** This chapter describes QuickDraw GX debugging functions and the GraphicsBug debugging utility that you should use when you are writing and debugging applications.
- **Collection Manager.** This chapter describes the Collection Manager, which provides an abstract data type you can use to store collections of information. Read this chapter if you need to work with some advanced features of QuickDraw GX printing, including print dialog boxes, or if you want to create collections for purposes specific to your application.
- **Message Manager.** This chapter describes the Message Manager, which is a part of the message-passing printing architecture of QuickDraw GX. Read this chapter if you want to use the Message Manager to develop printing extensions or printer drivers.
- **QuickDraw GX Stream Format.** This chapter describes the format of the compressed data stream that results when the QuickDraw GX `GXFlattenShape` function is used. It also describes the use of such data streams by print files and portable digital documents (PDDs). Read this chapter if you need to uncompress QuickDraw GX stream format data and cannot use the QuickDraw GX `GXUnflattenShape` function.
- **QuickDraw GX Mathematics.** This chapter describes QuickDraw GX number formats, number-format conversions, mathematical functions, and functions that operation on mappings (transformation matrices). Read this chapter if your application requires the explicit use of any of the mathematical capabilities of QuickDraw GX.

Chapter Organization

Most chapters in this book follow a standard general structure. For example, the chapter “QuickDraw GX and the Macintosh Environment” contains these major sections:

- “About QuickDraw GX and the Macintosh Environment.” This section provides an overview of the Macintosh interface and describes the QuickDraw–to–QuickDraw GX translator.
- “Using QuickDraw GX in the Macintosh Environment.” This section describes how you can test for the presence and version of QuickDraw GX and use the Macintosh interface functions. It describes how to use the most common functions, gives related user interface information, provides code samples, and supplies additional information.

- “QuickDraw GX and the Macintosh Environment Reference.” This section provides a complete reference for the topics described in this chapter by describing their related constants, data types, and functions. Each function description follows a standard format, which gives the function declaration; a description of every parameter; the function result, if any; and a list of errors, warnings, and notices. Most function descriptions give additional information about using the function and include cross-references to related information elsewhere.
- “Summary of QuickDraw GX and the Macintosh Environment Reference.” This shows the C interface for the constants, data types, and functions associated with the Macintosh interface and the QuickDraw–to–QuickDraw GX translator.

Conventions Used in This Book

This book uses various conventions to present certain types of information.

Special Fonts

All code listings, reserved words, and the names of data structures, constants, fields, parameters, and functions are shown in Courier `(this is Courier)`.

When new terms are introduced, they are in **boldface.** These terms are also defined in the glossary.

Types of Notes

There are several types of notes used in this book.

Note

A note formatted like this contains information that is interesting but possibly not essential to an understanding of the main text. The wording in the title may say something more descriptive than just “Note”; for example, “Terminology Note.” u

s **WARNING**

Warnings like this indicate potentially serious problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes and loss of data. s

Numerical Formats

Hexadecimal numbers are shown in this format: 0x0008.

The numerical values of constants are shown in decimal, unless the constants are flag or mask elements that can be summed, in which case they are shown in hexadecimal.

Type Definitions for Enumerations

Enumeration declarations in this book are commonly followed by a type definition that is not strictly part of the enumeration. You can use the type to specify one of the enumerated values for a parameter or field. The type name is usually the singular of the enumeration name, as in the following example:

```
enum gxDashAttributes {
   gxBendDash        = 0x0001,
   gxBreakDash       = 0x0002,
   gxClipDash        = 0x0004,
   gxLevelDash       = 0x0008,
   gxAutoAdvanceDash = 0x0010
};
typedef long gxDashAttribute;
```

Development Environment

The QuickDraw GX functions described in this book are available using C interfaces. How you access these functions depends on the development environment you are using.

Code listings in this book are shown in ANSI C. They suggest methods of using various functions and illustrate techniques for accomplishing particular tasks. Although most code listings have been compiled and tested, Apple Computer, Inc., does not intend for you to use these code samples in your applications.

Developer Products and Support

APDA is Apple's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. APDA offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	APDA
America Online	APDAorder
CompuServe	76666,2405
Internet	APDA@applelink.apple.com

C H A P T E R 1

QuickDraw GX and the Macintosh Environment

Contents

This chapter describes those aspects of QuickDraw GX that relate specifically to the Macintosh Toolbox, Macintosh programming environment, and Macintosh image data format. The chapter addresses the following topics:

- the Macintosh interface to QuickDraw GX
- the QuickDraw–to–QuickDraw GX translator

Before reading this chapter, you should be generally familiar with QuickDraw GX and QuickDraw GX objects, as described in the chapter "Introduction to QuickDraw GX" in *Inside Macintosh: QuickDraw GX Objects*. Additional specific information related to view ports and view devices is in the "View-Related Objects" chapter in *Inside Macintosh: QuickDraw GX Objects*.

Because this chapter describes the interface between QuickDraw GX and the rest of the Macintosh Toolbox, it uses many terms defined elsewhere. For a general picture of the Macintosh Toolbox, see *Inside Macintosh: Overview* or the introductory chapter of *Inside Macintosh: Macintosh Toolbox Essentials*. For information on Macintosh windows, see the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*. Mouse location and mouse handling is described in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*. QuickDraw, QuickDraw coordinates, the QuickDraw picture format, picture comments, graphics ports, and Macintosh graphics devices are all described in *Inside Macintosh: Imaging With QuickDraw*.

About QuickDraw GX and the Macintosh Environment

QuickDraw GX provides a number of useful functions that assist your application development on the Macintosh computer. The Macintosh interface provides functions specific to the Macintosh platform that allow you to use information provided by other parts of Macintosh system software. The QuickDraw–to–QuickDraw GX translator allows you to convert from QuickDraw pictures to QuickDraw GX objects.

The Macintosh Interface

Most QuickDraw GX functions are designed for implementation on any platform. However, there are specific functions that are wrappers for Macintosh system software functions or have meaning only in the Macintosh environment. QuickDraw GX contains **Macintosh interface functions** to convert between QuickDraw and QuickDraw GX coordinate systems, find the mouse position in QuickDraw GX coordinates, associate view ports with Macintosh windows, map between view devices and Macintosh `GDevice` records, and intercept drawing commands to a view port.

The QuickDraw–to–QuickDraw GX Translator

The **QuickDraw–to–QuickDraw GX translator** can be used to convert QuickDraw drawing commands into QuickDraw GX objects. It allows you to create a set of QuickDraw GX **objects** that have a similar appearance to that intended by the original QuickDraw picture. This capability is useful, for example, for importing QuickDraw data from the Clipboard into a QuickDraw GX–based application.

It is important to note that the translator does not provide a completely faithful, pixel-by-pixel mapping of the image defined by the QuickDraw commands. However, it does closely approximate the original image, and you can even control the closeness of the approximation. In most cases the differences are subtle and not apparent to the eye.

Using QuickDraw GX in the Macintosh Environment

This section describes how you can

- determine QuickDraw GX versions
- use the Macintosh interface functions
- use the QuickDraw–to–QuickDraw GX translator

Testing for the Presence and Version of QuickDraw GX

You can use the `Gestalt` function in your application to determine which parts of QuickDraw GX are installed and their version numbers. The `Gestalt` function returns a 32-bit value that indicates the version of QuickDraw GX that is installed. The graphics and typography part of QuickDraw GX has one version number, the printing part of QuickDraw GX has another, and there is an overall version number that applies to all of QuickDraw GX. In addition, you can determine if the debugging version or the non-debugging version of QuickDraw GX is installed, and if the installed version is native to PowerPC system software.

To determine the current version of QuickDraw GX in general, you call the `Gestalt` function with the `gestaltGXVersion` selector. The function returns a value indicating the version of QuickDraw GX printing currently installed. For version 1.0, the value returned is 0x00010000.

To determine the current version of the graphics and typography parts of QuickDraw GX, you call the `Gestalt` function with the `gestaltGraphicsVersion` selector. The function returns a value indicating the version of QuickDraw GX graphics and typography currently installed. For version 1.0, the value returned is 0x00010000.

To determine the current version of the printing part of QuickDraw GX, you call the `Gestalt` function with the `gestaltPrintingMgrVersion` selector. The function returns a value indicating the version of QuickDraw GX printing currently installed. For version 1.0, the value returned is 0x00010000.

To determine if the debugging or non-debugging version of QuickDraw GX is currently installed, or if the installed version is native to PowerPC system software, you call the `Gestalt` function with the `gestaltGraphicsAttr` selector. The `gestaltGraphicsisDebugging` attribute value is returned if the debugging version of QuickDraw GX is installed. The `gestaltGraphicsisLoaded` attribute value is returned if the non-debugging version of QuickDraw GX is installed. The `gestaltGraphicsIsPowerPC` attribute value is returned if the installed version of QuickDraw GX is PowerPC-native. The return value can be any combination of those attributes.

Listing 1-1 uses the `gestaltGraphicsVersion` and `gestaltPrintingMgrVersion` selectors to determine whether QuickDraw GX graphics and typography as well as QuickDraw GX printing are installed. This listing also uses the `gestaltGraphicsAttr` selector to determine whether the installed version of QuickDraw GX is the debugging or non-debugging version.

Listing 1-1 Determining the presence and features of QuickDraw GX

```
Boolean QuickDrawGXAvailable(Boolean *pIsDebugging)
{
   Boolean returnValue = false;
   long theFeature, flags;
   if(Gestalt(gestaltGraphicsVersion, &theFeature) == noErr)
      {
         returnValue = true;
         if (Gestalt(gestaltPrintingMgrVersion, &theFeature) ==
                                                         noErr)
            gQDGXPrintingInstalled = true;
      }
   else
      returnValue = false;
   if (Gestalt(gestaltGraphicsAttr, &theFeature) == noErr)
      {
         if (flags & gestaltGraphicsisDebugging)
            pIsDebugging = true;
         else
            pIsDebugging = false;
      }
   return returnValue;
}
```

For additional details concerning the use of the `Gestalt` function to determine features of the QuickDraw GX environment, see the chapter “Gestalt Manager” in *Inside Macintosh: Operating System Utilities.*

Using the Macintosh Interface Functions

The QuickDraw GX Macintosh interface functions allow you to integrate QuickDraw GX with the Macintosh Toolbox. These functions allow you to

- create and use view ports associated with Macintosh windows
- retrieve a QuickDraw graphics device associated with a QuickDraw GX view device.
- convert between QuickDraw and QuickDraw GX coordinate systems
- intercept QuickDraw GX drawing functions for a view port

Creating and Using View Ports with Macintosh Windows

QuickDraw GX drawing takes place in **view ports.** You can associate a view port with a window in order to clip drawing to the window's visible region. Once you've created a window, you can create a view port that is associated with that window by the use of the `GXNewWindowViewPort` function. When you attach a view port to a window, you guarantee that all the shapes that you draw to the view port will be drawn in the correct location within the window, even when the window is moved. You also guarantee that if the window is underneath others that the QuickDraw GX drawing will be clipped to the QuickDraw GX window's visible region. You can attach a view port to your window with the call

```
windowParentViewPort = GXNewWindowViewPort(theWindow);
```

The resulting `windowParentViewPort` contains the view port attached to the window. You cannot change either the mapping or the **clip** of the window view port. If you need to do either—for example, if you need to control position within the view port (as when scrolling) or clip drawing within the window (so you don't draw over scroll bars), you need to create a **child view port** of your window view port and draw only to it. Child view ports, **view port hierarchies,** and how to use them are described in detail in the chapter "View-Related Objects" in *Inside Macintosh: QuickDraw GX Objects*.

Once you've created a view port, you can determine the view port that is associated with a specific window by using the `GXGetWindowViewPort` function. If you haven't associated a view port to that window, the function returns `nil`.

You can find out which window is associated with a view port by using the `GXGetViewPortWindow` function. The function returns `nil` if the view port is not associated with any window.

The `GXNewWindowViewPort` function is described on page 1-24. The `GXGetWindowViewPort` function is described on page 1-26.
The `GXGetViewPortWindow` function is described on page 1-25.

Using View Devices With Graphics Devices

On the Macintosh, every monitor gets a **graphics device,** described by a `GDevice` record. So when QuickDraw GX creates a screen **view device** for each monitor, there is already a graphics device for it. The `GXGetViewDeviceGDevice` and `GXGetGDeviceViewDevice` functions link the two worlds together, so that you can work with either description of a display device.

These functions work only with Macintosh system graphics devices. If you have a screen view device, you can call the `GXGetViewDeviceGDevice` function to get the graphics device that corresponds to that view device. If you create your own offscreen view device, it will not have an associated graphics device. Likewise, there is no view device associated with an offscreen `GDevice` record.

The `GXGetViewDeviceGDevice` function is described on page 1-27; the `GXGetGDeviceViewDevice` function is described on page 1-28.

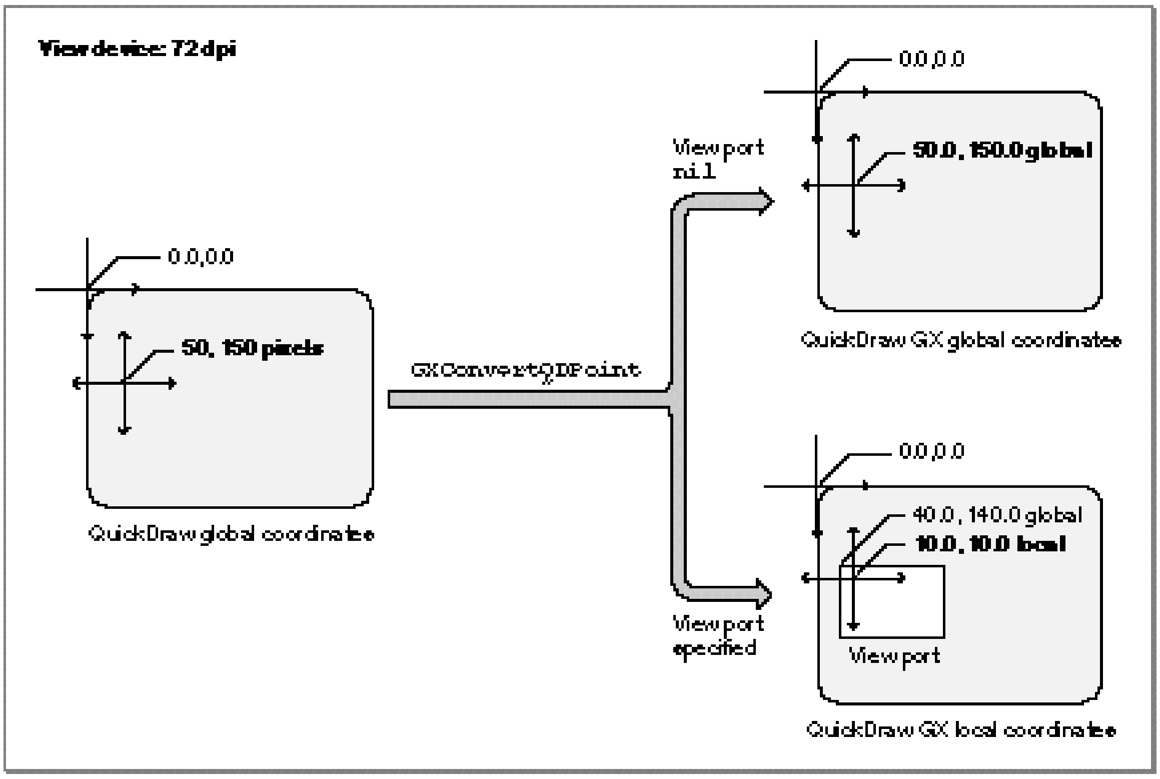
Converting From QuickDraw to QuickDraw GX Coordinates

QuickDraw GX provides several functions that involve conversion of locations on the QuickDraw coordinate plane into locations expressed in QuickDraw GX **local** or **global coordinates.**

Converting from QuickDraw Global to QuickDraw GX Local or Global Coordinates

You can use the `GXConvertQDPoint` function to convert a point having QuickDraw global coordinates to either QuickDraw GX global or QuickDraw GX local coordinates. If a view port is specified in the function's parameters, the QuickDraw point coordinates are converted to the corresponding QuickDraw GX local coordinates. If the view port parameter is `nil`, the QuickDraw point coordinates are converted to corresponding QuickDraw GX global coordinates. Figure 1-1 shows how the `GXConvertQDPoint` function converts a point having QuickDraw global coordinates of (50, 150) pixels on a monitor to QuickDraw GX coordinates in points (in which 1 point equals 1/72 inch).

Figure 1-1 Converting from QuickDraw global to QuickDraw GX local and global coordinates



When the view port parameter is `nil`, the QuickDraw global coordinates are converted to QuickDraw GX global coordinates (50.0, 150.0). When the view port is specified, the QuickDraw global coordinates are converted to QuickDraw local coordinates (10.0, 10.0) when the view port is located at QuickDraw GX global coordinates (40.0, 140.0). The local coordinates are local relative to the specified view port.

The `GXConvertQDPoint` function is described on page 1-29. For additional information about QuickDraw GX local, global, and device space, see the chapter “View-Related Objects” in *Inside Macintosh: QuickDraw GX Objects.*

Obtaining Mouse Location in Global Coordinates

The `GXGetGlobalMouse` function returns the location of the Macintosh cursor (mouse) in QuickDraw GX global coordinates. If a QuickDraw GX view device has a resolution of 72 dpi and a cursor is located at point (500, 150) pixels in QuickDraw coordinates, the `GXGetGlobalMouse` function would return the QuickDraw GX coordinates (500.0, 150.0) in points. If the resolution of the QuickDraw GX view device is 144 dpi and the cursor were at (1000, 300) pixels, the `GXGetGlobalMouse` function would again return coordinates (500.0, 150.0). No matter what the resolution of the device, the QuickDraw GX global coordinates are the same for a cursor located at a given absolute position.

The GXGetGlobalMouse function is described on page 1-30. For additional information about local, global, and device spaces, see the chapter “View-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*.

Obtaining Mouse Location in Local Coordinates

For a given view port, you can use the GXGetViewPortMouse function to obtain the mouse position in the coordinate system (local coordinates) of that view port. This function takes any scaling of local space into account; if, for example, you have a zoomed-in view, the coordinates would be relative to the zoomed coordinate system.

If you obtain the mouse point in QuickDraw global coordinates, you can take the result of the GXGetViewPortMouse function and immediately turn it into a shape. You can use the GXNewShape function with the returned point as the shape origin, and QuickDraw GX will draw the shape at the point where the mouse is located with the correct scale. If the scale factor is 10, the shape is drawn enlarged by a factor of 10.

The GXGetViewPortMouse function is described on page 1-30. For additional information about local, global, and device spaces, see the chapter “View-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*.

Intercepting Drawing Calls to a View Port

The GXSetViewPortFilter function causes QuickDraw GX to intercept all drawing function calls to a specified view port and pass them instead to an application-defined callback function that you supply. You can use the filter function to perform actions other than screen drawing, or perhaps to collect information about them.

QuickDraw GX uses this function to install a view port filter for printing. When a page is open and a call is made to draw a shape, instead of actually drawing it to the screen, the printing view port filter records (spools) it to the print file. You can use this kind of function if you want to achieve a similar result or if you otherwise want to manipulate shapes that would be drawn to a view port.

When you use the GXGetViewPortFilter function, you get back what you set with the GXSetViewPortFilter function. If you want to get rid of your view port filter, use the GXSetViewPortFilter function and specify a nil filter function.

The GXSetViewPortFilter function is described on page 1-31. The GXGetViewPortFilter function is described on page 1-32. The application-defined callback filter function is described on page 1-40.

Using the QuickDraw–to–QuickDraw GX Translator

The QuickDraw–to–QuickDraw GX translator converts QuickDraw drawing commands into QuickDraw GX shapes. There are two ways to use the translator:

- The first way is to pass the translator a handle to a QuickDraw picture. The translator returns a QuickDraw GX picture shape that approximates the original QuickDraw picture. The section "Using the Translator With QuickDraw Pictures" beginning on page 1-20 describes how to use the translator in this way.
- The second way is to use a pair of functions to install and remove the translator. After you install the translator in a given graphics port, it intercepts all subsequent QuickDraw drawing commands sent to that port and converts them to QuickDraw GX shapes. After you are finished converting, you remove the translator. The section "Installing and Removing the Translator" beginning on page 1-21 describes how to use the translator in this way.

The next section, "Factors in Translation," describes how translation works and how you can influence it by setting various translation parameters.

IMPORTANT

In order to use the QuickDraw-to-QuickDraw GX translator, you first must have called the `GXInitPrinting` function. The `GXInitPrinting` function is described in the core printing features chapter of *Inside Macintosh: QuickDraw GX Printing.* s

Factors in Translation

This section describes some of the factors that influence the translation process, and how you can manipulate them.

Graphics Port and View Port

The translation from QuickDraw to QuickDraw GX takes into account the current QuickDraw grafPort origin. Therefore, each resulting QuickDraw GX shape incorporates, either in its shape geometry or in its transform mapping, the origin of the graphics port that was active at the time of translation.

The QuickDraw GX shape that results from the translation must be associated with a view port. This can be accomplished by

- setting the view port for each shape
- setting the view port for the parent picture shape of the individual shapes contained in a picture

Scaling During Translation

The translator allows you to scale the QuickDraw data as it is converted. For example, you can use scaling to convert from a screen resolution of 72 dpi to a printer resolution of 300 dpi. You specify the scaling factor in the form of source and destination rectangles.

Also, in order to allow the translator to properly scale dash picture comments and other items, you can supply a pair of integer scale factors, which may be different in the x and y directions. The scale factors for both the source and destination rectangles and the pattern-stretch parameters are usually the destination resolution divided by the screen resolution (72 dpi), rounded to the nearest integer. Typical examples are shown in Table 1-1.

Table 1-1 Translation scaling factors

Source	**Destination**	**Scale Factor**
72 × 72	72 × 72	1 × 1
72 × 72	72 × 80	1 × 1
72 × 72	144 × 144	2 × 2
72 × 72	150 × 150	2 × 2
72 × 72	300 × 300	4 × 4

Translation Options

When you translate QuickDraw data to QuickDraw GX shapes, you specify one or more **translation options.** You can use either the default translation option provided by QuickDraw GX or a combination of the other available options. Some translation options provide simpler and faster translations, but with a resulting loss of pixel-for-pixel matching. Table 1-2 lists and describes the available translation options; the constants are defined in the `gxTranslationOptions` enumeration.

Table 1-2 Translation options settings

Constant	Value	Explanation
`gxDefaultOptionsTranslation`	0x0000	This is the default setting used for translation. This option generates the most accurate representation of the QuickDraw data that the translator is capable of producing.
`gxOptimizedTranslation`	0x0001	This option allows for optimizations to be applied during translation. For example, a sequence of QuickDraw lines can be combined into one polygon. In most cases, this results in the generation of a smaller number of QuickDraw GX shapes.
`gxReplaceLineWidthTranslation`	0x0002	The width of a resulting QuickDraw GX line is the average of the original pen's width and height. This option also affects the way in which the SetLineWidth `PicComment` is interpreted. The LaserWriter driver scales the current line width with the newly specified picture comment. The translator normally uses the LaserWriter mechanism. When you specify this option, the translator uses a mechanism in which the line is replaced with the newly specified width; this mimics the behavior of the LaserWriter SC driver.
`gxSimpleScalingTranslation`	0x0004	This option causes the translator to scale data from source resolution to destination resolution by using a simple multiplication, which is incorporated into the shape's transform. The translator makes no attempt to compensate for this increase in resolution. The resulting scaled image will not render the original QuickDraw data accurately, but will be similar to what QuickDraw would have produced when it attempted to scale the data.
`gxSimpleGeometryTranslation`	0x0008	This option results in a translation of QuickDraw data without taking into account the QuickDraw hanging pen. Normally the translator reproduces a QuickDraw triangle, for example, as a 6-sided or 7-sided polygon. This option sacrifices accuracy in order to produce an image that draws faster with QuickDraw GX and can be more useful for pen-based output devices. For example, QuickDraw lines become QuickDraw GX lines with flat endcaps. This option also turns on the simple lines translation and the simple scaling translation.

Table 1-2 Translation options settings (continued)

Constant	Value	Explanation
`gxSimpleLinesTranslation`	0x000C	This option results in simple geometry and scaling. The translator maintains the width of lines that are at an angle. Because QuickDraw uses a hanging pen, a diagonal line appears to be thicker than a horizontal or vertical line with the same pen size. Using this option causes the line width to be the same as the pen width, at the expense of the accuracy of the original QuickDraw data. This option also turns on the simple scaling translation and the simple geometry translation.
`gxLayoutTextTranslation`	0x0010	Normally the translator turns off layout-shape-specific capabilities when translating QuickDraw text into layout shapes. This option restores layout features such as default glyph substitutions. This results in a more attractive text; however it can be different from the original QuickDraw data.
`gxRasterTargetTranslation`	0x0020	This option causes PostScript picture comments to be discarded. The bitmap proxies sent along with such comments are preserved.
`gxPostScriptTargetTranslation`	0x0040	This option causes PostScript picture comments to be incorporated as tags attached to picture shapes. The bitmap proxies sent along with such comments are discarded.
`gxVectorTargetTranslation`	0x0080	This option causes PostScript picture comments to be discarded. The bitmap proxies sent along with such comments are preserved. Also, when this option is combined with the option `gxOptimizedTranslation`, lines are preserved and not combined in thick framed polygons.

How Option Settings Affect Translation of Lines

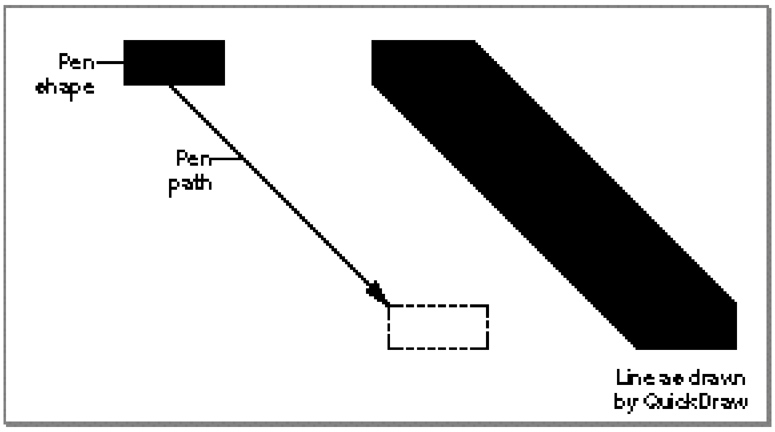
The translation of QuickDraw lines is affected by the translation options setting you choose. Consider the simple line generated by the QuickDraw commands given in Listing 1-2.

Listing 1-2 QuickDraw commands to draw a simple line

```
PenSize(5, 3);
MoveTo(100, 40);
LineTo(120, 70);
```

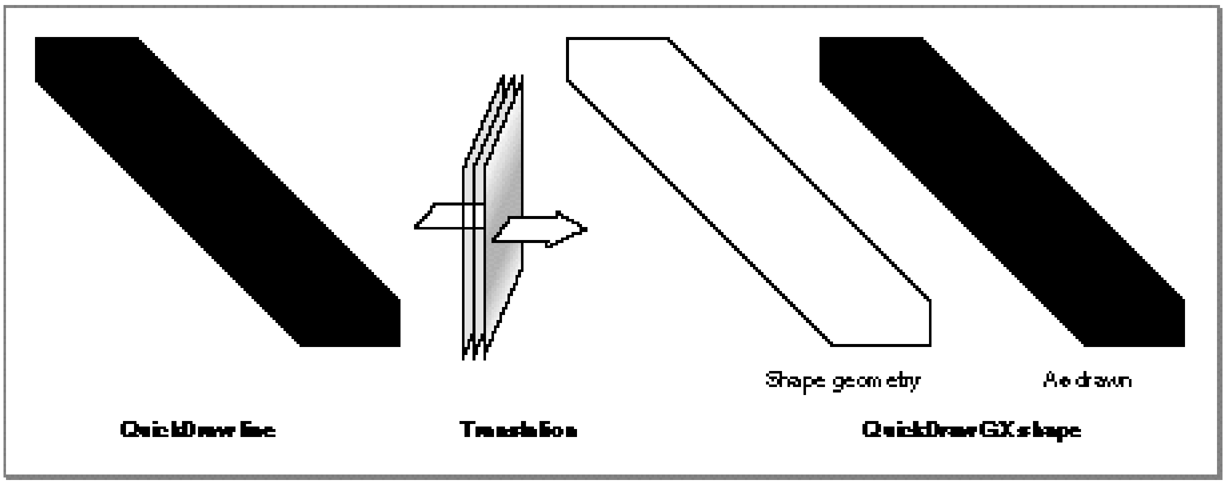
The QuickDraw commands in Listing 1-2 produce the line shown in Figure 1-2.

Figure 1-2 A QuickDraw line



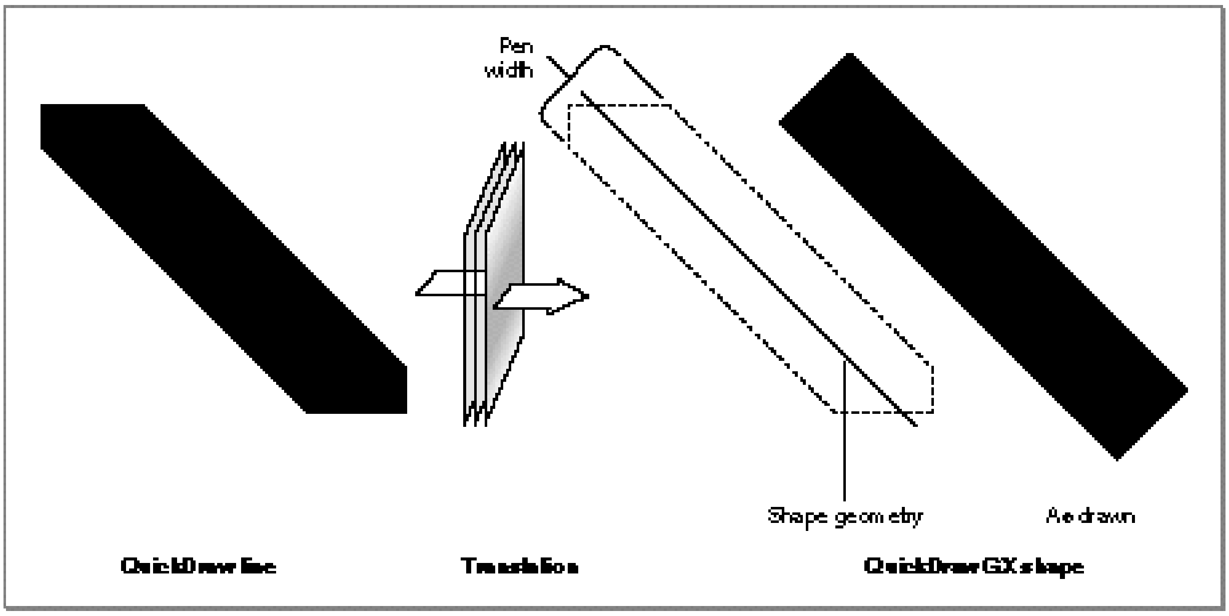
The `gxDefaultOptionsTranslation` setting produces the best replication of the original QuickDraw picture. However, it is also the slowest translation. If you use the `gxDefaultOptionsTranslation` setting for the translation of the original QuickDraw line shown in Figure 1-2, the resulting QuickDraw GX polygon shape mimics the QuickDraw hanging pen. Furthermore, any scaling between the source and destination rectangles is incorporated into the translated shape's geometry. The original QuickDraw line would be translated to the QuickDraw GX shape shown in Figure 1-3.

Figure 1-3 Translation of the QuickDraw line using `gxDefaultOptionsTranslation`



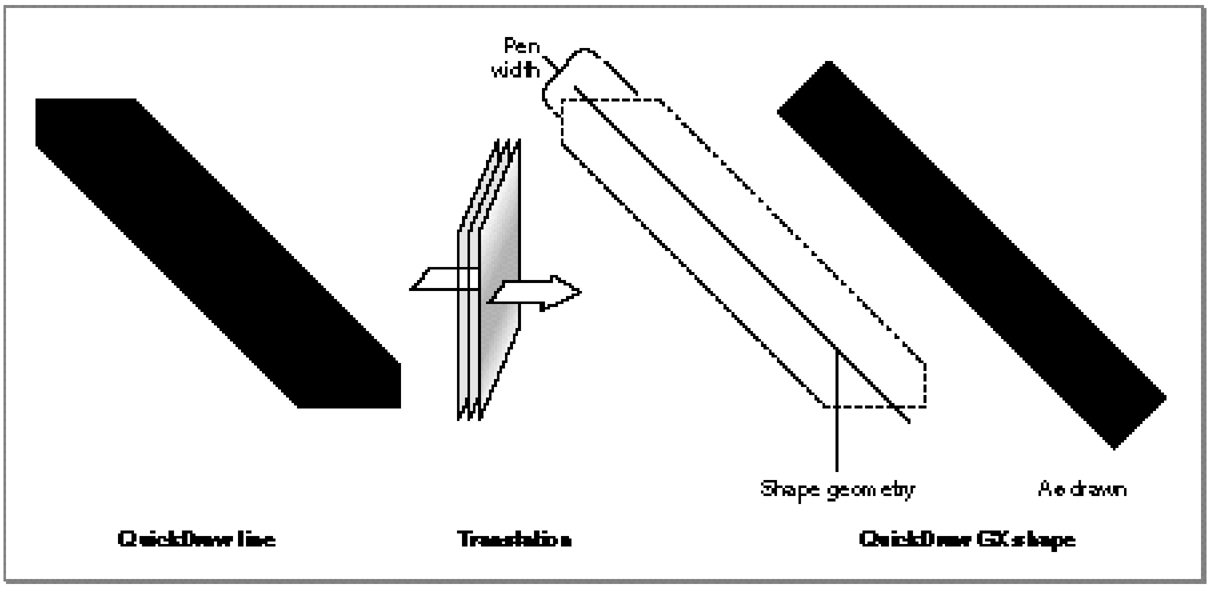
If you use the `gxSimpleGeometryTranslation` option setting, the resulting QuickDraw GX line shape runs along the center of the original QuickDraw line and covers all the pixels of the QuickDraw line and more; it is a superset. The resulting QuickDraw GX shape looks like the line shape shown in Figure 1-4.

Figure 1-4 Translation of the QuickDraw line using `gxSimpleGeometryTranslation`



If you use the `gxReplaceLineWidthTranslation` option setting, the resulting QuickDraw GX line shape has a width that is the average of the QuickDraw pen width and height. The line runs along the center of the original QuickDraw line between the extreme pixels at each end of the original QuickDraw line. The translation results in the QuickDraw GX shape shown in Figure 1-5.

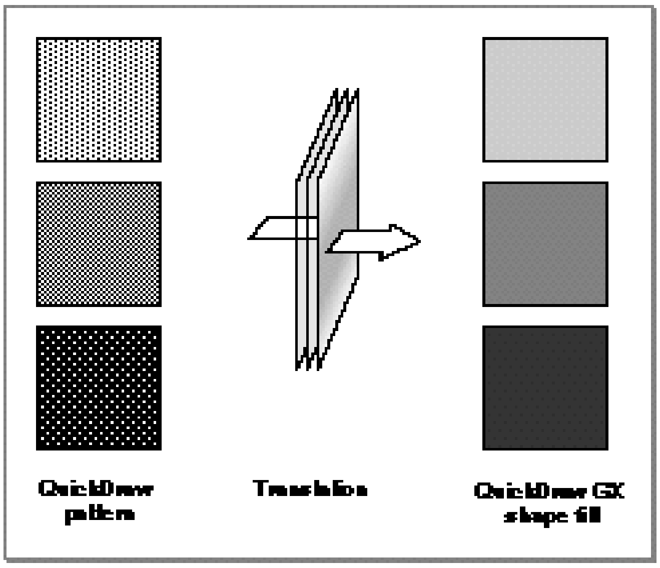
Figure 1-5 Translation of the QuickDraw line using `gxReplaceLineWidthTranslation`



Translation of Fill Patterns

The QuickDraw–to–QuickDraw GX translator converts those 8-bit × 8-bit QuickDraw fill patterns that are commonly used to represent gray patterns to colors that are blends of the foreground and background colors. In the case of QuickDraw black-and-white patterns, a uniform grayscale shade that ranges from 0 to 100 percent black is produced, depending on the overall apparent density of the original pattern, as shown in Figure 1-6.

Figure 1-6 Conversion of standard QuickDraw fill patterns to QuickDraw GX shape fills



Translation of QuickDraw Picture Comments

The capabilities of QuickDraw GX exceed those of QuickDraw. This means that a picture comment (`picComment`) can be incorporated into the translated shapes as part of the conversion process. With QuickDraw alone, picture comments can only be seen when the picture is printed, because the comments are interpreted at the printer level.

It is common practice for developers to include QuickDraw drawing commands (usually one or more **bitmaps**) within a picture comment as a proxy that provides an alternate representation of the picture comment. That way, if the `picComment` is not supported by a printer, some output—although at a lower resolution— is produced.

When processing a picture comment, the QuickDraw–to–QuickDraw GX translator typically discards the QuickDraw proxy and applies the `picComment` to the object—for example, by rotating the shape's transform or setting a dash in the shape's style. When processing PostScript picture comments, however, the translator creates a picture shape that contains QuickDraw GX **shape objects** (based on the QuickDraw proxies) as well as **tag objects** (containing the PostScript data). In this way, QuickDraw GX can render the picture both on a raster device (by drawing the items in the picture shape) and on a PostScript device (by applying the information in the tag objects).

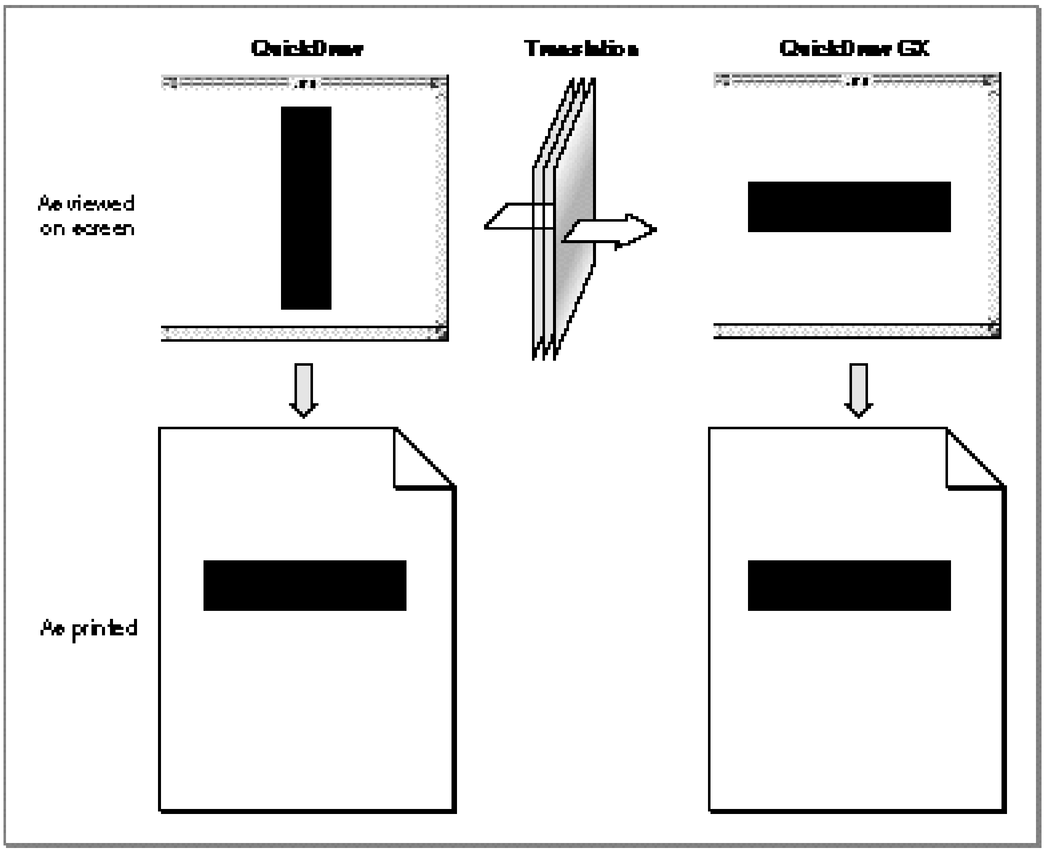
Sample code for applying a `picComment` for rotation is shown in Listing 1-3.

Listing 1-3 QuickDraw picture data that includes a `picComment`

```
RotComHandle   rInfo = NewHandle(sizeof(RotComRecord));

(*rInfo)->rFlip = 0
(*rInfo)->rAngle = 90;

MoveTo(100,100);
PicComment(RotateBegin, sizeof(RotComRecord), (Handle)rInfo);
LineTo(100, 200);
PicComment(RotateEnd, 0, nil);
```

The output of the sample code in Listing 1-3 is shown in Figure 1-7. Notice that the QuickDraw screen output is not rotated. This is because QuickDraw picture comments are interpreted by the printer. In contrast, the printed QuickDraw output and the translated QuickDraw GX shape (both printed and displayed onscreen) correctly represent the intent of the original QuickDraw data.

Figure 1-7 Translating QuickDraw data containing a rotation `picComment`



Translation Statistics

The translator keeps various statistics about the QuickDraw picture data that it translates. You can examine these statistics after the translation if you are interested in this information. The statistics information is returned in the form of bit flags, as shown in Table 1-3.

Table 1-3 Translation statistics options

Constant	Value	Explanation
`gxContainsFormsBegin`	0x0001	The data that was translated contained "`formsBegin`" picture comments.
`gxContainsFormsEnd`	0x0002	The data that was translated contained "`formsEnd`" picture comments.
`gxContainsPostScript`	0x0004	The data that was translated contained PostScript picture comments.
`gxContainsEmptyPostScript`	0x0008	The data that was translated contained PostScript picture comments in which there was no actual PostScript data.

Using the Translator With QuickDraw Pictures

If you have a handle to QuickDraw picture data, such as from a file or on the Clipboard, you can convert that data into a QuickDraw GX picture shape with a single call to the QuickDraw–to–QuickDraw GX translator.

You use the `GXConvertPICTToShape` function to translate an entire QuickDraw picture into a QuickDraw GX shape. You pass the picture handle of the QuickDraw picture you wish to translate and a reference to a shape into which the translated data is to be placed. Listing 1-4 is a sample that uses the `GXConvertPICTToShape` function to perform the translation, and then draws the resultant picture shape to the view port specified in the view port array `thePorts`.

Listing 1-4 Translating QuickDraw picture data with `GXConvertPICTToShape`

```
aPicShape = GXNewShape(gxPictureType);

GXConvertPICTToShape(thePicHdl, gxDefaultOptionsTranslation,
                     &theRect, &theRect, styleStretch,
                     aPicShape, nil);

GXSetShapeViewPorts(aPicShape,1,thePorts);
GXDrawShape(aPicShape);
GXDisposeShape(aPicShape);
```

Installing and Removing the Translator

If you want to capture QuickDraw commands as they are executed, convert them, and either draw them immediately or save them, you need to install the QuickDraw–to–QuickDraw GX translator in the graphics port to which the QuickDraw drawing commands will be sent.

You install the translator with the `GXInstallQDTranslator` function. Once installed, the translator intercepts all QuickDraw drawing commands to that port, converts them to QuickDraw GX shapes, and sends them to a callback function (that you supply) for drawing or saving. When you are finished capturing QuickDraw commands, you remove the translator with the `GXRemoveQDTranslator` function.

Note

There is not necessarily a one-to-one match between a QuickDraw function call and the generation of a QuickDraw GX shape. u

Listing 1-5 is a sample that uses the functions `GXInstallQDTranslator` and `GXRemoveQDTranslator` to convert the bounded QuickDraw commands. The application-defined callback function, `aShapeProc`, sets the view port and draws each translated shape. The `aShapeProc` function is shown in Listing 1-6 on page 1-22.

Listing 1-5 Installing and removing the translator

```
/* first, install the translator */
GXInstallQDTranslator(window, gxDefaultOptionsTranslation,
                        &theRect, &theRect, theStyleStretch,
                        aShapeProc, (void *) &theWindViewPort);

/* now, make QuickDraw calls */
PenSize(20, 10);

MoveTo(100, 100);
LineTo(200, 100);

MoveTo(100, 150);
LineTo(200, 250);

/* when finished drawing, remove the translator */
GXRemoveQDTranslator(window, nil);
```

When using the `GXInstallQDTranslator` function, you must supply an application-defined function that gives you control over what is to be done with the QuickDraw GX shapes resulting from the translation. For example, you may want to draw each shape as it is translated, or you may want to spool multiple shapes and draw after you have completed the picture.

Listing 1-6 is the sample shape-spooling function used by the code in Listing 1-5. This function sets the view port, which is passed to it in the `reference` parameter, and then draws the shape passed to it in the parameter `theShape`.

Listing 1-6 Sample application-defined shape-spooling function

```
OSErr aShapeProc( gxShape theShape,
                                 void *reference)
{
   GXSetShapeViewPorts(theShape, 1, (gxViewPort *) &reference);
   GXDrawShape(theShape);
   GXDisposeShape(theShape);
   return(GXGetGraphicsError(nil));
}
```

The prototype for the shape-spooling function, and how to use it, are described in the section “Handling Translated QuickDraw Data” beginning on page 1-41.

QuickDraw GX and the Macintosh Environment Reference

This section contains constants, data types, and functions that are specific to the QuickDraw GX environment.

Constants and Data Types

This section describes the constants that you can use with the `Gestalt` function and the constants you can use to control translation with the QuickDraw–to–QuickDraw GX translator.

Gestalt Selectors and Attributes

The selector `'grfx'` can be used with the `Gestalt` function to determine whether the graphics and typography portions of QuickDraw GX have been installed. The `'pmgr'` selector can be used to determine whether QuickDraw GX printing is installed. `Gestalt` returns the version number in either case.

If you call `Gestalt` with the `gestaltGraphicsAttr` selector, it returns an attribute that specifies whether the debugging or nondebugging version of QuickDraw GX is installed, and what platform it is installed on. You can use the `'qdgx'` selector to determine if QuickDraw GX is installed.

```
#define gestaltGXVersion                'qdgx'
#define gestaltGraphicsVersion          'grfx'
#define gestaltPrintingMgrVersion       'pmgr'
#define gestaltCurrentGraphicsVersion   0x00010000

#define gestaltGraphicsAttr             'gfxa'
#define gestaltGraphicsisDebugging      0x00000001
#define gestaltGraphicsisLoaded         0x00000002
#define gestaltGraphicsIsPowerPC        0x00000004
```

These selectors and attributes are described in the section “Testing for the Presence and Version of QuickDraw GX” beginning on page 1-4. The `Gestalt` function is described in the chapter “Gestalt Manager” in *Inside Macintosh: Operating System Utilities*.

Translator Options and Statistics

The `gxTranslationOptions` enumeration defines constants that control various aspects of the translation from QuickDraw to QuickDraw GX:

```
   enum gxTranslationOptions {
      gxDefaultOptionsTranslation   = 0x0000,
      gxOptimizedTranslation        = 0x0001,
      gxReplaceLineWidthTranslation = 0x0002,
      gxSimpleScalingTranslation    = 0x0004,
      gxSimpleGeometryTranslation   = 0x0008,
      gxSimpleLinesTranslation      = 0x000C,
      gxLayoutTextTranslation       = 0x0010,
      gxRasterTargetTranslation     = 0x0020,
      gxPostScriptTargetTranslation = 0x0040,
      gxVectorTargetTranslation     = 0x0080
   };
   typedef long gxTranslationOption;
```

The individual constants for the enumeration are described in Table 1-2 on page 1-12.

The `gxTranslationStatistics` enumeration defines constants that are used as masks, any of which you can combine using an AND operation to interpret the statistics gathered during translation:

```
enum gxTranslationStatistics {
   gxContainsFormsBegin       = 0x0001,
   gxContainsFormsEnd         = 0x0002,
   gxContainsPostScript       = 0x0004,
   gxContainsEmptyPostScript  = 0x0008
};
typedef long gxTranslationStatistic;
```

The individual constants for the enumeration are described in Table 1-3 on page 1-20.

Macintosh Interface Functions

This section describes the QuickDraw GX functions you can use to

- associate view ports with Macintosh windows
- associate view devices with Macintosh graphics devices (`GDevice` records)
- convert QuickDraw coordinates and mouse locations to QuickDraw GX coordinates
- install and remove view port filters that intercept QuickDraw GX drawing commands

Associating View Ports With Macintosh Windows

This section describes the function you use to

- create a new view port object associated with a specific Macintosh window
- retrieve the Macintosh window associated with a view port, or the view port associated with a Macintosh window

GXNewWindowViewPort

You can use the `GXNewWindowViewPort` function to create a new view port for a specified Macintosh window.

```
gxViewPort GXNewWindowViewPort(WindowPtr qdWindow);
```

`qdWindow`	A pointer to the window for which the new view port is to be created.
function result	A reference to the new view port.

DESCRIPTION

The `GXNewWindowViewPort` function creates a new view port associated with the specified window. All drawing in the window view port will be clipped to the visible region of the window.

View ports associated with windows are clipped by the visible region (`visRgn`), but not the clip region (`clipRgn`) of the window. The origin of the window doesn't affect the view port. The clip shape of the view port doesn't affect drawing in the window.

SPECIAL CONSIDERATIONS

You cannot alter the mapping or clip properties of view ports created with this function. Most typically, you use this function to create a view port attached to a window, and then—if you support scrolling or otherwise need to change the clip or mapping—you create one or more child view ports of the window view port and draw into them.

Do not attach more than one view port to a window through this function; unpredictable behavior results.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`

SEE ALSO

View ports, child view ports, and the limitations on access to window view ports are discussed in the chapter "View-Related Objects" in *Inside Macintosh: QuickDraw GX Objects.*

To obtain the window associated with a view port, use the `GXGetViewPortWindow` function, described next. To obtain the view port associated with a window, use the `GXGetWindowViewPort` function, described on page 1-26.

GXGetViewPortWindow

You can use the `GXGetViewPortWindow` function to return the Macintosh window of a specified view port.

```
gxWindowPtr GXGetViewPortWindow(gxViewPort portOrder);
```

`portOrder` A reference to the specified view port.

function result A pointer to the window associated with the specified view port.

DESCRIPTION

The function returns `nil` if the view port is not associated with a window.

This function returns `nil` if you pass it a reference to a child view port of a window view port. To determine the window ultimately associated with a child view port, use the `GXGetViewPortParent` function to find the parent view port at the top of the view port hierarchy, and pass that view port reference to the `GXGetViewPortWindow` function.

ERRORS, WARNINGS, AND NOTICES

Errors
```
out_of_memory
invalid_viewport_reference
```

SEE ALSO

To create a view port associated with a window, use the `GXNewWindowViewPort` function, described in the previous section. To obtain the view port associated with a window, use the `GXGetWindowViewPort` function, described next.

GXGetWindowViewPort

You can use the `GXGetWindowViewPort` function to return the view port of a specified Macintosh window.

```
gxViewPort GXGetWindowViewPort(WindowPtr qdWindow);
```

`qdWindow`	A pointer to the specified window.
function result	A reference to the view port associated with the specified window.

DESCRIPTION

The function returns `nil` if the window has no associated view port.

ERRORS, WARNINGS, AND NOTICES

Errors
```
out_of_memory
```

SEE ALSO

To create a view port associated with a window, use the `GXNewWindowViewPort` function, described on page 1-24. To obtain the window associated with a view port, use the `GXGetViewPortWindow` function, described in the previous section.

Associating View Devices With Macintosh Graphics Devices

This section describes the functions you use to retrieve the Macintosh graphics device (`GDevice` record) associated with a QuickDraw GX view device object, or the view device associated with a Macintosh graphics device.

GXGetViewDeviceGDevice

You can use the `GXGetViewDeviceGDevice` function to return the Macintosh graphics device associated with a specified view device object.

```
GDHandle GXGetViewDeviceGDevice(gxViewDevice theDevice);
```

`theDevice`	A reference to the view device whose graphics device is requested.
function result	A handle to the `GDevice` record of the specified view device.

DESCRIPTION

The `GXGetViewDeviceGDevice` function returns a handle to the `GDevice` record associated with a specified view device. The function returns `nil` if the view device has no `GDevice` record.

ERRORS, WARNINGS, AND NOTICES

Errors
```
out_of_memory
invalid_viewdevice_reference
```

SEE ALSO

Macintosh graphics devices and the `GDevice` record are described in *Inside Macintosh: Imaging With QuickDraw*.

View devices are described in the chapter “View-Related Objects” in *Inside Macintosh: QuickD*raw GX Objects.

To obtain the view device associated with a graphics device, use the `GXGetGDeviceViewDevice` function, described next.

GXGetGDeviceViewDevice

You can use the `GXGetGDeviceViewDevice` function to return the view device object associated with a specified Macintosh graphics device.

```
gxViewDevice GXGetGDeviceViewDevice(GDHandle qdGDevice);
```

`qdGDevice`	A handle to the `GDevice` record of the graphics device whose view device is requested.
function result	A reference to the view device object associated with the specified graphics device.

DESCRIPTION

The `GXGetGDeviceViewDevice` function returns a reference to the view device object associated with a specified graphics device. The function returns `nil` if the graphics device has no view device.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`

SEE ALSO

Macintosh graphics devices and the `GDevice` record are described in *Inside Macintosh: Imaging With QuickDraw*.

View devices are described in the chapter "View-Related Objects" in *Inside Macintosh: QuickD*raw GX Objects.

To obtain the graphics device associated with a view device, use the `GXGetViewDeviceGDevice` function, described in the previous section.

Converting From QuickDraw to QuickDraw GX Coordinates

This section describes the functions you use to

- convert from QuickDraw coordinates to QuickDraw GX coordinates
- retrieve mouse locations in terms of QuickDraw GX coordinates

GXConvertQDPoint

You can use the `GXConvertQDPoint` function to convert from QuickDraw global coordinates to QuickDraw GX coordinates.

```
void GXConvertQDPoint(const Point *shortPt, gxViewPort portOrder,
                        gxPoint *fixedPt);
```

`shortPt`	A pointer to a point in QuickDraw global coordinates that is to be converted to QuickDraw GX coordinate space.
`portOrder`	A reference to a view port. If this parameter is `nil`, the conversion is to QuickDraw GX global coordinates; if it is other than `nil`, the conversion is to the local coordinates of that view port.
`fixedPt`	A pointer to a `gxPoint` structure. On return, the structure contains the result of the coordinate conversion.

DESCRIPTION

The `GXConvertQDPoint` function converts a point with QuickDraw global coordinates to a point with QuickDraw GX coordinates. If the `portOrder` parameter is `nil`, the QuickDraw global coordinates are converted to QuickDraw GX global coordinates. If the `portOrder` parameter is specified, the QuickDraw global coordinates are converted to QuickDraw GX local coordinates. The local coordinates are local within the specified view port.

The QuickDraw global coordinates are specified in pixels. The QuickDraw GX global and local coordinates are specified in a coordinate space in which 1.0 = 1/72 inch.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`invalid_viewPort_reference`

Warnings
`point_does_not_intersect_port`   (debugging version)

SEE ALSO

For more information about converting from QuickDraw to QuickDraw GX coordinate space, see the section "Converting From QuickDraw to QuickDraw GX Coordinates" beginning on page 1-7. For more information about the QuickDraw GX coordinate system and drawing, see the chapter "View-Related Objects" in *Inside Macintosh: QuickDraw GX Objects.*

GXGetGlobalMouse

You can use the `GXGetGlobalMouse` function to obtain the current cursor position in QuickDraw GX global coordinates.

```
void GXGetGlobalMouse(gxPoint *globalPt);
```

`globalPt`	A pointer to a `gxPoint` structure. On return, the structure contains the fixed-point global coordinates of the current cursor position.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`

GXGetViewPortMouse

You can use the `GXGetViewPortMouse` function to obtain the cursor position expressed in the QuickDraw GX local coordinates of the specified view port.

```
void GXGetViewPortMouse(gxViewPort portOrder, gxPoint *localPt);
```

`portOrder`	A reference to the view port for which the cursor position is requested.
`localPt`	A pointer to a `gxPoint` structure. On return, the structure contains the fixed-point local coordinates of the current cursor position.

SPECIAL CONSIDERATIONS

If the `portOrder` parameter is `nil`, this function posts an `invalid_viewPort_reference` error.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`invalid_viewPort_reference`

Warnings
`point_does_not_intersect_port`

Installing a View Port Filter

This section describes the functions you use to install and remove view port filters, which allow you to intercept drawing commands.

GXSetViewPortFilter

You can use the `GXSetViewPortFilter` function to intercept a shape being sent to a specified view port. Instead of drawing the shape, you can use an application-defined filter function to manipulate the shape.

```
void GXSetViewPortFilter(gxViewPort portOrder,
                         gxUserViewPortFilter filter,long refCon);
```

`portOrder`	A reference to the view port to be filtered.
`filter`	A pointer to an application-defined callback function that acts on the filtered shape. If you pass `nil` for this parameter, any installed filter function is removed.
`refCon`	A `long` value to be passed to the callback filter function. Your filter function can use the value in any manner.

DESCRIPTION

The `GXSetViewPortFilter` function allows you to install a filter function that intercepts shapes sent to a specified view port and manipulates them in any manner you wish. You must specify a valid view port reference in the `portOrder` parameter.

The filter parameter is a pointer to an application-defined callback function of type `gxUserViewPortFilter`:

```
typedef void (*gxUserViewPortFilterProcPtr)(gxShape toFilter,
                              gxViewPort portOrder, long refCon);
typedef gxUserViewPortFilterProcPtr gxUserViewPortFilter;
```

You must provide the callback filter function; its prototype is described in the section “Filtering Drawing Calls to a View Port” beginning on page 1-40.

When you call `GXSetViewPortFilter`, you can pass it any useful `long` value in the `refCon` parameter. That value will be passed to the filter function each time it is called.

To remove a filter function from a view port, call `GXSetViewPortFilter` with a `nil` value for the `filter` parameter.

SPECIAL CONSIDERATIONS

If you assign a filter function to a view port, it affects drawing to that specific view port only. Drawing to its child view ports or its parent view port (or any other view ports within its hierarchy) is unaffected.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
invalid_viewPort_reference
```

SEE ALSO

To obtain a pointer to the filter function currently installed in a given view port, use the `GXGetViewPortFilter` function, described next.

For a description of the application-defined view port filter function, see page 1-40.

GXGetViewPortFilter

You can use the `GXGetViewPortFilter` function to return the view device of a specified graphics device.

```
gxUserViewPortFilter GXGetViewPortFilter(gxViewPort portOrder,
                                         long *refCon);
```

`portOrder`	A reference to the view port whose currently installed view port filter you need.
`refCon`	A pointer to a `long` value. On return, the value is the reference constant that was passed to the `GXSetViewPortFilter` function when the filter function was installed.
function result	A pointer to the view port filter function that is installed in the specified view port.

DESCRIPTION

The `GXGetViewPortFilter` function returns a pointer to the view port filter that is currently installed in the specified view port. The function also returns, in the `refCon` parameter, the reference constant that was passed to `GXSetViewPortFilter` when the filter function was installed.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`invalid_viewPort_reference`

SEE ALSO

The `GXSetViewPortFilter` function is described in the previous section.

For a description of the prototype of the view port filter function, see page 1-40.

QuickDraw–to–QuickDraw GX Translator Functions

This section describes the functions you can use to

- Convert the specification for a font and face in the GrafPort into a gxStyle
- Convert QuickDraw picture data into QuickDraw GX shapes
- Convert QuickDraw commands to QuickDraw GX shapes

Converting a GrafPort Font and Face Specification

You use the function described in this section to convert the specification for a font and face in the GrafPort into a gxStyle.

GXConvertQDFont

You use the `GXConvertQDfont` function to translate the specification for a font and face in the GrafPort into a gxStyle.

```
long GXConvertQDfont(gxStyle theStyle, long txFont, long txFace);
```

`theStyle`	A gxStyle.
`txFont`	A long specifying a text font; same as in the GrafPort.
`txFace`	A long specifying a text style; same as in the GrafPort.

DESCRIPTION

The `GXConvertQDfont` function picks the `gxFont` that is the closest match for the `txFont` and `txFace` parameters. If it does not find an exact match, `GXConvertQDfont` might also set the style's font variation.

The `GXConvertQDfont` function also sets the style's encoding. It returns any style bits from `txFace` that were not accounted for in the `gxFont` and variation. This permits the caller to construct a `gxTextFace` based on the returned style bits. Potentially all of the style bits can be matched. Currently only bold, italic, condense, and extended bits are matched. However, in the future more might be matched. You should not make any assumptions about what will or will not be matched.

SPECIAL CONSIDERATIONS

If the translator calls `GXConvertQDFont`, it will already have mapped `txFont==0` to the correct font by calling the Script Manager.

Converting QuickDraw Pictures

You use the function described in this section to convert QuickDraw picture data to a QuickDraw GX picture shape.

GXConvertPICTToShape

You can use the `GXConvertPICTToShape` function to convert a QuickDraw picture to a QuickDraw GX shape.

```
gxShape GXConvertPICTToShape(const PicHandle pict,
                        gxTranslationOptions options,
                        const Rect *srcRect,
                        const Rect *dstRect,
                        Point styleStretch,
                        gxShape destination,
                        gxTranslationStatistic *statistics);
```

`pict`	A handle to the QuickDraw picture image to be converted to QuickDraw GX.
`options`	The translation options to use for the translation.
`srcRect`	A pointer to the source rectangle (normally the QuickDraw picture frame) of the QuickDraw image, in QuickDraw coordinates.
`dstRect`	A pointer to the destination rectangle of the QuickDraw image, in QuickDraw coordinates.
`styleStretch`	The scale factor (both horizontal and vertical) to apply to certain items, such as dashes, in QuickDraw picture comments.

`destination`
: A reference to the destination shape—the shape in which to store the translated picture.

`statistics`
: A pointer to the location in which to store the translation statistics.

function result A reference to the destination shape. This is the same shape referenced in the `destination` parameter, or a newly created picture shape if the `destination` parameter was `nil`.

DESCRIPTION

The `GXConvertPICTToShape` function provides a conversion of a QuickDraw GX picture image within the boundaries of the source rectangle `srcRect` to a QuickDraw GX shape within the boundaries of the destination rectangle `dstRect`. You pass QuickDraw picture to the translator and it returns a QuickDraw GX picture shape that approximates the original QuickDraw picture.

The `srcRect` and `dstRect` parameters represent the source and destination space of the image, expressed in QuickDraw coordinates. The relation between the source and destination rectangles specifies the amount of scaling to be applied during translation. You can use this scaling capability to ensure that the resulting QuickDraw GX shape is a good representation of the original QuickDraw object. The `srcRect` parameter is normally the QuickDraw picture frame of the original picture data. If scaling is not required, you can pass in the source rectangle of the original data for both parameters.

If the `destination` parameter is `nil`, a QuickDraw GX picture shape is created and returned by the function. If the destination parameter is not `nil`, the function returns the same shape reference that it was passed. The QuickDraw GX shapes resulting from the translation are contained in that picture shape.

The translator keeps various statistics about the QuickDraw picture data it is translating. You can examine the mask that describes these statistics after the translation if you are interested in this information. The flags in the translation statistics masks are defined in the `gxTranslationStatistics` enumeration. If you are not interested in this information, you should pass `nil` for the `stats` parameter.

SPECIAL CONSIDERATIONS

If you pass `nil` for the `destination` parameter and if no error occurs, this function creates a QuickDraw GX shape object. You are responsible for disposing of that object when you no longer need it.

Before using the translator, you must first call the `GXInitPrinting` function.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`

SEE ALSO

For an example of the use of this function, see Listing 1-4 on page 1-20.

To translate individual QuickDraw drawing commands as they are executed, use the `GXInstallQDTranslator` function, described next.

Translation options from the `gxTranslationOptions` enumeration are described in Table 1-2 on page 1-12.

Translation statistics flags from the `gxTranslationStatistics` enumeration are described in Table 1-3 on page 1-20.

For a general description of the QuickDraw–to–QuickDraw GX translator, see the section "Using the QuickDraw–to–QuickDraw GX Translator" beginning on page 1-10.

The `GXInitPrinting` function is described in the chapter "Core Printing Features" in *Inside Macintosh: QuickDraw GX Printing*.

Installing and Removing the Translator

This section describes the functions you use to install the QuickDraw–to–QuickDraw GX translator in order to intercept QuickDraw drawing commands as they are executed. To use the translator in this way, you also need to supply a callback shape-spooling function, described on page 1-41.

GXInstallQDTranslator

You can use the `GXInstallQDTranslator` function to initiate translation of the QuickDraw drawing commands to QuickDraw GX shapes. Subsequent QuickDraw drawing commands are translated into equivalent QuickDraw GX shapes.

```
void GXInstallQDTranslator(GrafPtr port,
                           gxTranslationOptions options,
                           const Rect *srcRect,
                           const Rect *dstRect,
                           Point styleStretch,
                           gxShapeSpoolFunction userFunction,
                           void *reference);
```

`port`	A pointer to the QuickDraw graphics port into which to install the translator.
`options`	The translation options to use for the translation.

`srcRect`	A pointer to a rectangle defining the QuickDraw source dimensions for drawing, in QuickDraw coordinates.
`dstRect`	A pointer to a rectangle defining the QuickDraw destination dimensions for drawing, in QuickDraw coordinates.
`styleStretch`	The amount of scaling that certain picture-comment items, such as dashes, will be given in the x and y directions.
`userFunction`	A pointer to an application-defined callback function to which the translator passes each translated shape.
`reference`	A pointer to a reference constant that can be used for any purpose. QuickDraw GX passes the reference constant to the application-defined callback function each time it calls the function.

DESCRIPTION

You can use the `GXInstallQDTranslator` function to install the QuickDraw–to–QuickDraw GX translator into a QuickDraw graphics port. QuickDraw commands that draw into that port are translated to equivalent QuickDraw GX shapes.

All QuickDraw drawing commands executed after you call this function and before you call `GXRemoveQDTranslator` are converted into QuickDraw GX shapes and passed to an application-defined function. There is not necessarily a one-to-one match between a QuickDraw function call and the generation of a QuickDraw GX shape; the translation algorithm can combine several QuickDraw items into one QuickDraw GX shape.

The `srcRect` and `dstRect` parameters represent the source and destination space of the image, expressed in QuickDraw coordinates. The relation between the source and destination rectangles specifies the amount of scaling to be applied during translation. If scaling is not required, you can pass identical rectangles for both parameters.

The `styleStretch` parameter represents the amount of scaling that QuickDraw bitmap patterns are given by the translator in the x and y directions in order to scale them up to the destination space. The x scale factor is stored in `styleStretch.h` and the y scale factor is stored in `styleStretch.v`. These values are usually the destination resolution divided by the screen resolution (72 dpi), rounded to the nearest integer.

The `userFunction` parameter is a pointer to an application-defined callback function of type `gxShapeSpoolFunction`:

```
typedef OSErr (*gxShapeSpoolProcPtr)(gxShape toSpool,
                                     void *refCon);
typedef gxShapeSpoolProcPtr gxShapeSpoolFunction;
```

The translator calls the function every time that it generates a translated QuickDraw GX shape. You must provide the function; its prototype is described in the section “Handling Translated QuickDraw Data” beginning on page 1-41.

The `reference` parameter of the `GXInstallQDTranslator` function can be used by your application in any manner you desire, within the constraints of a `long` data field.

The translator passes the parameter to the application-defined callback function. For example, you can use `reference` to specify where the translated picture is to be displayed, by passing a view port reference in the `reference` parameter.

SPECIAL CONSIDERATIONS

If you call the `GXInstallQDTranslator` function to install the translator, you must subsequently call the `GXRemoveQDTranslator` function to remove it when you are finished.

The `port` parameter to this function must be an old-style graphics-port pointer (`GrafPtr`); however, the translator depends on the existence of a color graphics port. Therefore, you should always create a color graphics port (using `NewCWindow`, `NewGWorld` or `OpenCPort`), and coerce the `CGrafPtr` to a `GrafPtr` when you call this function.

Before installing the translator, you must first call the `GXInitPrinting` function.

Installation of the translator cannot cause an `out_of_memory` error, but the process of translation can result in an out-of-memory condition. Your application should be prepared to handle that condition.

ERRORS, WARNINGS, AND NOTICES

Notices (debugging version)
`translator_already_installed_on_this_grafport`

SEE ALSO

For an example of the use of this function, see Listing 1-5 on page 1-21.

Translation options from the `gxTranslationOptions` enumeration are described in Table 1-2 on page 1-12.

The application-defined shape-spooling function called by the translator is described on page 1-41.

To translate an entire QuickDraw picture, use the `GXConvertPICTToShape` function, described in the previous section.

For a general description of the QuickDraw–to–QuickDraw GX translator, see the section “Using the QuickDraw–to–QuickDraw GX Translator” beginning on page 1-10.

The `GXInitPrinting` function is described in the chapter “Core Printing Features” in *Inside Macintosh: QuickDraw GX Printing.*

For descriptions of color graphics ports and the functions you use to create them, see *Inside Macintosh: Imaging With QuickDraw.*

GXRemoveQDTranslator

You can use the `GXRemoveQDTranslator` function to terminate the translation of QuickDraw drawing commands.

```
translationStatistics GXRemoveQDTranslator(GrafPtr port,
                                gxTranslationStatistic *statistic);
```

`port`	A pointer to the QuickDraw graphics port in which the translator is currently installed.
`statistic`	A pointer to the location in which to return the translation statistics.
function result	The translation statistics.

DESCRIPTION

The `GXRemoveQDTranslator` function removes the translator from the QuickDraw graphics port in which it was installed, and flushes the internal translation buffer.

The translator keeps various statistics about the QuickDraw picture data it is translating. After removing the translator, you can examine the mask that describes these statistics if you are interested in this information. The flags in the translation statistics masks are defined in the `gxTranslationStatistics` enumeration. If you are not interested in this information, you should pass `nil` for the statistic parameter.

SPECIAL CONSIDERATIONS

Always be sure to call the `GXRemoveQDTranslator` function when you are finished translating.

EERRORS, WARNINGS, AND NOTICES

Warnings

`translator_not_installed_on_this_grafport`	(debugging version)

SEE ALSO

For an example of the use of this function, see Listing 1-5 on page 1-21.

Translation statistics flags from the `gxTranslationStatistics` enumeration are described in Table 1-3 on page 1-20.

To translate an entire QuickDraw picture, use the `GXConvertPICTToShape` function, described on page 1-34.

For a general description of the QuickDraw–to–QuickDraw GX translator, see the section “Using the QuickDraw–to–QuickDraw GX Translator” beginning on page 1-10.

Application-Defined Functions

This section describes callback functions that your application must provide for QuickDraw GX to call, in two situations:

- If you intercept shape-drawing calls to a view port
- If you install the QuickDraw–to–QuickDraw GX translator in a graphics port

Filtering Drawing Calls to a View Port

The callback function described in this section is a view port filter, which handles shape-drawing calls that have been intercepted in a given view port.

MyViewPortFilter

You can create a filter function that handles intercepted QuickDraw GX drawing calls. The filter function must have a prototype of this form:

```
void MyViewPortFilter(gxShape toFilter, gxViewPort portOrder,
                      long refCon);
```

`toFilter`	A reference to the shape to be filtered—that is, the shape that would have been drawn to the view port specified in the `portOrder` parameter.
`portOrder`	A reference to the view port in which this filter function has been installed.
`refCon`	A reference constant that your filter function can use in any manner.

DESCRIPTION

Once this filter function is installed, QuickDraw GX calls it any time a function is executed that draws a shape in the view port referenced by the `portOrder` parameter. Instead of drawing the shape, QuickDraw GX passes the shape to this function. Your filter function can perform actions other than drawing (such as spooling), or it can otherwise modify or process the shape.

Your application installs this filter function by providing a pointer to it when calling the `GXSetViewPortFilter` function. When your application calls the function `GXSetViewPortFilter`, it also provides the `refCon` value that will be passed to this filter function.

The value passed to you in the `refCon` parameter can be used for any purpose; for example, it might contain a reference to a different view port for drawing to, a pointer to a buffer for collecting spooled data, or anything else useful to the filter function. The `portOrder` parameter allows you to identify the view port being intercepted, in case the filter function is installed on more than one view port.

Your application can get a pointer to this installed filter function at any time by calling the `GXGetViewPortFilter` function. After you are finished intercepting drawing calls, your application can remove the filter function by calling the `GXSetViewPortFilter` function with a `nil` filter-function pointer.

SEE ALSO

The `GXSetViewPortFilter` function is described on page 1-31. The to `GXGetViewPortFilter` function is described on page 1-32.

Handling Translated QuickDraw Data

The callback function described in this section is a shape-spooling function, which handles QuickDraw GX shapes that have been translated from QuickDraw drawing commands.

MyShapeSpooler

You can create a shape-spooling function that handles QuickDraw drawing commands that have been translated into QuickDraw GX shapes. The shape-spooling function must have a prototype of this form:

```
OSErr MyShapeSpooler(gxShape toSpool, void *refCon);
```

`toSpool`	A reference to the shape just translated by the translator.
`refCon`	A pointer to a reference constant, passed to your spool function by the translator, that you can use for any purpose.

function result An result code of type `OSErr`. You should pass 0 if no error occurs.

DESCRIPTION

When you install the QuickDraw–to–QuickDraw GX translator with the `GXInstallQDTranslator` function, the translator intercepts all subsequent QuickDraw drawing commands, converts them to QuickDraw GX shapes, and passes those shapes to this shape-spooling function. Your shape-spooling function can draw each shape, add it to a picture, or perform any other action you wish.

You install this function by providing a pointer to it when you call the `GXInstallQDTranslator` function. When you call `GXInstallQDTranslator`, you also provide the `refCon` value that will be passed to this filter function. The `refCon` value can be used for any purpose; for example, it might contain a view port reference for drawing, a pointer to a buffer for collecting spooled data, or anything else useful to the shape-spooling function.

After your application has finished intercepting and converting QuickDraw calls and passing them to this shape-spooling function, your application must remove the translator by calling the `GXRemoveQDTranslator` function.

If your shape-spooling function encounters an error during processing, it should return a nonzero value (usually the error code). If the shape-spooling function returns a nonzero value, the translator ceases translating QuickDraw commands. If an error occurs, your application must still call the `GXRemoveQDTranslator` function to remove the translator.

SEE ALSO

The `GXInstallQDTranslator` function is described on page 1-36. The `GXRemoveQDTranslator` function is described on page 1-39.

For a general description of the QuickDraw–to–QuickDraw GX translator, see the section "Using the QuickDraw–to–QuickDraw GX Translator" beginning on page 1-10.

Summary of QuickDraw GX and the Macintosh Environment

Constants and Data Types

Gestalt Selectors and Attributes

```
#define gestaltGXVersion            'qdgx'   /* Overall GX vers. selector */
#define gestaltGraphicsVersion      'grfx'   /* GX graphics vers. selector */
#define gestaltPrintingMgrVersion   'pmgr'   /* GX printing vers. selector */
#define gestaltCurrentGraphicsVersion  0x00010000          /* version 1.0 */

#define gestaltGraphicsAttr         'gfxa'       /* GX attributes selector */
#define gestaltGraphicsisDebugging  0x00000001
#define gestaltGraphicsisLoaded     0x00000002
#define gestaltGraphicsIsPowerPC    0x00000004
```

Translator Options and Statistics

```
   enum gxTranslationOptions {
      gxDefaultOptionsTranslation   = 0x0000,
      gxOptimizedTranslation        = 0x0001,
      gxReplaceLineWidthTranslation = 0x0002,
      gxSimpleScalingTranslation    = 0x0004,
      gxSimpleGeometryTranslation   = 0x0008,
      gxSimpleLinesTranslation      = 0x000C,
      gxLayoutTextTranslation       = 0x0010,
      gxRasterTargetTranslation     = 0x0020,
      gxPostScriptTargetTranslation = 0x0040,
      gxVectorTargetTranslation     = 0x0080
   };
   typedef long gxTranslationOption;

   enum gxTranslationStatistics {
      gxContainsFormsBegin       = 0x0001,
      gxContainsFormsEnd         = 0x0002,
      gxContainsPostScript       = 0x0004,
      gxContainsEmptyPostScript  = 0x0008
   };
   typedef long gxTranslationStatistic;
```

Macintosh Interface Functions

Associating View Ports With Macintosh Windows

```
gxViewPort GXNewWindowViewPort
                             (WindowPtr qdWindow);
gxWindowPtr GXGetViewPortWindow
                             (gxViewPort portOrder);
gxViewPort GXGetWindowViewPort
                             (WindowPtr qdWindow);
```

Associating View Devices With Macintosh Graphics Devices

```
GDHandle GXGetViewDeviceGDevice
                             (gxViewDevice theDevice);
gxViewDevice GXGetGDeviceViewDevice
                             (GDHandle qdGDevice);
```

Converting From QuickDraw to QuickDraw GX Coordinates

```
void GXConvertQDPoint        (const Point *shortPt, gxViewPort portOrder,
                              gxPoint *fixedPt);
void GXGetGlobalMouse        (gxPoint *globalPt);
void GXGetViewPortMouse      (gxViewPort portOrder, gxPoint *localPt);
```

Installing a View Port Filter

```
void GXSetViewPortFilter     (gxViewPort portOrder,
                              gxUserViewPortFilter filter, long refCon)
gxUserViewPortFilter GXGetViewPortFilter
                             (gxViewPort portOrder, long *refCon)
```

QuickDraw–to–QuickDraw GX Translator Functions

Converting QuickDraw Font and Style

```
long GXConvertQDFont         (gxStyle theStyle, long txFont, long txFace);
```

Converting QuickDraw Pictures

```
gxShape GXConvertPICTToShape
                            (const PicHandle pict,
                             gxTranslationOptions options,
                             const Rect *srcRect,
                             const Rect *dstRect,
                             Point styleStretch,
                             gxShape destination,
                             gxTranslationStatistics *stats);
```

Installing and Removing the Translator

```
void GXInstallQDTranslator  (GrafPtr port, gxTranslationOptions options,
                             const Rect *srcRect, const Rect *dstRect,
                             Point styleStretch,
                             gxShapeSpoolFunction userFunction,
                             long refCon);
translationStatistics *GXRemoveQDTranslator
                            (GrafPtr port,
                             translationStatistic *statistics);
```

Application-Defined Functions

Filtering Drawing Calls to a View Port

```
void MyViewPortFilter       (gxShape toFilter, gxViewPort portOrder,
                             long refCon);
```

Handling Translated QuickDraw Data

```
OSErr MyShapeSpooler        (gxShape toSpool, void *refCon);
```

C H A P T E R 2

QuickDraw GX Memory Management

Contents

This chapter describes the aspects of QuickDraw GX memory management that your application can control. QuickDraw GX manages the memory blocks used by your application automatically. Read this chapter if you want to understand how QuickDraw GX memory works or to supplement QuickDraw GX memory management operations.

Before reading this chapter, you should be familiar with QuickDraw GX objects. For more information on objects, see *Inside Macintosh: QuickDraw GX Objects*.

For more information regarding Macintosh memory, see *Inside Macintosh: Memory*.

This chapter starts by providing an overview of the QuickDraw GX memory management system. It then tells how to:

- create and dispose of a graphics client and its heap
- determine memory requirements for a graphics client heap

Additional memory management topics of concern to few developers are also addressed, such as how to:

- respond to low-memory conditions
- load and unload objects
- work with multiple graphics clients

This chapter also contains reference information for the constants, data types, and functions associated with QuickDraw GX memory management.

About QuickDraw GX Memory Management

QuickDraw GX works with the Macintosh Memory Manager to manage the memory used by your application for creating and manipulating objects. QuickDraw GX memory management is automatic. Memory blocks are allocated and deallocated as your application needs them.

Nevertheless, the more memory that QuickDraw GX has available, the faster your application can run. As a result, you may be able to improve the performance of your application by using some of the QuickDraw GX memory management operations.

Memory Heaps

QuickDraw GX applications use an **application heap** and one or more graphics client heaps. The application heap memory holds your code and data structures. This is the part of memory where you allocate variables and your application executes. You can access any data structure in the application heap. Your application manages its own structures in the application heap and makes function calls to obtain or modify the contents of the graphics client heap.

The QuickDraw GX **graphics client heap** memory holds the QuickDraw GX objects you create. The graphics client heap consists of one or more blocks of **discontiguous memory** that QuickDraw GX uses to allocate its objects, structures, and variables. QuickDraw GX memory is private so, in general, you cannot directly access the contents of a graphics client heap.

The graphics client heap and the application heap work independently. For example, QuickDraw GX can execute from the memory on an accelerator card. As a result, QuickDraw GX never moves application memory. In addition, Macintosh Memory Manager functions cannot modify QuickDraw GX objects. QuickDraw GX has its own internal memory manager and memory management functions for interacting with its objects.

Graphics Clients and Graphics Client Heaps

At system startup, QuickDraw GX does not have any memory available in which to do work. To allocate memory, it needs a special QuickDraw GX object called a **graphics client.** This object is associated with a graphics client heap.

Because a graphics client stores bookkeeping data for its heap, including the memory starting address and the size of all of the heap's memory blocks, a graphics client can be associated with only one QuickDraw GX graphics client heap. A graphics client also stores the error, warning, and notice state.

An application heap is allocated by the Macintosh Memory Manager when an application is launched. This is the memory region used by your application for its own code and data structures. Graphics clients and their heaps are never allocated from memory blocks that have been allocated to the application heap.

QuickDraw GX provides functions to create a new graphics client and its graphics client heap. If you don't use these functions in your application, QuickDraw GX will call them for you to create a graphics client having a default memory size and location.

When you no longer need a graphics client and its heap, you dispose of them to release memory blocks. QuickDraw GX provides functions so that your application can dispose of a graphics client and its graphics client heap, but if you don't use these functions in your application, QuickDraw GX calls them for you at the appropriate time. Whenever an application requires **memory allocation,** QuickDraw GX automatically deallocates as many graphics client heap memory blocks as it needs to in order to satisfy the application's memory requirements. QuickDraw GX never deallocates graphics client heaps while they are in use. If QuickDraw GX cannot find sufficient memory blocks, it may automatically unload objects from memory to storage disk. QuickDraw GX automatically reloads these objects from storage disk to memory as it needs them.

Additional Topics

The latter part of this chapter discusses some issues that are relevant in only a few very specific and uncommon situations: handling low-memory problems, manual loading and unloading of objects, using functions that do not require a graphics client or its heap, specifying a memory starting location for a graphics client and its heap, and working with multiple graphics clients.

Under normal circumstances, QuickDraw GX resolves low-memory conditions and handles the loading of objects as needed, performing these tasks automatically and in a way that is transparent to your application. However, you can affect some of its processing in these areas. You can also set an attribute that prevents QuickDraw GX from allocating additional memory blocks to your graphics client heap and you can manually **load** and **unload** objects. These topics are described in the section “Additional Memory Management Topics” beginning on page 2-10.

Using Graphics Clients and Graphics Client Heaps

QuickDraw GX provides most memory management services automatically. This section describes how your application can override this automatic control to

- create a graphics client and its heap
- determine memory requirements for a graphics client heap
- dispose of a graphics client and its heap

Creating a Graphics Client and Its Graphics Client Heap

Either QuickDraw GX or you can create a new graphics client and its heap. This section discusses how you can control these tasks.

Implicit Creation

If your application does not explicitly create a graphics client or a graphics client heap, QuickDraw GX creates them for you when needed. QuickDraw GX calls the `GXNewGraphicsClient` and the `GXEnterGraphics` functions when the first function call is made in your application that requires a graphics client heap. Almost all QuickDraw GX functions require a graphics client heap. The few exceptions are listed in the section “Functions That Do Not Require a Graphics Client or Heap” beginning on page 2-14.

When QuickDraw GX calls the `GXNewGraphicsClient` function, it selects the starting memory location for the heap and creates a graphics client to provide the bookkeeping for the heap. When QuickDraw GX calls the `GXEnterGraphics` function, it uses the memory location and heap size stored in the graphics client to create the new heap.

QuickDraw GX looks for a resource of type `'gasz'` with an ID of 0 and uses the first long word of that resource as the number of bytes to be allocated to the graphics client heap. If your application does not provide this resource, QuickDraw GX version 1.0 uses a **default memory size** value of 600 KB. For additional information, see the description of the `GXNewGraphicsClient` routine beginning on page 2-19.

A `'gasz'` resource can only provide one graphics client heap size in a single application. QuickDraw GX uses this size for every graphics client with a `memoryLength` parameter of zero. Listing 2-1 shows how to create a type '`gasz`' resource for a 512 KB graphics client heap.

Listing 2-1 Creating a `'gasz'` resource

```
resource 'gasz' (0) {
    0x00080000        /* 512KB graphics client heap */
};
```

The `GXNewGraphicsClient` function is described on page 2-19. The `GXEnterGraphics` function is described on page 2-22.

Explicit Creation

If you want to specify the characteristics of the graphics client heap, you can use the `GXNewGraphicsClient` function explicitly to create a graphics client.

The `GXNewGraphicsClient` function has parameters that specify the heap's starting memory location, **memory size** in bytes, and whether or not QuickDraw GX is permitted to later increase the heap's size by allocating additional memory blocks. The graphics client stores the data passed by the `GXNewGraphicsClient` function, but does not allocate memory to the heap. This requires the `GXEnterGraphics` function call.

Most applications should allow QuickDraw GX to select the memory starting location by passing `nil` for the `memoryStart` parameter. If you need to specify the memory starting location, see the section "Specifying the Starting Location of a Graphics Client" beginning on page 2-14.

If you pass zero for the `memoryLength` parameter, QuickDraw GX looks for a resource of type `'gasz'` with an ID of 0 and uses the first long word of that resource as the heap size (the number of bytes to allocate). If your application does not provide this resource, QuickDraw GX version 1.0 uses a default size of 600 KB. Alternatively, you can specify the requested heap size in bytes. To determine how many bytes to specify for your graphics client heap, see the next section. The '`gasz`' resource is described in the previous section.

If you pass zero for the attribute parameter, QuickDraw GX can later add additional memory blocks to the heap when more memory is required. If the `attribute` parameter has value 1, indicating the `gxStaticHeapConstant` constant, QuickDraw GX cannot add more memory blocks to the graphics client heap allocated.

Once a graphics client is created, you use the `GXEnterGraphics` function to allocate memory for its heap. If you don't explicitly make the call, QuickDraw GX implicitly calls the `GXEnterGraphics` function for you when it executes the next function that requires a graphics client heap. Almost all QuickDraw GX functions require a graphics client heap. The exceptions are given in the section "Functions That Do Not Require a Graphics Client or Heap" beginning on page 2-14.

Listing 2-2 shows how to explicitly create a graphics client and allocate 10 KB of memory for its heap. Since the `attribute` parameter is 0, QuickDraw GX performs its default behavior to add **memory blocks** to the graphics client heap created, as required. For example, additional memory may be required as your application creates new objects. You should allocate your graphics client at the beginning of your application and poll for errors to ensure that the graphics client is allocated.

Listing 2-2 Explicitly creating a graphics client and its heap

```
gxGraphicsClient  newClient;
long              graphicsHeapSizeRequested = 50K; // 50K GX heap

newClient = GXNewGraphicsClient(nil, graphicsHeapSizeRequested
                                 * 1024,OL );

   // After we attempted to create the graphics client, we need to
   // determine if the call succeeded. If the call did not (as in
   // the case for all GX functions), “newClient” will be nil. If
   // it is, we alert the user to the problem, Otherwise, we will
   // attempt to allocate the GX heap.

if ( newClient ) {
   GXEnterGraphiccs();
   // Calling GXEnterGraphics allocates the memory within the GX
   // heap. The call would fail only if there is not enough
   // memory. In this case, the graphics error posted is -27999
   // (out of memory). At this point, we have not installed an
   // error handler, so we check for the error number
   // corresponding to the out-of-memory error.
```

```
if ( GXGetGraphicsError( nil ) == -27999 ) {

   // Because we cannot allocate the requested size for our GX
   // heap, we need to throw away the client we created and alert
   // the user that there is not enough memory to continue.

   GXDisposeGraphicsClient( newClient);

   >> application code to alert user and shut down app


} else {
   // Application error code to tell the user there is not enough
   // memory to create the graphics client. No error is
   // posted from GX because a graphics client does not
   // exist. The only reason you would not be able to create
   // a graphics client is if there is not enough memory.

   >> application code to alert user and shut down app
```

Determining Memory Requirements for a Graphics Client Heap

Using the optimal heap size increases the performance of your application. If your application does not allocate sufficient memory, QuickDraw GX will need to add additional memory blocks to the initial graphics client heap. If your heap is sized too large, you are wasting memory space.

You can use the QuickDraw GX GraphicsBug utility to check the actual size of your graphics client heap to ensure that your application has allocated sufficient, but not excessive, memory. Once you determine the optimal graphics client heap size for your application, you can specify this size at the beginning of your application by using the `GXNewGraphicsClient` function.

You can use the following procedure to determine the memory requirements of your graphics client heap:

1. Start your application with the `GXNewGraphicsClient` function and specify a memory size, such as 1 MB.
2. Run your application and create a document. QuickDraw GX allocates or deallocates memory blocks to a size that it deems necessary and sufficient to accommodate the number and complexity of the objects you have created.

3. Use the Heap Total (HT) GraphicsBug command to determine the memory size that QuickDraw GX is currently using. This is the size of the graphics client heap.
4. Use the `GXNewGraphicsClient` function to specify the size of the QuickDraw GX graphics client heap to accommodate the actual memory required.

Repeat these steps varying the size of the document used in step 2.

By running your application with what you would consider to be a document of average size and then with a document of large size, you can arrive at an optimum graphics client heap size that is probably somewhere between these two heap sizes. One important consideration is to ensure that your largest objects have sufficient memory allocated for the graphics client heap that they reside in. This is because an object cannot be split into multiple memory blocks in the heap.

Because QuickDraw GX can grow the heap to accommodate the needs of your application, you don't need to allocate sufficient space for your largest document. Assuming you have not passed the `gxStaticHeapClient` attribute to `GXNewGraphicsClient`. This procedure provides only a preliminary evaluation of the memory requirements for your application.

For additional information on how to use the GraphicsBug utility, see the section "Debugging With GraphicsBug" in the chapter "QuickDraw GX Debugging," in this book.

Disposing of a Graphics Client and Graphics Client Heap

When your application no longer needs a graphics client and its heap, you should **dispose** of them to free memory blocks. You can use the `GXExitGraphics` and `GXDisposeGraphicsClient` functions to do this.

While you are writing and debugging your application, it is a good idea to be meticulous about disposing of all graphics clients and graphics client heaps at the end of your application. As a result, you should use the `GXExitGraphics` function to dispose of the currently active graphics client heap and the `GXDisposeGraphicsClient` function to dispose of each active graphics client. If your application does not make these calls, QuickDraw GX automatically disposes of all graphics clients and heaps that belong to the exiting application. However, in this case, the graphics clients and heaps are considered aborted instead of being disposed of normally, and therefore QuickDraw GX does not report any errors that occur during the process of disposing of these graphics clients and heaps. Listing 2-3 shows how to properly dispose of a graphics client and its heap.

Listing 2-3 Disposing of graphics clients and graphics client heaps

```
.
. /* QuickDraw GX application code */
.
GXExitGraphics(void);
GXDisposeGraphicsClient(client);
}
```

Once your application is ready to ship, be sure to remove the terminating `GXExitGraphics` and `GXDisposeGraphicsClient` function calls and rely on QuickDraw GX to automatically dispose of all of your graphics clients and their heaps for your exiting application. When your application quits, it is much faster for QuickDraw GX to throw away all of the graphics clients and their graphics client heaps, rather than to dispose of each of them sequentially. This approach is analogous to quitting an application rather than taking the extra time to close multiple application windows.

The `GXExitGraphics` function is described on page 2-23. The `GXDisposeGraphics` function is described on page 2-21.

Additional Memory Management Topics

This section describes some additional memory management topics. Your application can supplement QuickDraw GX automatic memory management operations to

- respond to low-memory conditions
- load and unload objects
- work with graphics clients and graphics client heaps

Low-Memory Conditions

QuickDraw GX may post memory-related errors, warnings, and notices while trying to allocate additional memory. These notifications indicate the status of QuickDraw GX memory management operations and, in some cases, provide the opportunity for your application to respond accordingly.

Freeing Up Already Allocated Memory

When QuickDraw GX needs one or more additional memory blocks for a graphics client heap, it responds to the situation by performing one or more of the following sequential steps. If insufficient memory is freed in one step, QuickDraw GX proceeds to the next step in the sequence. When sufficient memory blocks are freed, QuickDraw GX allocates the memory blocks and processing continues. QuickDraw GX memory management steps 1 through 4 affect memory blocks that have already been allocated.

1. QuickDraw GX disposes of dead caches: A **QuickDraw GX cache** is temporary memory used by QuickDraw GX. A cache that contains out of date, and therefore invalid, information is called a **dead cache.** If it disposes of dead caches, QuickDraw GX posts a `disposed_dead_caches` notice in the debugging init when the operation is complete. This notice is posted once per graphics client. This notice is a one-time-only alert indicating that your graphics client heap is low on memory.
2. QuickDraw GX unloads objects in pictures that have the `gxDiskShape` shape attribute: All of the objects within the picture are unloaded before any other objects are unloaded. The picture object is not unloaded. The `gxDiskShape` shape attribute is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.
3. QuickDraw GX disposes of live caches: A QuickDraw GX cache that contains current valid drawing information is called a **live cache.** After live caches are disposed of, they need to be rebuilt before the next time you draw the object. QuickDraw GX posts a `disposed_live_caches` notice in the debugging init when the operation is complete. This notice is only posted once per graphics client. This notice is a one-time-only alert indicating that your graphics client heap is low on memory.
4. QuickDraw GX relocates bit images: Bit images are moved in memory in order to free memory space adjacent to them. No memory error, warning, or notice is posted to notify you of this step.

Allocating New Memory and Unloading Objects

If QuickDraw GX has not released sufficient memory after step 4, it attempts to add additional memory blocks to the graphics client heap. If sufficient memory is not available after step 5, QuickDraw GX begins to unload objects to disk storage.

5. QuickDraw GX adds additional memory blocks: QuickDraw GX adds additional memory blocks to the graphics client heap. Prior to adding memory blocks, QuickDraw GX posts an `about_to_grow_heap` warning. If the `gxStaticHeapClient` attribute is set for the graphics client heap, QuickDraw GX does not perform this step.
6. QuickDraw GX unloads objects: Prior to unloading objects, QuickDraw GX posts an `about_to_unload_objects` warning. First, shapes with the `gxDiskShape` shape attribute are unloaded. Then, objects without either the `gxDiskShape` or the `gxMemoryShape` attributes are unloaded. Finally, shapes with the `gxMemoryShape` attribute are unloaded. Unlike disposing of caches, unloading occurs without information loss, but it does take time and disk space. For additional information about object loading and unloading, see the section “Loading and Unloading Objects” beginning on page 2-12. If an object cannot be unloaded, QuickDraw GX posts a `could_not_create_backing_store` error or the appropriate system error.

When your application has received the `about_to_grow_heap` warning or the `could_not_create_backing_store` error, you can decide to free up some memory before GX does. However, you must be very careful if you decide to dispose of a GX object. You cannot dispose of anything that is currently in use. The only way to determine if something is in use would be by carefully tracking the GX objects used within your application. Most likely, you would only want to dispose of off-screen worlds used by your application and let GX free up memory by releasing other blocks. GX knows what is and is not busy.

If steps 1 through 6 fail to release sufficient memory to accommodate the allocation of the required additional blocks of memory, QuickDraw GX posts an `out_of_memory` error.

Functions That Create Additional Memory Demands

Individual QuickDraw GX functions have different memory-allocation consequences:

- Many QuickDraw GX functions explicitly allocate memory. For example, the `GXNewShape`, `GXCopyToShape`, and `GetShapeParts` functions allocate memory. An `out_of_memory` error may be posted when a memory allocation fails.
- Most QuickDraw GX functions can implicitly allocate memory to load a required object. For example, the `GXGetShapeAttributes` function may need to load a shape into memory to retrieve its attributes. QuickDraw GX loads objects automatically and does not post an error, warning, or notice. The exception is when QuickDraw GX posts an `out_of_memory` error when a memory allocation fails or a disk error occurs.
- Some functions do not allocate memory explicitly or implicitly. These functions might require a graphics client heap and do not post an `out_of_memory` error. These include math routines, error routines, and the `GXClone`*Object*, `GXDispose`*Object*, `GXUnload`*Object*, `GXValidate`*Object* function sets and all of the functions listed in the second part of Table 2-1 on page 2-14.

The GraphicsBug utility is useful in debugging memory problems. This utility is described in the chapter "QuickDraw GX Debugging" in this book.

QuickDraw GX errors, warnings, and notices are described in the chapter "Errors, Warnings, and Notices" in this book.

Loading and Unloading Objects

If your application needs more memory during execution, QuickDraw GX automatically unloads objects to disk storage to free memory. QuickDraw GX automatically reloads previously unloaded objects when it needs them.

QuickDraw GX only begins to unload objects after it has failed to free sufficient memory by disposing of dead caches, unloading picture shape objects, disposing of live caches, relocating bit images, and adding additional memory blocks to the graphics client heap. Unless you choose to control loading and unloading of objects to memory, QuickDraw GX performs these tasks for you automatically. Your application never needs to load or unload an object.

The order in which QuickDraw GX automatically loads and unloads objects depends upon the objects' shape attributes. QuickDraw GX first unloads shape objects with the `gxDiskShape` attribute. QuickDraw GX then unloads shapes without special attributes, style, ink, transform, color set, color profile, and tag objects. Finally, after all other objects are unloaded, QuickDraw GX unloads objects with the `gxMemoryShape` attribute.

You can use the `GXSetShapeAttribute` function to set or clear an object's shape attribute and the `GXGetShapeAttribute` function to determine which attributes of a shape object are set. Shape attributes are described in the chapter "Shape Objects" in *Inside Macintosh: QuickDraw GX Objects*.

Objects are unloaded to a temporary file created on the startup disk in the invisible temporary items folder. When an object is unloaded, a 4-byte stub remains in memory to describe the location of each object on disk so that it can be reloaded when required. Sufficient disk storage space must be available to accommodate all of the objects that are unloaded or a file system error is posted.

You can supplement QuickDraw GX automatic loading and unloading operations by using function calls. These functions may be useful in increasing application performance. For example, a multimedia application with time-critical processing may need to control specific objects to ensure that they are resident in memory when they are to be displayed and removed from memory when they are no longer required.

You can use the `GXUnloadShape` function to move a shape from memory to disk storage and the `GXLoadShape` function to move a shape from disk storage to memory. QuickDraw GX provides loading and unloading functions for shape, style, ink, transform, color set, color profile, and tag objects.

When you unload an object, QuickDraw GX always first disposes of all of the live and dead caches for the object.

A recommended approach is to initially write your application without the use of object loading and unloading functions. Then, experiment with loading and unloading functions to improve performance.

The QuickDraw GX loading and unloading functions are described in the section "Loading and Unloading Objects" beginning on page 2-26.

When objects are loaded and unloaded by QuickDraw GX is discussed in the section "Low-Memory Conditions" beginning on page 2-10.

Functions That Do Not Require a Graphics Client or Heap

Almost all QuickDraw GX functions require both a graphics client and a graphics client heap to execute. Table 2-1 lists the functions that either do not require a graphics client or require a graphics client but not its heap to execute.

Table 2-1 QuickDraw GX functions that do not require a graphics client or heap

Memory requirements	Function

graphics client not required	`GXValidateGraphicsClient`
	`GXGetUserGraphicsDebug`
	`GXSetUserGraphicsDebug`
	`GXNewGraphicsClient`
	`GXGetGraphicsClient`
	`GXSetGraphicsClient`
	`GXDisposeGraphicsClient`
	`GXGetGraphicsClients`
	`GXGetConvertQDFont`
	`GXSetConvertQDFont`

graphics client required; graphics client heap not required	All of the functions described in the chapter "Errors, Warnings, and Notices" (excluding the application-defined functions)
	All of the functions described in the chapter "QuickDraw GX Mathematics"

	`GXSetValidation`
	`GXGetValidation`
	`GXSetValidationError`
	`GXGetGraphicsPollingHandler`
	`GXSetGraphicsPollingHandler`
	`GXEnterGraphics`

Specifying the Starting Location of a Graphics Client

If you use the `GXNewGraphicsClient` function to specify the starting location of a new graphics client, you must also specify the requested size of the graphics client heap. In this case, the size in bytes of the graphics client heap requested is used for a contiguous block of memory for both the graphics client and heap. In all other cases, the graphics client heap is allocated as a discontiguous memory block and the entire memory allocation requested by specifying a `memoryLength` parameter for the `GXNewGraphicsClient` function is assigned to the new graphics client heap.

Use the `GXNewGraphicsClient` function if you need to create a graphics client without allocating any memory. This allows you to draw at interrupt time. For example, you may want to report `out_of_memory` errors in a dialog box.

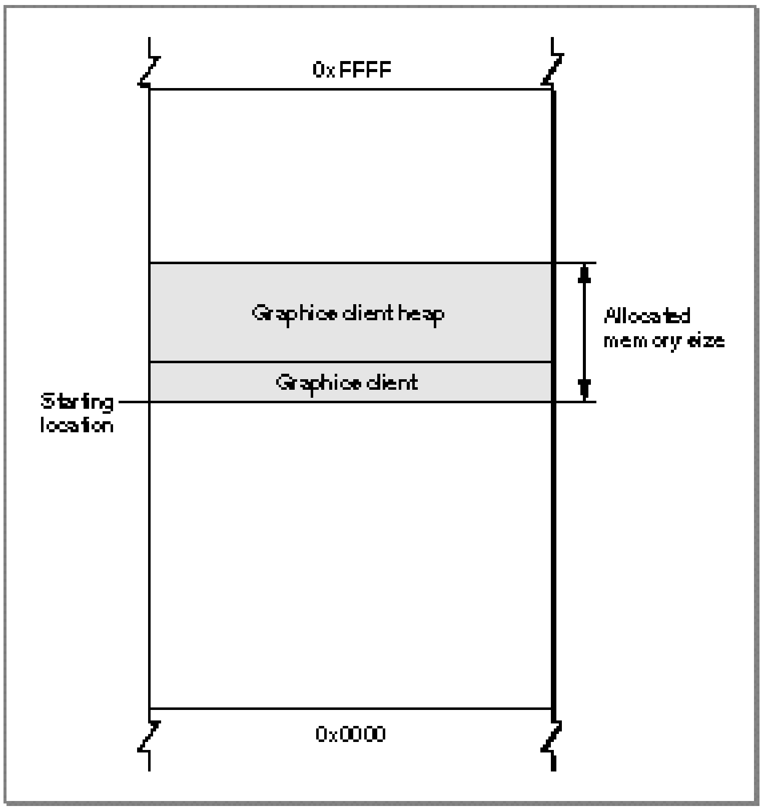
Listing 2-4 demonstrates how to specify a memory size and a memory starting location for a graphics client and its heap.

Listing 2-4 Specifying the starting location and size for a graphics client and its heap

```
gxGraphicsClient     newClient;
char                 memoryBuffer[10000]

newClient = GXNewGraphicsClient(&memoryBuffer[0],
                              sizeof (memoryBuffer),
                              gxNoAttributes);
   // After we attempted to create the graphics client, we need
   // to determine if the call succeeded.If it did not ( as in the
   // case for all GX functions), “newClient” will be nil. If it
   // is, we alert the user to the problem. Otherwise, we attempt
   // to allocate the GX heap.

if (newClient)

   GXEnterGraphics();
   // Calling GXEnterGraphics allocates the memory within the GX
   // heap. The only reason the call would not succeed is if
   // there is not enough memory. In this case, the graphics
   // error which is posted is -27999 (out of memory). At this
   // point, we have not installed an error handler, so we check
   // for the error number corresponding to the out-of-memory
   // error.

if ( GXGetGraphicsError( nil ) == -27999 ) {
   // Because we canot allocate the requested size for our GX
   // heap, we need to throw away the client we created and alert
   // the user that there is not enough memory to continue..
   //
   GXDisposeGraphicsClient(newClient);
      >>application code to alert user and shut down app
} else {
      >>application code to alert user and shut down app
}}
```

The `myClient` variable holds the new graphics client. You can use this variable to access the graphics client any time you need it. The combined size of the graphics client and the graphics client heap is 10 KB and its starting location in memory is at the starting location of the buffer. Since the memory starting location is specified, the new graphics client and its heap use contiguous memory, as shown in Figure 2-1.

Figure 2-1 Creating a graphics client by specifying the memory starting location



Working With Multiple Graphics Clients

The exceptional QuickDraw GX application may need multiple graphics clients to provide special features. For example, an application may want to create multiple graphics clients to provide a QuickDraw GX environment with

- segmented memory to allow some sets of objects to have more memory than others
- different error states so that one state would have an error condition and the other would have a normal condition.

Another example is a QuickDraw GX application that needs to display a dialog box to convey status information while continuing to perform other tasks. By using separate graphics clients for the dialog box and the other task-oriented part of the application, you can guarantee that QuickDraw GX will not affect the memory being used for the dialog box.

Two disadvantages of having multiple graphics clients are that

- objects cannot be shared between graphics clients
- memory may become fragmented as the memory size grows

Without object sharing, if an object is to be used by more than one graphics client, the object must be duplicated and this requires additional memory overhead. Fragmented memory results from QuickDraw GX objects being initially allocated to a large block of memory and subsequent addition of multiple discontiguous memory blocks.

If you are going to have multiple graphics clients, you must explicitly create them using the `GXNewGraphicsClient` and `GXEnterGraphics` functions. This assures that a reference is returned for each new graphics client. If you allow QuickDraw GX to implicitly create a graphics client, QuickDraw GX has no way of returning a reference.

The `GXGetGraphicsClient`, `GXGetGraphicsClients`, and `GXSetGraphicsClient` functions allow you to work with the graphics clients that you create.

You can use the `GXGetGraphicsClients` function to return some or all of the graphics client references that have been allocated by QuickDraw GX. The `GXGetGraphicsClients` function is described on page 2-25.

You can use the `GXSetGraphicsClient` function to change the active graphics client for your application and the `GXGetGraphicsClient` function to return the active graphics client for your application. These functions may be used prior to calling `GXEnterGraphics` and `GXExitGraphics` to specify the active graphics client. The `GXSetGraphicsClient` function is described on page 2-26 and the `GXGetGraphicsClients` function is described on page 2-25.

Creating graphics clients and graphics client heaps explicitly is described in the section “Explicit Creation” beginning on page 2-6.

QuickDraw GX Memory Management Reference

This section describes the constants, data types, and functions related to QuickDraw GX memory management. The section “Constants and Data Types” gives the type definition of the graphics client and the graphics client attributes enumeration. The section “Functions” describes the functions used for creating and disposing of a graphics client, working with multiple graphics clients, and loading and unloading objects.

Constants and Data Types

This section describes the constants and data types that are used for memory management.

Graphics Client Object

QuickDraw GX provides you with access to the graphics client object through a graphics client reference:

```
typedef struct gxPrivateGraphicsClientRecord *gxGraphicsClient;
```

In this type definition, `gxGraphicsClient` is a type-checked reference, not an actual pointer to any defined structure. The contents of the graphics client object are private.

Graphics Client Attributes

The options for the `attribute` parameter of the `GXNewGraphicsClient` function are defined in the `gxClientAttributes` enumeration:

```
enum gxClientAttributes {
   gxStaticHeapClient= 0x0001
};

typedef long gxClientAttribute;
```

Constant descriptions

`gxStaticHeapClient`
QuickDraw GX will never add additional memory blocks to the graphics client heap.

A graphics client having a `gxClientAttributes` value of 0 may add additional memory blocks to its heap, as required. This is the standard default behavior.

For additional information, see the section "Creating a Graphics Client and Its Graphics Client Heap" beginning on page 2-5. The `GXNewGraphicsClient` function is described on page 2-19.

Functions

This section describes the Quickdraw GX functions you can use to

- create and dispose of graphics clients
- allocate and dispose of graphics client heaps
- load and unload objects
- work with multiple graphics clients

Creating and Disposing of a Graphics Client

This section describes the QuickDraw GX functions you can use to

- create a new graphics client
- dispose of a graphics client

GXNewGraphicsClient

You can use the `GXNewGraphicsClient` function to create a new graphics client.

```
gxGraphicsClient GXNewGraphicsClient(void *memoryStart,
                    long memoryLength, gxClientAttribute attribute);
```

`memoryStart`	A pointer to the memory location where the new graphics client will begin.
`memoryLength`	The requested size in bytes of the QuickDraw GX graphics client heap.
`attribute`	The attributes flag set for the new graphics client.
function result	A reference to the new graphics client object.

DESCRIPTION

The `GXNewGraphicsClient` function creates a new graphics client and makes it the active graphics client for this application. The graphics client specifies the memory location, the size in bytes, and the attributes of its graphics client heap. When additional memory blocks are allocated to the graphics client heap, their locations and sizes are also stored in the graphics client. The `GXNewGraphicsClient` function does not allocate memory for the graphics client heap. Calling the `GXEnterGraphics` functionallocates the heap.

If you are going to make a `GXNewGraphicsClient` call, it must be the first QuickDraw GX call in your application. Otherwise, a Quickdraw GX call may implicitly create the first graphics client and any subsequent `GXNewGraphicsClient` call creates another graphics client. If you want to create multiple graphics client objects, you can call this routine several times.

The `memoryStart` parameter specifies the starting location in memory for the graphics client and its graphics client heap. If you specify `nil`, QuickDraw GX selects the location for you. This is the most common selection. Since QuickDraw GX is managing memory, it selects what it believes is the optimum location in memory for the new graphics client. However, in the rare case in which you need to specify the memory location, you can use the memoryStart parameter to specify the exact location of the graphics client. If you specify the memoryStart parameter, you must also specify the memoryLength parameter.

The memoryLength parameter specifies the size of the heap in bytes. If you pass 0 and there is no 'gasz' resource, QuickDraw GX version 1.0 creates a graphics client with a default heap size of 600 KB. If there is a 'gasz' resource, QuickDraw GX uses its size value instead.

The attributes parameter is a flag from the gxClientAttributes enumeration that defines whether QuickDraw GX will or will not add additional memory blocks to the newly defined, but not allocated, graphics client heap. A flag of default value 0 indicates that QuickDraw GX may add memory blocks to the graphics client heap, as required. A flag of value 1 is the gxStaticHeapClient constant and indicates that QuickDraw GX will never add memory blocks to the initially allocated graphics client heap.

If QuickDraw GX is unable to create a graphics client, there probably is not sufficient memory. As a result, the function returns nil. Note that QuickDraw GX does not post an error since there is no graphics client to post the error to.

SPECIAL CONSIDERATIONS

If no error results, the GXNewGraphicsClient function creates a graphics client object; you are responsible for disposing of that object when you no longer need it.

SEE ALSO

The use of the GXNewGraphicsClient function to create a new graphics client is described in the section “Creating a Graphics Client and Its Graphics Client Heap” beginning on page 2-5.

To determine the correct size of the memory for your graphics client, see the section “Determining Memory Requirements for a Graphics Client Heap” beginning on page 2-8.

The gxClientAttribute enumeration is described in the section “Graphics Client Attributes” beginning on page 2-18.

QuickDraw GX functions that do not require a graphics client or a graphics client heap are described in the section “Functions That Do Not Require a Graphics Client or Heap” beginning on page 2-14.

If you need to specify the memory starting location of the graphics client and its graphics client heap, see the section “Specifying the Starting Location of a Graphics Client” beginning on page 2-14.

GXDisposeGraphicsClient

You can use the `GXDisposeGraphicsClient` function to dispose of a specific graphics client.

```
void GXDisposeGraphicsClient(gxGraphicsClient client);
```

`client`	A reference to the graphics client to be disposed of.

DESCRIPTION

The `GXDisposeGraphicsClient` function is the last QuickDraw GX call that an application being debugged should make. It disposes of all the data structures associated with the passed graphics client, including its heap. If the application does not make this call, QuickDraw GX automatically disposes of all graphics clients that belong to the exiting application. However, in this case the graphics clients are considered aborted instead of being disposed of normally, and therefore QuickDraw GX does not report any errors that occur during the process of disposing of these graphics clients.

SPECIAL CONSIDERATIONS

If your `GXNewGraphicsClient` call failed to create a graphics client and returned `nil`, this function accepts `nil` as a valid graphics client and disposes of the referenced graphics client.

When your application is ready to ship, you should remove the terminating `GXDisposeGraphicsClient` function and rely on QuickDraw GX to automatically dispose of your graphics clients.

SEE ALSO

The role of the `GXDisposeGraphicsClient` function in disposing of a graphics client is described in the section "Disposing of a Graphics Client and Graphics Client Heap" beginning on page 2-9.

The `GXNewGraphicsClient` function is used to create a new graphics client from memory and is described on page 2-19.

Allocating and Disposing of a Graphics Client Heap

This section describes the QuickDraw GX functions you can use to

- allocate a graphics client heap
- obtain a list of all of the allocated graphics clients
- dispose of a graphics client heap

GXEnterGraphics

You can use the `GXEnterGraphics` function to allocate memory for a QuickDraw GX graphics client heap.

```
void GXEnterGraphics(void);
```

DESCRIPTION

The `GXEnterGraphics` function allocates memory for a graphics client heap and initializes the default data structures. QuickDraw GX obtains the memory starting location, memory length, and attributes for the new graphics client heap from the active graphics client.

Normally, you never need to call `GXEnterGraphics`. You should call this function only in the specific instance that you want to isolate the QuickDraw GX call to a specific part of the application. You then usethe `GXExitGraphics` function to remove all memory used by QuickDraw GX and then use the `GXEnterGraphics` function to begin using QuickDraw GX memory again.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`

SEE ALSO

The use of the `GXEnterGraphics` function to allocate memory to a graphics client is described in the section “Creating a Graphics Client and Its Graphics Client Heap” beginning on page 2-5.

The `GXExitGraphics` function deallocates memory for the QuickDraw GX graphics client heap and removes the default data structures. This function is described in the next section.

QuickDraw GX functions that do not require a graphics client or a graphics client heap are described in the section “Functions That Do Not Require a Graphics Client or Heap” beginning on page 2-14.

GXExitGraphics

You can use the `GXExitGraphics` function to dispose of the default structures and the active QuickDraw GX graphics client heap.

```
void GXExitGraphics(void);
```

DESCRIPTION

The `GXExitGraphics` function disposes of all of the default data structures that you have created in your QuickDraw GX application and disposes of the active graphics client heap. If a notice handler routine has been installed, it is called to report any objects allocated by the application that have not been disposed of.

Normally, you never need to call the `GXExitGraphics` function if you use the `GXDisposeGraphicsClient` function.

SPECIAL CONSIDERATIONS

In the debugging version of QuickDraw GX, you can call the `GXExitGraphics` function if you want to confirm that all QuickDraw GX objects that you allocated have been disposed of.

When your application is ready to ship, you should remove the terminating `GXExitGraphics` function and rely on QuickDraw GX to automatically dispose of your graphics client heaps.

ERRORS, WARNINGS, AND NOTICES

Notices (debugging only)

```
shape_not_disposed
font_not_disposed
style_not_disposed
ink_not_disposed
transform_not_disposed
colorSet_not_disposed
colorProfile_not_disposed
```

SEE ALSO

The role of the `GXExitGraphicsClient` function in disposing of a graphics client heap is described in the section “Disposing of a Graphics Client and Graphics Client Heap” beginning on page 2-9.

The `GXDisposeGraphicsClient` function is described on page 2-21.

The `GXEnterGraphics` function allocates memory for the QuickDraw GX graphics client heap and initializes the default data structures. This function is described on page 2-22.

Working With Multiple Graphics Clients

This section describes the QuickDraw GX functions you can use to

- return the active graphics client
- change the active graphics client

GXGetGraphicsClient

You can use the `GXGetGraphicsClient` function to return the active graphics client to your application.

```
gxGraphicsClient GXGetGraphicsClient(void);
```

function result The active graphics client.

DESCRIPTION

The `GXGetGraphicsClient` function returns the active graphics client. Each application has its own active graphics client. The only way that the active graphics client is changed within an application is when the application calls the `GXSetGraphicsClient` function or when a new graphics client is created by the `GXNewGraphicsClient` call.

SEE ALSO

For additional information about graphics clients, see the section “About QuickDraw GX Memory Management” beginning on page 2-3.

Multiple graphics clients are discussed in the section“Working With Multiple Graphics Clients” beginning on page 2-16.

The `GXGetGraphicsClients` function returns all or some of the graphics clients that have been allocated by QuickDraw GX. This function is described in the next section.

GXGetGraphicsClients

You can use the `GXGetGraphicsClients` function to list all of the graphics clients that have been allocated by QuickDraw GX.

```
long GXGetGraphicsClients(long index, long count,
                          gxGraphicsClient clients[]);
```

`index`	The one-based index into the list of all graphics clients that indicates the first client to return.
`count`	The number of graphics clients to be returned.
`clients`	An array of graphics client references. On return, the array contains references to the allocated graphics clients.
function result	The number of graphics clients returned.

DESCRIPTION

The `GXGetGraphicsClients` function copies the graphics client references specified by the `index` and `count` parameters into the array. It will return the graphics clients that are owned by other applications in addition to the ones owned by the calling application. Specifying the value 1 for the `index` parameter returns the first client. Specifying the `gxSelectToEnd` constant for the `count` parameter returns all remaining graphics clients, starting with the indexed graphics client. If `nil` is passed for the `clients` parameter, no graphics clients are returned.

SEE ALSO

For additional information about graphics clients, see the section "About QuickDraw GX Memory Management" beginning on page 2-3.

Multiple graphics clients are discussed in the section"Working With Multiple Graphics Clients" beginning on page 2-16.

GXSetGraphicsClient

You can use the `GXSetGraphicsClient` function to change the active graphics client for your application.

```
void GXSetGraphicsClient(gxGraphicsClient client);
```

`client`	A reference to the graphics client that is to become active.

DESCRIPTION

The `GXSetGraphicsClient` function can be used to switch to any of the graphics clients that your application owns; it may not switch to graphics clients that other applications own.

The active graphics client determines which QuickDraw GX graphics client heap to use for subsequent QuickDraw GX calls. Note that if you create a QuickDraw GX object with one graphics client active and switch to another one, you may not make calls that use the object. This is because an object cannot be shared by graphics clients. The object must be duplicated.

SEE ALSO

See the section“Working With Multiple Graphics Clients” beginning on page 2-16 for more information about multiple graphics clients.

Loading and Unloading Objects

This section describes the functions you use to load objects from disk storage to memory and to unload objects from memory to disk storage.

GXLoadShape

You can use the `GXLoadShape` function to load a shape into memory.

```
void GXLoadShape(gxShape target);
```

`target`	A reference to the shape object to be loaded into memory.

DESCRIPTION

The `GXLoadShape` function moves a shape object from disk storage to the active graphics client heap. When you or QuickDraw GX unload a shape object from memory to disk storage using the `GXUnloadShape` function, QuickDraw GX creates a 4-byte stub that remains in the active graphics client heap. When you use the `GXLoadShape` function to retrieve the stored object, QuickDraw GX obtains the location of the stored shape object from the stub.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`shape_is_nil`

SEE ALSO

For additional information about loading objects from disk storage to memory and unloading objects from memory to disk storage, see the sections "Loading and Unloading Objects" beginning on page 2-12.

The `GXUnloadShape` function is described in the next section.

GXUnloadShape

You can use the `GXUnloadShape` function to unload a shape from memory.

```
void GXUnloadShape(gxShape target);
```

`target`	A reference to the shape object to be unloaded from memory.

DESCRIPTION

The `GXUnloadShape` function moves a shape object from the active graphics client heap to disk storage. When you or QuickDraw GX use the `GXUnloadShape` function to unload a shape object from memory to disk storage, QuickDraw GX stores its location in a 4-byte stub in the active graphics client heap. When you use the `GXLoadShape` function to reload the object from disk storage to memory, QuickDraw GX uses the stub to find the stored shape object.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`shape_is_nil`

SEE ALSO

For additional information about loading objects from disk storage to memory and unloading objects from memory to disk storage, see the section "Loading and Unloading Objects" beginning on page 2-12.

The `GXLoadShape` function is described in the previous section.

GXLoadStyle

You can use the `GXLoadStyle` function to load a style into memory.

```
void GXLoadStyle(gxStyle target);
```

`target`	A reference to the style object to be loaded into memory.

DESCRIPTION

The `GXLoadStyle` function moves a style object from disk storage to the active graphics client heap of your application. When you or QuickDraw GX unload a style object from memory to disk storage using the `GXUnloadStyle` function, QuickDraw GX creates a 4-byte stub that remains in the graphics client heap. When you use the `GXLoadStyle` function to retrieve the stored style, QuickDraw GX obtains the location of the stored style object from the stub.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`style_is_nil`

SEE ALSO

For additional information about loading objects from disk storage to memory and unloading objects from memory to disk storage, see the section "Loading and Unloading Objects" beginning on page 2-12.

The `GXUnloadStyle` function is described in the next section.

GXUnloadStyle

You can use the `GXUnloadStyle` function to unload a style from memory.

```
void GXUnloadStyle(gxStyle target);
```

`target`	A reference to the style object to be unloaded from memory.

DESCRIPTION

The `GXUnloadStyle` function moves a style object from the active graphics client heap to disk storage. When you or QuickDraw GX use the `GXUnloadStyle` function to unload a style object from memory to disk storage, QuickDraw GX stores its location in a 4-byte stub in the active graphics client heap. When you use the `GXLoadStyle` function to reload the object from disk storage to memory, QuickDraw GX uses the stub to find the stored style object.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`style_is_nil`

SEE ALSO

For additional information about loading objects from disk storage to memory and unloading objects from memory to disk storage, see the section “Loading and Unloading Objects” beginning on page 2-12.

The `GXLoadStyle` function is described in the previous section.

GXLoadInk

You can use the `GXLoadInk` function to load an ink into memory.

```
void GXLoadInk(gxInk target);
```

`target`	A reference to the ink object to be loaded into memory.

DESCRIPTION

The `GXLoadInk` function moves an ink object from disk storage to the active graphics client heap of your application. When you or QuickDraw GX unload an ink object from memory to disk storage using the `GXUnloadInk` function, QuickDraw GX creates a

4-byte stub that remains in the graphics client heap. When you use the `GXLoadInk` function to retrieve the stored object, QuickDraw GX obtains the location of the stored ink object from the stub.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`ink_is_nil`

SEE ALSO

For additional information about loading objects from disk storage to memory and unloading objects from memory to disk storage, see the section “Loading and Unloading Objects” beginning on page 2-12.

The `GXUnloadInk` function is described in the next section.

GXUnloadInk

You can use the `GXUnloadInk` function to unload an ink from memory.

```
void GXUnloadInk(gxInk target);
```

`target`	A reference to the ink object to be unloaded from memory.

DESCRIPTION

The `GXUnloadInk` function moves an ink object from the active graphics client heap to disk storage. When you or QuickDraw GX use the `GXUnloadInk` function to unload an ink object from memory to disk storage, QuickDraw GX stores its location in a 4-byte stub in the active graphics client heap. When you use the `GXLoadInk` function to reload the object from disk storage to memory, QuickDraw GX uses the stub to find the stored ink object.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`ink_is_nil`

SEE ALSO

For additional information about loading objects from disk storage to memory and unloading objects from memory to disk storage, see the section "Loading and Unloading Objects" beginning on page 2-12.

The `GXLoadInk` function is described in the previous section.

GXLoadTransform

You can use the `GXLoadTransform` function to load a transform into memory.

```
void GXLoadTransform(gxTransform target);
```

`target`	A reference to the transform object to be loaded into memory.

DESCRIPTION

The `GXLoadTransform` function moves a transform object from disk storage to the active graphics client heap of your application. When you or QuickDraw GX unload a transform object from memory to disk storage using the `GXUnloadTransform` function, QuickDraw GX creates a 4-byte stub that remains in the graphics client heap. When you use the `GXLoadTransform` function to retrieve the stored object, QuickDraw GX obtains the location of the stored transform object from the stub.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`transform_is_nil`

SEE ALSO

For additional information about loading objects from disk storage to memory and unloading objects from memory to disk storage, see the section "Loading and Unloading Objects" beginning on page 2-12.

The `GXUnloadTransform` function is described in the next section.

GXUnloadTransform

You can use the `GXUnloadTransform` function to unload a transform from memory.

```
void GXUnloadTransform(gxTransform target);
```

`target`	A reference to the transform object to be unloaded from memory.

DESCRIPTION

The `GXUnloadTransform` function moves a transform object from the active graphics client heap to disk storage. When you or QuickDraw GX use the `GXUnloadTransform` function to unload a transform object from memory to disk storage, QuickDraw GX stores its location in a 4-byte stub in the active graphics client heap. When you use the `GXLoadTransform` function to reload the object from disk storage to memory, QuickDraw GX uses the stub to find the stored transform object.

ERRORS, WARNINGS, AND NOTICES

Errors
```
out_of_memory
transform_is_nil
```

SEE ALSO

For additional information about loading objects from disk storage to memory and unloading objects from memory to disk storage, see the section “Loading and Unloading Objects” beginning on page 2-12.

The `GXLoadTransform` function is described in the previous section.

GXLoadColorSet

You can use the `GXLoadColorSet` function to load a color set into memory.

```
void GXLoadColorSet(gxColorSet target);
```

`target`	A reference to the color set object to be loaded into memory.

DESCRIPTION

The `GXLoadColorSet` function moves a color set object from disk storage to the active graphics client heap of your application. When you or QuickDraw GX unload a color set object from memory to disk storage using the `GXUnloadColorSet` function, QuickDraw GX creates a 4-byte stub that remains in the graphics client heap. When you use the `GXLoadColorSet` function to retrieve the stored object, QuickDraw GX obtains the location of the stored color set object from the stub.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`color_set_is_nil`

SEE ALSO

For additional information about loading objects from disk storage to memory and unloading objects from memory to disk storage, see the section "Loading and Unloading Objects" beginning on page 2-12.

The `GXUnloadColorSet` function is described in the next section.

GXUnloadColorSet

You can use the `GXUnloadColorSet` function to unload a color set from memory.

```
void GXUnloadColorSet(gxColorSet target);
```

`target`	A reference to the color set object to be unloaded from memory.

DESCRIPTION

The `GXUnloadColorSet` function moves a color set object from the active graphics client heap to disk storage. When you or QuickDraw GX use the `GXUnloadColorSet` function to unload a color set object from memory to disk storage, QuickDraw GX stores its location in a 4-byte stub in the active graphics client heap. When you use the `GXLoadColorSet` function to reload the object from disk storage to memory, QuickDraw GX uses the stub to find the stored color set object.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`color_set_is_nil`

SEE ALSO

For additional information about loading objects from disk storage to memory and unloading objects from memory to disk storage, see the section “Loading and Unloading Objects” beginning on page 2-12.

The `GXLoadColorSet` function is described in the previous section.

GXLoadColorProfile

You can use the `GXLoadColorProfile` function to load a color profile into memory.

```
void GXLoadColorProfile(gxColorProfile target);
```

`target`	A reference to the color profile object to be loaded into memory.

DESCRIPTION

The `GXLoadColorProfile` function moves a color profile object from disk storage to the active graphics client heap of your application. When you or QuickDraw GX unload a color profile object from memory to disk storage using the `GXUnloadColorProfile` function, QuickDraw GX creates a 4-byte stub that remains in the graphics client heap. When you use the `GXLoadColorProfile` function to retrieve the stored object, QuickDraw GX obtains the location of the stored color profile object from the stub.

ERRORS, WARNINGS, AND NOTICES

Errors
```
out_of_memory
color_profile_is_nil
```

SEE ALSO

For additional information about loading objects from disk storage to memory and unloading objects from memory to disk storage, see the section “Loading and Unloading Objects” beginning on page 2-12.

The `GXUnloadColorProfile` function is described in the next section.

GXUnloadColorProfile

You can use the `GXUnloadColorProfile` function to unload a color profile from memory.

```
void GXUnloadColorProfile(gxColorProfile target);
```

`target`	A reference to the color profile object to be unloaded from memory.

DESCRIPTION

The `GXUnloadColorProfile` function moves a color profile object from the active graphics client heap to disk storage. When you or QuickDraw GX use the `GXUnloadColorProfile` function to unload a color profile object from memory to disk storage, QuickDraw GX stores its location in a 4-byte stub in the active graphics client heap. When you use the `GXLoadColorProfile` function to reload the object from disk storage to memory, QuickDraw GX uses the stub to find the stored color profile object.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`color_profile_is_nil`

SEE ALSO

For additional information about loading objects from disk storage to memory and unloading objects from memory to disk storage, see the section “Loading and Unloading Objects” beginning on page 2-12.

The `GXLoadColorProfile` function is described in the previous section.

GXLoadTag

You can use the `GXLoadTag` function to load a tag into memory.

```
void GXLoadTag(gxTag target);
```

`target`	A reference to the tag object to be loaded into memory.

DESCRIPTION

The `GXLoadTag` function moves a tag object from disk storage to the active graphics client heap of your application. When you or QuickDraw GX unload a tag object from memory to disk storage using the `GXUnloadTag` function, QuickDraw GX creates a 4-byte stub that remains in the graphics client heap. When you use the `GXLoadTag` function to retrieve the stored object, QuickDraw GX obtains the location of the stored tag object from the stub.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`tag_is_nil`

SEE ALSO

For additional information about loading objects from disk storage to memory and unloading objects from memory to disk storage, see the section “Loading and Unloading Objects” beginning on page 2-12.

The `GXUnloadTag` function is described in the next section.

GXUnloadTag

You can use the `GXUnloadTag` function to unload a tag from memory.

```
void GXUnloadTag(gxTag target);
```

`target`	A reference to the tag object to be unloaded from memory.

DESCRIPTION

The `GXUnloadTag` function moves a tag object from the active graphics client heap to disk storage. When you or QuickDraw GX use the `GXUnloadTag` function to unload a tag object from memory to disk storage, QuickDraw GX stores its location in a 4-byte stub in the active graphics client heap. When you use the `GXLoadTag` function to reload the object from disk storage to memory, QuickDraw GX uses the stub to find the stored tag object.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
tag_is_nil
```

SEE ALSO

For additional information about loading objects from disk storage to memory and unloading objects from memory to disk storage, see the section “Loading and Unloading Objects” beginning on page 2-12.

The `GXLoadTag` function is described in the previous section.

Summary of QuickDraw GX Memory Management

Constants and Data Types

Graphics Client Object

```
typedef struct gxPrivateGraphicsClientRecord *gxGraphicsClient;
```

Graphics Client Attributes

```
enum gxClientAttributes {
   gxStaticHeapClient = 0x0001
};

typedef long gxClientAttribute;
```

Functions

Creating and Disposing of a Graphics Client

```
gxGraphicsClient GXNewGraphicsClient
                             (void *memoryStart, long memoryLength,
                              gxClientAttribute attribute);
void GXDisposeGraphicsClient(gxGraphicsClient client);
```

Allocating and Disposing of a Graphics Client Heap

```
void GXEnterGraphics         (void);
void GXExitGraphics          (void);
```

Working With Multiple Graphics Clients

```
gxGraphicsClient GXGetGraphicsClient
                             (void);
long GXGetGraphicsClients    (long index, long count,
                              gxGraphicsClient clients[]);
void GXSetGraphicsClient     (gxGraphicsClient client);
```

Loading and Unloading Objects

```
void GXLoadShape              (gxShape target);
void GXUnloadShape            (gxShape target);
void GXLoadStyle              (gxStyle target);
void GXUnloadStyle            (gxStyle target);
void GXLoadInk                (gxInk target);
void GXUnloadInk              (gxInk target);
void GXLoadTransform          (gxTransform target);
void GXUnloadTransform        (gxTransform target);
void GXLoadColorSet           (gxColorSet target);
void GXUnloadColorSet         (gxColorSet target);
void GXLoadColorProfile       (gxColorProfile target);
void GXUnloadColorProfile     (gxColorProfile target);
void GXLoadTag                (gxTag target);
void GXUnloadTag              (gxTag target);
```

C H A P T E R 3

Errors, Warnings, and Notices

Contents

This chapter describes the errors, warnings, and notices that can be posted by QuickDraw GX functions, and how you can manipulate them. In addition, this chapter describes how you can use application-defined handlers to provide alternative or complementary processing of errors, warnings, and notices. The reference sections of the *Inside Macintosh: QuickDraw GX* books list the errors, warnings, and notices for each function that they describe.

Before reading this chapter, you should be familiar with the information in the chapter "Introduction to QuickDraw GX" in *Inside Macintosh: QuickDraw GX Objects*.

The errors, warnings, and notices and their related functions that are discussed in this chapter pertain to the graphic and typography parts of QuickDraw GX and do not, in general, apply to printing. For more information on printing errors, see *Inside Macintosh: QuickDraw GX Printing* and *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers.*

This chapter starts by introducing you to the errors, warnings, and notices provided in the debugging and non-debugging versions of QuickDraw GX. It then shows you how to use their related functions to

- obtain the QuickDraw GX errors, warnings, and notices posted
- change the QuickDraw GX errors, warnings, and notices posted
- ignore QuickDraw GX warnings and notices
- install application-defined error, warning, and notice handlers

This chapter also contains reference information for all data types, application-defined handlers, and functions associated with QuickDraw GX errors, warnings, and notices.

About QuickDraw GX Errors, Warnings, and Notices

QuickDraw GX posts **errors**, **warnings**, or **notices**, depending upon the severity of the problem that was detected when your application was running. The three types of QuickDraw GX execution problems are

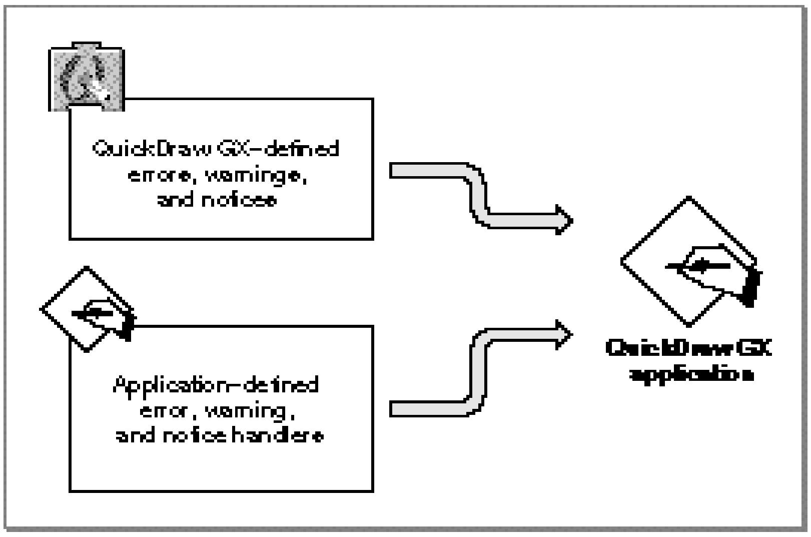
- **Errors.** QuickDraw GX posts errors whenever a function in your application is unable to execute. An error indicates that an operation cannot continue. Execution terminates at the nonexecutable function. When an error is posted inside a QuickDraw GX function, the function returns immediately with a function result (if any) of 0 or `nil`.
- **Warnings.** QuickDraw GX posts warnings whenever your application executes a function that doesn't provide the result that you expect. Execution continues internally, as if the warning had not been posted.
- **Notices.** QuickDraw GX posts notices to alert you to the fact that it has performed an unnecessary or redundant function. Execution continues as if the notice had not been posted. Graphics notices are posted only in the debugging version of QuickDraw GX.

In addition to the **posting** of errors, warnings, and notices, QuickDraw GX supports application-defined error, warning, and notice **handlers.** You can use your own handlers or QuickDraw GX's errors, warnings, and notices either separately or together.

To obtain errors, warnings, and notices, either check for QuickDraw GX errors, warnings, and notices or install your application's error, warning, and notice handlers. The use of error, warning, and notice handlers is a simple and efficient method of managing errors, warnings, and notices. Error handlers are described in the section "Installing an Error, Warning, or Notice Handler" beginning on page 3-40.

Figure 3-1 shows the relationship of the two problem-management approaches.

Figure 3-1 QuickDraw GX and application-defined error, warning, and notice management


QuickDraw GX–defined errors, warnings, and notices
Application–defined error, warning, and notice handlers
QuickDraw GX application

There are two versions of QuickDraw GX.

- **Non-debugging version.** This version of QuickDraw GX is intended for debugged applications used by the end user. The number of QuickDraw GX errors and warnings is limited. Notices are not posted. This version of QuickDraw GX is smaller and faster than the debugging version.
- **Debugging version.** This version of QuickDraw GX is intended for developers that are writing and debugging new applications. This version provides an extensive set of QuickDraw GX errors, warnings, and notices to assist in debugging and optimizing the performance of your application. Special functions are provided to assist in the posting, utilization, and control of debugging errors.

To determine if the debugging or non-debugging version is installed, use the `Gestalt` function described in the chapter "QuickDraw GX and the Macintosh Environment" in this book.

QuickDraw GX posts most errors and warnings only in the debugging version. The non-debugging version posts errors and warnings if the error could not be anticipated at compile time—for example, running out of memory or disk space. You should correct application problems that result in errors and warnings while developing your application. The non-debugging version does not include most of the errors and warnings that the debugging version provides.

QuickDraw GX non-debugging and debugging errors are defined by the `gxGraphicErrors` enumeration given in the section "Errors" beginning on page 3-42. QuickDraw GX non-debugging and debugging warnings are defined by the `gxGraphicWarnings` enumeration given in the section "Warnings" beginning on page 3-50. QuickDraw GX debugging notices are defined by the `gxGraphicNotices` enumeration given in the section "Notices" beginning on page 3-53.

Non-Debugging Version

When you install the non-debugging version, QuickDraw GX provides a reduced set of errors and warnings. Since the amount of testing is less, the non-debugging version of QuickDraw GX runs significantly faster than the debugging version. Use the non-debugging version for debugged applications that you have extensively tested using the debugging version of QuickDraw GX.

When the non-debugging version is installed and corrupt data is used, drawings may execute with undesirable results, including crashes, without the posting of errors and warnings. With other execution problems, the application may not crash, but the drawing may not yield the expected result.

In the non-debugging version, typical problem messages indicate that there is insufficient memory, insufficient storage space, or that the required fonts are not installed. If problems persist, you can always install the debugging version to assist in the analysis of the errors that are occurring.

For a complete list of errors, please see the graphics `errors.h` interface file. The many notices, warnings, and errors defined between `#ifdef debugging` and `#endif` in that file are available only with the debugging version.

A debugged application should encounter only errors like

```
out_of_memory
not_enough_memory_for_graphics_client_heap
graphics_client_too_small
could_not_create_backing_store
```

A debugged application should encounter warnings like

```
character_substitution_occurred
map_shape_out_of_range
move_shape_out_of_range
scale_shape_out_of_range
```

```
rotate_shape_out_of_range
skew_shape_out_of_range
map_transform_out_of_range
move_transform_out_of_range
scale_transform_out_of_range
rotate_transform_out_of_range
skew_transform_out_of_range
```

Both the debugging and non-debugging versions of QuickDraw GX provide a debugging utility called GraphicsBug. This versatile utility allows you to examine the details of each graphics object. GraphicsBug is described in the chapter "QuickDraw GX Debugging" in this book.

Errors

This section describes the errors that may be posted by both the debugging and non-debugging versions of QuickDraw GX. These errors can be grouped into the following categories:

- fatal errors
- internal errors
- recoverable errors
- font management errors
- bad parameter errors
- implementation limit errors
- font scaler errors

Each QuickDraw GX error has an error number and an error name. Table 3-1 gives the non-debugging error number ranges.

Table 3-1 Non-debugging error number ranges

Number	Name
–27999	`gxFirstSystemError`
–27999	`gxFirstFatalError`
–27951	`gxLastFatalError`
–27950	`gxFirstNonFatalError`
–27900	`gxFirstFontScalerError`
–27851	`gxLastFontScalerError`
–27850	`gxFirstParameterError`
–27800	`gxFirstImplementationLimitError`

QuickDraw GX **fatal errors** terminate operation and automatically call the `GXExitGraphics` function. Control returns to the calling application after the error is posted. If the function that caused the error returns a function result, its value is either 0 or `nil`. Table 3-2 lists fatal errors. They are included in both the debugging and non-debugging versions of QuickDraw GX.

Table 3-2 Fatal errors

Number	Name
–27999	`out_of_memory`
–27998	`internal_fatal_error`
–27997	`no_outline_font_found`
–27996	`not_enough_memory_for_graphics_client_heap`
–27995	`could_not_create_backing_store`

QuickDraw GX nonfatal **internal errors** indicate damaged files, memory problems, or incorrect implementation of QuickDraw GX. Table 3-3 lists the internal errors.

Table 3-3 Internal errors

Number	Name
–27950	`internal_error`
–27949	`internal_font_error`
–27948	`internal_layout_error`

Table 3-4 lists the QuickDraw GX recoverable errors.

Table 3-4 Recoverable errors

Number	Name
–27946	`could_not_dispose_backing_store`
–27945	`unflattening_interrupted_by_client`

Table 3-5 lists the QuickDraw GX **font management errors.**

Table 3-5 Font management errors

Number	Name
–27944	`font_cannot_be_changed`
–27943	`illegal_font_parameter`

Table 3-6 lists the QuickDraw GX **font scaler errors.**

Table 3-6 Font scaler errors

Number	Name
–27900	`null_font_scaler_context`
–27899	`null_font_scaler_input`
–27988	`invalid_font_scaler_context`
–27897	`invalid_font_scaler_input`
–27896	`invalid_font_scaler_font_data`
–27895	`font_scaler_newblock_failed`
–27894	`font_scaler_getfonttable_failed`
–27893	`font_scaler_bitmap_allocation_failed`
–27892	`font_scaler_outline_allocation_failed`
–27891	`required_font_scaler_table_missing`
–27890	`unsupported_font_scaler_outline_format`
–27889	`unsupported_font_scaler_stream_format`
–27888	`unsupported_font_scaler_font_format`
–27887	`font_scaler_hinting_error`
–27886	`font_scaler_rasterizer_error`
–27885	`font_scaler_internal_error`
–27884	`font_scaler_invalid_matrix`
–27883	`font_scaler_fixed_overflow`
–27882	`font_scaler_api_version_mismatch`
–27881	`font_scaler_streaming_aborted`
–27880	`unknown_font_scaler_error`

QuickDraw GX posts **bad parameter errors** when a required parameter is out of range, invalid, or is passed with the value of `nil`. Table 3-7 lists bad parameter errors.

Table 3-7 Bad parameter errors

Number	Name
–27850	`parameter_is_nil`
–27849	`shape_is_nil`
–27848	`style_is_nil`
–27847	`transform_is_nil`
–27846	`ink_is_nil`
–27845	`transferMode_is_nil`
–27844	`color_is_nil`
–27843	`colorProfile_is_nil`
–27842	`colorSet_is_nil`
–27841	`spoolProcedure_is_nil`
–27840	`tag_is_nil`
–27839	`type_is_nil`
–27838	`mapping_is_nil`
–27837	`invalid_viewDevice_reference`
–27836	`invalid_viewGroup_reference`
–27835	`invalid_viewPort_reference`

QuickDraw GX posts **implementation limit errors** to indicate that the size or number exceeds the size or number supported by the current version of QuickDraw GX. Table 3-8 lists the implementation limit errors.

Table 3-8 Implementation limit errors

Number	Name
–27800	`number_of_contours_exceeds_implementation_limit`
–27799	`number_of_points_exceeds_implementation_limit`
–27798	`size_of_polygon_exceeds_implementation_limit`
–27797	`size_of_path_exceeds_implementation_limit`
–27796	`size_of_text_exceeds_implementation_limit`
–27795	`size_of_bitmap_exceeds_implementation_limit`
–27794	`number_of_colors_exceeds_implementation_limit`
–27793	`procedure_not_reentrant`

Warnings

This section describes the warnings that the debugging and non-debugging versions of QuickDraw GX may post. These errors can be grouped into the following categories:

- stack, heap, and object warnings
- result is out of range warnings
- parameter is out of range warnings
- font scaler warnings
- unexpected result warnings
- storage warnings

Each QuickDraw GX warning has a unique **warning number** and **warning name.** Table 3-9 gives the non-debugging warning number ranges.

Table 3-9 Non-debugging warning number ranges

Number	Description
–26999	`gxFirstSystemWarning`
–26950	`gxFirstResultOutOfRangeWarning`
–26900	`gxFirstParameterOutOfRangeWarning`
–26850	`gxFirstFontScalerWarning`

QuickDraw GX **overflow warnings** occur when the number of warnings that have been added to the warning or notice stack exceeds the current implementation limit. An **underflow warning** occurs when the `GXPopGraphicsNotice` or `GXPopGraphicsWarning` function attempts to remove an error or warning on its ignore stack and there is no error or warning to remove. This topic is discussed in the section “Ignoring Warnings and Notices” beginning on page 3-37.

Table 3-10 lists QuickDraw GX stack, heap, and object warnings.

Table 3-10 Stack, heap, and object warnings

Number	Name
–26999	`warning_stack_underflow`
–26998	`warning_stack_overflow`
–26997	`notice_stack_underflow`
–26996	`notice_stack_overflow`
–26995	`about_to_grow_heap`
–26994	`about_to_unload_objects`

QuickDraw GX **result out of range warnings** occur when a function result is out of the usable or defined QuickDraw boundaries. Table 3-11 lists result out of range warnings.

Table 3-11 Result out of range warnings

Number	Name
–26950	`map_shape_out_of_range`
–26949	`move_shape_out_of_range`
–26948	`scale_shape_out_of_range`
–26947	`rotate_shape_out_of_range`
–26946	`skew_shape_out_of_range`
–26945	`map_transform_out_of_range`
–26944	`move_transform_out_of_range`
–26943	`scale_transform_out_of_range`
–26942	`rotate_transform_out_of_range`
–26941	`skew_transform_out_of_range`
–26940	`map_points_out_of_range`

QuickDraw GX **parameter out of range warnings** occur when a function parameter is out of the usable range. Table 3-12 lists parameter out of range warnings.

Table 3-12 Parameter out of range warnings

Number	Name
–26900	`contour_out_of_range`
–26899	`index_out_of_range_in_contour`
–26898	`picture_index_out_of_range`
–26897	`color_index_requested_not_found`
–26896	`colorSet_index_out_of_range`
–26895	`index_out_of_range`
–26894	`count_out_of_range`
–26893	`length_out_of_range`
–26892	`font_table_index_out_of_range`
–26891	`font_glyph_index_out_of_range`
–26890	`point_out_of_range`
–26889	`profile_response_out_of_range`

Table 3-13 lists QuickDraw GX **font scaler warnings.**

Table 3-13 Font scaler warnings

Number	Name
–26850	`font_scaler_no_output`
–26849	`font_scaler_fake_metrics`
–26848	`font_scaler_fake_linespacing`
–26847	`font_scaler_glyph_substitution`
–26846	`font_scaler_no_kerning_applied`

Table 3-14 lists QuickDraw GX **unexpected result warnings.**

Table 3-14 Unexpected result warnings

Warning number	Warning name
–26845	`character_substitution_took_place`
–26844	`unable_to_bounds_on_multiple_devices`
–26843	`font_language_not_found`
–26842	`font_not_found_during_unflattening`

Table 3-15 lists QuickDraw GX data stream **storage warnings.**

Table 3-15 Storage warnings

Number	Name
–26841	`unrecognized_stream_version`
–26840	`bad_data_in_stream`

Debugging Version

When you install the debugging version, QuickDraw GX posts errors, warnings, and notices in addition to those posted by the non-debugging version. The debugging analysis and resulting number of errors, warnings, and notices posted is far more extensive than can be provided by the non-debugging version of QuickDraw GX. As a result, the debugging version executes significantly slower than the non-debugging version.

The errors and warnings posted by both the debugging and non-debugging versions of QuickDraw GX are listed in the sections “Errors” beginning on page 3-6 and “Warnings” beginning on page 3-10. The errors, warnings, and notices described in the following sections are posted only in the debugging version of QuickDraw GX.

The debugging version also provides a number of useful functions that you can use to analyze your code and that assist in determining the cause of a wide variety of problems. These are described in the section “Using Errors, Warnings, and Notices” beginning on page 3-30.

The debugging version of QuickDraw GX also provides validation functions and GraphicsBug so that you can examine the details of each graphics object. These are described in the chapter “QuickDraw GX Debugging” in this book.

Errors

This section describes the errors that the debugging version of QuickDraw GX may post. QuickDraw GX debugging errors can be grouped into the following categories:

- internal errors
- font parameter errors
- bad parameter errors
- restricted access errors
- wrong type or bad reference errors
- validation errors

Table 3-16 gives the debugging error number range.

Table 3-16 Debugging error number range

Number	Name
–27700	`gxFirstSystemDebuggingError`
–27000	`gxLastSystemError`

Table 3-17 lists the internal debugging errors.

Table 3-17 Internal debugging errors

Number	Name
–27700	`functionality_unimplemented`
–27699	`clip_to_frame_shape_unimplemented`

Table 3-18 lists the font parameter debugging errors.

Table 3-18 Font parameter debugging errors

Number	Name
–27698	`illegal_font_storage_type`
–27697	`illegal_font_storage_reference`
–27696	`illegal_font_attributes`

QuickDraw GX bad parameter errors are posted when a required parameter is out of range, invalid, or is passed with the value of `nil`. Table 3-19 lists the bad parameter debugging errors.

Table 3-19 Bad parameter debugging errors

Number	Name
–27695	`parameter_out_of_range`
–27694	`inconsistent_parameters`
–27693	`index_is_less_than_zero`
–27692	`index_is_less_than_one`
–27691	`count_is_less_than_zero`
–27690	`count_is_less_than_one`
–27689	`contour_is_less_than_zero`
–27688	`length_is_less_than_zero`
–27687	`invalid_client_reference`
–27686	`invalid_graphics_heap_start_pointer`
–27685	`invalid_nongraphic_globals_pointer`
–27684	`colorSpace_out_of_range`
–27683	`pattern_lattice_out_of_range`
–27682	`frequency_parameter_out_of_range`
–27681	`tinting_parameter_out_of_range`
–27680	`method_parameter_out_of_range`
–27679	`space_may_not_be_indexed`
–27678	`glyph_index_too_small`
–27677	`no_glyphs_added_to_font`
–27676	`glyph_not_added_to_font`
–27675	`point_does_not_intersect_bitmap`
–27674	`required_font_table_not_present`
–27673	`unknown_font_table_format`
–27672	`shapeFill_not_allowed`
–27671	`inverseFill_face_must_set_clipLayer_flag`
–27670	`invalid_transferMode_colorSpace`
–27669	`colorProfile_must_be_nil`

continued

Table 3-19 Bad parameter debugging errors (continued)

Number	Name
–27668	`bitmap_pixel_size_must_be_1`
–27667	`empty_shape_not_allowed`
–27666	`ignorePlatformShape_not_allowed`
–27665	`nil_style_in_glyph_not_allowed`
–27664	`complex_glyph_style_not_allowed`
–27663	`invalid_mapping`
–27662	`cannot_set_item_shapes_to_nil`
–27661	`cannot_use_original_item_shapes_when_growing_picture`
–27660	`cannot_add_unspecified_new_glyphs`
–27659	`cannot_dispose_locked_tag`
–27658	`cannot_dispose_locked_shape`

Table 3-20 lists the QuickDraw GX **restricted access** debugging errors.

Table 3-20 Restricted access debugging errors

Number	Name
–27657	`shape_access_not_allowed`
–27656	`colorSet_access_restricted`
–27655	`colorProfile_access_restricted`
–27654	`tag_access_restricted`
–27653	`viewDevice_access_restricted`
–27652	`graphic_type_does_not_have_a_structure`
–27651	`style_run_array_does_not_match_number_of_characters`
–27650	`rectangles_cannot_be_inserted_into`
–27649	`unknown_graphics_heap`
–27648	`graphics_routine_selector_is_obsolete`
–27647	`cannot_set_graphics_client_memory_without_setting_size`
–27646	`graphics_client_memory_too_small`
–27645	`graphics_client_memory_is_already_allocated`
–27644	`viewPort_is_a_window`

Table 3-21 lists the QuickDraw GX **wrong type** and **bad reference** debugging errors.

Table 3-21 Wrong type and bad reference debugging errors

Number	Name
–27643	`illegal_type_for_shape`
–27642	`shape_does_not_contain_a_bitmap`
–27641	`shape_does_not_contain_text`
–27640	`picture_expected`
–27639	`bitmap_is_not_resizable`
–27638	`shape_may_not_be_a_bitmap`
–27637	`shape__may_not_be_a_picture`
–27636	`graphic_type_does_not_contain_points`
–27635	`graphic_type_does_not_have_multiple_contours`
–27634	`graphic_type_cannot_be_mapped`
–27633	`graphic_type_cannot_be_moved`
–27632	`graphic_type_cannot_be_scaled`
–27631	`graphic_type_cannot_be_rotated`
–27630	`graphic_type_cannot_be_skewed`
–27629	`graphic_type_cannot_be_reset`
–27628	`graphic_type_cannot_be_dashed`
–27627	`graphic_type_cannot_be_reduced`
–27626	`graphic_type_cannot_be_inset`
–27625	`shape_cannot_be_inverted`
–27624	`shape_does_not_have_area`
–27623	`shape_does not_have_length`
–27622	`first_glyph_advance_must_be_absolute`
–27621	`picture_cannot_contain_itself`
–27620	`viewPort_cannot_contain_itself`
–27619	`cannot_set_unique_items_attribute_when_picture_contains_items`
–27618	`layer_style_cannot_contain_a_face`
–27617	`layer_glyph_shape_cannot_contain_nil_styles`

QuickDraw GX posts validation errors only when QuickDraw GX validation error functions activate validation error checking. Validation error checking is discussed the chapter "QuickDraw GX Debugging" in this book. Table 3-22 lists the type validation debugging errors.

Table 3-22 Type validation debugging errors

Number	Name
–27616	`object_wrong_type`
–27615	`shape_wrong_type`
–27614	`style_wrong_type`
–27613	`ink_wrong_type`
–27612	`transform_wrong_type`
–27611	`device_wrong_type`
–27610	`port_wrong_type`

Table 3-23 lists the QuickDraw GX **cache validation** debugging errors.

Table 3-23 Cache validation debugging errors

Number	Name
–27609	`shape_cache_wrong_type`
–27608	`style_cache_wrong_type`
–27607	`ink_cache_wrong_type`
–27606	`transform_cache_wrong_type`
–27605	`port_cache_wrong_type`
–27604	`shape_cache_parent_mismatch`
–27603	`style_cache_parent_mismatch`
–27602	`ink_cache_parent_mismatch`
–27601	`transform_cache_parent_mismatch`
–27600	`port_cache_parent_mismatch`
–27599	`invalid_shape_cache_port`
–27598	`invalid_shape_cache_device`
–27597	`invalid_ink_cache_port`
–27596	`invalid_ink_cache_device`
–27595	`invalid_style_cache_port`

Table 3-23 Cache validation debugging errors (continued)

Number	Name
–27594	`invalid_style_cache_device`
–27593	`invalid_transform_cache_port`
–27592	`invalid_transform_cache_device`
–27591	`recursive_caches`

Table 3-24 lists the QuickDraw GX **shape cache** validation debugging errors.

Table 3-24 Shape cache validation shape debugging errors

Number	Name
–27590	`invalid_fillShape_ownerCount`
–27589	`recursive_fillShapes`

Table 3-25 lists the QuickDraw GX memory block validation debugging errors.

Table 3-25 Memory block validation debugging errors

Number	Name
–27588	`indirect_memory_block_too_small`
–27587	`indirect_memory_block_too_large`
–27586	`unexpected_nil_pointer`
–27585	`bad_address`

Table 3-26 lists the QuickDraw GX object validation debugging errors.

Table 3-26 Object validation debugging errors

Number	Name
–27584	`no_owners`
–27583	`invalid_pointer`
–27582	`invalid_seed`
–27581	`invalid_frame_seed`
–27580	`invalid_text_seed`
–27579	`invalid_draw_seed`
–27578	`bad_printer_flags`

Table 3-27 lists the QuickDraw GX path and polygon validation debugging errors.

Table 3-27 Path and polygon validation debugging errors

Number	Name
–27577	`invalid_vector_count`
–27576	`invalid_contour_count`

Table 3-28 lists the QuickDraw GX bitmap validation debugging errors.

Table 3-28 Bitmap validation debugging errors

Number	Name
–27575	`bitmap_ptr_too_small`
–27574	`bitmap_ptr_not_aligned`
–27573	`bitmap_rowBytes_negative`
–27572	`bitmap_width_negative`
–27571	`bitmap_height_negative`
–27570	`invalid_pixelSize`
–27569	`bitmap_rowBytes_too_small`
–27568	`bitmap_rowBytes_not_aligned`
–27567	`bitmap_rowBytes_must_be_specified_for_user_image_buffer`

Table 3-29 lists the QuickDraw GX bitmap image validation debugging errors.

Table 3-29 Bitmap image validation debugging errors

Number	Name
–27566	`invalid_bitImage_fileOffset`
–27565	`invalid_bitImage_owners`
–27564	`invalid_bitImage_rowBytes`
–27563	`invalid_bitImage_internal_flag`

Table 3-30 lists the QuickDraw GX text validation debugging errors.

Table 3-30 Text validation debugging errors

Number	Name
–27562	`text_bounds_cache_wrong_size`
–27561	`text_metrics_cache_wrong_size`
–27560	`text_index_cache_wrong_size`

Table 3-31 lists the QuickDraw GX glyph validation debugging errors.

Table 3-31 Glyph validation debugging errors

Number	Name
–27559	`glyph_run_count_negative`
–27558	`glyph_run_count_zero`
–27557	`glyph_run_counts_do_not_sum_to_character_count`
–27556	`glyph_first_advance_bit_set_not_allowed`
–27555	`glyph_tangent_vectors_both_zero`

Table 3-32 lists the QuickDraw GX layout validation debugging errors.

Table 3-32 Layout validation debugging errors

Number	Name
–27554	`layout_run_length_negative`
–27553	`layout_run_length_zero`
–27552	`layout_run_level_negative`
–27551	`layout_run_lengths_do_not_sum_to_text_length`

Table 3-33 lists the QuickDraw GX picture validation debugging errors.

Table 3-33 Picture validation debugging errors

Number	Name
–27550	`bad_shape_in_picture`
–27549	`bad_style_in_picture`
–27548	`bad_ink_in_picture`
–27547	`bad_transform_in_picture`
–27546	`bad_shape_cache_in_picture`
–27545	`bad_seed_in_picture`
–27544	`invalid_picture_count`

Table 3-34 lists the QuickDraw GX text face validation debugging errors.

Table 3-34 Text face validation debugging errors

Number	Name
–27543	`bad_textLayer_count`
–27542	`bad_fillType_in_textFace`
–27541	`bad_style_in_textFace`
–27540	`bad_transform_in_textFace`

Table 3-35 lists the QuickDraw GX transform validation debugging errors.

Table 3-35 Transform validation debugging errors

Number	Name
–27539	`invalid_matrix_flag`
–27538	`transform_clip_missing`

Table 3-36 lists the QuickDraw GX font cache validation debugging errors.

Table 3-36 Font cache validation debugging errors

Number	Name
–27537	`metrics_wrong_type`
–27536	`metrics_point_size_probably_bad`
–27535	`scalar_block_wrong_type`
–27534	`scalar_block_parent_mismatch`
–27533	`scalar_block_too_small`
–27532	`scalar_block_too_large`
–27531	`invalid_metrics_range`
–27530	`invalid_metrics_flags`
–27529	`metrics_maxWidth_probably_bad`
–27528	`font_wrong_type`
–27527	`font_wrong_size`
–27526	`invalid_font_platform`
–27525	`invalid_lookup_range`
–27524	`invalid_lookup_platform`
–27523	`font_not_in_font_list`
–27522	`metrics_not_in_metrics_list`

Table 3-37 lists the QuickDraw GX view device validation debugging errors.

Table 3-37 View device validation debugging errors

Number	Name
–27521	`bad_device_private_flags`
–27520	`bad_device_attributes`
–27519	`invalid_device_number`
–27518	`invalid_device_viewGroup`
–27517	`invalid_device_bounds`
–27516	`invalid_bitmap_in_device`

Table 3-38 lists the QuickDraw GX color set validation debugging errors.

Table 3-38 Color set validation debugging errors

Number	Name
–27515	`colorSet_wrong_type`
–27514	`invalid_colorSet_viewDevice_owners`
–27513	`invalid_colorSet_colorSpace`
–27512	`invalid_colorSet_count`

Table 3-39 lists the QuickDraw GX color profile validation debugging errors.

Table 3-39 Color profile validation debugging errors

Number	Name
–27511	`colorProfile_wrong_type`
–27510	`invalid_colorProfile_flags`
–27509	`invalid_colorProfile_response_count`

Table 3-40 lists the QuickDraw GX internal backing store validation debugging errors.

Table 3-40 Internal backing store validation debugging errors

Number	Name
–27508	`backing_free_parent_mismatch`
–27507	`backing_store_parent_mismatch`

Warnings

This section describes the warnings that the debugging version of QuickDraw GX may post. QuickDraw GX debugging warnings can be grouped into the following categories:

- invalid data warnings
- can’t find warnings
- other warnings

Table 3-41 gives the range of debugging warning numbers.

Table 3-41 Debugging warning number range

Number	Description
–26700	`gxFirstSystemDebuggingWarning`
–26000	`gxLastSystemWarning`

Table 3-42 lists the QuickDraw GX **invalid data** debugging warnings.

Table 3-42 Invalid data debugging warnings

Number	Name
–26700	`new_shape_contains_invalid_data`
–26699	`new_tag_contains_invalid_data`
–26698	`extra_data_passed_was_ignored`
–26697	`font_table_not_found`
–26696	`font_name_not_found`
–26695	`unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve`
–26694	`unable_to_draw_open_contour_that_starts_or_ends_off_the_curve`
–26693	`cannot_dispose_default_shape`
–26692	`cannot_dispose_default_style`
–26691	`cannot_dispose_default_ink`
–26690	`cannot_dispose_default_transform`
–26689	`cannot_dispose_default_colorProfile`
–26688	`cannot_dispose_default_colorSet`
–26687	`shape_direct_attribute_not_set`

Table 3-43 lists the QuickDraw GX can't find debugging warnings.

Table 3-43 Can't find debugging warnings

Number	Name
–26686	`point_does_not_intersect_port`
–26685	`cannot_dispose_non_font`
–26684	`face_override_style_font_must_match_style`
–26683	`union_of_area_and_and_length_returns_area_only`
–26682	`insufficient_coordinate_space_for_new_device`

Table 3-44 lists the QuickDraw GX other debugging warnings.

Table 3-44 Other debugging warnings

Number	Name
–26681	`shape_passed_has_no_bounds`
–26680	`tags_of_type_flst_removed`
–26679	`translator_not_installed_on_this_grafport`

Notices

QuickDraw GX provides notices only in the debugging version. This section describes the notices that the debugging version of QuickDraw GX may post. Each QuickDraw notice has a unique **notice number** and a **notice name.** Table 3-45 gives the debugging notice number range.

Table 3-45 Debugging version notice number summary

Number	Description
–25999	`gxFirstSystemNotice`
–25500	`gxLastSystemNotice`

Table 3-46 lists the QuickDraw GX debugging notices.

Table 3-46 Debugging notices

Number	Name
–25999	`parameters_have_no_effect`
–25998	`attributes_already_set`
–25997	`caps_already_set`
–25996	`clip_already_set`
–25995	`color_already_set`
–25994	`curve_error_already_set`
–25993	`dash_already_set`
–25992	`default_colorProfile_already_set`
–25991	`default_ink_already_set`
–25990	`default_transform_already_set`

continued

Table 3-46 Debugging notices (continued)

Number	Name
–25989	`default_shape_already_set`
–25988	`default_style_already_set`
–25987	`dither_already_set`
–25986	`encoding_already_set`
–25985	`face_already_set`
–25984	`fill_already_set`
–25983	`font_already_set`
–25982	`font_variations_already_set`
–25981	`glyph_positions_are_already_set`
–25980	`glyph_tangents_are_already_set`
–25979	`halftone_already_set`
–25978	`hit_test_already_set`
–25977	`ink_already_set`
–25976	`join_already_set`
–25975	`justification_already_set`
–25974	`mapping_already_set`
–25973	`pattern_already_set`
–25972	`pen_already_set`
–25971	`style_already_set`
–25970	`tag_already_set`
–25969	`text_attributes_already_set`
–25968	`text_size_already_set`
–25967	`transfer_already_set`
–25966	`translator_already_installed_on_this_grafport`
–25965	`transform_already_set`
–25964	`type_already_set`
–25963	`validation_level_already_set`
–25962	`viewPorts_already_set`
–25961	`viewPorts_already_in_viewGroup`
–25960	`viewDevice_already_in_viewGroup`
–25959	`geometry_unaffected`
–25958	`mapping_unaffected`

Table 3-46 Debugging notices (continued)

Number	Name
–25957	`tags_in_shape_ignored`
–25956	`shape_already_in_primitive_form`
–25955	`shape_already_in_simple_form`
–25954	`shape_already_broken`
–25953	`shape_already_joined`
–25952	`cache_already_cleared`
–25951	`shape_not_disposed`
–25950	`style_not_disposed`
–25949	`ink_not_disposed`
–25948	`transform_not_disposed`
–25947	`colorSet_not_disposed`
–25946	`colorProfile_not_disposed`
–25945	`font_not_disposed`
–25944	`glyph_tangents_have_no_effect`
–25943	`glyph_positions_determined_by_advance`
–25942	`transform_viewPorts_already_set`
–25941	`directShape_attribute_set_as_side_effect`
–25940	`lockShape_called_as_side_effect`
–25939	`lockTag_called_as_side_effect`
–25938	`shapes_unlocked_as_side_effect`
–25937	`shape_not_locked`
–25936	`tag_not_locked`
–25935	`disposed_dead_caches`
–25934	`disposed_live_caches`
–25933	`low_on_memory`
–25932	`very_low_on_memory`
–25931	`transform_references_disposed_viewPort`

Using Errors, Warnings, and Notices

This section describes how to control and utilize QuickDraw GX errors, warnings, and notices and how to include an application-defined function to provide complementary or alternative error, warning, and notice processing. This section describes how you can

- obtain the QuickDraw GX errors, warnings, and notices posted
- change the QuickDraw GX errors, warnings, and notices posted
- ignore QuickDraw GX warnings and notices
- install application-defined error, warning, and notice handlers

Obtaining Errors, Warnings, and Notices

You can use the `GXGetGraphicsError`, `GXGetGraphicsWarning`, and `GXGetGraphicsNotice` functions to obtain QuickDraw GX error, warning, and notice messages describing problems that occur during the execution of your application. These three functions return the last problem encountered during execution. If no problem has been posted, the function returns 0 until a problem message is posted.

The `stickyError`, `stickyWarning`, or `stickyNotice` parameters of the respective function, if not `nil`, are pointers to the first execution problem that QuickDraw GX encountered after the last time that the `GXGetGraphicsError`, `GXGetGraphicsWarning`, or `GXGetGraphicsNotice` function was called. These functions thereby allow you to determine both the original problem and the final problem that was detected by QuickDraw GX during execution of your application.

Note
Notices are posted only in the debugging version of QuickDraw GX. u

Figure 3-2 shows the use of these polling functions to obtain the errors, warnings, and notices of selected blocks of your code.

Figure 3-2 Polling for errors, warnings, and notices

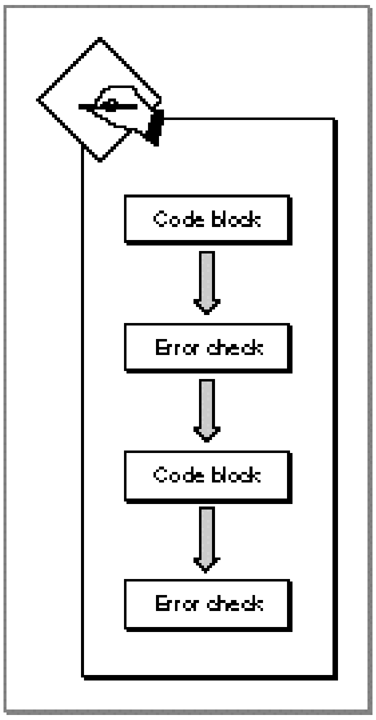
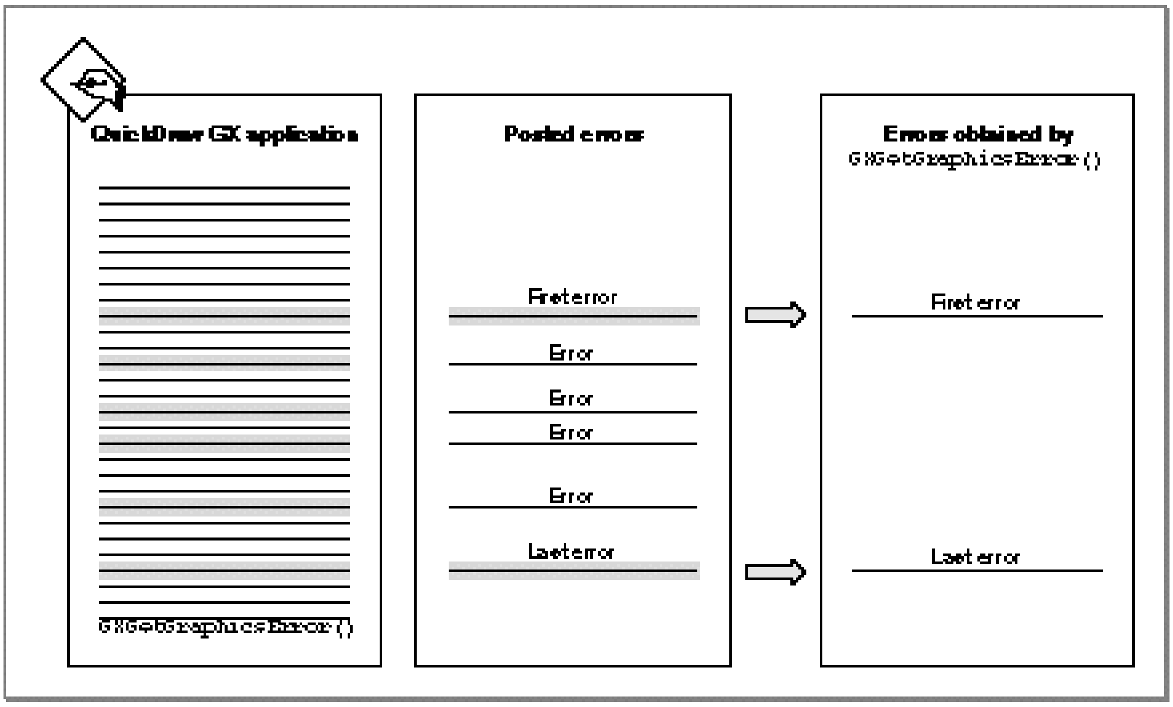


Figure 3-3 shows the use of the `GXGetGraphicsError` function to obtain the first and last errors posted when you test your QuickDraw GX application.

Figure 3-3 Obtaining the first and last posted QuickDraw GX error



Listing 3-1 shows the use of the `GXGetGraphicsError` function to obtain the first error posted after the execution of a block of code.

Listing 3-1 Obtaining the first posted error

```
static void ObtainOriginalError(void)
{

/* block of application code */

/*
If an error occurred, then see if the orginal error was
out_of_memory. Note that you need to look at the original error,
not the last error returned, since if the NewLine fails, then
the next two functions (DrawShape and DisposeShape) will
generate a shape_is_nil error.
*/

   {  graphicsError myError, originalError;
      if( myError = GetGraphicsError(&originalError) ) {
         if( originalError == out_of_memory ) {
            /* post out of memory dialog box */
         } else {
/* post generic error dialog box */
         }
      }
   }
}
```

Listing 3-2 shows the use of the `GXGetGraphicsWarning` function to obtain the first and last warning posted after the execution of a block of code.

Listing 3-2 Obtaining the first and last QuickDraw GX warning

```
static void ObtainFirstLastWarning(void)
{

/* block of application code */

/*
It might be valuable to look at both myWarning (last warning
posted) and originalWarning (first warning posted), although the
last warning is usually the most important warning posted.
*/

   {  graphicsWarning myWarning, originalWarning;
      if( myWarning = GXGetGraphicsWarning(&originalWarning) ) {
         DebugStr("\pa warning occurred");
      }
   }
}
```

Listing 3-3 shows the use of the `GXGetGraphicsNotice` function to obtain the first and last notices posted after the execution of a block of code.

Listing 3-3 Obtaining the first and last posted notices

```
static void ObtainFirstLastNotice(void)
{

/* block of application code */

/*
It might be useful to look at both myNotice (last notice
posted)and originalNotice (first notice posted), although the last
notice is usually the most important notice posted.
*/
```

```
   {  graphicsNotice myNotice, originalNotice;
      if( myNotice = GXGetGraphicsNotice(&originalNotice) ) {
         DebugStr("\pa notice occurred");
      }
   }
}
```

The `GXGetGraphicsError` function is described on page 3-56. The QuickDraw GX errors that may be posted are listed in the section “Errors” beginning on page 3-6. QuickDraw GX allows you to ignore warnings and notices, but does not provide a function that will ignore errors.

The `GXGetGraphicsWarning` function is described on page 3-60. The QuickDraw GX warnings that may be posted are listed in the section “Warnings” beginning on page 3-10. QuickDraw GX allows you to ignore warnings that would otherwise be posted. How to ignore warnings is discussed in the section “Ignoring Warnings and Notices” beginning on page 3-37. The `GXIgnoreGraphicsWarning` function is discussed on page 3-64.

The `GXGetGraphicsNotice` function is described on page 3-66. The QuickDraw GX notices that may be posted are listed in the section “Notices” beginning on page 3-27. QuickDraw GX allows you to ignore notices that would otherwise be posted. How to ignore notices is discussed in the section “Ignoring Warnings and Notices” beginning on page 3-37. The `GXIgnoreGraphicsNotice` function is discussed on page 3-70.

Note

An alternative or complementary approach to the use of the `GXGetGraphicsError`, `GXGetGraphicsWarning`, and `GXGetGraphicsNotice` functions is to include an application-defined error, warning, or notice handler. This topic is discussed in the section “Installing an Error, Warning, or Notice Handler” beginning on page 3-40. u

Changing the Error, Warning, or Notice Posted

You can use the `GXPostGraphicsError`, `GXPostGraphicsWarning`, and `GXPostGraphicsNotice` functions to post your own errors, warnings, and notices from inside your application.

Note

Notices are posted only in the debugging version of QuickDraw GX. u

The `GXPostGraphicsError` function replaces the current QuickDraw GX error with any error message you provide as the `error` parameter. The error you substitute may be one of the QuickDraw GX errors or your own error message. This function stores the new error message so that subsequent calls to `GXGetGraphicsError` return the error substituted by this function.

Listing 3-4 shows the use of the GXPostGraphicsError function to change the posted error to an error having the name special_user_error and the error number 2097152. This is the gxFirstAppError constant.

Listing 3-4 Changing the error posted

```
static long SampleCode4(void)
{
   #define special_user_error 2097152L
   #define end_of_file -1L
   long myFilePosition = 0;

/* block of application code */

   if( myFilePosition == end_of_file ) {

/* indicate that an error occurred */

      PostGraphicsError(special_user_error);
   }

   /* block of application code */

/*
You need to check for errors only once; this will catch errors
generated by QuickDraw GX and any user-defined errors that were
posted.
*/


   {  graphicsError myError;
      if( myError = GXGetGraphicsError(nil) )
         return myError;
   }

   /* block of application code */

}
```

The GXPostGraphicsError function is described on page 3-57. The QuickDraw GX errors are listed in the section "Errors" beginning on page 3-6.

The `GXPostGraphicsWarning` function replaces the current QuickDraw GX warning with any warning message you provide as the `warning` parameter. The warning you substitute may be one of the QuickDraw GX warnings or your own warning message. This function stores the new warning message so that subsequent calls to `GXGetGraphicsWarning` return the warning substituted by this function.

The `GXPostGraphicsWarning` function is described on page 3-61. The QuickDraw GX warnings are listed in the section "Warnings" beginning on page 3-10.

The `GXPostGraphicsNotice` function replaces the current QuickDraw GX notice with any notice message you provide as the `notice` parameter. The notice you substitute may be one of the QuickDraw GX notices or your own notice message. This function stores the new notice message so that subsequent calls to `GXGetGraphicsNotice` return the notice substituted by this function.

The `GXPostGraphicsNotice` function is described on page 3-67. The QuickDraw GX notices are listed in the section "Notices" beginning on page 3-27.

Ignoring Warnings and Notices

You can use the `GXIgnoreGraphicsWarning` and `GXIgnoreGraphicsNotice` functions to selectively ignore, and thereby suppress, the posting of specific QuickDraw GX warnings and notices in parts of your application. There is no analogous function to ignore errors.

Note

Notices are posted only in the debugging version of QuickDraw GX. u

The `GXIgnoreGraphicsWarning` function places the warning to be ignored on the **ignore warning stack.** The posting of all QuickDraw GX warnings that are on the ignore warning stack is suppressed, just as if the problem that resulted in the warning message never occurred.

When a QuickDraw GX warning is about to be posted, QuickDraw GX determines if the specific warning is on the ignore warning stack. If the warning to be posted is on the stack, QuickDraw GX does not post this warning. If the warning to be posted is not on the ignore warning stack, QuickDraw GX does post the warning. QuickDraw GX does not change the stack when it checks for the presence or absence of a warning.

The `GXPopGraphicsWarning` function removes warnings from the ignore warning stack in the reverse order that they are placed on the stack by the `GXIgnoreGraphicsWarning` function. You don't need to specify which warning to remove. You remove one ignored warning code from the top of the ignore warning stack each time that you call the `GXPopGraphicsWarning` function.

Note

There is an implementation limit on the number of times that you can use the `GXIgnoreGraphicsWarning` and `GXPopGraphicsWarning` functions. When the implementation limit is exceeded, QuickDraw GX posts a `warning_stack_overflow` warning message. If there are no warnings on the ignore warning stack and the `GXPopGraphicsWarning` function is called, QuickDraw GX posts a `warning_stack_underflow` warning message. u

Since there is an implementation limit on the number of warnings and notices that you can ignore, you should use the `GXIgnoreGraphicsWarning` and `GXPopGraphicsWarning` functions only when you need to debug specific parts of your application code.

The `GXIgnoreGraphicsNotice` function provides the same feature for notices that the `GXIgnoreGraphicsWarning` function provides for warnings.

The `GXIgnoreGraphicsNotice` function places the notice to be ignored on the **ignore notice stack.** The posting of all QuickDraw GX notices on the ignore notice stack is suppressed, just as if the problem that resulted in the notice message never occurred.

When a QuickDraw GX notice is about to be posted, QuickDraw GX determines if the specific notice is on the ignore notice stack. If the notice to be posted is on the stack, QuickDraw GX does not post this notice. If the notice to be posted is not on the ignore notice stack, QuickDraw GX does post it. QuickDraw GX does not change the stack when it checks for the presence or absence of a notice.

The `GXPopGraphicsNotice` function removes notices from the ignore notice stack in the reverse order that they are placed on the stack by the `GXIgnoreGraphicsNotice` function. You don't need to specify which notice to remove. You remove one ignored notice code from the top of the ignore notice stack each time you call the `GXPopGraphicsNotice` function.

Note

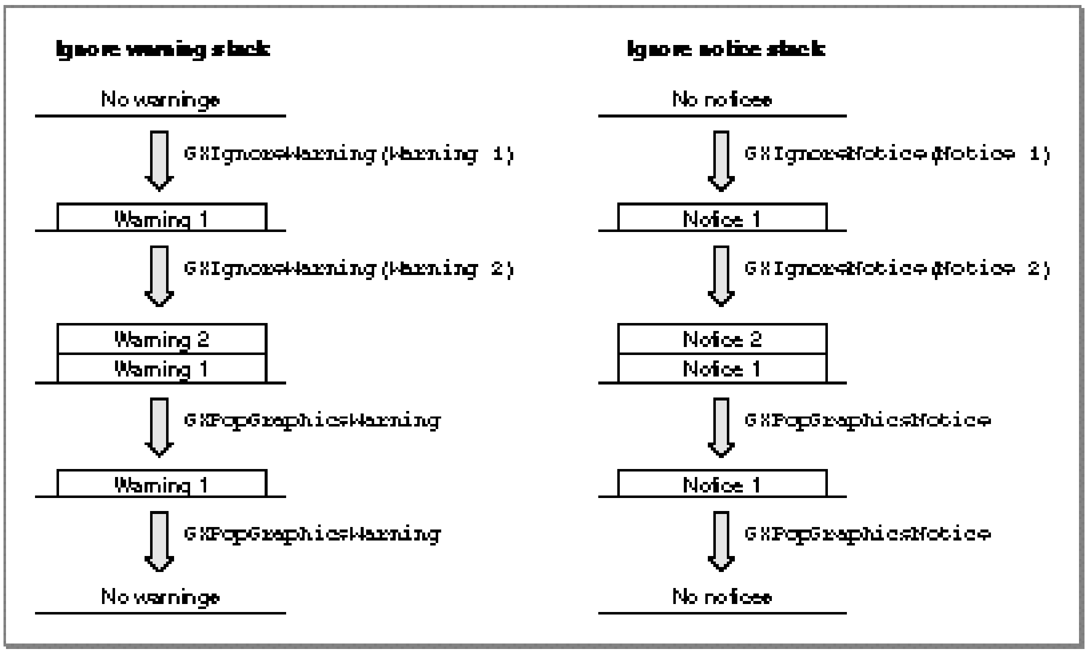
There is an implementation limit on the number of times that you can use the `GXIgnoreGraphicsNotice` and `GXPopGraphicsNotice` functions. When the implementation limit is exceeded, QuickDraw GX will post a `notice_stack_overflow` warning message. If there are no notices on the notice warning stack and the `GXPopGraphicsNotice` function is called, QuickDraw GX posts a `notice_stack_underflow` warning message. u

For example, if you wanted to suppress the `attributes_already_set` notice posted by QuickDraw GX, you could use the `GXIgnoreGraphicsNotice` function to push its notice number, –25998, onto the ignore notice stack. When QuickDraw GX is about to post a notice, it looks on the ignore notice stack to determine if its notice number is on the ignore notice stack. If the notice to be posted is `attributes_already_set`, then the notice is not posted. QuickDraw GX posts any notice that is not on the ignore notice stack.

If you also wanted to ignore the `color_already_set` notice, then you could use the `GXIgnoreGraphicsNotice` function to push its notice number, –25995, onto the ignore notice stack. QuickDraw GX would then ignore, and therefore not post, the `attributes_already_set` and `color_already_set` notices. Since you added the notices to the ignore notice stack in the order `attributes_already_set` and then `color_already_set`, the `color_already_set` notice would be on top of the ignore notice stack. When you use the `GXPopGraphicsNotice` function to remove a notice from the stack, the first notice to be removed is `color_already_set`, the one on top of the ignore notice stack. To remove the `attributes_already_set` notice, you need to call the `GXPopGraphicsNotice` function a second time. After the second call to the `GXPopGraphicsNotice` function, no notices are on the ignore notice stack. As a result, QuickDraw GX resumes posting all notices.

Figure 3-4 illustrates how warnings and notices are added to and removed from the ignore warning stack and the ignore notice stack.

Figure 3-4 Adding and removing warnings and notices from the ignore warning and ignore notice stacks



You should ignore warnings and notices only if you are confident that you understand why they are being issued and the consequences of ignoring these warnings and notices.

For example, if your program asks for 100 points in a polygon and there are fewer points available, QuickDraw GX posts a warning and returns all of the points that are available. You can add the `GXIgnoreGraphicsNotice` function to your code to suppress this warning, but your application needs to be smart enough to accommodate the fact that less than the requested number of points may be returned.

The `GXIgnoreGraphicsWarning` function is discussed on page 3-64. The `GXPopGraphicsWarning` function is discussed on page 3-65. The QuickDraw warning names and numbers that may be ignored are listed in the section "Warnings" beginning on page 3-10.

The `GXIgnoreGraphicsNotice` function is discussed on page 3-70. The `GXPopGraphicsNotice` function is discussed on page 3-71. The QuickDraw GX warning names and numbers that can be ignored are listed in the section "Notices" beginning on page 3-27.

Installing an Error, Warning, or Notice Handler

You can use the `GXSetUserGraphicsError`, `GXSetUserGraphicsWarning`, and `GXSetUserGraphicsNotice` functions to install an application-defined function that you want to call whenever an error, warning, or notice occurs. QuickDraw GX will pass this function the error, warning, or notice when it is generated. Your function can then respond accordingly. You may use the `reference` argument to pass an associated `long` value parameter to your function. If you want to disable your handler, you just pass `nil`.

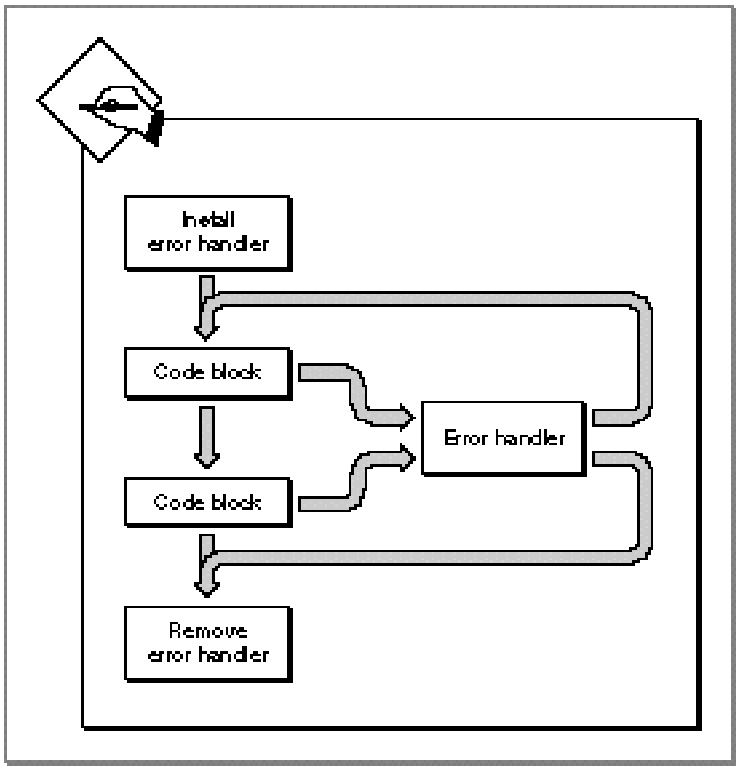
Your application can then take advantage of these errors. For example, QuickDraw GX may post an error indicating that your application has run out of memory or has tried to use a font that is not installed. As a result, your application may be able to recommend corrective action via the application-defined error handling function and the application's human interface.

You can use the `GXGetUserGraphicsError`, `GXGetUserGraphicsWarning`, and `GXGetUserGraphicsNotice` functions to obtain the application-defined handler functions that have been previously installed by `GXSetUserGraphicsError`, `GXSetUserGraphicsWarning`, and `SetUserGraphicsNotices`. These functions return `nil` if no function has been installed.

You usually install handlers at the beginning of your application code. You can install error, warning, and notice handlers before any graphics operations have occurred and before the `GXEnterGraphics` function has been called. If you do, QuickDraw GX will call the `GXEnterGraphics` function for you. In contrast, you can't install error, warning, and notice handlers before calling the `GXNewGraphicsClient` function.

Alternatively, you may selectively enable and disable error, warning, and notice handlers at different sections of the application code. Figure 3-5 shows how an error handler can be enabled and disabled within the application. This is an effective method for ignoring errors, warnings, and notices, analogous to the use of the `GXIgnoreGraphicsError`, `GXIgnoreGraphicsWarning`, and `GXIgnoreGraphicsNotice` functions.

Figure 3-5 Enabling and disabling an error handler



The handler should respond to the problems that occur during typical application scenarios. A friendly application should let the user know when it is taking action in response to errors and warnings that have occurred. For example, if an application runs out of memory, it may let the user know that it is out of memory and that it is responding in a particular manner to alleviate the problem. If it cannot solve the problem, it may need to notify the user that it needs to abort processing. Such an application would need to install an error handler that looks for `out_of_memory` errors.

In general, in the non-debugging version of your application, the handler might be relatively simple. If the handler doesn't have a response to an error or warning, it should just return and continue execution.

In contrast, the debugging version of the handler may be relatively complex to accommodate special error, warning, and notice conditions. In general, you should stop and print the errors, warnings, and notices whenever a problem occurs.

An application can have more than one error handler. A simple application might have just one error handler to handle specific problems. However, a more complicated application may have multiple error handlers. For example, an application might have one error handler that takes care of memory problems and another error handler for other types of errors. The special error handler may be installed only when a particular type of processing is to occur, like animation or QuickTime movies.

Errors, Warnings, and Notices Reference

This section provides reference information related to the data types and functions that allow you to control the generation of errors, warnings, and notices.

Constants and Data Types

This section describes the error, warning, and notice data types that you may use in your application.

Errors

QuickDraw GX provides you with an extended set of errors in the debugging version and a reduced set of errors in the non-debugging version. Each QuickDraw GX error constant has an error number described by the `gxGraphicsError` type definition and the `gxGraphicErrors` enumeration:

```
typedef long gxGraphicsError;

enum gxGraphicErrors {

   /* truly fatal errors */
   out_of_memory = -27999,
   internal_fatal_error,
   no_outline_font_found,
   not_enough_memory_for_graphics_client_heap,
   could_not_create_backing_store,

   /* internal errors */
   internal_error = -27950,
   internal_font_error,
   internal_layout_error,

   /* recoverable errors */
   could_not_dispose_backing_store = internal_layout_error + 2,
   unflattening_interrupted_by_client,

   /* font manager errors */
   font_cannot_be_changed,
   illegal_font_parameter,
/* gxFont scaler errors */
null_font_scaler_context = -27900,
null_font_scaler_input,
invalid_font_scaler_context,
invalid_font_scaler_input,
invalid_font_scaler_font_data,
font_scaler_newblock_failed,
font_scaler_getfonttable_failed,
font_scaler_bitmap_allocation_failed,
font_scaler_outline_allocation_failed,
required_font_scaler_table_missing,
unsupported_font_scaler_outline_format,
unsupported_font_scaler_stream_format,
unsupported_font_scaler_font_format,
font_scaler_hinting_error,
font_scaler_rasterizer_error,
font_scaler_internal_error,
font_scaler_invalid_matrix,
font_scaler_fixed_overflow,
font_scaler_api_version_mismatch,
font_scaler_streaming_aborted,
unknown_font_scaler_error,

/* bad parameters */
parameter_is_nil = -27850,
shape_is_nil,
style_is_nil,
transform_is_nil,
ink_is_nil,
transferMode_is_nil,
color_is_nil,
colorProfile_is_nil,
colorSet_is_nil,
spoolProcedure_is_nil,
tag_is_nil,
type_is_nil,
mapping_is_nil,
invalid_viewDevice_reference,
invalid_viewGroup_reference,
invalid_viewPort_reference,

/* implementation limits */
number_of_contours_exceeds_implementation_limit = -27800,
number_of_points_exceeds_implementation_limit,
size_of_polygon_exceeds_implementation_limit,
size_of_path_exceeds_implementation_limit,
size_of_text_exceeds_implementation_limit,
size_of_bitmap_exceeds_implementation_limit,
number_of_colors_exceeds_implementation_limit,
procedure_not_reentrant

#ifdef debugging
,
/* internal debugging errors: following available only in */
/* the debugging init */
functionality_unimplemented = -27700,
clip_to_frame_shape_unimplemented,

/* font parameter debugging errors */
illegal_font_storage_type,
illegal_font_storage_reference,
illegal_font_attributes,

/* parameter debugging errors */
parameter_out_of_range,
inconsistent_parameters,
index_is_less_than_zero,
index_is_less_than_one,
count_is_less_than_zero,
count_is_less_than_one,
contour_is_less_than_zero,
length_is_less_than_zero,

invalid_client_reference,
invalid_graphics_heap_start_pointer,
invalid_nongraphic_globals_pointer,

colorSpace_out_of_range,

pattern_lattice_out_of_range,
frequency_parameter_out_of_range,
tinting_parameter_out_of_range,
method_parameter_out_of_range,
space_may_not_be_indexed,
   glyph_index_too_small,
   no_glyphs_added_to_font,
   glyph_not_added_to_font,
   point_does_not_intersect_bitmap,

   required_font_table_not_present,
   unknown_font_table_format,

/* the styles encoding is not present in the font */
   shapeFill_not_allowed,
   inverseFill_face_must_set_clipLayer_flag,
   invalid_transferMode_colorSpace,
   colorProfile_must_be_nil,
   bitmap_pixel_size_must_be_1,
   empty_shape_not_allowed,
   ignorePlatformShape_not_allowed,
   nil_style_in_glyph_not_allowed,
   complex_glyph_style_not_allowed,
   invalid_mapping,

   cannot_set_item_shapes_to_nil,
   cannot_use_original_item_shapes_when_growing_picture,
   cannot_add_unspecified_new_glyphs,
   cannot_dispose_locked_tag,
   cannot_dispose_locked_shape,

   /* restricted access */
   shape_access_not_allowed,
   colorSet_access_restricted,
   colorProfile_access_restricted,
   tag_access_restricted,
   viewDevice_access_restricted,

   graphic_type_does_not_have_a_structure,
   style_run_array_does_not_match_number_of_characters,
   rectangles_cannot_be_inserted_into,

   unknown_graphics_heap,
   graphics_routine_selector_is_obsolete,
   cannot_set_graphics_client_memory_without_setting_size,
   graphics_client_memory_too_small,
   graphics_client_memory_is_already_allocated,
   viewPort_is_a_window,

   /* wrong type/bad reference */
   illegal_type_for_shape,
   shape_does_not_contain_a_bitmap,
   shape_does_not_contain_text,
   picture_expected,
   bitmap_is_not_resizable,
   shape_may_not_be_a_bitmap,
   shape_may_not_be_a_picture,
   graphic_type_does_not_contain_points,
   graphic_type_does_not_have_multiple_contours,
   graphic_type_cannot_be_mapped,
   graphic_type_cannot_be_moved,
   graphic_type_cannot_be_scaled,
   graphic_type_cannot_be_rotated,
   graphic_type_cannot_be_skewed,
   graphic_type_cannot_be_reset,
   graphic_type_cannot_be_dashed,
   graphic_type_cannot_be_reduced,
   graphic_type_cannot_be_inset,
   shape_cannot_be_inverted,
   shape_does_not_have_area,
   shape_does_not_have_length,
   first_glyph_advance_must_be_absolute,
   picture_cannot_contain_itself,
   viewPort_cannot_contain_itself,

   cannot_set_unique_items_attribute_when_picture_
   contains_items,
   layer_style_cannot_contain_a_face,
   layer_glyph_shape_cannot_contain_nil_styles,

      /* validation errors */
   object_wrong_type,
   shape_wrong_type,
   style_wrong_type,
   ink_wrong_type,
   transform_wrong_type,
   device_wrong_type,
   port_wrong_type,
/*cache validation errors */
shape_cache_wrong_type,
style_cache_wrong_type,
ink_cache_wrong_type,
transform_cache_wrong_type,
port_cache_wrong_type,
shape_cache_parent_mismatch,
style_cache_parent_mismatch,
ink_cache_parent_mismatch,
transform_cache_parent_mismatch,
port_cache_parent_mismatch,
invalid_shape_cache_port,
invalid_shape_cache_device,
invalid_ink_cache_port,
invalid_ink_cache_device,
invalid_style_cache_port,
invalid_style_cache_device,
invalid_transform_cache_port,
invalid_transform_cache_device,
recursive_caches,

/*shape cache validation errors */
invalid_fillShape_ownerCount,
recursive_fillShapes,

/*memory block validation errors */
indirect_memory_block_too_small,
indirect_memory_block_too_large,
unexpected_nil_pointer,
bad_address,

/* object validation errors */
no_owners,
invalid_pointer,
invalid_seed,
invalid_frame_seed,
invalid_text_seed,
invalid_draw_seed,
bad_private_flags,

/* path and polygon validation errors */
invalid_vector_count,
invalid_contour_count,
/* validation bitmap errors */
bitmap_ptr_too_small,
bitmap_ptr_not_aligned,
bitmap_rowBytes_negative,
bitmap_width_negative,
bitmap_height_negative,
invalid_pixelSize,
bitmap_rowBytes_too_small,
bitmap_rowBytes_not_aligned,
bitmap_rowBytes_must_be_specified_for_user_image_buffer,

/* bitmap validation image errors */
invalid_bitImage_fileOffset,
invalid_bitImage_owners,
invalid_bitImage_rowBytes,
invalid_bitImage_internal_flag,

/* text validation errors */
text_bounds_cache_wrong_size,
text_metrics_cache_wrong_size,
text_index_cache_wrong_size,

/* glyph validation errors */
glyph_run_count_negative,
glyph_run_count_zero,
glyph_run_counts_do_not_sum_to_character_count,
glyph_first_advance_bit_set_not_allowed,
glyph_tangent_vectors_both_zero,

/* layout validation errors */
layout_run_length_negative,
layout_run_length_zero,
layout_run_level_negative,
layout_run_lengths_do_not_sum_to_text_length,

/* picture validation errors */
bad_shape_in_picture,
bad_style_in_picture,
bad_ink_in_picture,
bad_transform_in_picture,
bad_shape_cache_in_picture,
bad_seed_in_picture,
invalid_picture_count,
/* text face validation errors */
bad_textLayer_count,
bad_fillType_in_textFace,
bad_style_in_textFace,
bad_transform_in_textFace,

/* transform validation errors */
invalid_matrix_flag,
transform_clip_missing,

/* font cache validation errors */
metrics_wrong_type,
metrics_point_size_probably_bad,
scalar_block_wrong_type,
scalar_block_parent_mismatch,
scalar_block_too_small,
scalar_block_too_large,
invalid_metrics_range,
invalid_metrics_flags,
metrics_maxWidth_probably_bad,
font_wrong_type,
font_wrong_size,
invalid_font_platform,
invalid_lookup_range,
invalid_lookup_platform,
font_not_in_font_list,
metrics_not_in_metrics_list,

/* view device validation errors */
bad_device_private_flags,
bad_device_attributes,
invalid_device_number,
invalid_device_viewGroup,
invalid_device_bounds,
invalid_bitmap_in_device,
/* color set validation errors */
colorSet_wrong_type,
invalid_colorSet_viewDevice_owners,
invalid_colorSet_colorSpace,
invalid_colorSet_count,
   /* color profile validation errors */
   colorProfile_wrong_type,
   invalid_colorProfile_flags,
   invalid_colorProfile_response_count,

   /* internal backing store validation errors */
   backing_free_parent_mismatch,
   backing_store_parent_mismatch
   #endif
   };
```

QuickDraw GX non-debugging errors are listed in the section “Errors” beginning on page 3-6. Debugging errors are listed in the section “Errors” beginning on page 3-6.

Warnings

QuickDraw GX provides you with an extended set of warnings in the debugging version and a reduced set of warnings in the non-debugging version. Each QuickDraw GX warning has a warning number described by the `gxGraphicsWarning` type definition and the `gxGraphicWarnings` enumeration:

```
typedef long gxGraphicsWarning;

enum gxGraphicWarnings {

   /* warnings about warnings */
   warning_stack_underflow = -26999,
   warning_stack_overflow,
   notice_stack_underflow,
   notice_stack_overflow,
   about_to_grow_heap,
   about_to_unload_objects,

   /* result went out of range */
   map_shape_out_of_range = -26950,
   move_shape_out_of_range,
   scale_shape_out_of_range,
   rotate_shape_out_of_range,
   skew_shape_out_of_range,
   map_transform_out_of_range,
   move_transform_out_of_range,
   scale_transform_out_of_range,
   rotate_transform_out_of_range,
   skew_transform_out_of_range,
   map_points_out_of_range,
   /* gave a parameter out of range */
   contour_out_of_range = -26900,
   index_out_of_range_in_contour,
   picture_index_out_of_range,
   color_index_requested_not_found,
   colorSet_index_out_of_range,
   index_out_of_range,
   count_out_of_range,
   length_out_of_range,
   font_table_index_out_of_range,
   font_glyph_index_out_of_range,
   point_out_of_range,
   profile_response_out_of_range,

      /* gxFont scaler warnings */
   font_scaler_no_output = -26850,
   font_scaler_fake_metrics,
   font_scaler_fake_linespacing,
   font_scaler_glyph_substitution,
   font_scaler_no_kerning_applied,

   /* might not be what you expected */
   character_substitution_took_place,
   unable_to_get_bounds_on_multiple_devices,
   font_language_not_found,
   font_not_found_during_unflattening,

   /*storage */
   unrecognized_stream_version,
   bad_data_in_stream

#ifdef debugging
   /*available only in debugging init */
   ,
   /* nonsense data */
   new_shape_contains_invalid_data = -26700,
   new_tag_contains_invalid_data,
   extra_data_passed_was_ignored,
   font_table_not_found,
   font_name_not_found,
/* doesn't make sense to do */
unable_to_traverse_open_contour_that_starts_or_
   ends_off_the_curve,
unable_to_draw_open_contour_that_starts_or_ends_
   off_the_curve,
cannot_dispose_default_shape,
cannot_dispose_default_style,
cannot_dispose_default_ink,
cannot_dispose_default_transform,
cannot_dispose_default_colorProfile,
cannot_dispose_default_colorSet,
shape_direct_attribute_not_set,

/* couldn't find what you were looking for */
point_does_not_intersect_port,
cannot_dispose_non_font,
face_override_style_font_must_match_style,
union_of_area_and_length_returns_area_only,
insufficient_coordinate_space_for_new_device,

/* other */
shape_passed_has_no_bounds,
tags_of_type_flst_removed,
translator_not_installed_on_this_grafport
#endif
};
```

Non-debugging warnings are listed in the section “Warnings” beginning on page 3-10. Debugging warnings are listed in the section “Warnings” beginning on page 3-10.

Notices

QuickDraw GX provides you with a set of notices in the debugging version, but no notices in the non-debugging version. Each QuickDraw GX notice has a notice number described by the `gxGraphicsNotice` type definition and the `gxGraphicNotices` enumeration:

```
typedef long gxGraphicsNotice;

#ifdef debugging
enum gxGraphicNotices {
   parameters_have_no_effect = -25999,
   attributes_already_set,
   caps_already_set,
   clip_already_set,
   color_already_set,
   curve_error_already_set,
   dash_already_set,
   default_colorProfile_already_set,
   default_ink_already_set,
   default_transform_already_set,
   default_shape_already_set,
   default_style_already_set,
   dither_already_set,
   encoding_already_set,
   face_already_set,
   fill_already_set,
   font_already_set,
   font_variations_already_set,
   glyph_positions_are_already_set,
   glyph_tangents_are_already_set,
   halftone_already_set,
   hit_test_already_set,
   ink_already_set,
   join_already_set,
   justification_already_set,
   mapping_already_set,
   pattern_already_set,
   pen_already_set,
   style_already_set,
   tag_already_set,
   text_attributes_already_set,
   text_size_already_set,
   transfer_already_set,
    translator_already_installed_on_this_grafport,
    transform_already_set,
    type_already_set,
    validation_level_already_set,
    viewPorts_already_set,
    viewPort_already_in_viewGroup,
    viewDevice_already_in_viewGroup,
    geometry_unaffected,
    mapping_unaffected,
    tags_in_shape_ignored,
    shape_already_in_primitive_form,
    shape_already_in_simple_form,
    shape_already_broken,
    shape_already_joined,
    cache_already_cleared,
    shape_not_disposed,
    style_not_disposed,
    ink_not_disposed,
    transform_not_disposed,
    colorSet_not_disposed,
    colorProfile_not_disposed,
    font_not_disposed,
    glyph_tangents_have_no_effect,
    glyph_positions_determined_by_advance,
    transform_viewPorts_already_set,
    directShape_attribute_set_as_side_effect,
    lockShape_called_as_side_effect,
    lockTag_called_as_side_effect,
    shapes_unlocked_as_side_effect,
    shape_not_locked,
    tag_not_locked,
    disposed_dead_caches,
    disposed_live_caches,
    low_on_memory,
    very_low_on_memory
    transform_references_disposed_viewPort
};
```

Debugging notices are listed in the section “Notices” beginning on page 3-27.

Error, Warning, and Notice Number Ranges

QuickDraw GX specifies the defined ranges of error, warning, and notice numbers. The `gxFirstAppError`, `gxLastAppError`, `gxFirstAppWarning`, `gxLastAppWarning`, `gxFirstAppNotice`, and `gxLastAppNotice` types define the allowable ranges for application-defined errors, warnings, and notices.

```
#define gxFirstSystemError                  –27999
#define gxFirstFatalError                   –27999
#define gxLastFatalError                    –27951
#define gxFirstNonfatalError                –27950
#define gxFirstFontScalerError              –27900
#define gxLastFontScalerError               –27851
#define gxFirstParameterError               –27850
#define gxFirstImplementationLimitError     –27800
#define gxFirstSystemDebuggingError         –27700
#define gxLastSystemError                   –27000
#define gxFirstAppError                     2097152
#define gxLastAppError                      4194303

#define gxFirstSystemWarning                –26999
#define gxFirstResultOutOfRangeWarning      –26950
#define gxFirstParameterOutOfRangeWarning –26900
#define gxFirstFontScalerWarning            –26850
#define gxFirstSystemDebuggingWarning       –26700
#define gxLastSystemWarning                 –26000
#define gxFirstAppWarning                   5242880
#define gxLastAppWarning                    7340031

#define gxFirstSystemNotice                 –25999
#define gxLastSystemNotice                  –25500
#define gxFirstAppNotice                    7602146
#define gxLastAppNotice                     8388607
```

Functions

This section describes the QuickDrawGX functions you can use to control the generation of errors, warnings, and notices.

Error Posting and Handling

This section describes the QuickDraw GX functions you can use to

- obtain the first and last QuickDraw GX errors posted
- replace the current error name with another error name
- install the application-defined error handler function
- obtain the installed application-defined error handler function

GXGetGraphicsError

You can use the `GXGetGraphicsError` function to obtain the first and last QuickDraw GX errors posted.

```
gxGraphicsError GXGetGraphicsError(gxGraphicsError *stickyError);
```

`stickyError`
On return, a pointer to the first error posted.

function result The last error posted.

DESCRIPTION

The `GXGetGraphicsError` function returns the last error posted, or 0 if no error has been posted. This function clears the last error so that all calls to this function return 0 until an error is posted.

The `stickyError` parameter, if not `nil`, is a pointer to the first error posted since the last call to the `GXGetGraphicsError` function. QuickDraw GX clears the `stickyError` parameter at the end of every call to the `GXGetGraphicsError` function.

SEE ALSO

The use of this function is described in the section “Obtaining Errors, Warnings, and Notices” beginning on page 3-30. Non-debugging errors that may be posted are listed in the section “Errors” beginning on page 3-6. Debugging errors that may be posted are listed in the section “Errors” beginning on page 3-6.

An alternative method of posting errors is to include an application-defined error handler. This topic is described in the section “Installing an Error, Warning, or Notice Handler” beginning on page 3-40.

The `GXSetUserGraphicsError` function is used to install the error handler and is described on page 3-58.

GXPostGraphicsError

You can use the `GXPostGraphicsError` function to replace the current QuickDraw GX error with another error.

```
void GXPostGraphicsError(gxGraphicsError error);
```

`error`	The error to be posted.

DESCRIPTION

The `GXPostGraphicsError` function replaces the QuickDraw GX error about to be posted with an error message defined by the `error` parameter. You may use the QuickDraw GX errors or define your own error number and error name. This function stores the error posted so that subsequent calls to the `GXGetGraphicsError` function return the error substituted by this function.

The `GXPostGraphicsError` function is available only when the debugging version is installed.

SPECIAL CONSIDERATIONS

The error number must be within the range defined by QuickDraw GX. This range is bounded by error numbers –27999 through –27000, or is in the application range.

SEE ALSO

The use of this function is described in the section “Changing the Error, Warning, or Notice Posted” beginning on page 3-35.

Non-debugging errors that can be replaced are listed in the section “Errors” beginning on page 3-6. Non-debugging errors that can be replaced are listed in the section “Errors” beginning on page 3-6.

GXSetUserGraphicsError

You can use the GXSetUserGraphicsError function to install an error handling function.

```
void GXSetUserGraphicsError(gxUserErrorFunction userFunction,
                           long reference);
```

userFunction	The application's error handling function that is to be passed the error code.
reference	A long value that is passed each time an error occurs. This value can be used by the application for any purpose.

DESCRIPTION

The GXSetUserGraphicsError function installs an application-defined error handling function. This function installs a function pointer that is called whenever an error is posted. Setting the userFunction parameter to nil removes the error handling function.

The userFunction parameter points to an application-defined error handler defined by the following type:

```
typedef void (*gxUserErrorProcPtr)(gxGraphicsError status,
               long reference);

typedef gxUserErrorProcPtr gxUserErrorFunction;
```

The second parameter is the long reference number. Whenever the application posts an error, the installed error handling function is called with the error number. The reference number is passed to the GXSetUserGraphicsError function.

You can install an error handler before calling the GXEnterGraphics function, but you should call the GXNewGraphicsClient function first. If you don't, GXNewGraphicsClient will be called for you.

SPECIAL CONSIDERATIONS

If the error number posted by the application is within the QuickDraw GX range of fatal errors, execution continues with undefined results. The fatal error range is bounded by error numbers –27999 and –27951.

If the error number posted by the application is within the QuickDraw GX range of nonfatal errors, execution continues, but results may be other than that expected. The nonfatal error range is bounded by error numbers –27950 and –27000.

SEE ALSO

The use of this function is described in the section “Installing an Error, Warning, or Notice Handler” beginning on page 3-40.

The `GXGetUserGraphicsError` function used to return a pointer to the application-defined error-handling function is described in the next section.

An alternative method of posting errors is to use the QuickDraw GX error messages. This topic is discussed in the section “Obtaining Errors, Warnings, and Notices” beginning on page 3-30.

The `GXGetGraphicsError` function described on page 3-56 is used to obtain the first and last QuickDraw GX errors posted.

The application-defined error handler is described on page 3-72.

GXGetUserGraphicsError

You can use the `GXGetUserGraphicsError` function to obtain the currently installed application-defined error handler.

```
gxUserErrorFunction GXGetUserGraphicsError(long *reference);
```

`reference`	A pointer to a `long` value that gets called each time an error occurs. This value can be used by the application for any purpose.
function result	A pointer to the installed application-defined error handler function.

DESCRIPTION

The `GXGetUserGraphicsError` function returns a pointer to the function that the application uses to handle errors. The function returns `nil` if no application-defined error handler is provided.

If an error-handling function is installed and the `reference` parameter is not `nil`, then the `reference` parameter passed to the `GXSetUserGraphicsError` function is returned.

SEE ALSO

The use of this function is described in the section “Installing an Error, Warning, or Notice Handler” beginning on page 3-40.

The `GXSetUserGraphicsError` function used to install the error handler is described in the previous section.

An alternative method to the use of an application-defined error handler is the use of the QuickDraw GX error set.

The `GXGetGraphicsError` function, described in the section "Obtaining Errors, Warnings, and Notices" beginning on page 3-30, returns the first and last QuickDraw GX errors that have been posted.

Warning Posting and Handling

This section describes the QuickDraw GX functions you can use to

- obtain the first and last QuickDraw GX warnings posted
- replace the current error name with another error name
- install the application-defined warning handler function
- obtain the installed application-defined warning handler function
- add a warning to the ignore warning stack
- remove the last warning to be added to the warning stack

GXGetGraphicsWarning

You can use the `GXGetGraphicsWarning` function to obtain the first and last warning posted.

```
gxGraphicsWarning GXGetGraphicsWarning
                        (gxGraphicsWarning *stickyWarning);
```

`stickyWarning`
On return, a pointer to the first warning posted.

function result The last warning posted.

DESCRIPTION

The `GXGetGraphicsWarning` function returns the last warning posted, or 0 if none.

The `stickyWarning` parameter, if not `nil`, receives the first warning posted since the last call to the `GXGetGraphicsWarning` function. QuickDraw GX clears the `stickyWarning` parameter at the end of every call to the `GXGetGraphicsWarning` function.

SEE ALSO

The use of this function is described in the section“Obtaining Errors, Warnings, and Notices” beginning on page 3-30.

QuickDraw GX non-debugging warnings that may be posted are listed in the section “Warnings” beginning on page 3-10. Debugging warnings are listed in the section “Warnings” beginning on page 3-10.

An alternative method of posting warnings is to include an application-defined warning handler. This topic is described in the section “Changing the Error, Warning, or Notice Posted” beginning on page 3-35.

The `GXSetUserGraphicsWarning` function is used to install the warning handler and is described on page 3-62.

GXPostGraphicsWarning

You can use the `GXPostGraphicsWarning` function to post your own warnings from your application.

```
void GXPostGraphicsWarning(gxGraphicsWarning warning);
```

`warning`	The warning to be posted.

DESCRIPTION

The `GXPostGraphicsWarning` function replaces the QuickDraw GX warning about to be posted with a warning message defined by the `warning` parameter.

You may use the QuickDraw GX warnings or define your own warning number and warning name. This function stores the warning posted so that subsequent calls to the `GXGetGraphicsWarning` function return the warning substituted by this function.

If the warning to be posted is in the ignore warning stack, the warning is not posted and execution continues.

If an application-defined warning handler is provided, the warning is passed to the warning handler.

SPECIAL CONSIDERATIONS

The warning number must be within the range defined by QuickDraw GX. This range is bounded by warning numbers –26999 through –26000 or is in an application range.

SEE ALSO

The use of this function is described in the section “Changing the Error, Warning, or Notice Posted” beginning on page 3-35.

QuickDraw GX non-debugging warnings that may be replaced are listed in the section “Warnings” beginning on page 3-10. Debugging warnings are listed in the section “Warnings” beginning on page 3-10.

Ignoring warnings is discussed in the section “Ignoring Warnings and Notices” beginning on page 3-37.

GXSetUserGraphicsWarning

You can use the `GXSetUserGraphicsWarning` function to install an application-defined warning handler.

```
void GXSetUserGraphicsWarning(gxUserWarningFunction userFunction,
                              long reference);
```

`userFunction`
The application’s warning function that is to be passed the warning code.

`reference` A `long` value that gets called each time a warning occurs. This value can be used by the application for any purpose.

DESCRIPTION

The `GXSetUserGraphicsWarning` function installs an application-defined warning handler. This function installs a function pointer that is called whenever a warning is posted. Setting the `userFunction` parameter to `nil` removes the error function.

The `userFunction` parameter points to an application-defined warning handler defined by the following type:

```
typedef void (*gxUserWarningProcPtr)(gxGraphicsWarning status,
               long refcon)

typedef gxUserWarningProcPtr gxUserWarningFunction;
```

The second parameter is the `long` reference parameter. Whenever a warning is posted by the application, the installed warning handler is called with the warning number. The reference number is passed to the `GXSetUserGraphicsError` function.

You can install a warning handler before calling the `GXEnterGraphics` function , but you should call the `GXNewGraphicsClient` function first. If you don’t, `GXNewGraphicsClient` will be called for you.

SEE ALSO

The use of this function is described in the section "Installing an Error, Warning, or Notice Handler" beginning on page 3-40.

The `GXGetUserGraphicsWarning` function described in the next section is used to return a pointer to the application-defined warning handler.

An alternative method of posting warnings is to use the QuickDraw GX warning messages. This topic is discussed in the section "Obtaining Errors, Warnings, and Notices" beginning on page 3-30.

The `GXGetGraphicsError` function described on page 3-56 is used to obtain the first and last QuickDraw GX errors posted.

The application-defined warning handler is described on page 3-73.

GXGetUserGraphicsWarning

You can use the `GXGetUserGraphicsWarning` function to obtain the currently installed application-defined warning handler.

```
gxUserWarningFunction GXGetUserGraphicsWarning(long *reference);
```

`reference`	A `long` value that gets called each time a warning occurs. This value can be used by your application for any purpose.
function result	A pointer to the installed application-defined warning handler.

DESCRIPTION

The `GXGetUserGraphicsWarning` function returns a pointer to the function that the application uses to handle warnings. The function returns `nil` if no application-defined warning handler is provided.

If a warning handler is installed and the `reference` parameter is not `nil`, then the `reference` parameter passed to the `GXSetUserGraphicsWarning` function is returned.

SEE ALSO

The use of this function is described in the section "Installing an Error, Warning, or Notice Handler" beginning on page 3-40.

The `GXSetUserGraphicsWarning` function used to install the warning handler is described in the previous section.

An alternative method to the use of an application-defined warning handler is the use of the QuickDraw GX warnings.

The `GXGetGraphicsWarning` function, described in the section "Obtaining Errors, Warnings, and Notices" beginning on page 3-30, returns the first and last QuickDraw GX warnings that have been posted.

GXIgnoreGraphicsWarning

You can use the `GXIgnoreGraphicsWarning` function to ignore warnings.

```
void GXIgnoreGraphicsWarning(gxGraphicsWarning warning);
```

`warning`	The warning number or warning name to ignore.

DESCRIPTION

The `GXIgnoreGraphicsWarning` function adds the warning to be ignored to the ignore warning stack. The posting of warnings is suppressed for all warnings on the ignore warning stack. Warnings may be removed from the ignore warnings stack by the use of the `GXPopGraphicsWarning` function.

You may use any Quickdraw GX warning numbers and warning names or, if you have installed an application-defined warning handler, you may use your own warning numbers and warning names, as long as they use a numbering system different than that provided by QuickDraw GX.

SPECIAL CONSIDERATIONS

The `GXIgnoreGraphicsWarning` function saves warning numbers in a warning stack of limited size, so that a limited number of warnings can be ignored at one time. If the `GXIgnoreGraphicsWarning` function has been called too many times with no matching calls to the `GXPopGraphicsWarning` function, subsequent calls to the `GXIgnoreGraphicsWarning` function do not cause the warning to be ignored and a `warning_stack_overflow` warning is posted.

ERRORS, WARNINGS, AND NOTICES

Warnings
`warning_stack_overflow`

SEE ALSO

The use of this function is described in the section “Ignoring Warnings and Notices” beginning on page 3-37.

QuickDraw GX non-debugging warnings that may be posted are listed in the section “Warnings” beginning on page 3-10. Debugging warnings are listed in the section “Warnings” beginning on page 3-10.

The `GXPopGraphicsWarning` function is described in the next section.

GXPopGraphicsWarning

You can use the `GXPopGraphicsWarning` function to remove ignore warnings from the ignore warning stack.

```
void GXPopGraphicsWarning(void);
```

DESCRIPTION

The `GXPopGraphicsWarning` function removes the last warning placed on the ignore warning stack by the `GXIgnoreGraphicsWarning` function. The `GXPopGraphicsWarning` function removes warnings from the stack in the opposite order that they were added to the stack (last in, first out). Calls to the `GXIgnoreGraphicsWarning` and `GXPopGraphicsWarning` functions can be nested.

SPECIAL CONSIDERATIONS

If no warning is on the warning stack when you call this function, a `warning_stack_underflow` warning is posted.

ERRORS, WARNINGS, AND NOTICES

Warnings
`warning_stack_underflow`

SEE ALSO

The use of this function is described in the section “Ignoring Warnings and Notices” beginning on page 3-37.

QuickDraw GX non-debugging warnings that may be added to and removed from the ignore warning stack are listed in the section “Warnings” beginning on page 3-10. Debugging warnings are listed in the section “Warnings” beginning on page 3-10.

The `GXIgnoreGraphicsWarning` function is described in the previous section.

Notice Posting and Handling

This section describes the QuickDraw GX functions you can use to

- obtain the first and last notice posted
- install the current notice
- install an application-defined function for posted notices
- obtain an application-defined notice handler function for posted notices
- add a notice to the ignore notice stack
- remove the last notice to be added to the notice stack

GXGetGraphicsNotice

You can use the `GXGetGraphicsNotice` function to obtain the first and last notices posted.

```
gxGraphicsNotice GXGetGraphicsNotice
                                (gxGraphicsNotice *stickyNotice);
```

`stickyNotice`
On return, a pointer to the first notice posted.

function result The last notice posted.

DESCRIPTION

The `GXGetGraphicsNotice` function returns the last notice posted, or 0 if none. The `stickyNotice` parameter, if not `nil`, receives the first notice posted since the last call to the `GXGetGraphicsNotice` function.

SPECIAL CONSIDERATIONS

QuickDraw GX clears the `stickyNotice` argument at the end of every call to the `GXGetGraphicsNotice` function. It always returns 0 on non-debugging versions.

SEE ALSO

The use of this function is described in the section“Obtaining Errors, Warnings, and Notices” beginning on page 3-30.

An alternative method of posting notices is to include an application-defined notice handler. This topic is described in the section "Changing the Error, Warning, or Notice Posted" beginning on page 3-35.

The `GXSetUserGraphicsNotice` function that is used to install the notice handler is described on page 3-68.

GXPostGraphicsNotice

You can use the `GXPostGraphicsNotice` function to post your own notices from inside your application.

```
void GXPostGraphicsNotice(gxGraphicsNotice notice);
```

`notice`	The notice to be posted.

DESCRIPTION

The `GXPostGraphicsNotice` function replaces the QuickDraw GX notice about to be posted with a notice message defined by the `notice` parameter.

You may use the QuickDraw GX notices or define your own notice number and notice name. This function stores the posted notice so that subsequent calls to the `GXGetGraphicsNotice` function return the notice substituted by this function.

If the notice to be posted is in the ignore notice stack, the notice is not posted and execution continues. Ignoring notices is discussed in the section "Ignoring Warnings and Notices" beginning on page 3-37.

If an application-defined notice handler is provided, the notice is passed to the handler.

The `GXIgnoreGraphicsNotice` function has no effect in the non-debugging version.

SPECIAL CONSIDERATIONS

The notice number must be within the range defined by QuickDraw GX. This range is bounded by notice numbers –25999 through –25500 or is in an application range.

SEE ALSO

The use of this function is described in the section "Changing the Error, Warning, or Notice Posted" beginning on page 3-35.

Notice handlers are discussed in the section "Installing an Error, Warning, or Notice Handler" beginning on page 3-40.

GXSetUserGraphicsNotice

You can use the `GXSetUserGraphicsNotice` function to install a notice handler.

```
void GXSetUserGraphicsNotice(gxUserNoticeFunction userFunction,
                             long reference);
```

`userFunction`
The application function that is to be passed the notice result code.

`reference` A `long` value that is called each time a notice occurs. This value can be used by the application for any purpose.

DESCRIPTION

The `GXSetUserGraphicsNotice` function installs an application-defined notice-handling function. This function installs a function pointer that is called whenever a notice is posted. Setting the `userFunction` parameter to `nil` removes the notice function.

The `userFunction` parameter points to an application-defined notice handler defined by the following type:

```
typedef void (*gxUserNoticeProcPtr)(gxGraphicsNotice status,
               long reference)

typedef gxUserNoticeProcPtr gxUserNoticeFunction;
```

The second parameter is the `long` reference. Whenever a notice is posted by the application, the installed notice handler is called with the notice number. The reference number is passed to the `GXSetUserGraphicsNotice` function.

You can install a notice handler before calling the `GXEnterGraphics` function, but you should call the `GXNewGraphicsClient` function first. If you don't, it will be called for you.

The `GXSetUserGraphicsNotice` function has no effect in the non-debugging version.

SEE ALSO

The use of this function is described in the section "Installing an Error, Warning, or Notice Handler" beginning on page 3-40.

The `GXGetUserGraphicsNotice` function used to return a pointer to the application-defined notice handler is described in the next section.

The application-defined notice handler is described on page 3-74.

GXGetUserGraphicsNotice

You can use the `GXGetUserGraphicsNotice` function to obtain the currently installed application-defined notice handler.

```
gxUserNoticeFunction GXGetUserGraphicsNotice(long *reference);
```

`reference`	A `long` value that is called each time a notice occurs. This value can be used by the application for any purpose.
function result	A pointer to the installed application-defined notice handler.

DESCRIPTION

The `GXGetUserGraphicsNotice` function returns a pointer to the function that the application uses to handle notices. The function returns `nil` if no application-defined notice handler is installed.

If a notice handler function is installed and the `reference` parameter is not `nil`, then the `reference` parameter passed to the `GXSetUserGraphicsNotice` function is returned.

The `GXGetUserGraphicsNotice` function has no effect in the non-debugging version.

SEE ALSO

The use of this function is described in the section "Installing an Error, Warning, or Notice Handler" beginning on page 3-40.

The `GXSetUserGraphicsNotice` function used to install the notice handler is described in the previous section.

An alternative method to the use of an application-defined notice handler is the use of QuickDraw GX notices. The `GXGetGraphicsNotice` function, described in the section "Obtaining Errors, Warnings, and Notices" beginning on page 3-30, returns the first and last QuickDraw GX notices that have been posted.

GXIgnoreGraphicsNotice

You can use the `GXIgnoreGraphicsNotice` function to ignore QuickDraw GX notices that may occur when specific parts of your application execute.

```
void GXIgnoreGraphicsNotice(gxGraphicsNotice notice);
```

`notice`	The graphics notice number or name to ignore.

DESCRIPTION

The `GXIgnoreGraphicsNotice` function adds the notice to be ignored to the ignore notice stack. The posting of notices is suppressed for all notices on the ignore notice stack. Notices may be removed from the ignore notice stack by the use of the `GXPopGraphicsNotice` function.

You may use any QuickDraw GX notice numbers and notice names or, if you have installed an application-defined notice handler, you may use your own notice numbers and notice names, as long as they use a numbering system different than that provided by QuickDraw GX.

This function has no effect in non-debugging versions

SPECIAL CONSIDERATIONS

The `GXIgnoreGraphicsNotice` function saves notice numbers in a warning stack of limited size. If the `GXIgnoreGraphicsNotice` function has been called too many times with no matching calls to the `GXPopGraphicsNotice` function, subsequent calls to the `GXIgnoreGraphicsNotice` function do not cause the notice to be ignored and a `notice_stack_overflow` warning is be posted.

ERRORS, WARNINGS, AND NOTICES

Warnings
`notice_stack_overflow`

SEE ALSO

The use of this function is described in the section “Ignoring Warnings and Notices” beginning on page 3-37.

QuickDraw GX notices that may be posted are listed in the section “Notices” beginning on page 3-27.

The `GXPopGraphicsNotice` function is described in the next section.

GXPopGraphicsNotice

You can use the `GXPopGraphicsNotice` function to remove notices from the ignore notice stack.

```
void GXPopGraphicsNotice(void);
```

DESCRIPTION

The `GXPopGraphicsNotice` function removes the last notice added to the ignore notice stack by the `GXIgnoreGraphicsNotice` function. The `GXPopGraphicsNotice` function removes notices from the stack in the opposite order that they were added to the stack (last in, first out). Calls to the `GXIgnoreGraphicsNotice` function and the `GXPopGraphicsNotice` function can be nested.

The `GXPopGraphicsNotice` function has no effect in the non-debugging version.

SPECIAL CONSIDERATIONS

If no notice is on the ignore notice stack when you call this function, a `notice_stack_underflow` warning is posted.

ERRORS, WARNINGS, AND NOTICES

Warnings
`notice_stack_underflow`

SEE ALSO

The use of this function is described in the section "Ignoring Warnings and Notices" beginning on page 3-37.

QuickDraw GX notices that may be added and removed from the ignore notice stack are listed in the section "Notices" beginning on page 3-27.

The `GXIgnoreGraphicsNotice` function is described in the previous section.

Application-Defined Functions

QuickDraw GX supports application-defined error, warning, and notice handlers. These handlers are installed by the use of the `GXSetUserGraphicsError`, `GXSetUserGraphicsWarning`, and `GXSetUserGraphicsNotice` functions.

MyUserGraphicsError

You can use the `MyUserGraphicsError` function to provide an application-defined error handler for your application.

```
void MyUserGraphicsError(gxGraphicsError error, long reference);
```

`error`	The QuickDraw GX error being passed to the handler.
`reference`	A `long` value passed each time that an error occurs. This value can be used by the error handler for any purpose.

DESCRIPTION

The `MyUserGraphicsError` function is called with the error number posted by the failed function. The `MyUserGraphicsError` function can evaluate the error and respond in any appropriate manner.

The error handler is enabled and disabled by the use of the `GXSetUserGraphicsError` function. If its parameter is set to `nil`, the error handler is disabled. If its parameter is not `nil`, the error handler is enabled and all errors detected by QuickDraw GX are passed to the error handler for processing and possible response.

The `GXGetUserGraphicsError` function returns the currently installed application-defined error handler.

SEE ALSO

The use of this function is described in the section “Installing an Error, Warning, or Notice Handler” on page 3-40.

QuickDraw GX non-debugging errors that may be sent to the error handler are listed in section “Errors” beginning on page 3-6. Debugging errors are listed in the section “Errors” beginning on page 3-6.

The `GXSetUserGraphicsError` function is described on page 3-58.

The `GXGetUserGraphicsError` function is described on page 3-59.

MyUserGraphicsWarning

You can use the `MyUserGraphicsWarning` function to provide an application-defined warning handler for your application.

```
void MyUserGraphicsWarning(gxGraphicsWarning warning,
                           long reference);
```

`warning`	The QuickDraw GX warning being passed to the handler.
`reference`	A `long` value passed each time that a warning occurs. This value can be used by the warning handler for any purpose.

DESCRIPTION

The `MyUserGraphicsWarning` function is called with the warning number posted by the defective function. The `MyUserGraphicsWarning` function can evaluate the warning and respond in any appropriate manner.

The warning handler is enabled and disabled by the use of the `GXSetUserGraphicsWarning` function. If its parameter is set to `nil`, the warning handler is disabled. If its parameter is not `nil`, the warning handler is enabled and all warnings detected by QuickDraw GX are passed to the warning handler for processing and possible response.

The `GXGetUserGraphicsWarning` function returns the currently installed application-defined warning handler.

SEE ALSO

The use of this function is described in the section “Installing an Error, Warning, or Notice Handler” on page 3-40.

Warnings that may be sent to the warning handler are listed in the section “Warnings” beginning on page 3-10.

The `GXSetUserGraphicsWarning` function is described on page 3-62.

The `GXGetUserGraphicsWarning` function is described on page 3-63.

MyUserGraphicsNotice

You can use the `MyUserGraphicsNotice` function to provide an application-defined notice handler for your application.

```
void MyUserGraphicsNotice(gxGraphicsNotice notice,
                          long reference);
```

`notice`	The QuickDraw GX notice being passed to the handler.
`reference`	A `long` value passed each time that a notice occurs. This value can be used by the notice handler for any purpose.

DESCRIPTION

The `MyUserGraphicsNotice` function is called with the notice number posted by the defective function. The `MyUserGraphicsNotice` function can evaluate the notice and respond in any appropriate manner.

The notice handler is enabled and disabled by the use of the `GXSetUserGraphicsNotice` function. If its parameter is set to `nil`, the notice handler is disabled. If its parameter is not `nil`, the notice handler is enabled and all notices detected by QuickDraw GX are passed to the notice handler for processing and possible response.

The `GXGetUserGraphicsNotice` function returns the currently installed application-defined notice handler. This function will never be called in the non-debugging version of QuickDraw GX.

SEE ALSO

The use of this function is described in the section “Installing an Error, Warning, or Notice Handler” on page 3-40.

Notices that may be sent to the notice handler are listed in the section “Notices” beginning on page 3-27.

The `GXSetUserGraphicsNotice` function is described on page 3-68.

The `GXGetUserGraphicsNotice` function is described on page 3-69.

Summary of Errors, Warnings, and Notices

Constants and Data Types

QuickDraw GX Errors

```
typedef long gxGraphicsError
```

QuickDraw GX Warnings

```
typedef long gxGraphicsWarning
```

QuickDraw GX Notices

```
typedef long gxGraphicsNotice
```

Application-Defined Handlers

```
typedef void                (*gxUserErrorProcPtr)(gxGraphicsError status,
                             long refcon)
typedef gxUserErrorProcPtr gxUserErrorFunction;
typedef void                (*gxUserWarningProcPtr)(gxGraphicsWarning
                             status,long refcon)
typedef gxUserWarningProcPtr gxUserWarningFunction;
typedef void                (*gxUserNoticeProcPtr)(gxGraphicsNotice status,
                             long refcon)
typedef gxUserNoticeProcPtr gxUserNoticeFunction;
```

Functions

Error Posting and Handling

```
gxGraphicsError GXGetGraphicsError
                             (gxGraphicsError *stickyError);
void GXPostGraphicsError     (gxGraphicsError error);
void GXSetUserGraphicsError  (gxUserErrorFunction userFunction,
                             long reference);
gxUserErrorFunction GXGetUserGraphicsError
                             (long *reference);
```

Warning Posting and Handling

```
gxGraphicsWarning GXGetGraphicsWarning
                             (gxGraphicsWarning *stickyWarning);
void GXPostGraphicsWarning  (gxGraphicsWarning warning);
void GXSetUserGraphicsWarning
                             (gxUserWarningFunction userFunction,
                              long reference);
gxUserWarningFunction GXGetUserGraphicsWarning
                             (long *reference);
void GXIgnoreGraphicsWarning
                             (gxGraphicsWarning warning);
void GXPopGraphicsWarning   (void);
```

Notice Posting and Handling

```
gxGraphicsNotice GXGetGraphicsNotice
                             (gxGraphicsNotice *stickyNotice);
void GXPostGraphicsNotice   (gxGraphicsNotice notice);
void GXSetUserGraphicsNotice
                             (gxUserNoticeFunction userFunction,
                              long reference);
gxUserNoticeFunction GXGetUserGraphicsNotice
                             (long *reference);
void GXIgnoreGraphicsNotice
                             (gxGraphicsNotice notice);
void GXPopGraphicsNotice    (void);
```

Application-Defined Functions

```
void MyUserGraphicsError    (gxGraphicsError error, long reference);
void MyUserGraphicsWarning  (gxGraphicsWarning warning, long reference);
void MyUserGraphicsNotice   (gxGraphicsNotice notice, long reference);
```

C H A P T E R 4

QuickDraw GX Debugging

Contents

This chapter describes the QuickDraw GX application debugging environment and the functions and utilities that you can use to debug your application. Read this chapter if you are developing a QuickDraw GX application and want to use these features.

Before reading this chapter, you should be familiar with the debugging and non-debugging versions of QuickDraw GX described in the chapter “Errors, Warnings, and Notices” in this book. You should also read the chapter “Introduction to QuickDraw GX” in *Inside Macintosh: QuickDraw GX Objects*.

For more information on debugging printing applications, see *Inside Macintosh: QuickDraw GX Printing* and *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers*.

This chapter introduces the QuickDraw GX debugging environment. It then describes how to use this environment during application development to

- analyze drawing problems
- validate public and internal function parameters for all allocated objects
- validate public and internal function parameters for specific objects
- distinguish between application and QuickDraw GX bugs
- detect corrupted objects
- install a debugging function
- use the GraphicsBug utility

This chapter also contains reference information for all data types and functions associated with QuickDraw GX debugging.

About QuickDraw GX Debugging

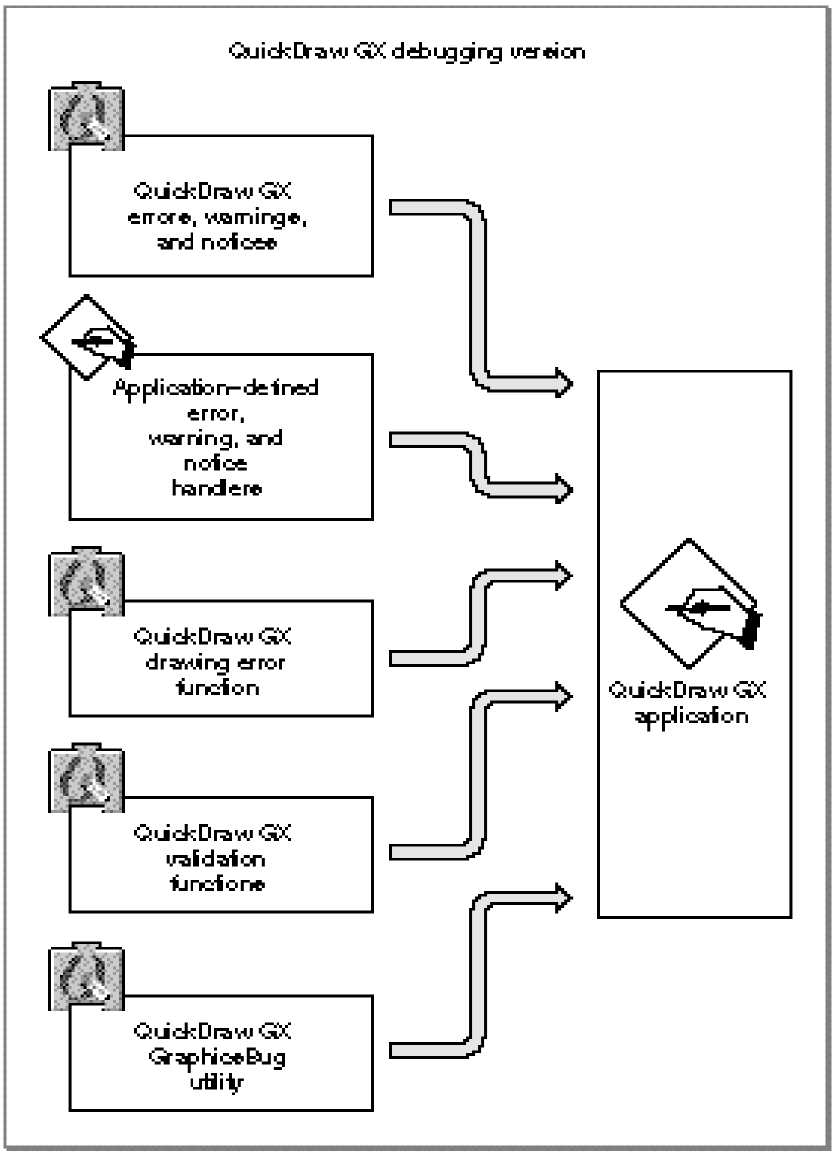
QuickDraw GX provides both a **debugging environment** and a **non-debugging environment.** The non-debugging environment is present whenever you install the non-debugging version of QuickDraw GX. You install the non-debugging version after completely debugging your application. Users of your application will use the non-debugging version of QuickDraw GX.

You can develop applications that use QuickDraw GX graphics and typography functions using the QuickDraw GX debugging environment. The debugging environment consists of

- the QuickDraw GX debugging version
- QuickDraw GX errors, warnings, and notices
- application-defined error, warning, and notice handlers
- a QuickDraw GX drawing error function
- QuickDraw GX validation functions
- the QuickDraw GX GraphicsBug utility

Figure 4-1 shows the QuickDraw GX application development environment.

Figure 4-1 The QuickDraw GX debugging environment


QuickDraw GX debugging version
QuickDraw GX errors, warnings, and notices
Application-defined error, warning, and notice handlers
QuickDraw GX drawing error function
QuickDraw GX validation functions
QuickDraw GX GraphicsBug utility
QuickDraw GX application

As a direct result of the extensive error, warning, and notice checking the debugging environment performs, the debugging version of QuickDraw GX is significantly slower than that of the non-debugging environment. Invoking additional optional error checking using the validation functions further affects performance.

Debugging Version of QuickDraw GX

You should use the debugging version of QuickDraw GX when you are writing and debugging applications. This version provides an extensive set of errors, warnings, and notices to help you understand the problems you may encounter during the execution of your application. In addition, this version provides special functions that allow you to manage errors, warnings, and notices and to provide public and private error validation.

The debugging version runs slower than the non-debugging version. The reasons for this are that the debugging version:

- performs additional error checking
- posts additional errors, warnings, and notices
- does not provide speed optimization, such as in-line functions
- generates MacsBug messages
- provides additional debugging functions, such as validation

To determine if the debugging or non-debugging version of QuickDraw GX is installed, see the chapter “QuickDraw GX and the Macintosh Environment.”

QuickDraw GX Errors, Warnings, and Notices

QuickDraw GX posts errors, warnings, and notices whenever an execution problem occurs while an application is running. You can obtain errors, warnings, and notices by polling or by the use of application-defined error, warning, and notice handlers.

The debugging and non-debugging versions of QuickDraw GX and the errors, warnings, and notices that may be posted from each version are described in the chapter “Errors, Warnings, and Notices.”

Application-Defined Error, Warning, and Notice Handlers

You can use error, warning, and notice handlers to manage problems that occur when your application is running. When QuickDraw GX detects an error, warning, or notice it will call your handler. Your function can then respond accordingly. You can also use error and warning handlers with the non-debugging version of QuickDraw GX to provide part of your user interface.

Application-defined error, warning, and notice handlers are described in the section “Installing an Error, Warning, or Notice Handler” beginning on page 3-40.

The Drawing Error Function

The debugging version of QuickDraw GX provides a drawing error function that you can use if you have run your application and get an unexpected result. This function reparses the entire QuickDraw GX operation, analyzes your application's draw procedure, and posts a single error that will assist you in determining what went wrong with your application. The drawing error function is described in the section "Analyzing Drawing Problems" beginning on page 4-8.

Validation Functions

The debugging version of QuickDraw GX provides validation for applications using graphics and typographic functions, but does not provide validation for QuickDraw GX printing functions.

The validation functions check function parameters of allocated objects to see if they are valid. If QuickDraw GX finds one or more parameters of a function to be invalid, it posts a validation error. All of the validation errors that may be posted are listed in the chapter "Errors, Warnings, and Notices."

There are two modes of validation that control when validation occurs:

- public validation
- internal validation

Public validation occurs whenever the public validation flag is set and your application uses a public function. A public function is any function that you use in your application. This is the mode of validation developers use most.

Internal validation occurs whenever the internal validation flag is set and your application uses a public function and whenever QuickDraw GX uses one of its internal (private) functions. Application developers do not usually use internal validation. Internal validation performs checking on functions that you have no control over. As a result, you will rarely need to perform this type of validation. However, QuickDraw GX provides internal validation to allow you to distinguish between bugs that appear in public functions and bugs that are present in the QuickDraw GX internal functions, as discussed in the section "Distinguishing Between Application Bugs and QuickDraw GX Bugs" beginning on page 4-22.

There are three levels of validation that control what is checked during validation:

- type validation
- structure validation
- all object validation

The validation these three levels provide is cumulative and progressively more complex. For example, all object validation includes type validation and structure validation.

In addition to these three levels, there are separate object validation functions.

Type validation confirms the validity of references to object types. For example, when you call the `GXDrawShape` function, type validation confirms that a shape type is passed.

Structure validation confirms the validity of references to object types and the properties of the function, and also checks internal caches. For example, when you call the `GXDrawShape` function, structure validation confirms not only that the function passes a shape, but also confirms the validity of the properties specified in the shape's style, ink, and transform objects.

All object validation confirms the validity of references to a specific object type, the validity of the properties of all objects, and all internal caches.

Specific object validation functions are used to confirm that all references to a specific object type are valid, that the properties of all objects are valid, and that all internal caches built for the specific object type are valid. Specific object validation functions are provided for shapes, styles, inks, transforms, color sets, color profiles, tags, view devices, view ports, view groups, and graphics clients.

It is important to note that not all parameters of all functions are checked by validation. Validation does not check scalars and structures, such as bitmaps and dash records.

For example, the second parameter of the `GXSetShapePen` function is the pen size. If you pass a negative value to the second parameter, QuickDraw GX will not post a validation error. Fortunately, QuickDraw GX often provides an overlap in its debugging capabilities, and in this case, the `GXSetShapePen` function would post an error indicating that the size is invalid.

Validation does check

- objects that are indicated by pointer values, such as shapes
- objects that are indicated by references, such as view devices

Note

You should not make an application dependent on whether an object is referred to by pointer or reference. This is subject to change in future versions of QuickDraw GX. u

You can enable validation selectively over the selected problem area of code. Rather than turning validation on at the beginning of your application, you may find it is more useful to concentrate on an area where a problem is suspected and to turn validation on and off selectively in that area or selectively use the specific object validation functions.

MacsBug and GraphicsBug

Both the debugging and non-debugging versions of QuickDraw GX support MacsBug and GraphicsBug. **MacsBug** is Apple Computer, Inc.'s, assembly-language debugger that was developed for Macintosh programmers. MacsBug is not very useful for debugging QuickDraw GX applications because GX data structures are private. For additional information about MacsBug, see *MacsBug Reference and Debugging Guide.*

GraphicsBug is Apple Computer Inc.'s symbolic debugger for QuickDraw GX applications. This utility assists in finding bugs by allowing you to display and check QuickDraw GX objects. GraphicsBug is modeled after MacsBug. In fact, many of the commands are similar.

The use of GraphicsBug to analyze a QuickDraw GX graphics client heap is described in the section "Debugging With GraphicsBug" beginning on page 4-23.

Using QuickDraw GX Debugging

You can use the QuickDraw GX debugging environment to help you debug your application. This section shows how you can

- determine why a shape didn't draw
- validate using public, internal, or object modes of validation
- validate types, structures, and all objects
- validate memory
- distinguish between QuickDraw GX bugs and application bugs
- validate public objects
- analyze the QuickDraw GX graphics heap with the GraphicsBug utility

Analyzing Drawing Problems

If you have run your application and a shape didn't draw as you anticipated, you can use the `GXGetShapeDrawError` function to have QuickDraw GX analyze why the shape didn't draw correctly. This function checks the content of a shape and all of the objects referenced by the shape for a condition that explains why the shape has no visible effect when drawn. As a result, `GXGetShapeDrawError` returns a single **drawing error** from the `gxDrawErrors` enumeration that may describe why the shape failed to draw correctly. The `gxDrawErrors` enumeration is listed in the section "Drawing Errors" beginning on page 4-29. The `GXGetShapeDrawError` function is described on page 4-33.These errors should not be confused with `gxGraphicsErrors`.

If the drawing was completed successfully, QuickDraw GX posts the `NoDrawError` drawing error. If you don't see the drawing, remember that it may have been drawn to a different view device or may have just redrawn over the previous shape that was drawn. The posting of a `NoDrawError` drawing error does not mean that the shape drawn is the one you expected or the correct shape. It just means that QuickDraw GX detected no drawing problems during the processing of the shape drawn.

The drawing error QuickDraw GX posts is selected from a special subset of the QuickDraw GX error codes. This set of drawing error codes is structured with respect to the stage in the **drawing process sequence** that the drawing failed. The earliest stage of failure will be described in the posted drawing error. The single error code posted attempts to indicate the reason that you do not see the drawing that you anticipated.

Drawing errors are grouped into categories that correspond to the approximate sequence of QuickDraw GX processing, as shown in Table 4-1.

Table 4-1 QuickDraw GX drawing process sequence

Drawing process sequence	Object processed
1	Shape type
2	Style
3	Ink
4	Transform
5	View port
6	View device

The processing sequence is also the sequence of drawing errors posted. QuickDraw GX posts the first drawing error that is detected. It does not post subsequent drawing errors until the error posted earlier in the process sequence is corrected. For example, if an application attempts to draw a defective shape with a defective view port, QuickDraw GX posts a single shape type drawing error and does not post a view port drawing error. This is because QuickDraw GX analyzes the integrity of the shape earlier in the drawing process. It analyzes the integrity of the view port toward the end of the process. Once you correct the defective shape, QuickDraw GX can detect the defective view port in subsequent analysis with the `GXGetShapeDrawError` function.

Table 4-2 shows the GXGetShapeDrawError function **shape type** drawing errors that QuickDraw GX may post.

Table 4-2 Shape type drawing errors

Error	Description
shape_emptyType	An empty type doesn't have an area to draw.
shape_inverse_fullType	An inverse full type doesn't have an area to draw.
rectangle_zero_width	The rectangle doesn't have an area to draw.
rectangle_zero_height	The rectangle doesn't have an area to draw.
polygon_empty	There is no contour to draw.
path_empty	There is no contour to draw.
bitmap_zero_width	The bitmap doesn't have an area to draw.
bitmap_zero_height	The bitmap doesn't have an area to draw.
text_empty	There is no character to draw.
glyph_empty	There is no glyph to draw.
layout_empty	There is no layout to draw.
picture_empty	There is no shape in the picture.
shape_no_fill	The shape fill is set to gxNoFill, which will not draw.
shape_no_enclosed_area	There is no enclosed area to draw.
shape_no_enclosed_pixels	There is an enclosed area, but it is so small that it does not cross any pixel centers.
shape_very_small	There is a shape to draw, but it is extremely small (on the order of the size of a pixel).
shape_very_large	Part of the shape may be drawn outside the bounds of the coordinate system (±32,768).
shape_contours_cancel	The shapes contours overlap and cancel each other out.

Table 4-3 shows the GXGetShapeDrawError function style drawing errors.

Table 4-3 Style drawing errors

Error	Description
pen_too_small	The pen width is so small that it doesn't enclose any pixels and therefore doesn't draw.
text_size_too_small	The text size is so small that it doesn't enclose any pixels and therefore doesn't draw.
dash_empty	The dash shape was specified as an empty type shape.
start_cap_empty	The start cap shape was specified as an empty type shape.
pattern_empty	The pattern shape was specified as an empty type shape.
textFace_Empty	Each layer of the text face has a shape fill equal to gxNoFill.
shape_primitive_empty	The original shape enclosed an area. There is no stylized shape to draw. An example is a pattern shape that contains overlapping patterns that cancel.
shape_primitive_very_small	There is a shape to draw, but it is extremely small (on the order of the size of a pixel). An example is a scaled transform that shrinks the shape.

Table 4-4 shows the `GXGetShapeDrawError` function ink drawing errors.

Table 4-4 Ink drawing errors

Error	Description
`transfer_equals_noMode`	The transfer mode `gxNoMode` suppresses drawing.
`transfer_matrix_ignores_source`	The transfer mode's mapping scales all values greater than 1 or less than 0 and the `overComponent` flag is not set.
`transfer_matrix_ignores_device`	The transfer mode's mapping scales all values greater than 1 or less than 0 and the `overComponent` flag is not set.
`transfer_source_reject`	The color is not within the source minimum and the source maximum.
`transfer_mode_ineffective`	The transfer mode has no effect on the device. An example is a blend with an operand of 0.
`colorSet_no_entries`	There are no colors in the color set so there is nothing to draw.
`bitmap_colorSet_one_entry`	The bitmap drew, but it is probably not the desired result, since all colors map to the one color of the entry. An example is when the colors are off the end of the color set.

Table 4-5 shows the `GXGetShapeDrawError` function transform drawing errors.

Table 4-5 Transform drawing errors

Error	Description
`transform_scale_too_small`	The transform has reduced the shape to less than 1/72 inch. You may see a few pixels drawn, depending on the resolution of your view port.

`transform_map_too_large`
`transform_move_too_large`
`transform_scale_too_large`
`transform_rotate_too_large`
`transform_perspective_too_large`
`transform_skew_too_large`
	The transform has moved all or part of the shape outside the bounds of the coordinate system (±32,768). This may be the result of a move, scale, rotate, perspective, or skew transformation.

`transform_clip_no_intersection`	The clip shape does not intersect any view port.
`transform_clip_empty`	The transform clip is an empty type shape.
`transform_no_viewPorts`	The number of entries in the view port list is zero.

Table 4-6 shows the `GXGetShapeDrawError` function view port drawing errors.

Table 4-6 View port drawing errors

Error	Description
`viewPort_disposed`	The view port that was to be drawn to has already been disposed of. There is no view port to draw to.
`viewPort_clip_empty`	The view port clip is an empty type shape.
`viewPort_clip_no_intersection`	The view port clip does not intersect the view device.
`viewPort_scale_too_small`	The map to global space has been completed. The object is less than 1/72 inch. You may see a few pixels drawn, depending on the resolution of your view port.

`viewPort_map_too_large`
`viewPort_move_too_large`
`viewPort_scale_too_large`
`viewPort_rotate_too_large`
`viewPort_perspective_too_large`
`viewPort_skew_too_large,`
	The view port mapping has moved all or part of the shape outside the bounds of the coordinate system (±32,768). This may be the result of a move, scale, rotate, perspective, or skew transformation.

`viewPort_viewGroup_offscreen`	The shape is drawn to an off-screen view device. This may be normal. This error is returned to alert you in the event that the drawing result was unexpected.

Table 4-7 shows the `GXGetShapeDrawError` function view device drawing errors.

Table 4-7 View device drawing errors

Error	Description
`viewDevice_clip_no_intersection`	The view device clip does not intersect the bounds described by the view device bitmap shape.
`viewDevice_scale_too_small`	The mapping to global space has been completed. The object is less than 1/72 inch. You may see a few pixels drawn, depending on the resolution of your draw view port.

`viewDevice_map_too_large`
`viewDevice_move_too_large`
`viewDevice_scale_too_large`
`viewDevice_rotate_too_large`
`viewDevice_perspective_too_large`
`viewDevice_skew_too_large`
	The view port mapping has moved the shape outside the bounds of the coordinate system (±32,768). This may be the result of a move, scale, rotate, perspective, or skew transformation.

Using Validation Functions

QuickDraw GX provides validation functions that check the function parameters of all allocated objects. You can validate the public functions that you use in your application or choose to validate the internal QuickDraw GX functions.

Type validation is the simplest level of validation. QuickDraw GX provides successively more complicated levels of validation when you also check structures and internal caches. The various validation modes and validation levels are described in the section "Validation Functions" beginning on page 4-6.

Controlling Validation

You can use the `GXSetValidation` function to control the validation of public and private functions used by your application. You control validation by using the `GXSetValidation` function to set validation level flags for the `gxValidationLevel` parameter.

```
void GXSetValidation(gxValidationLevel, level);
```

You set one flag from the modes in Table 4-8, one flag from the options in Table 4-9, and one or more flags from Table 4-10. The validation modes and levels are defined in the `gxValidationLevel` enumeration that appears in the section "Drawing Errors" beginning on page 4-29. The `GXSetValidation` function is described on page 4-34

Once you set the `gxValidationLevel` parameter, you can use the `GXGetValidation` function to return the current `gxValidationLevel` parameter.

The three validation mode options are validation off, public validation, and internal validation. You may choose only one of these validation options. Table 4-8 summarizes the public and internal validation mode options.

Table 4-8 Validation modes

Constant	Value	Explanation
`gxNoValidation`	0x00	Turns off QuickDraw GX validation.
`gxPublicValidation`	0x01	Performs validation whenever your application uses a public function.
`gxInternalValidation`	0x02	Performs validation whenever your application uses a public function or an internal function.

The validation mode flags allow you to selectively turn validation options on and off. You should experience reduction in performance only when validation is on. In the non-debugging version, validation is not operational. However, it is best just to turn validation off by setting the parameter of the `GXSetValidation` function to `gxNoValidation`.

If you activate either public validation or internal validation mode, then you must also specify either type validation, structure validation, or all object validation. You may choose only one option. Table 4-9 summarizes the type, structure, and object validation level options.

Table 4-9 Validation levels

Constant	Value	Explanation
`gxTypeValidation`	0x00	Validates object types of function parameters.
`gxStructureValidation`	0x10	Validates object structures, caches and function parameters.
`gxAllObjectValidation`	0x20	Validates object types, structures, and all internal caches built for all objects.

Type Validation

You can select the `gxTypeValidation` level to check the type passed to all objects. The type validation errors are listed in the section “Debugging Version” in the chapter “Errors, Warnings, and Notices.”

The simplest and most commonly used `gxValidationLevel` parameter value combination is of the `gxPublicValidation` and `gxTypeValidation` options:

```
GXSetValidation(gxPublicValidation | gxTypeValidation);
```

This combination of options causes QuickDraw GX to verify that the objects used by all public functions your application calls are the correct type. For example, if you call the `GXDrawShape` function and pass it a style, the `GXSetValidation` function posts a `shape_wrong_type` error.

If you want to check the type of all objects that your application passes to both public and internal functions, you can use the `gxInternalValidation` option plus the `gxTypeValidation` option for the `gxValidationLevel` parameter:

```
GXSetValidation(gxInternalValidation | gxTypeValidation);
```

This is useful only for detecting GX internal errors.

Structure Validation

You can set the `gxStructureValidation` validation parameter to check the type and structure for all objects. The structure validation errors are listed in the section “Debugging Version” in the chapter “Errors, Warnings, and Notices.”

If you want to check the type of all objects and structure your application passes to public functions, you can use the `gxPublicValidation` and `gxStructureValidation` options for the `gxValidationLevel` parameter:

```
GXSetValidation(gxPublicValidation | gxStructureValidation);
```

If you want to check the type of all objects and the structure your application passes to public and internal functions, you can use the `gxInternalValidation` and `gxStructureValidation` options for the `gxValidationLevel` parameter:

```
GXSetValidation(gxInternalValidation | gxStructureValidation);
```

This is useful only for detecting internal GX errors.

The gxStructureValidation option might generate validation errors that are not part of the public interface. For example, these options may post a shape_cache_wrong_type error. This suggests only that the application erroneously changed the internal information that identifies a specific shape cache or an internal GX error occured. The correct shape and the correct value for a shape cache are private. The bad_private_flags error means that the application corrupted the flags internal to some structure. This is a private structure and QuickDraw GX provides no additional information for these posted errors. However it is useful for a developer to report the circumstances that produced these errors so that Apple Computer, Inc. can investigate them.

All Object Validation

You can use the gxAllObjectValidation validation level to check the type, structure, and internal caches built for all objects. In addition, it checks objects written to disk and the file structure itself to see if they are corrupt. The all object validation errors are listed in the section "Debugging Version" in the chapter "Errors, Warnings, and Notices."

If an application has an error that randomly writes to some portion of memory, the error can corrupt one object as easily as another. As a result, it is necessary to check all objects to detect this type of error. If a random write occurs in a free memory block or the value is already in the shape type, QuickDraw GX doesn't detect it. Again, this validation allows the developer to discriminate between QuickDraw GX and application problems.

If you want to check the type of all objects, the structure, and the internal caches for all objects each time public functions are called by your application, you can use the gxPublicValidation and gxAllObjectValidation options for the gxValidationLevel parameter:

```
GXSetValidation(gxPublicValidation | gxAllObjectValidation);
```

As an alternative to using the gxAllObjectValidation options for the gxValidationLevel parameter of the GXSetValidation function, you can use the GXValidateAll function, described in the section "Validating Objects" beginning on page 4-20. The GXValidateAll function is described on page 4-43.

Memory Validation

Once you pick a validation mode and a validation level, you can then also choose to include or not include memory validation options. Memory validation does not post validation errors. If QuickDraw GX detects a memory validation problem, it drops you into Macsbug or the debugging utility that is installed on your system.

Table 4-10 summarizes the memory validation options, all of which are associated with QuickDraw GX private data structures.

Table 4-10 Memory validation options

Constant	Value	Explanation
`gxNoMemoryManagerValidation`	0x0000	Turns off memory validation.
`gxApBlockValidation`	0x0100	Enables additional error checking on application blocks passed as parameters to internal memory routines.
`gxFontBlockValidation`	0x0200	Enables additional error checking on system blocks, often font caches, passed as parameters to internal memory routines.
`gxApHeapValidation`	0x0400	Checks all objects in a heap for validity each time an internal memory routine is called.
`gxFontHeapValidation`	0x0800	Checks all font objects in a heap for validity each time an internal memory routine is called.
`gxCheckApHeapValidation`	0x1000	When used with `gxInternalValidation`, checks the application heap on every internal function call. When used with `gxPublicValidation`, checks the application heap on every public function call.
`gxCheckFontHeapValidation`	0x2000	When used with `gxInternalValidation`, checks the font heap on every internal function call. When used with `gxPublicValidation`, checks the font heap on every public function call.

If you want to check the type of all objects that your application passes to public functions and also check the application heap on every public call, you can use the `gxPublicValidation` option plus the `gxTypeValidation` option plus the `gxCheckApHeapValidation` option for the `gxValidationLevel` parameter:

```
GXSetValidation(gxPublicValidation | gxTypeValidation |
                gxCheckApHeapValidation);
```

s **WARNING**
If the `gxApHeapValidation` or `gxFontHeapValidation` flag is enabled and the platform that it is running on locates the graphics memory below the bottom 14 megabytes of memory, then the addresses on the stack and master pointers that refer to QuickDraw GX objects will be scrambled. This is a method of finding internal errors that may lead to unexpected erroneous behavior. For example, if the application has a path type shape and one `long` parameter of the path data happens to exactly equal the address of a graphics object, then QuickDraw GX might scramble the one `long` of path data and the path may draw one point off of the screen. This is expected behavior. These functions can scramble addresses without knowing that the addresses are really points on a path. Since these two validation types produce these apparent bugs, an application cannot use the `gxApHeapValidation` and `gxFontHeapValidation` options to ensure that QuickDraw GX has no internal bugs. These validation types are useful in tracking down bugs related to QuickDraw GX memory management. s

For additional information about using QuickDraw GX memory, see the chapter “QuickDraw GX Memory Management.”

The `GXSetValidation` function is described on page 4-34. The `GXGetValidation` function is described on page 4-35.

Validating Objects

QuickDraw GX also provides separate functions that validate the parameters passed to specific objects, their structures, and any internal caches built for specific objects.

You can use the `GXValidateAll` function to check the type, structure, and internal caches built for all objects. This is an alternative to using the `GXSetValidation` function with the `gxInternalValidation` and `gxAllObjectValidation` options selected, as described in the section “Using Validation Functions” beginning on page 4-15.

The following functions validate specific objects:

- The `GXValidateColorSet` function checks parameters for the color space, color-value array, owner count, and tag list properties for the specified color set object. The `GXValidateColorSet` function is described on page 4-38.
- The `GXValidateColorProfile` function checks the specified color profile object. The `GXValidateColorProfile` function is described on page 4-39.

- The `GXValidateGraphicsClient` function checks all properties of a specified graphics client object. The `GXValidateGraphicsClient` function is described on page 4-42.
- The `GXValidateInk` function checks parameters for the color, transfer mode, attributes, owner count, and tag list properties for a specified ink object. The `GXValidateInk` function is described on page 4-37.
- The `GXValidateShape` function checks parameters for the type, geometry, fill, style, ink, transform, attributes, owner count, and tag list properties for a specified shape object. The `GXValidateShape` function is described on page 4-36.
- The `GXValidateStyle` function checks parameters for the pen size, cap, join, dash, pattern, curve error, attributes, text face, text size, justification, font variations, platform, text attributes properties, run controls, run features array, glyph substitutions array, kerning adjustments, priority justification override, and glyph justification overrides array properties for the specified style object. The `GXValidateStyle` function is described on page 4-36.
- The `GXValidateTag` function checks the parameters for the tag type, size, contents, and owner count properties for a specified tag object. The `GXValidateTag` function is described on page 4-39.
- The `GXValidateTransform` function checks the parameters for the clip, mapping, view port list, hit-test parameters, attributes, owner count, and tag list properties for a specified transform object. The `GXValidateTransform` function is described on page 4-38.
- The `GXValidateViewDevice` function checks parameters for the clip, mapping, bitmap, attributes, and tag list properties for a specified view device object. The `GXValidateViewDevice` function is described on page 4-40.
- The `GXValidateViewPort` function checks parameters for the clip, mapping, dither, halftone, parent view port, child view port list, view device, attributes, owner count, and tag list properties for all view port objects. The `GXValidateViewPort` function is described on page 4-40.
- The `GXValidateViewGroup` function checks parameters for the clip, mapping, dither, halftone, parent view port, child view port list, view device, attributes, owner count, and tag list properties of the view port object and the clip, mapping, bitmap, attributes, and tag list properties of the view device object. The `GXValidateViewGroup` function is described on page 4-41.

Analyzing the Cause of Validation Errors

You can use the `GXGetValidationError` function to determine the function and parameter that caused the last validation error. This function works like other QuickDraw GX functions that return variable-length data. There are three steps:

1. Call the function to determine the length of data that will be returned. If no validation error is posted, a 0 is returned.
2. Allocate memory to store the data that will be returned.
3. Call the function a second time to obtain pointers to the function, parameter name, and parameter number that caused the validation error.

Listing 4-1 gives an example of using the `GXGetValidationError` function to obtain the function and parameter that caused the last validation error. The `GXGetValidationError` function is described on page 4-35.

Listing 4-1 Determining the function and parameter that caused the last validation error

```
static void DisplayErrorMessage(gxGraphicsError errorID,
long context)
{
   char buffer[255];
   void * graphicsObject;
   long argNum;

   if (GXGetValidationError(buffer, &thing, &argNum)) {
      GXValidationError(buffer, nil, nil);
      printf(“gxValidationError: %ld (routine: %s) “,
         errorID, buffer);
      printf(“(argument[%ld]: 0x081x)\n”,argNum, graphicsObject);
   } else
      printf(“gxGraphicsError: 0x%081x\n”, errorID);
}
```

Distinguishing Between Application Bugs and QuickDraw GX Bugs

All QuickDraw GX functions have been extensively tested prior to shipment. However, during your application debugging process, you may find anomalous behavior that you attribute to QuickDraw GX private functions.

Validation checking allows you to distinguish between your application bugs and QuickDraw GX bugs. If QuickDraw GX posts validation errors when internal validation is set, but not when public validation is set, it is possible that you have found an error in the QuickDraw GX internal private code. Please contact Apple Developer Technical Support and provide a detailed report of the bug encountered. For more information concerning public and internal validation modes, see the section “Controlling Validation” beginning on page 4-15.

Detecting Corrupted Objects

Normally, there is no way for an application using the public interface to corrupt the content of an object. If an error occurs with structure validation and not with type validation, either the error is a QuickDraw GX error or the application has corrupted memory. The most probable method of corrupting memory is by calling the `GXGetShapeStructure` function and altering the content directly or by writing randomly into memory. For more information concerning type and structure validation levels, see the section “Controlling Validation” beginning on page 4-15.

Debugging With GraphicsBug

GraphicsBug reads and verifies only graphics objects. It does not create objects, dispose of objects, or modify objects in any manner. GraphicsBug never interferes with an application and does not cause bugs to appear or disappear.

Table 4-11 summarizes the GraphicsBug commands. This list is available online by typing “?”, “help”, or “HELP” when in the command line of GraphicsBug. You can copy or save the brief explanations as a text file.

Table 4-11 GraphicsBug commands and responses

Command	Response

`DA`
`[bu(sy)`
`di(rect)`
`fr(ee)`
`i(ndirect)`
`t(emp) u(n)b(usy)`
` u(n)l(oaded)]`
`[<type>[type>...]]`
	Display all blocks in the heap, or all that match parameters. Example: `DA bu line layout polygon`.

`DM addr[n|t(ype)]`	Display memory from `addr` for *n* bytes or as a type. Example: `DM 1b2358 t`.
`DV`	Display version.
`ER number`	Display error name that matches this number.

`F addr[number[start[end]]]`
`[bu(sy)`
`di(rect)`
`fr(ee)`
`i(ndirect)`
`t(emp) u(n)b(usy)`
` u(n)l(oaded)]`
`[<type>[type>...]]`
	Find references to `addr` in the heap blocks that match parameters. Example: `F 0x4456A 3 ul picture`.

`FL addr[filename]`	Display the stream produced by flattening this shape. Example: `FL 0x3321A “flat shapes”`.
`GG`	Display graphics globals
`HC`	Check the heap.

`HD`
`[bu(sy)`
`di(rect)`
`fr(ee)`
`i(ndirect)`
`t(emp) u(n)b(usy) u(n)l(oaded)]`
`HD [<type> [<type>...]]`
	Dump the heap or the heap parts that match parameters. Example: `HD bu line layout polygon`.

continued

Table 4-11 GraphicsBug commands and responses (continued)

Command	Response
`HT`	Total the heap.
`HX addr|<heapname>`	Switch to the heap containing `addr`, or named `<heapname>`. Example: `HX System`.
`HZ`	List the known heaps.
`IG`	Display initialization globals.
`LC (process)`	List the known graphics clients.
`LP`	List the known processes that have a graphics client.
`CG`	Display other (generic, nongraphic) globals.
`Q`	Quit.
`UF filename[page number]`	Display the contents of the file by flattening it. Use page number to specify a page of a print file.
`V [addr]`	Validate all (no parameters) or validate specific block.
`GG`	Graphics globals.
`WH addr`	Display the block containing `addr`. Operators: –, +, *, / , %, ^, | , &, [, @, *,], ~, (,) Numbers: .0x$#3 “strings: “”

In addition to the GraphicsBug commands above, you can Option-double-click (hold down the Option key and double-click) on a memory address to display memory as a type, use the up / down arrow keys to set the scrolling speed, use dot ‘.’ to represent the last displayed address, and use shape as an argument to the `DA`, `F`, and `HD` commands to display all graphics client-owned shapes.

Analyzing a Picture Shape

The following sections demonstrate the use of GraphicsBug for the analysis of a picture containing seven shapes. The code that creates the picture and the analysis of the data stream for each flattened shape is given in the section “Analyzing the Data Streams of Flattened Shapes” in the chapter “QuickDraw GX Stream Format.”

Determining the Heap Size for All Shapes in the Picture

You can use the GraphicsBug `HT` command to display the heap total in bytes for a specified graphics client heap. First run the application, then select the graphics client heap from the GraphicsBug heap menu, then apply the `HT` command. Listing 4-2 showsa sample output of the `HT` command: the GraphicsBug heap size in bytes. Note that the size of the graphics client and its heap is 86724 bytes. You can use this procedure to select the initial size of your application’s graphics client heap or heaps. For additional information about specifying the size of your graphics client heap, see the section “Creating a Graphics Client and its Graphics Client Heap” in the chapter “QuickDraw GX Memory Management.”

Listing 4-2 Totaling the graphics client and its heap

```
Totaling the heap at 00c07de8 (all shapes heap).
          Total Blocks                     Total of Block Sizes
Free      0000001b    #       27       0000fff8    #    65528
Direct    00000044    #       68       00001dbc    #     7612
Indirect  00000047    #       71       000031bc    #    12732
Sub Heaps 00000000    #        0       00000000    #        0
Heap Size 000000a6    #      166       000152c4    #    86724
```

Analyzing the Shapes in the Picture

You can use the GraphicsBug `HD PIC` command to display the memory locations of the seven shapes in the picture. Listing 4-3 shows the GraphicsBug output for the picture shape created by the application “all shapes.” User input is shown in boldface.

The GraphicsBug command lines shown in Listing 4-3 are used as follows:

- hd pic command turns `pic` into `picture.`
- `dm 00c0886c t` command displays default picture data.
- dm 00c0a4a0 t displays the data for the picture with seven shapes. Note that there are multiple text shapes displayed because the `gxUniqueItemsShape` attribute was set.

Listing 4-3 Determining the memory locations of the shapes in the picture

```
hx "all shapes"
heap set to 00c07de8  "all shapes"
hd pic
 Start    Length  Δ   Typ Busy Mstr Ptr Temp TBsy Disk   Object
00c0886c 00000048+00   i       00c1d02c                picture
00c0a4a0 00000108+00   i       00c1d010                picture
          Total Blocks            Total of Block Sizes
Blocks    00000002  #        2    00000150    #      336
dm 00c0886c t
displaying picture gxShape from 00c0886c
  devShape        nil
  owners            1
  seed              0
  flags      isDefaultShape
  attributes gxMapTransformShape
  gxStyle      00c083b0
  gxInk        00c08460
  gxTransform  00c088b4
  tagList         nil
  cacheList       nil
  geo.flags         0
  fillType   evenOddFill
  entries    0
  references 00000000
      gxShape        (type)       gxStyle           gxInk         gxTransform
dm 00c0a4a0 t
displaying picture gxShape from 00c0a4a0
  devShape   00c0a98c
  owners            1
  seed              0
  flags             0
  attributes
/*
There are multiple text shapes because the gxUniqueItemsShape attribute was
set.
*/
  gxStyle      00c083b0
  gxInk        00c08460
  gxTransform  00c088b4
  tagList         nil
  cacheList       nil
```

```
geo.flags        0
fillType   evenOddFill
entries    12
references 00c08d1c
    gxShape   (type)      gxStyle        gxInk      gxTransform
  00c08dd0  (line)       00000000       00000000       00000000
  00c0949c  (rectangle)  00000000       00000000       00000000
  00c099e4  (curve)      00000000       00000000       00000000
  00c09bd0  (path)       00000000       00000000       00000000
  00c0a220  (text)       00000000       00000000       00000000
  00c0a220  (text)       00c0a268       00c08e1c       00c0997c
  00c0a220  (text)       00c0a268       00c09b98       00c0a350
  00c0a220  (text)       00c0a268       00c0a640       00c0bc30
  00c0a220  (text)       00c0a268       00c0a678       00c0a6b0
  00c0a220  (text)       00c0a268       00c0a750       00c0a788
  00c0a828  (polygon)    00000000       00000000       00000000
  00c0bc94  (bitmap)     00000000       00000000       00000000
```

Analyzing the Rectangle in the Picture

You can use the `dm` command or Option-double-click command on the memory location of one of the seven shapes from Listing 4-3 to display information about the shape. Listing 4-4 shows the GraphicsBug output for the rectangle shape. The command line is shown in boldface.

Listing 4-4 Analyzing the rectangle shape in the picture

```
dm 00c0949c t
Displaying rectangle gxShape from 00c0949c
  devShape        nil
  owners            1
  seed              0
  flags             0
  attributes no attributes
  gxStyle      00c0984c
  gxInk        00c098fc
  gxTransform  00c0961c
  tagList         nil
  cacheList       nil
  geo.flags         0
  fillType   closedFrameFill
{   150.0000,    25.0000}  {   200.0000,    75.0000}
```

Analyzing the Ink in the Rectangle

You can select a memory location of one of the objects in the rectangle from Listing 4-4 and use the `dm` command or GraphicsBug Option-double click command to display information about the object. Listing 4-5 shows the GraphicsBug output for the ink in the rectangle shape. The command line is shown in boldface.

Listing 4-5 Analyzing the ink in the rectangle shape

```
dm 00c098fc t
Displaying gxInk from 00c098fc
  devInk       00c094e8
  privateFlags        0
  attributes          0
  owners              1
  seed                0
  tagList           nil
  space        gxRGBSpace
  profile           nil
  value(s)       1.0000 (ffff)  0.0000 0x0000  0.0000 0x0000
  mode         gxCopyMode
```

QuickDraw GX Debugging Reference

This section describes the data structures and routines that are specific to the QuickDraw GX debugging environment.

The “Constants and Data Types” section shows the enumerations and structures for drawing errors and GraphicsBug parameters. A cross-reference is provided to the enumerated validation levels.

Constants and Data Types

This section describes the constants and data structures that you use to provide information to debugging functions.

Drawing Errors

QuickDraw GX posts drawing errors when you use the `GXGetShapeDrawError` function after an unsuccessful drawing operation. The `gxDrawError` enumeration defines the posted drawing errors.

```
enum gxDrawErrors {
   no_draw_error,

   /* gxShape type errors */
   shape_emptyType,
   shape_inverse_fullType,
   rectangle_zero_width,
   rectangle_zero_height,
   polygon_empty,
   path_empty,
   bitmap_zero_width,
   bitmap_zero_height,
   text_empty,
   glyph_empty,
   layout_empty,
   picture_empty,

   /* general gxShape errors */
   shape_no_fill,
   shape_no_enclosed_area,
   shape_no_enclosed_pixels,
   shape_very_small,
   shape_very_large,
   shape_contours_cancel,

   /* gxStyle errors */
   pen_too_small,
   text_size_too_small,
   dash_empty,
   start_cap_empty,
   pattern_empty,
   textFace_empty,
   shape_primitive_empty,
   shape_primitive_very_small,
/* gxInk errors */
transfer_equals_noMode,
transfer_matrix_ignores_source,
transfer_matrix_ignores_device,
transfer_source_reject,
transfer_mode_ineffective,
colorSet_no_entries,
bitmap_colorSet_one_entry,

/* gxTransform errors */
transform_scale_too_small,
transform_map_too_large,
transform_move_too_large,
transform_scale_too_large,
transform_rotate_too_large,
transform_perspective_too_large,
transform_skew_too_large,
transform_clip_no_intersection,
transform_clip_empty,
transform_no_viewPorts,

/* gxViewPort errors */
viewPort_disposed,
viewPort_clip_empty,
viewPort_clip_no_intersection,
viewPort_scale_too_small,
viewPort_map_too_large,
viewPort_move_too_large,
viewPort_scale_too_large,
viewPort_rotate_too_large,
viewPort_perspective_too_large,
viewPort_skew_too_large,
viewPort_viewGroup_offscreen,

/* gxViewDevice errors */
viewDevice_clip_no_intersection,
viewDevice_scale_too_small,
viewDevice_map_too_large,
viewDevice_move_too_large,
viewDevice_scale_too_large,
viewDevice_rotate_too_large,
   viewDevice_perspective_too_large,
   viewDevice_skew_too_large
};
```

```
typedef long gxDrawError;
```

Table 4-2 through Table 4-7 list the drawing errors and give a description of each error.

Validation Levels

The `GXSetValidation` function uses the `gxValidationLevel` enumeration to turn off or to control the QuickDraw GX validation.

```
typedef long gxValidationLevel;
```

```
enum gxValidationLevels {
/*
These levels tell how to validate routines.  Choose one.
*/
   gxNoValidation               = 0x00,
   gxPublicValidation           = 0x01,
   gxInternalValidation         = 0x02,
/*
These levels tell how to validate types.  Choose one.
*/
   gxTypeValidation             = 0x00,
   gxStructureValidation        = 0x10,
   gxAllObjectValidation        = 0x20,
/*
These levels tell how to validate memory manager blocks.  Choose
any combination.
*/
   gxNoMemoryManagerValidation = 0x0000,
   gxApBlockValidation          = 0x0100,
   gxFontBlockValidation        = 0x0200
   gxApHeapValidation           = 0x0400,
   gxFontHeapValidation         = 0x0800,
   gxCheckApHeapValidation      = 0x1000,
   gxCheckFontHeapValidation    = 0x2000
} ;
```

Field descriptions

`gxNoValidation`
: If set, QuickDraw GX performs no validation checking.

`gxPublicValidation`
: If set, QuickDraw GX checks parameters to public routines.

`gxInternalValidation`
: If set, QuickDraw GX checks parameters to internal routines.

`gxTypeValidation`
: If set, QuickDraw GX checks types of objects.

`gxStructureValidation`
: If set, QuickDraw GX checks fields of private structures.

`gxAllObjectValidation`
: If set, QuickDraw GX checks every object for each public routine called.

`gxNoMemoryManagerValidation`
: If set, QuickDraw GX does not check Memory Management calls.

`gxApBlockValidation`
: If set, QuickDraw GX checks the relevant block structures before each Memory Manager call.

`gxFontBlockValidation`
: If set, QuickDraw GX also checks the system heap block structures..

`gxApHeapValidation`
: If set, QuickDraw GX also checks all application heap blocks every time the heap changes.

`gxFontHeapValidation`
: If set, QuickDraw GX also checks all system heap blocks every time the heap changes..

`gxCheckApHeapValidation`
: If set, QuickDraw GX also checks all application heap blocks for each public or internal routine called.

`gxCheckFontHeapValidation`
: If set, QuickDraw GX also checks the system heap blocks for each public or internal routine called.

For information on how to use QuickDraw GX validation, see the section “Using Validation Functions” beginning on page 4-15. The `GXSetValidation` function is described on page 4-34.

Functions

The functions described in this section allow you to detect drawing errors, perform validation, and install debugging utility functions.

Obtaining Drawing Errors

This section describes the function that allows you to obtain a single error message that describes why a shape did not draw correctly.

GXGetShapeDrawError

You can use the `GXGetShapeDrawError` function to determine why a shape failed to draw.

```
gxDrawError GXGetShapeDrawError(gxShape source);
```

`source`	A reference to the shape that didn't draw.
function result	An error result code indicating why a shape didn't draw.

DESCRIPTION

The `GXGetShapeDrawError` function returns a single error code that indicates why a shape didn't draw. The error returneddepends on the step in the drawing process in which the drawing error occurred. QuickDraw GX returns the first drawing error it detects in the drawing process. A drawing error that may occur later in the drawing process is not returned until all prior drawing errors detected are resolved.

If you run your application and it does not draw what you expect, you can add the `GXGetShapeDrawError` function to the end of your application code and rerun your application. QuickDraw GX returns a single error from the `gxDrawErrors` enumeration that may assist in determining the drawing problem. If a drawing error is not detected, QuickDraw GX returns a `gxNoDrawError` error.

SEE ALSO

The use of the `GXGetShapeDrawError` is discussed in the section “Analyzing Drawing Problems” beginning on page 4-8.

The `gxDrawError` enumeration is described in the section “Drawing Errors” beginning on page 4-29.

Table 4-2 through Table 4-7 provide a description of each drawing error.

Table 4-1 gives the object processing sequence that determines which drawing error is posted.

Setting and Getting Validation Options and Errors

This section describes the functions that control QuickDraw GX validation. QuickDraw GX validation checks public and internal function parameters to ensure that they are valid. You can use validation functions and flag options to check types, structures, all objects, memory, and specific objects.

When validation error checking is on, QuickDraw GX may post the validation errors listed in the section "Debugging Version" in the chapter "Errors, Warnings, and Notices."

GXSetValidation

You can use the `GXSetValidation` function to control the type and level of validation checking.

```
void GXSetValidation(gxValidationLevel);
```

`gxValidationLevel`
The validation flags.

DESCRIPTION

The `GXSetValidation` function allows you to set the validation mode, as well as the validation levels, for type, structure, all object, and memory block validation options. You may pick one mode, one level, and any combination of memory options. The options are defined by the `gxValidationLevel` enumeration.

The `GXSetValidation` function turns validation on when you select any flags other than 0x00. If you set the `gxValidationLevel` flag to `gxNoAttributes`, validation is off.

This function has no effect in the non-debugging version of QuickDraw GX.

As an alternative to the use of the `GXSetValidation` function with the internal and all object validation flags set, you can use the `GXValidateAll` function.

SEE ALSO

To get the current `gxValidationLevel` parameter, use the `GXGetValidation` function, described on page 4-35.

The `gxValidationLevel` enumeration is described in the section "Validation Levels" beginning on page 4-31.

Table 4-8 on page 4-16 lists the public and internal validation options.

Table 4-9 on page 4-16 lists the type, structure, and all object validation options.

Table 4-10 on page 4-19 lists the memory validation options.

The `GXValidateAll` function is described on page 4-43.

GXGetValidation

You can use the `GXGetValidation` function to obtain the current validation flags that are set.

```
gxValidationLevel GXGetValidation(void);
```

function result The current flags set for validation error checking.

DESCRIPTION

The `GXGetValidation` function returns the `gxValidationLevel` parameter set by the `GXSetValidation` function.

This function always returns 0 in the non-debugging version of QuickDraw GX.

SEE ALSO

The `GXSetValidation` function is described in the previous section.

GXGetValidationError

You can use the `GXGetValidationError` function to determine the application function and parameter that caused the last validation error.

```
void GXGetValidationError(char *procedureName, void **argument,
                           long *argumentNumber);
```

`procedureName`
A pointer to the name of the function that produced the validation error.

`argument` A pointer to a list of the function's arguments.

`argumentNumber`
A pointer to the number of the argument that produced the validation error.

DESCRIPTION

The `GXGetValidationError` function provides the name of the function, a list of the function's parameters, and the number of the parameter that produced the last validation error. The `argumentNumber` parameter for the *n*th parameter is *n*. For example, the `argumentNumber` for the third parameter is 3. If you call the `GXGetValidationError` function and no validation errors have been posted, the function returns `nil`.

This function leaves its arguments unchanged in the non-debugging version of QuickDraw GX.

SEE ALSO

The use of the `GXGetValidationError` function is described in the section "Analyzing the Cause of Validation Errors" beginning on page 4-21.

Validating Objects

This section describes the functions that allow you to validate the function parameters of allocated QuickDraw GX objects. QuickDraw GX provides functions for specific object validation and all object validation.

When validation error checking is on, QuickDraw GX may post the validation errors listed in the section "Debugging Version" in the chapter "Errors, Warnings, and Notices."

GXValidateShape

You can use the `GXValidateShape` function to check the parameters of a shape object.

```
void GXValidateShape(gxShape target);
```

`target`	A reference to a shape object to be validated.

DESCRIPTION

The `GXValidateShape` function checks parameters for the type, geometry, fill, style, ink, transform, attributes, owner count, and tag list properties for all shape objects. In addition, this function checks any internal caches built for the shape. If one or more of the parameters are not valid, a validation error is posted.

This function is not operational in the non-debugging version of QuickDraw GX.

SEE ALSO

QuickDraw GX validation is introduced in the section "Validation Functions" beginning on page 4-6. The use of validation functions is discussed in the section "Controlling Validation" beginning on page 4-15.

GXValidateStyle

You can use the `GXValidateStyle` function to check the parameters of a style object.

```
void GXValidateStyle(gxStyle target);
```

`target`	A reference to a style object to be validated.

DESCRIPTION

The `GXValidateStyle` function checks parameters for the pen size, cap, join, dash, pattern, curve error, and attributes properties for all graphics style objects. It also checks parameters for the text face, text size, justification, font variations, platform, and text attributes properties for all typographic style objects. In addition, it confirms parameters for the run controls, run features array, glyph substitutions array, kerning adjustments, priority justification override, and glyph justification overrides array typographic properties for all layout shapes objects. In addition, this function checks any internal caches built for the style. If one or more parameters are not valid, QuickDraw GX posts a validation error.

This function is not operational in the non-debugging version of QuickDraw GX. If a discrepancy is found, QuickDraw GX posts an error.

SEE ALSO

QuickDraw GX validation is introduced in the section “Validation Functions” beginning on page 4-6. The use of validation functions is described in the section “Controlling Validation” beginning on page 4-15.

GXValidateInk

You can use the `GXValidateInk` function to check the parameters of an ink object.

```
void GXValidateInk(gxInk target);
```

`target`	A reference to an ink object to be validated.

DESCRIPTION

The `GXValidateInk` function checks parameters for the color, transfer mode, attributes, owner count, and tag list properties for all ink objects. In addition, this function checks any internal caches built for the ink. If one or more of the parameters are not valid, QuickDraw GX posts a validation error.

This function is not operational in the non-debugging version of QuickDraw GX.

SEE ALSO

QuickDraw GX validation is introduced in the section “Validation Functions” beginning on page 4-6. The use of validation functions is described in the section “Controlling Validation” beginning on page 4-15.

GXValidateTransform

You can use the `GXValidateTransform` function to check the parameters of a transform object.

```
void GXValidateTransform(gxTransform target);
```

`target`	A reference to a transform object to be validated.

DESCRIPTION

The `GXValidateTransform` function checks the parameters for the clip, mapping, view port list, hit-test parameters, attributes, owner count, and tag list properties for all transform objects. In addition, this function checks any internal caches built for the transform. If one or more of the parameters are not valid, QuickDraw GX posts a validation error.

This function is not operational in the non-debugging version of QuickDraw GX.

SEE ALSO

QuickDraw GX validation is introduced in the section “Validation Functions” beginning on page 4-6. The use of validation functions is described in the section “Controlling Validation” beginning on page 4-15.

GXValidateColorSet

You can use the `GXValidateColorSet` function to check the parameters of a color set object.

```
void GXValidateColorSet(gxColorSet target);
```

`target`	A reference to a color set object to be validated.

DESCRIPTION

The `GXValidateColorSet` function checks parameters for the color space, color-value array, owner count, and tag list properties for all color set objects. In addition, this function checks any internal caches built for the color set. If one or more of the parameters are not valid, QuickDraw GX posts a validation error.

This function is not operational in the non-debugging version of QuickDraw GX.

SEE ALSO

QuickDraw GX validation is introduced in the section “Validation Functions” beginning on page 4-6. The use of validation functions is described in the section “Controlling Validation” beginning on page 4-15.

GXValidateColorProfile

You can use the `GXValidateColorProfile` function to check the parameters of a color profile object.

```
void GXValidateColorProfile(gxColorProfile target);
```

`target`	A reference to a color profile object to be validated.

DESCRIPTION

The `GXValidateColorProfile` function checks the content of the target color profile object. In addition, this function checks any internal caches built for the color profile. If one or more of the parameters are not valid, QuickDraw GX posts a validation error.

This function is not operational in the non-debugging version of QuickDraw GX.

SEE ALSO

QuickDraw GX validation is introduced in the section “Validation Functions” beginning on page 4-6. The use of validation functions is described in the section “Controlling Validation” beginning on page 4-15.

GXValidateTag

You can use the `GXValidateTag` function to check the parameters of a tag object.

```
void GXValidateTag(gxTag target);
```

`target`	A reference to a tag object to be validated.

DESCRIPTION

The `GXValidateTag` function checks the parameters for the tag type, size, contents, and owner count properties for all tag objects. In addition, this function checks any internal caches built for the tag. If one or more of the parameters are not valid, QuickDraw GX posts a validation error.

This function is not operational in the non-debugging version of QuickDraw GX.

SEE ALSO

QuickDraw GX validation is introduced in the section “Validation Functions” beginning on page 4-6. The use of validation functions is described in the section “Controlling Validation” beginning on page 4-15.

GXValidateViewDevice

You can use the `GXValidateViewDevice` function to check the parameters of a view device object.

```
void GXValidateViewDevice(gxViewDevice target);
```

`target`	A reference to a view device object to be validated.

DESCRIPTION

The `GXValidateViewDevice` function checks parameters for the clip, mapping, bitmap, attributes, and tag list properties for all view device objects. In addition, this function checks any internal caches built for the view device. If one or more of the parameters are not valid, QuickDraw GX posts a validation error.

This function is not operational in the non-debugging version of QuickDraw GX.

SEE ALSO

QuickDraw GX validation is introduced in the section “Validation Functions” beginning on page 4-6. The use of validation functions is described in the section “Controlling Validation” beginning on page 4-15.

GXValidateViewPort

You can use the `GXValidateViewPort` function to check the parameters of a view port object.

```
void GXValidateViewPort(gxViewPort target);
```

`target`	A reference to a view port object to be validated.

DESCRIPTION

The `GXValidateViewPort` function checks parameters for the clip, mapping, dither, halftone, parent view port, child view port list, view device, attributes, owner count, and tag list properties for all view port objects. In addition, this function checks any internal caches built for the view port. If one or more of the parameters are not valid, QuickDraw GX posts a validation error.

This function is not operational in the non-debugging version of QuickDraw GX.

SEE ALSO

QuickDraw GX validation is introduced in the section "Validation Functions" beginning on page 4-6. The use of validation functions is described in the section "Controlling Validation" beginning on page 4-15.

GXValidateViewGroup

You can use the `GXValidateViewGroup` function to check the parameters of a view group object.

```
void GXValidateViewGroup(gxViewGroup target);
```

`target`	A reference to a view group object to be validated.

DESCRIPTION

The `GXValidateViewGroup` function checks parameters for the clip, mapping, dither, halftone, parent view port, child view port list, view device, attributes, owner count, and tag list properties of the view port object and the clip, mapping, bitmap, attributes, and tag list properties of the view device object. In addition, this function checks any internal caches built for the view group. If one or more of the parameters are not valid, QuickDraw GX posts a validation error.

This function is not operational in the non-debugging version of QuickDraw GX.

SEE ALSO

QuickDraw GX validation is introduced in the section "Validation Functions" beginning on page 4-6. The use of validation functions is described in the section "Controlling Validation" beginning on page 4-15.

GXValidateGraphicsClient

You can use the `GXValidateGraphicsClient` function to check the parameters of a graphics client object.

```
void GXValidateGraphicsClient(gxGraphicsClient target);
```

`target`	A reference to a graphics client object to be validated.

DESCRIPTION

The `GXValidateGraphicsClient` checks all parameters for all properties of a graphics client object. In addition, this function checks any internal caches built for the graphics client. If one or more of the parameters are not valid, QuickDraw GX posts a validation error.

This function is not operational in the non-debugging version of QuickDraw GX.

SEE ALSO

QuickDraw GX validation is introduced in the section “Validation Functions” beginning on page 4-6. The use of validation functions is described in the section “Controlling Validation” beginning on page 4-15.

GXValidateAll

You can use the `GXValidateAll` function to validate all objects that are allocated.

```
void GXValidateAll(void);
```

DESCRIPTION

The `GXValidateAll` function allows you to validate the parameters of all objects that are allocated in the QuickDraw GX heap. It also checks additional structures in the backing store. In addition, this function checks any internal caches built for the objects. If one or more of the parameters are not valid, QuickDraw GX posts a validation error.

This function is not operational in the non-debugging version of QuickDraw GX.

An alternative method of validating all of the objects in the heap is to use the `GXSetValidation` function with the `gxValidationLevel` parameter set to the `gxPublicValidation` plus `gxAllObjectValidation` options.

SEE ALSO

QuickDraw GX validation is introduced in the section "Validation Functions" beginning on page 4-6. The use of validation functions is described in the section "Controlling Validation" beginning on page 4-15.

The `GXSetValidation` function is described on page 4-34.

Summary of QuickDraw GX Debugging

Constants and Data Types

Drawing Errors

```
typedef long gxDrawError;

enum gxDrawErrors {
   no_draw_error,

   /* gxShape type errors */
   shape_emptyType,
   shape_inverse_fullType,
   rectangle_zero_width,
   rectangle_zero_height,
   polygon_empty,
   path_empty,
   bitmap_zero_width,
   bitmap_zero_height,
   text_empty,
   glyph_empty,
   layout_empty,
   picture_empty,

   /* general gxShape errors */
   shape_no_fill,
   shape_no_enclosed_area,
   shape_no_enclosed_pixels,
   shape_very_small,
   shape_very_large,
   shape_contours_cancel,

   /* gxStyle errors */
   pen_too_small,
   text_size_too_small,
   dash_empty,
   start_cap_empty,
   pattern_empty,
   textFace_empty,
shape_primitive_empty,
shape_primitive_very_small,

/* gxInk errors */
transfer_equals_noMode,
transfer_matrix_ignores_source,
transfer_matrix_ignores_device,
transfer_source_reject,
transfer_mode_ineffective,
colorSet_no_entries,
bitmap_colorSet_one_entry,

/* gxTransform errors */
transform_scale_too_small,
transform_map_too_large,
transform_move_too_large,
transform_scale_too_large,
transform_rotate_too_large,
transform_perspective_too_large,
transform_skew_too_large,
transform_clip_no_intersection,
transform_clip_empty,
transform_no_viewPorts,

/* gxViewPort errors */
viewPort_disposed,
viewPort_clip_empty,
viewPort_clip_no_intersection,
viewPort_scale_too_small,
viewPort_map_too_large,
viewPort_move_too_large,
viewPort_scale_too_large,
viewPort_rotate_too_large,
viewPort_perspective_too_large,
viewPort_skew_too_large,
viewPort_viewGroup_offscreen,

/* gxViewDevice errors */
viewDevice_clip_no_intersection,
viewDevice_scale_too_small,
viewDevice_map_too_large,
viewDevice_move_too_large,
viewDevice_scale_too_large,
   viewDevice_rotate_too_large,
   viewDevice_perspective_too_large,
   viewDevice_skew_too_large
};
```

Validation Levels

```
typedef long gxValidationLevel;

enum gxValidationLevels {
/*
These levels tell how to validate routines.  Choose one.
*/
   gxNoValidation = 0x00,        /* no validation */
   gxPublicValidation = 0x01,    /* check parameters to public routines */
   gxInternalValidation = 0x02,  /* check parameters to internal routines */

/*
These levels tell how to validate types.  Choose one.
*/
   gxTypeValidation = 0x00,      /* check types of objects */
   gxStructureValidation = 0x10, /* check fields of private structures */
   gxAllObjectValidation = 0x20, /* check every object over every call */

/*
These levels tell how to validate memory manager blocks.  Choose any 
combination.
*/
   gxNoMemoryManagerValidation = 0x0000,/* no memory validation */
   gxApBlockValidation = 0x0100,       /* check the relevant block
                                        structures after each Memory Manager
                                        call */
   gxFontBlockValidation = 0x0200/* check the system gxHeap as well */
   gxApHeapValidation = 0x0400,  /* check the memory manager’s gxHeap after
                                 every memory call */
   gxFontHeapValidation= 0x0800, /* also check the system gxHeap */
   gxCheckApHeapValidation = 0x1000,
                                 /* check the memory manager’s
                                  gxHeap if checking routine
                                  parameters */
   gxCheckFontHeapValidation = 0x2000
                                  /* check the system gxHeap as
                                   well */
} ;
```

Functions

Obtaining Drawing Errors

```
gxDrawError GXGetShapeDrawError
                             (gxShape source);
```

Setting and Getting Validation Options and Errors

```
void GXSetValidation         (gxValidationLevel);
gxValidationLevel GXGetValidation
                             (void);
void GXGetValidationError    (char *procedureName, void **argument,
                              long *argumentNumber);
```

Validating Objects

```
void GXValidateShape         (gxShape target);
void GXValidateStyle         (gxStyle target);
void GXValidateInk           (gxInk target);
void GXValidateTransform     (gxTransform target);
void GXValidateColorSet      (gxColorSet target);
void GXValidateColorProfile
                             (gxColorProfile target);
void GXValidateTag           (gxTag target);
void GXValidateViewDevice    (gxViewDevice target);
void GXValidateViewPort      (gxViewPort target);
void GXValidateViewGroup     (gxViewGroup target);
void GXValidateGraphicsClient
                             (gxGraphicsClient target);
void GXValidateAll           (void);
```

C H A P T E R 5

Collection Manager

Contents

This chapter describes the Collection Manager, which provides an abstract data type you can use to store collections of information. Read this chapter if you need to work with some advanced features of QuickDraw GX printing, including print dialog boxes, or if you want to create collections for purposes specific to your application.

Before reading this chapter, you might want to familiarize yourself with the information in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*. For some examples of how the Collection Manager is used by other parts of QuickDraw GX, you should read the chapter "Page Formatting and Dialog Box Customization" in *Inside Macintosh: QuickDraw GX Printing*.

This chapter introduces the collection object as an abstract data type and describes the properties of this object. It then shows how to use the functions provided by the Collection Manager to

- create and manipulate collection objects
- add information to a collection object
- retrieve information from a collection object
- store a collection object to disk and retrieve a collection object from disk

This chapter also contains reference information for all data types, functions, and resources associated with the Collection Manager.

About the Collection Manager

The Collection Manager implements an abstract data type that allows you to store multiple pieces of related information. This abstract data type is called a collection object.

Collection Objects

A **collection object,** or simply a **collection,** is an abstract data type that allows you to store information. A collection is like an array in that it contains a number of individually accessible items. However, a collection offers some advantanges over an array:

- A collection allows for a variable number of data items. You can add items to a collection or remove items from a collection during run time, and the Collection Manager automatically resizes the collection.
- A collection allows for variable-size items. Each item in a collection can contain data of any size.

There are some corresponding disadvantages to using a collection versus using an array:

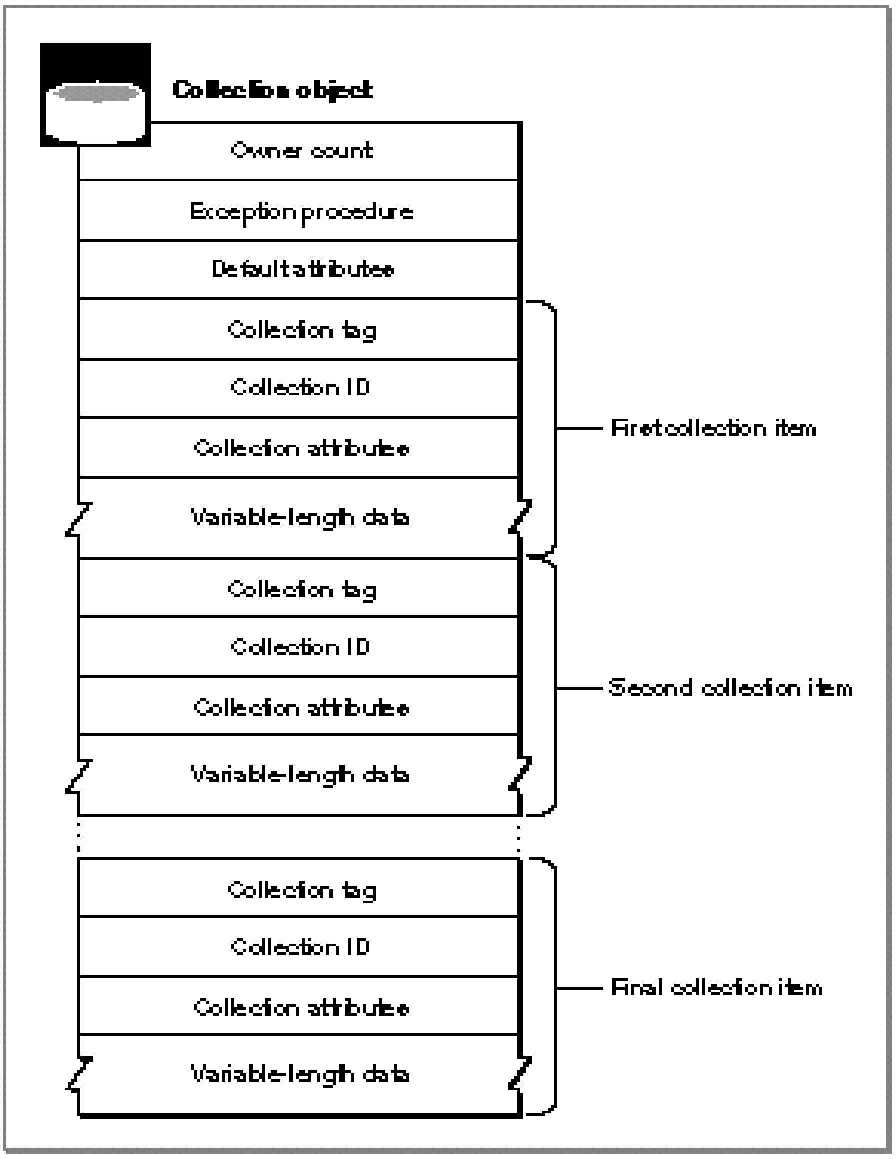
- You must store and retrieve information in a collection using Collection Manager functions, which is not as efficient as accessing an item of an array.
- The Collection Manager stores extra information about the collection and about each item in the collection, so a collection requires more memory than a comparable array.

A collection is also similar to a database, in that you can store information and retrieve it using a variety of search mechanisms. However, a collection has many more limitations than a real database. For example, the Collection Manager provides only a few mechanisms for searching a collection. Also, a collection is entirely memory-based. You can use a collection only when the entire contents of the collection are in memory, which makes a collection more like a powerful array than a database.

The internal structure of a collection object is private—you must store information in a collection and retrieve information from it by providing a Collection Manager function with a reference to the collection.

Figure 5-1 depicts the accessible properties of a collection object. Note that, because a collection is an object and not a public data structure, the order of the properties as shown is completely arbitrary.

Figure 5-1 The collection object



As Figure 5-1 shows, a collection object contains

- an **owner count,** which reflects the current number of references to the collection
- an exception procedure, which you can use to handle errors that occur while operating on the collection
- default attributes, which are described in “Collection Attributes” beginning on page 5-9.
- a number of collection items, which are described in the next section

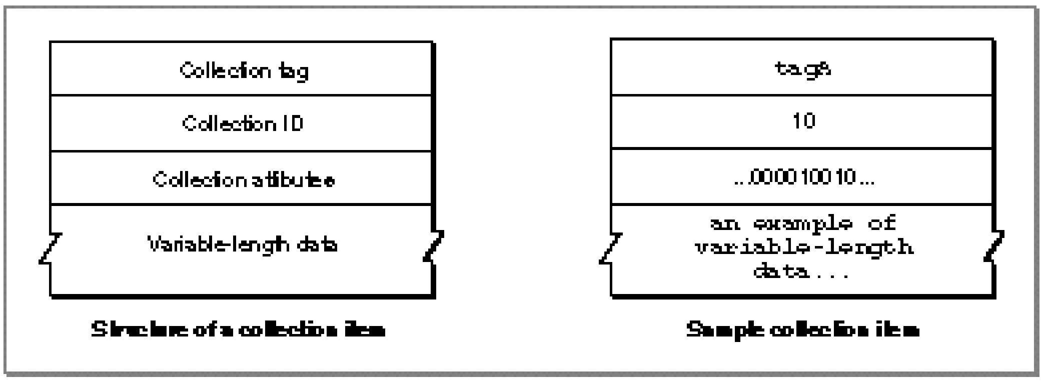
The Collection Manager maintains the owner count for you, although you can increment or decrement it by cloning or disposing of the collection, as described in “Creating or Disposing of a Collection” beginning on page 5-14 and “Cloning or Copying a Collection” beginning on page 5-14.

The Collection Manager allows you to install an **exception procedure** for each collection object. When the Collection Manager is operating on a collection and an error occurs, the Collection Manager calls the collection’s exception procedure (if you installed one) and passes to it the result code associated with the error that occurred. Your exception procedure can then respond to the error. For more information about exception procedures, see “Getting and Setting the Exception Procedure for a Collection” beginning on page 5-58 and the description of an application-defined exception procedure on page 5-101.

Collection Items

A collection is composed of **collection items.** Figure 5-2 shows the general structure of a collection item and also shows a sample collection item. Note that, because a collection item is always part of a collection object, you always access the information in a collection item using Collection Manager functions. Therefore, the order of the properties shown in Figure 5-2 is completely arbitrary.

Figure 5-2 The collection item



As Figure 5-2 shows, each collection item contains these properties:

- **Collection tag.** A collection tag is a four-character identifier that, in conjunction with the collection ID, uniquely identifies the collection item.
- **Collection ID.** A collection ID is a `long` value that, in conjunction with the collection tag, uniquely identifies the collection item.

- **Collection attributes.** The collection attributes are a set of 32 bit flags, each of which represents an attribute of the collection item. The Collection Manager defines the meaning of some of these attributes and leaves some of them for you to define. See the next section for more information about collection attributes.
- **Variable-length data.** The variable-length data contains the actual data of the item. This data corresponds to the contents of an item in an array, except items in the same collection can store data of different sizes, whereas items in a single array must all store data of the same size.

The Collection Manager uses a collection item's collection tag and collection ID to uniquely identify the item. As an example, in any collection there can be exactly one item with a collection tag of `'tagA'` and a collection ID of 10.

When you want to retrieve the data stored in an item, you can specify the item by providing a Collection Manager function with the item's collection tag and collection ID. You can also specify a collection item using one of the other methods provided by the Collection Manager. See "Methods of Identifying Collection Items" beginning on page 5-11 for more information.

Collection Attributes

Each collection item has 32 **attributes.** Each attribute is represented by one bit flag in the item's attributes property. Therefore each attribute is either set or clear. An item's attributes are stored in a 32-bit long word. The bits are numbered 0 through 31. Bit 31 is the high bit.

The upper 16 bits of an item's attributes property represent attributes that are reserved for use by Apple Computer, Inc. Currently, two of these attributes are defined:

- Bit 31 represents the **lock attribute.** When an item has this attribute set, attempts to replace the item result in an error. When this attribute is clear, you can replace the item.
- Bit 30 represents the **persistence attribute.** When an item has this attribute set, the Collection Manager includes this item when flattening the collection. When this attribute is clear, the Collection Manager ignores the item when flattening the collection. See "Flattening and Unflattening a Collection" beginning on page 5-37 for more information about flattening collections.

The other 14 reserved attribute bits are called the **reserved attributes.**

The lower 16 bits of an item's attributes property represent attributes that you can define for purposes suitable to your application. For example, you could use one of these attributes to mark all of the items that you wanted to write to disk and remove from the collection should you need more memory. These 16 attributes are called the **user attributes.**

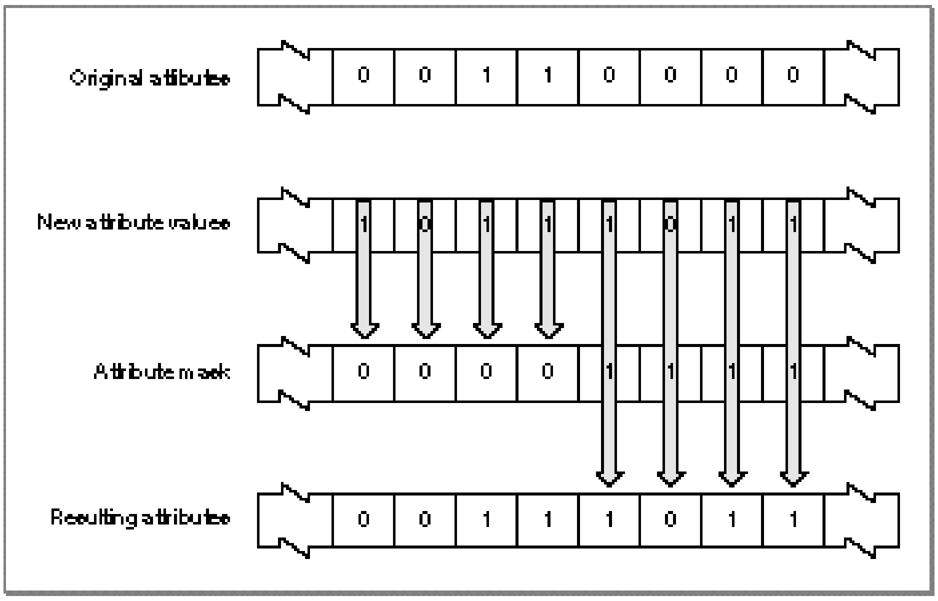
Depending on your application, you can set and examine the user attributes individually, or you can set and examine combinations of them. As an example, if your application uses collections that contain four distinct types of items, you could combine two user attributes to provide the four values (0–3) necessary to identify the four types of items.

Every collection object contains **default attributes.** A collection's default attributes determine the initial attribute values assigned to items added to that collection. For example, you could set the lock and persistence default attributes for a collection. From then on, when you added an item to the collection, the new item would have its lock and persistence attributes set. Of course, you would still be free to edit the attributes for the new item after adding it to the collection.

The Collection Manager provides a mechanism for editing attributes that allows you to set (or clear) the values of some attributes while leaving the values of other attributes alone. To edit attributes, you provide an **attribute mask,** in which you specify the attributes you want to edit, and new attribute values, in which you specify the new values for the attributes.

Figure 5-3 depicts this editing mechanism.

Figure 5-3 Editing attributes in a collection item



When editing an item's attributes, you provide an attribute mask and new attribute values. For every attribute:

- If you set the corresponding bit of the attribute mask to 0, the Collection Manager leaves the attribute unchanged from the original. The new value for the attribute (which you provide in the new attribute values) is ignored.
- If you set the corresponding bit in the attribute mask to 1, the Collection Manager copies the new attribute value you provide for this attribute. The original value of this attribute is overwritten.

You use this mechanism when editing an item's attributes, when editing a collection's default attributes, when searching for items whose attributes match a certain pattern, when flattening parts of a collection, and when purging items from a collection. For an example, see "Changing the Default Attributes of a Collection" beginning on page 5-15.

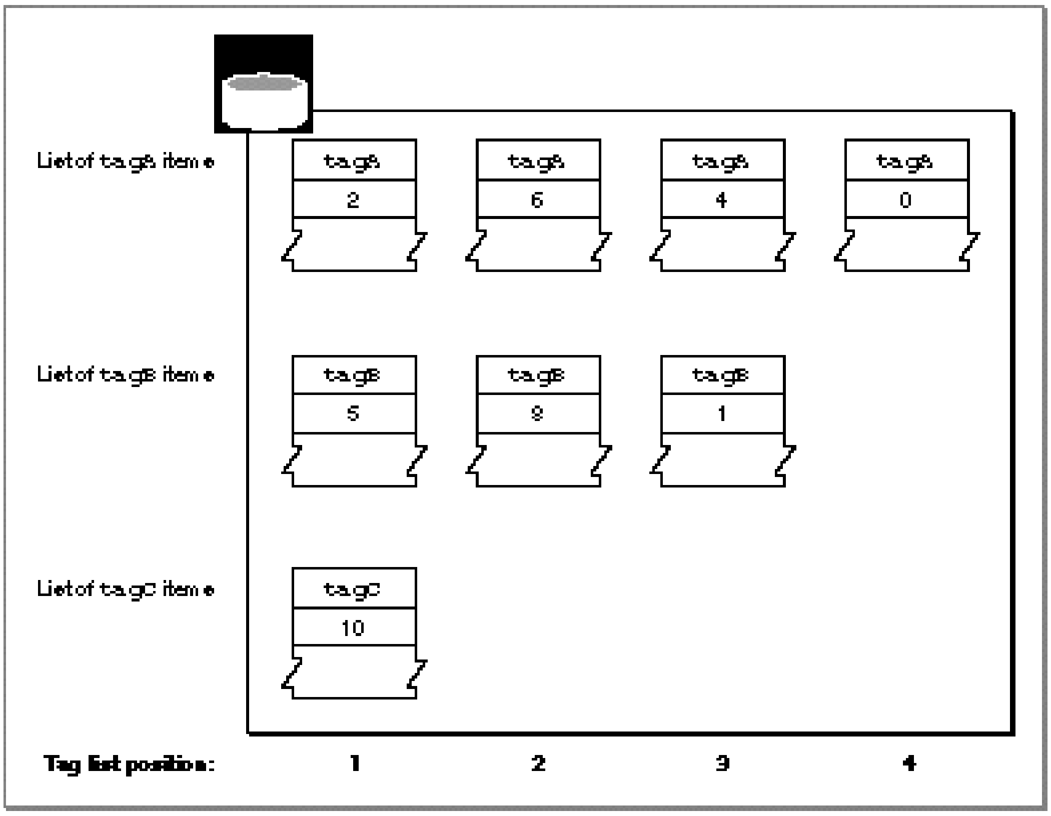
Methods of Identifying Collection Items

Many Collection Manager functions operate on an individual item within a collection. For example, the Collection Manager provides functions that allow you to replace the variable-length data for a particular item as well as functions that allow you to retrieve an item's variable-length data.

When calling these Collection Manager functions, you need to specify which collection item you want to examine or edit. The Collection Manager provides three methods of specifying a particular item in a collection:

- The collection tag and the collection ID. Together, these two properties uniquely identify an item in a collection. The collection IDs of collection items with the same collection tag do not have to be sequential. For example, the collection shown in Figure 5-4 has four items with the `'tagA'` collection tag. These four items have collection IDs 2, 6, 4, and 0.
- The collection tag and the tag list position. Each item in a collection has a tag list position as well as a collection ID. The **tag list position** of an item is the position of the item in the list of items with the same collection tag. Unlike a collection ID, the tag list positions of items with the same tag are sequential. For example, in Figure 5-4 the four items with the `'tagA'` collection tag have tag list positions 1, 2, 3, and 4. Unlike the collection ID, the tag list position of an item can change if another item with the same collection tag is added to or removed from the collection.
- The collection index. The Collection Manager assigns a **collection index** to each item in a collection. A collection index uniquely identifies its item within a collection. Indexes across a collection do not have to be sequential. The collection index of any item in a collection can change if another item is added to or removed from the collection.

Figure 5-4 Items in a collection



In Figure 5-4, the third item in the second row can be identified in three ways:

- It has a collection tag of `'tagB'` and a collection ID of 1.
- It has a collection tag of `'tagB'` and a tag list position of 3.
- It has a unique collection index assigned to it by the Collection Manager.

For examples of identifying collection items, see "Adding Items to a Collection" beginning on page 5-17, "Determining the Collection Index of an Item" beginning on page 5-19, and "Determining the Tag and ID of an Item" beginning on page 5-21.

Using the Collection Manager

This section describes how your application can

- create or dispose of a collection
- clone or copy a collection
- change the default attributes of a collection

- add items to a collection
- determine the collection index of an item
- determine the collection tag and collection ID of an item
- determine the size of an item's variable-length data
- get and set the attributes of a collection item
- replace items in a collection
- remove items from a collection
- retrieve the variable-length data from a collection item
- examine the collection tags of a collection
- flatten and unflatten collections
- read collections from and write collections to disk

Determining Whether the Collection Manager Is Available

The Collection Manager is not available in all system software versions. Therefore, before calling any Collection Manager functions, you should use the `Gestalt` function to determine whether the Collection Manager is available. To get information about the Collection Manager, you pass the `Gestalt` function the `gestaltCollectionMgrVersion` selector, as shown in Listing 5-1.

Listing 5-1 Determining whether the Collection Manager is available

```
Boolean CollectionMgrExists(long versionRequired) {

   long collectionMgrVersion;

   Boolean exists = (Gestalt(gestaltCollectionMgrVersion,
                             &collectionMgrVersion) == noErr);
   if (exists)
      exists = (collectionMgrVersion >= versionRequired);
   return(exists);
}
```

In Listing 5-1, the `CollectionMgrExists` sample function uses the `Gestalt` function to determine whether the Collection Manager is available and, if so, which version is available. If the Collection Manager is available, this sample function tests whether the version number is sufficiently high by comparing it with a specified minimum.

You would call this sample function with a line of code such as

```
exists = CollectionMgrExists(0x01008000); /* version 1.0f0 */
```

You can find out more about the `Gestalt` function in the chapter "Gestalt Manager" of *Inside Macintosh: Operating System Utilities*.

Creating or Disposing of a Collection

The Collection Manager provides a number of ways for you to create a collection object:

- You can create a new collection object using the `NewCollection` function, as described in this section.
- You can make a copy of an existing collection object using the `CopyCollection` function, as described in "Cloning or Copying a Collection" beginning on page 5-14.
- You can create a collection from a resource using the `GetNewCollection` function, as described in "Reading Collections From and Writing Collections to Disk" beginning on page 5-41.

The `NewCollection` function creates a new collection object containing no collection items. The section "Adding Items to a Collection" beginning on page 5-17 shows how you can add items to a new collection.

The default attributes of the new, empty collection are described by the `defaultCollectionAttributes` constant, in which only the persistence attribute is set. This constant is defined in "Attributes Masks" beginning on page 5-49. The section "Changing the Default Attributes of a Collection" beginning on page 5-15 shows how you can change the default attributes of the new collection.

The owner count of the new collection is 1. You can increment the collection's owner count using the `CloneCollection` function, as shown in the section "Cloning or Copying a Collection" beginning on page 5-14. You can decrement the collection's owner count using the `DisposeCollection` function. This function decrements the owner count of a collection and frees the memory used by the collection if the owner count becomes 0.

You can find more information about the `NewCollection` function on page 5-54. You can find more information about the `DisposeCollection` function on page 5-55.

Cloning or Copying a Collection

You use the `CloneCollection` and `CopyCollection` functions to clone and copy collection objects. You clone a collection object if you want to make a copy of the reference to the collection object, and you copy a collection object if you want to make a copy of the entire object, including all of its items.

For example, if you have a reference to a collection object stored in the variable `aCollection`, you can create a new reference to this collection using

```
newCollection = CloneCollection(aCollection);
```

which increments the owner count of the collection object referenced by the `aCollection` variable and returns a copy of the reference as the function result. After this call to the `CloneCollection` function, the `newCollection` and `aCollection` variables reference the same collection object, which has an incremented owner count.

You can create a copy of a collection object, including a copy of all its items, using

```
newCollection = CopyCollection(aCollection, nil);
```

The `CopyCollection` function does not increment the owner count of the `aCollection` collection. Instead, it creates a new collection object with an owner count of 1, copies all of the information from the `aCollection` collection into the new collection, and returns a reference to the new collection. After this call to the `CopyCollection` function, the `newCollection` and `aCollection` variables reference two distinct collections—you can make changes to one without affecting the other.

You can use the second parameter of the `CopyCollection` function to provide a reference to an existing collection object, in which case the function copies the information from the collection referenced by the first parameter into the collection referenced by the second parameter. If the collection referenced by the second parameter already has information in it, the function

- removes all of the items in the second collection—including locked items—before copying the items from the first collection into the second collection
- copies the default attribute values from the first collection into the second collection

The `CopyCollection` function does not copy the owner count or the exception procedure of the first collection; it leaves the owner count and the exception procedure of the second collection unchanged.

You can find more information about the `CloneCollection` function on page 5-56. You can find more information about the `CopyCollection` function on page 5-57.

Changing the Default Attributes of a Collection

Every collection object has default attributes. When you add a new item to a collection, the Collection Manager sets the attributes of the new item to match the default attributes of the collection. You can change the attributes of individual items in a collection using the functions described in “Getting and Setting the Attributes of an Item” beginning on page 5-24. You can change the default attributes for a collection using the `SetCollectionDefaultAttributes` function. This function allows you to change the value of a collection’s default attributes. With this function, you can change the value of every default attribute or you can choose to change only some of the default attributes.

This function takes three parameters:

- a reference to the collection object
- a mask specifying which attributes you want to edit
- the new values for the attributes

(See Figure 5-3 on page 5-10 for an overview of editing attributes.)

As an example, Listing 5-2 changes the default attributes for a collection object so that

- user attribute 0 and the lock attribute are set
- all other attributes are clear

Listing 5-2 Changing the default attributes of a collection

```
long newAttributes;
.
.
.
newAttributes = collectionUser0Mask  /* set user 0 bit */
                | collectionLockMask;  /* set lock bit */

anErr = SetCollectionDefaultAttributes(anyCollection,
                           allCollectionAttributes, /* mask */
                           newAttributes); /* new values */
```

In this example, the `allCollectionAttributes` mask, which is defined in "Attributes Masks" on page 5-49, specifies that you want to replace the value of every attribute in the collection's default attributes with the corresponding value in the `newAttributes` parameter. The value of the `newAttributes` parameter specifies that the user 0 attribute and the lock attribute are set while every other attribute is clear.

You can use different values for the second parameter of this function if you want to edit some of the collection's default attributes but leave other default attributes unchanged. For example, you could set the second parameter of this function to the `userCollectionAttributes` mask:

```
anErr = SetCollectionDefaultAttributes(anyCollection,
                           userCollectionAttributes, /* mask */
                           newAttributes); /* new values */
```

Using this mask specifies that you want to edit only the user attributes of the collection's default attributes. The function replaces the existing values for the collection's default user attributes with the values of the user attributes from the `newAttributes` parameter. In this example, the user 0 attribute is set while all the other user attributes are cleared. However, this call to the `SetCollectionDefaultAttributes` function does not change the values of any of the reserved attributes. For example, the value of the lock attribute in the collection's default attributes remains the same as it was—the value of the lock attribute in the `newAttributes` parameter makes no difference as it is not copied into the collection's default attributes.

You can find more information about the `SetCollectionDefaultAttributes` function on page 5-61.

If you want to examine the default attributes of a particular collection object, you can use the `GetCollectionDefaultAttributes` function, which is described on page 5-60.

Adding Items to a Collection

Once you've created a collection object, you can add new items to the collection using the `AddCollectionItem` function. With this function, you specify the collection tag and collection ID that you want associated with the new item, the size of the new item's variable-length data, and a pointer to the data.

Note

The Collection Manager also provides a utility function, `AddCollectionItemHdl`, which allows you to specify a handle, rather than a pointer, to the data. See page 5-92 for more information about this function. u

Listing 5-3 creates a new collection object and adds ten items to it. Each item has the collection tag `'GXPT'`, the items have collection IDs 0 through 9, and each item contains two coordinates that make up a QuickDraw GX point.

Listing 5-3 Adding items to a collection

```
OSErr anErr;
Collection pointsAndQuotes;
gxPoint location;
long count;
.
.
.
pointsAndQuotes = NewCollection();

location.x = ff(10);
location.y = ff(10);

for (count = 0; count < 10; count++) {
   anErr = AddCollectionItem(pointsAndQuotes,
                             'GXPT',  /* tag */
                             count,   /* id */
                             sizeof(gxPoint),     /* size */
                             &location); /* data */

   location.x += ff(1);  /* change data for next item */
   location.y += ff(1);
}
```

The collection resulting from the code in Listing 5-3 is very similar to an array: the items are numbered sequentially starting with 0, and all items are of the same size. Unlike arrays, however, collections are not limited to these properties. For example, you can add items to a collection dynamically, increasing the total number of collection items during run time. That is, you do not have to specify the capacity of the collection at compile time. Also, collection IDs do not have to be sequential. To demonstrate, the code

```
location.x = ff(100);
location.y = ff(100);
anErr = AddCollectionItem(pointsAndQuotes,
                          'GXPT', 20, /* tag and id */
                          sizeof(gxPoint),     /* size */
                          &location); /* data */
```

adds an eleventh item to the collection from Listing 5-3. The collection tag of the new item is `'GXPT'`, but the new item has a collection ID of 20.

When you add this item to the collection, the Collection Manager assigns it a tag list position, reflecting its position in the list of items with the same collection tag. This tag list position can change if you add a new item with the same collection tag. For example, the call

```
anErr = AddCollectionItem(pointsAndQuotes,
                          'GXPT', 15, /* tag and id */
                          sizeof(gxPoint),     /* size */
                          &location); /* data */
```

adds a new item with the `'GXPT'` collection tag and a collection ID of 15. Adding this item can change the tag list position of any other item with the `'GXPT'` collection tag.

So far, the example collection contains items of the same size. You can also use the `AddCollectionItem` function to add items with variable-length data, as shown in Listing 5-4.

Listing 5-4 Adding items with variable-length data to a collection

```
anErr = AddCollectionItem(pointsAndQuotes,
                          'QUOT', 0,  /* tag and id */
                          17,  /* size */
                          "Le plus ca change");  /* data */

anErr = AddCollectionItem(pointsAndQuotes,
                          'QUOT', 1,
                          29, "Fourscore and seven years ago");

anErr = AddCollectionItem(pointsAndQuotes,
                          'QUOT', 2,
                          18, "It's not the heat.");
```

The sample code from Listing 5-4 adds three new items to the example collection. Each of these items, which have collection tag `'QUOT'`, contains a string of characters as its variable-length data; however, each item contains a string of different length.

Note that the `AddCollectionItem` function copies the information from the block of memory pointed to by its final parameter into the specified collection item. After adding the item, you can change your copy of the information (the copy that the last parameter points to) without affecting the value of the item's variable-length data.

You can use the `CountCollectionItems` function to count the number of items in a collection. For example, after the call

```
totalItems = CountCollectionItems(pointsAndQuotes);
```

the `totalItems` variable contains the value 15 (12 items with points and 3 items with quotes).

You can use the `CountTaggedCollectionItems` function to count the number of items in a collection that have a specified collection tag. For example, after the call

```
quotedItems = CountTaggedCollectionItems(pointsAndQuotes, 'QUOT');
```

the `quotedItems` variable contains the value 3.

For more information about the `AddCollectionItem` function, see page 5-62.

For more information about the `CountCollectionItems` and `CountTaggedCollectionItems` functions, see "Counting Items in a Collection" beginning on page 5-69.

Determining the Collection Index of an Item

Once you have added an item to a collection, you can identify that item in three ways:

- You can specify its collection tag and ID.
- You can specify its collection tag and its tag list position.
- You can specify its collection index.

A collection index is a unique value that the Collection Manager assigns to each item in a collection. You can use an item's collection index to identify the item without specifying the item's collection tag or collection ID. In fact, once you've determined the collection index of an item, specifying that item using its collection index results in faster operations than specifying the item using its collection tag and collection ID.

There are two ways to determine the collection index that the Collection Manager has assigned to an item:

- You can use the `GetCollectionItemInfo` function. With this function, you specify the collection tag and collection ID of the item, and the function returns the item's collection index.
- You can use the `GetTaggedCollectionItemInfo` function. With this function, you specify the collection tag and the tag list position of the item, and the function returns the item's collection index.

Both of these functions optionally return other information about the specified item, as shown in the next two sections.

Listing 5-5 shows how to use the the `GetCollectionItemInfo` function to determine the collection index of an item from the sample collection created in "Adding Items to a Collection" beginning on page 5-17. This listing uses the `dontWantSize` and `dontWantAttributes` constants, which are equal to `nil` and specify that you don't want to determine certain information about the item. These constants are described in "Optional Return Value Constants" on page 5-49.

Listing 5-5 Determining the index of an item

```
long index;
.
.
.
anErr = GetCollectionItemInfo(pointsAndQuotes, /* collection */
                              'GXPT', 15,  /* tag and id */
                              &index,  /* returned index */
                              dontWantSize,
                              dontWantAttributes);
```

After this call to `GetCollectionItemInfo` function, the `index` variable contains the collection index of the item in the `pointsAndQuotes` collection with the `'GXPT'` collection tag and a collection ID of 15. You can use this value to identify this item when using certain Collection Manager functions, such as `RemoveIndexedCollectionItem` and `GetIndexedCollectionItem`.

You can also use the `GetTaggedCollectionItemInfo` function to determine the collection index of a collection item. This function allows you to specify the item using the item's collection tag and tag list position. For example, in Listing 5-5, the item is specified using the `'GXPT'` collection tag and collection ID 15. Assuming the Collection Manager has assigned this item a tag list position of 11, you could also use the call

```
anErr = GetTaggedCollectionItemInfo(pointsAndQuotes,
                                    'GXPT', /* collection tag */
                                    11, /* tag list position */
                                    dontWantId,
                                    &index, /* returned index */
                                    dontWantSize,
                                    dontWantAttributes);
```

to determine the collection index for that item.

For more information about identifying collection items, see "Methods of Identifying Collection Items" on page 5-11.

For more information about the `GetCollectionItemInfo` and `GetTaggedCollectionItemInfo` functions, see "Getting Information About a Collection Item" beginning on page 5-76.

Determining the Tag and ID of an Item

Just as you can determine the collection index of an item given its collection tag and ID, you can also determine the collection tag and ID of an item given its collection index. Listing 5-6 shows how to determine an item's collection tag and collection ID using the `GetIndexedCollectionItemInfo` function.

Listing 5-6 Determining the tag and ID of an item given the item's index

```
long tag, id;
.
.
.
anErr = GetIndexedCollectionItemInfo(pointsAndQuotes,
                                     index, /* index of item */
                                     &tag,  /* returned tag */
                                     &id,   /* returned id */
                                     dontWantSize,
                                     dontWantAttributes);
```

You need to set the value of the `index` variable to contain the collection index of the desired item before making this call to the `GetIndexedCollectionItemInfo` function. (See the previous section, "Determining the Collection Index of an Item" on page 5-19, for in the `GetCollectionItemInfo` function shown in Listing 5-6, the `tag` variable contains the collection tag and the `id` variable contains the collection ID of the item in the `pointsAndQuotes` collection with the collection index specified by the `index` variable.

If you know the collection tag of an item and you know its tag list position, you can use the `GetTaggedCollectionItemInfo` function to determine its collection ID. For example, you could also use the call

```
anErr = GetTaggedCollectionItemInfo(pointsAndQuotes,
                                    'GXPT', 11,
                                    &id,
                                    dontWantIndex,
                                    dontWantSize,
                                    dontWantAttributes);
```

to determine the collection ID of the eleventh item in the `pointsAndQuotes` collection with the tag `'GXPT'`. With the `pointsAndQuotes` collection defined in "Adding Items to a Collection" beginning on page 5-17, the collection ID of this item turns out to be 15.

For more information about identifying collection items, see "Methods of Identifying Collection Items" on page 5-11.

For more information about the `GetCollectionItemInfo` and `GetTaggedCollectionItemInfo` functions, see "Getting Information About a Collection Item" beginning on page 5-76.

Determining the Size of an Item's Variable-Length Data

The Collection Manager provides three functions that provide information about an item in a collection. These three functions differ in how they allow you to specify which item you want information about:

- The `GetCollectionItemInfo` function requires that you specify the collection tag and ID of the desired item.
- The `GetIndexedCollectionItemInfo` function requires that you specify the collection index of the desired item.
- The `GetTaggedCollectionItemInfo` function requires that you specify the collection tag and tag list position of the desired item.

These functions each return a variety of information about the specified item—for example, the item's attributes, the size of the item's variable length data, and so on. These functions return each piece of information in a separate parameter. You can specify that you do not want certain pieces of information returned by providing `nil` for the corresponding parameter. The Collection Manager provides the optional return value constants, each of which is equal to `nil`, to make your code easier to read.

Listing 5-7 shows how to use the `GetCollectionItemInfo` function to determine the size of an item's variable-length data, given that item's collection tag and ID.

Listing 5-7 Determining the size of an item's variable-length data

```
long theSize;
.
.
.
anErr = GetCollectionItemInfo(pointsAndQuotes,  /* collection */
                              'GXPT', 15,  /* tag and id */
                              dontWantIndex,
                              &theSize,  /* returned size */
                              dontWantAttributes);
```

(You can also use the `GetCollectionItemInfo` function to determine an item's collection index, as described in "Determining the Collection Index of an Item" beginning on page 5-19, or to examine an item's attributes, as described in "Getting and Setting the Attributes of an Item" beginning on page 5-24.)

Similarly, you can use the `GetIndexedCollectionItemInfo` function to determine the size of the item's variable-length data given the item's collection index:

```
anErr = GetIndexedCollectionItemInfo(pointsAndQuotes,
                                     index,  /* index of item */
                                     dontWantTag,
                                     dontWantId,
                                     &theSize, /* returned size */
                                     dontWantAttibutes);
```

(You can also use the `GetIndexedCollectionItemInfo` function to determine an item's collection tag and collection ID, as described in "Determining the Tag and ID of an Item" beginning on page 5-21, or to examine an item's attributes, as described in the next section, "Getting and Setting the Attributes of an Item.".)

Finally, you can use the `GetTaggedCollectionItemInfo` function to determine the size of the item's variable-length data given its collection tag and tag list position.

```
anErr = GetTaggedCollectionItemInfo(pointsAndQuotes,
                                    'GXPT',  /* tag of item */
                                    11,  /* tag list position */
                                    dontWantId,
                                    dontWantIndex,
                                    &theSize, /* returned size */
                                    dontWantAttributes);
```

(You can also use the `GetTaggedCollectionItemInfo` function to determine an item's collection index, as described in "Determining the Collection Index of an Item" beginning on page 5-19, to determine an item's collection ID, as described in "Determining the Tag and ID of an Item" beginning on page 5-21, or to examine an item's attributes, as described in the next section, "Getting and Setting the Attributes of an Item.")

For more information about identifying collection items, see "Methods of Identifying Collection Items" on page 5-11.

For more information about the `GetCollectionItemInfo`, `GetIndexedCollectionItemInfo`, and `GetTaggedCollectionItemInfo` functions, see "Getting Information About a Collection Item" beginning on page 5-76.

Getting and Setting the Attributes of an Item

The Collection Manager provides three functions that allow you to examine the attributes of a collection item:

- The `GetCollectionItemInfo` function requires that you specify the collection tag and ID of the desired item.
- The `GetIndexedCollectionItemInfo` function requires that you specify the collection index of the desired item.
- The `GetTaggedCollectionItemInfo` function requires that you specify the collection tag and tag list position of the desired item.

The Collection Manager provides two functions that allow you to edit the attributes of an item:

- The `SetCollectionItemInfo` function requires that you specify the collection tag and ID of the desired item.
- The `SetIndexedCollectionItemInfo` function requires that you specify the collection index of the desired item.

(There is no `SetTaggedCollectionItemInfo` function to correspond to the `GetTaggedCollectionItemInfo` function.)

The three information-retrieving functions allow you to determine a variety of information about the item—not just its attributes. You can find more information about the other values returned by these functions in "Determining the Collection Index of an Item" beginning on page 5-19, "Determining the Tag and ID of an Item" beginning on page 5-21, and "Determining the Size of an Item's Variable-Length Data" beginning on page 5-22.

The information-editing functions, however, allow you to edit the attributes of only the specified item. (You cannot, for instance, use these functions to change the index of an item, or the size of its variable-length data.)

Listing 5-8 shows how you can use the `GetCollectionItemInfo` function to examine the attributes of an item given the item's collection tag and collection ID. This listing uses the collection defined in "Adding Items to a Collection" beginning on page 5-17.

Listing 5-8 Examining the attributes of an item

```
long attributes;
.
.
.
anErr = GetCollectionItemInfo(pointsAndQuotes, /* collection */
                              'QUOT', 0,  /* tag and id */
                              dontWantIndex,
                              dontWantSize,
                              &attributes); /* returned attr's */
```

After this call to the `GetCollectionItemInfo` function, the `attributes` variable contains a copy of the attributes of item from the `pointsAndQuotes` collection with the collection tag `'QUOT'` and a collection ID of 0. You can examine specific attributes using the attribute bit masks, which are described in "Attribute Bit Masks" beginning on page 5-52. As an example, the expression

```
(attributes & collectionLockMask)
```

evaluates to `false` (0) if the lock attribute is not set and to `true` (not 0) if the lock attribute is set.

You can also use the `GetIndexedCollectionItemInfo` function to examine the attributes of an item, given its collection index rather than its collection tag and collection ID:

```
anErr = GetIndexedCollectionItemInfo(pointsAndQuotes,
                                     index, /* index of item */
                                     dontWantTag,
                                     dontWantId,
                                     dontWantSize,
                                     &attributes); /* returned */
```

Similarly, you can use the `GetTaggedCollectionItemInfo` function to examine the attributes of an item, given its collection tag and tag list position:

```
anErr = GetTaggedCollectionItemInfo(pointsAndQuotes,
                                    'QUOT',  /* tag of item */
                                    1,  /* tag list position */
                                    dontWantId,
                                    dontWantIndex,
                                    dontWantSize,
                                    &attributes);  /* returned */
```

You can edit the attributes of a collection item using the `SetCollectionItemInfo` and `SetIndexedCollectionItemInfo` functions. These functions require you to specify which attributes you want to edit and what the new values for those attributes should be.

You specify this information using two `long` parameters:

- The first is a mask. Each bit in this mask represents one of the item's attributes. A bit value of 0 in this mask signifies that you do not want to edit the corresponding attribute of the specified item. A bit value of 1 in this mask signifies that you do want to edit the corresponding attribute of the item.
- The second contains the new values. Each bit in this parameter represents the new value for the corresponding attribute of the item. Only the bits in this parameter that correspond to bits in the mask parameter that have a value of 1 are significant. The Collection Manager ignores the other bit values in this parameter.

Listing 5-9 shows how you can use the `SetCollectionItemInfo` to set the lock and persistence attributes of a collection item and clear all the other attributes.

Listing 5-9 Setting the lock and persistence bit attribute of an item

```
long newAttributes;
.
.
.
newAttributes = collectionLockMask
                | collectionPersistenceMask;

anErr = SetCollectionItemInfo(pointsAndQuotes,
                              'QUOT', 0, /* tag and id */
                              allCollectionAttributes, /* mask */
                              newAttributes); /* new values */
```

This example uses the `allCollectionAttributes` constant (which is defined in "Attributes Masks" beginning on page 5-49) to indicate that all the attributes of the specified collection item should be edited. As a result, the code in the example replaces the value of every attribute in the specified collection item with the corresponding value from the `newAttributes` parameter.

If you want to set the lock and persistence attributes of this collection item without affecting the values of the other attributes, you can use the `newAttributes` variable as both the mask and the values parameters:

```
anErr = SetCollectionItemInfo(pointsAndQuotes,
                              'QUOT', 0, /* tag and id */
                              newAttributes, /* mask */
                              newAttributes); /* new values */
```

In this case, the code uses the `newAttributes` parameter as the mask (which indicates that only the lock attribute and the persistence attribute should be affected) as well as the values (which indicate that both of these attributes should be set). The values of all the other attributes of the specified item remain as they were before the call.

You can also use the `SetIndexedCollectionItemInfo` function to set the attributes of an item, given the item's collection index rather than its collection tag and collection ID:

```
anErr = SetIndexedCollectionItemInfo(pointsAndQuotes,
                                     index,
                                     allCollectionAttributes,
                                     newAttributes);
```

For more information about identifying collection items, see "Methods of Identifying Collection Items" on page 5-11.

For more information about the `GetCollectionItemInfo`, `GetIndexedCollectionItemInfo`, and `GetTaggedCollectionItemInfo` functions, see "Getting Information About a Collection Item" beginning on page 5-76.

For more information about the `SetCollectionItemInfo` and `SetIndexedCollectionItemInfo` functions, see "Editing Item Attributes" beginning on page 5-82.

Replacing Items in a Collection

The Collection Manager provides two methods for replacing items in a collection:

- You can use the `AddCollectionItem` function, specifying the collection tag and collection ID of an existing item.
- You can use the `ReplaceIndexedCollectionItem` function, specifying the collection index of an existing item.

Note
The Collection Manager also provides the utility functions, `AddCollectionItemHdl` and `ReplaceCollectionItemHdl`, which allow you to specify a handle, rather than a pointer, to the item's data. See "Working With Macintosh Memory Manager Handles" beginning on page 5-92 for more information about these functions. u

The new item does not have to be the same size as the replaced item. For example, Listing 5-10 shows how you can use the `AddCollectionItem` function to replace the data in a collection item with a new data of a different length. (This example uses the collection created in "Adding Items to a Collection" beginning on page 5-17, in which the item identified by the collection tag `'QUOT'` and the collection ID 1 contains the 29-character string "Fourscore and seven years ago")

Listing 5-10 Replacing an item in a collection

```
anErr = AddCollectionItem(pointsAndQuotes,
                          'QUOT', 1,
                          22, "Eighty-seven years ago");
```

You cannot replace a collection item if its lock attribute is set. For example, the previous section shows how to set the lock attribute of the item with the collection tag `'QUOT'` and the collection ID 0. If you try to replace this item using

```
anErr = AddCollectionItem(pointsAndQuotes,
                          'QUOT', 0,
                          24, "Plus c'est la meme chose");
```

the code sets the `anErr` variable to the `collectionItemLockedErr` value and the Collection Manager does not replace the item.

If you know the collection index of an item, you can use the `ReplaceIndexedCollectionItem` function to replace the item. This function finds the specified item more efficiently than the `AddCollectionItem` function. Listing 5-11 shows an example of this function.

Listing 5-11 Replacing an item using the item's index

```
long index;
.
.
.
/* find the index. */
anErr = GetCollectionItemInfo(pointsAndQuotes,
                              'QUOT', 1,
                              &index,
                              dontWantSize,
                              dontWantAttributes);
.
.
.
/* replace the item. */
anErr = ReplaceIndexedCollectionItem(pointsAndQuotes,
                                     index,
                                     22,
                                     "Eighty-seven years ago");
```

The example in Listing 5-11 uses the `GetCollectionItemInfo` function to find the collection index that the Collection Manager has assigned to the item with a collection tag of `'QUOT'` and a collection ID of 1. Finding this collection index requires some processing time. However, once you've found the item's collection index, you can use it to find information about the item quickly, because functions that search for a collection item using the item's collection index operate more efficiently than functions that search using the item's collection tag and collection ID. Typically, if you want to search for an item only once, you use the item's collection tag and collection ID. If you know that you have to search for the same item repeatedly, you find the item's collection index and use the collection index when examining or editing the item.

For more information about identifying collection items, see "Methods of Identifying Collection Items" on page 5-11.

For more information about the `AddCollectionItem` function and the `ReplaceIndexedCollectionItem` function, see "Adding and Replacing Items in a Collection" beginning on page 5-62.

Removing Items From a Collection

The Collection Manager provides two methods for removing individual items from a collection:

- You can use the `RemoveCollectionItem` function, specifying the collection tag and collection ID of the item you want to remove.
- You can use the `RemoveIndexedCollectionItem` function, specifying the collection index of the item you want to remove.

The Collection Manager provides three methods for removing multiple items from a collection:

- You can use the `PurgeCollection` function to remove all the items in a collection whose attributes match criteria you specify.
- You can use the `PurgeCollectionTag` function to remove all the items in a collection that have a specified collection tag.
- You can use the `EmptyCollection` function to remove every item from a collection.

Listing 5-12 shows how you can use the `RemoveCollectionItem` function to remove an item from a collection. (This example uses the collection created in “Adding Items to a Collection” beginning on page 5-17.)

Listing 5-12 Removing an item in a collection

```
anErr = RemoveCollectionItem(pointsAndQuotes,
                             'QUOT', 1); /* tag and id */
```

You can remove a collection item even if its lock attribute is set—the lock attribute only affects replacing. For example, if you have set the lock attribute of the collection item with the collection tag `'QUOT'` and the collection ID 0, you can remove this item using

```
anErr = RemoveCollectionItem(pointsAndQuotes,
                             'QUOT', 0); /* tag and id */
```

You can also remove the item using

```
anErr = RemoveCollectionItem(pointsAndQuotes,
                             'QUOT', 0); /* tag and id */
```

If you know the index of an item, you can use the `RemoveIndexedCollectionItem` function to remove the item. This function finds the specified item more efficiently than the `RemoveCollectionItem` function. Listing 5-13 shows an example of this function.

Listing 5-13 Removing an item using the item's index

```
long index;
.
.
.
/* get the index */
anErr = GetCollectionItemInfo(pointsAndQuotes,
                              'QUOT', 1,
                              &index,
                              dontWantSize,
                              dontWantAttributes);
.
.
.
/* remove the item */
anErr = RemoveIndexedCollectionItem(pointsAndQuotes, index);
```

The example in Listing 5-13 uses the `GetCollectionItemInfo` function to find the collection index that the Collection Manager has assigned to the item with a collection tag of `'QUOT'` and a collection ID of 1. Finding this collection index requires some processing time. However, once you've found the item's collection index, you can use it to find information about the item quickly, because functions that search for a collection item using the item's collection index operate more efficiently than functions that search using the item's collection tag and collection ID.

The `PurgeCollection` function allows you to remove multiple items from a collection. You provide this function with a collection and a set of attribute values, and it removes any items in the collection whose attributes match these values. You specify which attributes to examine in the second parameter of this function, and you specify the values to compare those attributes against in the third parameter, as shown in Listing 5-14.

Listing 5-14 Removing multiple items with specific attributes

```
long whichAttributes, attributeValues;
.
.
.
/* specify which attributes to examine: user 0 and user 1 */
whichAttributes = collectionUser0Mask
                  | collectionUser1Mask;
```

```
/* specify the values to test for: user 0 set and user 1 clear */
attributeValues = collectionUser0Mask
                  & ~collectionUser1Mask;

/* purge all items with user 0 attribute set and user 1 clear */
PurgeCollection(pointsAndQuotes,
                whichAttributes,
                attributeValues);
```

This example sets two bits in the `whichAttributes` variable—the user 0 attribute and the user 1 attribute—and clears every other bit in this variable, which signifies that the function should test only the user 0 attribute and the user 1 attribute. The `attributeValues` variable sets the user attribute 0 flag and clears the user attribute 1 flag. Therefore, this call to `PurgeCollection` removes every item in the collection that has the user 0 attribute set and the user 1 attribute clear. It ignores the values of all the other attributes.

You can use the `PurgeCollectionTag` function to remove all of the items in a collection that share a collection tag—even the locked items. To remove all the items with the collection tag `'GXPT'` from the `pointsAndQuotes` collection (which is defined in "Adding Items to a Collection" beginning on page 5-17), you could use this line of code:

```
PurgeCollectionTag(pointsAndQuotes, 'GXPT');
```

Finally, you can remove all of the items in a collection—even the locked items—using the `EmptyCollection` function:

```
EmptyCollection(pointsAndQuotes);
```

For more information about identifying collection items, see "Methods of Identifying Collection Items" on page 5-11.

For more information about the `RemoveCollectionItem`, `RemoveIndexedCollectionItem`, `PurgeCollection`, `PurgeCollectionTag`, and `EmptyCollection` functions, see "Removing Items From a Collection" beginning on page 5-65.

Retrieving the Variable-Length Data From an Item

The Collection Manager provides three functions that return a copy of the information in an item's variable-length data. These three functions differ in how they allow you to specify which item you want information about:

- The `GetCollectionItem` function requires that you specify the collection tag and collection ID of the desired item.
- The `GetIndexedCollectionItem` function requires that you specify the collection index of the desired item.
- The `GetTaggedCollectionItem` function requires that you specify the collection tag and tag list position of the desired item.

Note

The Collection Manager also provides the utility function `GetCollectionItemHdl`, which returns a copy of the item's data in a block of memory referenced by a Macintosh Memory Manager handle, rather than a pointer. See page 5-94 for more information about this function. u

These functions each return two pieces of information about the specified item—the size of its variable-length data and a copy of the data itself. You can specify that you want to determine either the size or the data or both (or neither, actually, although that doesn't prove to be very useful).

Typically, you call these functions twice:

- once to determine the size of the data (if you don't already know the size)
- once (after allocating enough memory) to obtain a copy of the data.

Listing 5-15 shows how to use the `GetCollectionItem` function to retrieve the variable-length data from an item. This sample code uses the `pointsAndQuotes` collection defined in "Adding Items to a Collection" beginning on page 5-17.

Listing 5-15 Retrieving the variable-length data from an item

```
long theSize;
char *theData;
.
.
.
anErr = GetCollectionItem(pointsAndQuotes,
                          'QUOT', 0, /* tag and id */
                          &theSize,
                          dontWantData);

theData = (char *) NewPtr(theSize);
```

```
anErr = GetCollectionItem(pointsAndQuotes,
                          'QUOT', 0,
                          dontWantSize,
                          theData);
```

If you specify a non-NIL value for the size parameter, the GetCollectionItem function returns in the size parameter the actual number of bytes of the item's data.

If you specify non-NIL values for both the size and data parameters, the number of bytes returned in the data parameter is either the value specified by the size parameter or the actual number of bytes of the specified item's data, whichever is lower.

You can also use the GetIndexedCollectionItem function to retrieve an item's data, given the item's collection index rather than its collection tag and collection ID, as shown in Listing 5-16.

Listing 5-16 Retrieving the variable-length data from an item using the item's index

```
long index;
long theSize;
char *theData;
.
.
.
/* get the index and data size */
anErr = GetCollectionItemInfo(pointsAndQuotes,
                              'QUOT', 0, /* tag and id */
                              &index,
                              &theSize,
                              dontWantAttributes;
.
.
.
theData = (char *) NewPtr(theSize);

anErr = GetIndexedCollectionItem(pointsAndQuotes,
                                 index,
                                 dontWantSize,
                                 theData);
```

Similarly, you can use the `GetTaggedCollectionItem` function to retrieve an item's data, given the item's collection tag and tag list position, as shown in Listing 5-17.

Listing 5-17 Retrieving the variable-length data from an item using the tag and tag list position

```
long index;
long theSize;
char *theData;
.
.
.
anErr = GetTaggedCollectionItem(pointsAndQuotes,
                                'QUOT',
                                1, /* first of the 'QUOT' items */
                                &theSize,
                                dontWantData);

theData = (char *) NewPtr(theSize);

anErr = GetTaggedCollectionItem(pointsAndQuotes,
                                'QUOT',
                                1, /* first of the 'QUOT' items */
                                dontWantSize,
                                (void *) theData);
```

For more information about identifying collection items, see "Methods of Identifying Collection Items" on page 5-11.

For more information about the `GetCollectionItem`, `GetIndexedCollectionItem`, and `GetTaggedCollectionItem` functions, see "Retrieving the Variable-Length Data From an Item" beginning on page 5-70.

Examining the Collection Tags of a Collection

The Collection Manager provides three functions that allow you to examine the collection tags contained in a specific collection:

- You can use the `CollectionTagExists` function to determine if any of the items in a specific collection have a specified collection tag.
- You can use the `CountCollectionTags` function to determine the total number of distinct collection tags contained in the items of a collection.
- You can use the `GetIndexedCollectionTag` function to examine the value of one of the distinct collection tags contained in a collection.

Every collection has a list of distinct collection tags contained in that collection. The `GetIndexedCollectionTag` function allows you to step through this list of distinct collection tags, as shown in Listing 5-18.

Listing 5-18 Counting tags in a collection

```
long numTags, numItems, eachTag, eachItem;
.
.
.
numTags = CountCollectionTags(pointsAndQuotes);

/* iterate through each tag */
for (eachTag = 1; eachTag <= numTags; ++eachTag) {
   GetIndexedCollectionTag(pointsAndQuotes, eachTag, &theTag);
   numItems = CountTaggedCollectionItems(pointsAndQuotes, theTag);

   /* iterate through each item with that tag */
   for (eachItem = 1; eachItem <= numItems; ++eachItem) {

      /* find size of item data and obtain copy of data */
      GetTaggedCollectionItem(pointsAndQuotes,
                              theTag, eachItem,
                              &theSize, dontWantData);
      theData = (char *) NewPtr(theSize);
      GetTaggedCollectionItem(pointsAndQuotes,
                              theTag, eachItem,
                              dontWantSize, theData);
      .
      .
      .
      /* manipulate item data . . .*/
      .
      .
      .
      DisposePtr(theData);
   }
}
```

This sample code determines the total number of distinct tags in the `pointsAndQuotes` collection using the `CountCollectionTags` function. Then, it uses the `GetIndexedCollectionTag` function to step through each of the distinct collection tags in the collection.

With each collection tag, the sample code uses the `GetTaggedCollectionItem` function to retrieve the variable-length data from each item with the tag. In this manner, this sample code retrieves the data from every item in the collection.

For more information about the `CollectionTagExists`, `CountCollectionTags`, and `GetIndexedCollectionTag` functions, see "Getting Information About Collection Tags" beginning on page 5-85.

Flattening and Unflattening a Collection

The Collection Manager provides the `FlattenCollection` function for converting the information in a collection object into a flattened stream of bytes. With the `FlattenCollection` function, you provide a callback function that operates on the stream of bytes—you can use this callback function to write the stream out to disk, store the stream in a Macintosh Memory Manager handle, and so on.

The `FlattenCollection` function takes three parameters:

- a reference to the collection to flatten
- a pointer to the callback function that you provide to handle the returned stream of bytes
- a 32-bit reference constant that the Collection Manager passes back to your callback function

When you call the `FlattenCollection` function, the Collection Manager begins converting the collection into a stream of bytes. It repeatedly calls your callback function, each time sending it more of the flattened collection, until it has converted the entire collection.

Your callback function determines what happens to the flattened collection. This function must take three parameters: a `long` value that represents the size of the current block of data, a pointer to the current block of data, and a reference constant that you can use as a pointer to other information.

Listing 5-19 shows an example callback function. This function appends the block of data provided by the Collection Manager in the `theData` parameter to the end of a block of data referenced by a Macintosh Memory Manager handle. The handle and the current size of the block of data referenced by the handle are stored in a `TFlattenBlock` structure. (The sample code in Listing 5-20 passes a pointer to this structure as the reference constant when calling the `FlattenCollection` function, which passes the pointer back to your callback function.)

Listing 5-19 Flattening procedure

```
typedef struct {
   long position;
   Handle dataHandle;
} TFlattenBlock;

OSErr FlattenProc(long theSize, Ptr theData,
                  TFlattenBlock *flattenBlock) {

   register OSErr anErr = noErr;


   SetHandleSize(flattenBlock->dataHandle,
                    flattenBlock->position + theSize);
   anErr = MemError();
   if (anErr == noErr) {
      BlockMove(data,
                *flattenBlock->dataHandle +
                flattenBlock->position,
                theSize);
         flattenBlock->position += theSize;
      }
   }
   return anErr;
}
```

Listing 5-20 shows how you can use this callback function. The sample function in Listing 5-20 uses the `FlattenCollection` function to flatten a collection into a block of memory referenced by a Macintosh Memory Manager handle.

Listing 5-20 The `FlattenCollectionToHdl` function

```
/* possible implementation of FlattenCollectionToHdl */

OSErr FlattenCollectionToHdl(Collection anyCollection,
                             Handle flattenedCollection)
{

   register OSErr anErr;
   TFlattenBlock flattenBlock;

   flattenBlock.position = 0;
   flattenBlock.dataHandle = flattenedCollection;

   if (!(anErr = MemError())) {

      anErr = FlattenCollection(anyCollection,
                                FlattenProc,
                                &flattenBlock);

      if (anErr)
         flattenBlock.dataHandle = nil;
   }

   return anErr;
}
```

This function creates a `TFlattenBlock` structure, initializes the `position` field to 0, and initializes the `dataHandle` field to a newly allocated Macintosh Memory Manager handle. The function then calls the `FlattenCollection` function, specifying the collection to flatten, the callback function specified in Listing 5-19, and a pointer to the `TFlattenBlock` structure. In response, the Collection Manager flattens the specified collection one piece at a time, repeatedly calling the callback function with new blocks of the flattened collection. The Collection Manager provides a pointer to the `TFlattenBlock` structure when calling the callback function. The callback function uses this information to copy each new block of flattened collection data onto the end of the Macintosh Memory Manager handle.

Listing 5-21 shows the reverse process—using the `UnflattenCollection` function to convert a flattened collection from a Macintosh Memory Manager handle into a collection object.

Listing 5-21 A possible implementation of the `UnflattenCollectionFromHdl` function

```
void UnflattenProc(long theSize, Ptr theData,
                  TFlattenBlock *flattenBlock) {

   BlockMove(*flattenBlock->dataHandle +
             flattenBlock->position,
             theData, theSize)

   flattenBlock->position += theSize;
}


OSErr UnflattenCollectionFromHdl(Collection anyCollection,
                                 Handle flattenedCollection)
{
   register OSErr anErr;
   TFlattenBlock flattenBlock;

   flattenBlock.position = 0;
   flattenBlock.dataHandle = flattenedCollection;

   anErr = UnflattenCollection(anyCollection,
                               UnflattenProc,
                               &flattenBlock);

   return anErr;
}
```

Listing 5-21 shows a possible implementation of the `UnflattenCollectionFromHdl` function. The Collection Manager provides both the `FlattenCollectionToHdl` and `UnflattenCollectionFromHdl` functions for you—you do not have to define these yourself. For more information about the flattening and unflattening functions, see “Flattening and Unflattening a Collection” beginning on page 5-88.

Reading Collections From and Writing Collections to Disk

The Collection Manager provides a number of methods for storing collections on disk:

- You can store the collection's contents as a collection (`'cltn'`) resource and read the information into a collection object using the `GetNewCollection` function. For more information about the `'cltn'` resource, see "The Collection Resource" beginning on page 5-102, and for more information about the `GetNewCollection` function, see the description of that function on page 5-99. For an example of reading a collection object from a collection resource, see the next section, "Reading a Collection From a Collection Resource."
- You can flatten a collection using the `FlattenCollection` function and provide a callback function that writes the blocks of flattened data to a file. You can unflatten this collection using the `UnflattenCollection` function, providing a callback function that reads blocks of data from the file. For more information about these functions, see "Flattening and Unflattening a Collection" beginning on page 5-37 and the description of the `FlattenCollection` function on page 5-88 and the description of the `UnflattenCollection` function on page 5-90.
- You can flatten a collection to a handle using the `FlattenCollectionToHdl` function and write the contents of the handle to the resource fork of a file (using the Macintosh function `AddResource`) or to the data fork of a file (using the Macintosh function `FSWrite`). You can then read the contents of this file into a handle (using the Macintosh functions `GetResource` or `FSRead`) and unflatten the result using the `UnflattenCollectionFromHdl` function.

IMPORTANT

Although you may create a resource containing a flattened collection using the `FlattenCollectionToHdl` and `AddResource` functions, you cannot recreate the collection from this resource using the `GetNewCollection` function. The resource format created by the `FlattenCollectionToHdl` and `AddResource` functions is incompatible with the resource format expected by the `GetNewCollection` function. s

Listing 5-22 shows how to flatten a collection to a handle and then write the contents of the handle to the resource fork of a disk file.

Listing 5-22 Flattening a collection to a disk file as a resource

```
OSErr anErr;
Handle flattened;

/* write the collection out as a resource */

flattened = NewHandle(0);
anErr = FlattenCollectionToHdl(myCollection, flattened);

if (anErr == noErr) {
   AddResource(flattened, myType, myID, myName);
   anErr = ResError();
}
```

Listing 5-23 shows how to flatten a collection to a handle and then write the contents of the handle to the data fork of a disk file.

Listing 5-23 Flattening a collection to a data fork of a disk file

```
OSErr anErr;
Handle flattened;
long theSize;

/* write the collection out to the data fork */

flattened = NewHandle(0);
anErr = FlattenCollectionToHdl(myCollection, flattened);

if (anErr == noErr) {
   theSize = GetHandleSize(flattened);
   anErr = FSWrite(refNum, theSize, *flattened);
}
```

Listing 5-24 shows how to read a flattened collection from the resource fork of a disk file into a block of memory referenced by a Macintosh Memory Manager handle and then unflatten the information in that block of memory into a collection object.

Listing 5-24 Unflattening a collection from a disk file as a resource

```
Handle flattened;
Collection myCollection;

if (myCollection = NewCollection()) {

   /* read the collection in as a resource */

   flattened = GetResource(myType, myID);

   if ((anErr = ResError()) == noErr) {
   anErr = UnflattenCollectionFromHdl(myCollection, flattened);
   ReleaseResource(flattened);
   if (anErr == noErr)
      anErr = ResError();
   }
}
```

Listing 5-25 shows how to read a flattened collection from the data fork of a disk file into a block of memory referenced by a Macintosh Memory Manager handle and then unflatten the information in that block of memory into a collection object.

Listing 5-25 Unflattening a collection from the data fork of a disk file

```
OSErr anErr;
Handle flattened;
Collection myCollection;

if (myCollection = NewCollection()) {

   /* read the collection in from the data fork */

   flattened = NewHandle(theSize);

   if ((anErr = MemError()) == noErr) {
      if ((anErr = FSRead(refNum, theSize, *flattened)) == noErr)
         anErr = UnflattenCollectionFromHdl(myCollection,
                                            flattened);
      DisposHandle(flattened);

   }
}
```

To unflatten a collection using Listing 5-25, you must know the size of the collection before you can unflatten it. If you don't know the size of the collection, you unflatten a collection using the callback function mechanism described in "Flattening and Unflattening a Collection" beginning on page 5-37.

For more information about the `FlattenCollectionToHdl` function and the `UnflattenCollectionFromHdl` function, see "Flattening and Unflattening a Collection" beginning on page 5-37 as well as the descriptions of these functions starting on page 5-97.

For information about the Macintosh functions `AddResource` and `GetResource`, see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*. For information about the Macintosh functions `FSRead` and `FSWrite`, see the chapter "File Manager" in *Inside Macintosh: Files*.

Reading a Collection From a Collection Resource

To store a collection to disk, you can flatten a collection and write the flattened data to a file, as described in the previous section, or you can create a **collection (`'cltn'`) resource.** The format of the collection resource is shown in the section "The Collection Resource" beginning on page 5-102.

You can create a collection object from the information stored in a collection resource using the `GetNewCollection` function. Listing 5-26 gives an example.

Listing 5-26 Reading a collection from a collection resource

```
OSErr ReadCollectionFromResource(short refNum, short resID,
                                 Collection* pCollection)
{
   OSErr anErr = noErr;

   short saveResFile = CurResFile();
   UseResFile(refNum);
   *pCollection = GetNewCollection(resID);
   if (!*pCollection) {
      anErr = ResError();
      if (!anErr)            /* if ResErr returned noErr */
         anErr = resNotFound; /* then the error was resNotFound */
   }

   UseResFile(saveResFile);

   return anErr;
}
```

The `ReadCollectionFromResource` sample function requires three parameters:

- the reference number of the file containing the desired collection resource
- the resource ID of the desired collection resource
- a pointer to a collection object reference

The sample function uses the `CurResFile` function to determine the current resource file, saves the reference number of that resource file, and uses the `UseResFile` function to indicate that the current resource file should be the resource file specified by the reference number contained in the first parameter.

The sample function then uses the `GetNewCollection` function, which takes a resource ID as its only parameter, to read the information from the designated collection resource into the collection object referenced by the sample function's third parameter.

Finally, the sample function checks for errors and resets the current resource file.

For more information about resource files and the `CurResFile`, `UseResFile`, and `ResError` functions, see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.

For more information about the collection resource, see "The Collection Resource" beginning on page 5-102. For more information about the `GetNewCollection` function, see the description of that function on page 5-99.

Installing an Exception Procedure

The Collection Manager allows you to specify an exception procedure for each collection object. When you attempt to manipulate a collection object using a Collection Manager function and the function results in an error, the Collection Manager calls the exception procedure for the collection object and sends it two parameters: a reference to the collection object that caused the error and the error code that was generated.

In an exception procedure, you can handle the error and then change the error code to `noErr`, a process which indicates that the Collection Manager can return control to the place in your application that generated the error as if no error had occurred. You can also change the error from one error code to another. A third alternative is to use the

ANSI C functions `setjmp` and `longjmp` to jump out of the exception handler and into code to handle the error. Listing 5-27 shows a sample exception procedure.

Listing 5-27 A sample exception procedure

```
jmp_buf cpuState; /* global machine state */

pascal OSErr MyExceptionHandler(Collection errorCollection,
                                OSErr status)
{
   /* ignore collectionItemLockedErr errors */
   if (status == collectionItemLockedErr)
      return noErr;

   /* all other errors must be handled by caller’s setjmp block */
   /* jump back to callers setjmp block and return status */
   longjmp(cpuState, status);
}

void ExceptionTest(Collection anyCollection)
{
   OSErr result;

   SetCollectionExceptionProc(anyCollection, MyExceptionHandler);

   if (!(result = setjmp(cpuState))) {
      AddCollectionItem(anyCollection, 'tag1', 1, 4, "data");
      AddCollectionItem(anyCollection, 'tag1', 2, 9, "more data");
      AddCollectionItem(anyCollection, 'tag1', 3, 9, "last data");

      /* cause an error . . . */
      RemoveCollectionItem(anyCollection, 'tag1', 4);
   } else {
      .
      .
      .
      /* handle errors other than collectionItemLockedErr */
      /* use result local variable to determine which error */
      .
      .
      .
   }
}
```

In Listing 5-27, the ExceptionTest sample function takes a single parameter: a reference to a collection object. The sample function first calls the SetCollectionExceptionProc function to install an exception handler for this collection object. In this example, the call to SetCollectionExceptionProc installs the MyExceptionHandler function as the exception handler.

The next line of the ExceptionTest sample function calls the setjmp function. This function stores the current machine state, including the current position in the sample code, into the cpuState global variable. It also returns a value of 0 as its function result, and this value is assigned to the local variable result. This value is negated (by the ! operator), an operation that produces a Boolean value of true. Therefore, the block of code in the if clause begins to execute.

Imagine that the first call to the AddCollectionItem function completes successfully, but that the second call to AddCollectionItem generates a collectionItemLockedErr error. During the second call to AddCollectionItem, the Collection Manager responds to the error by calling the MyExceptionHandler function. The first parameter passed to this function indicates the collection that generated the error, and the second parameter passed to this function indicates the error that was generated. This sample exception handler determines whether the error is the collectionItemLockedErr error (which it is in this example) and then returns with the noErr error as the function result. The Collection Manager notices this change in error and returns control to the sample function as if no error had occurred. (Just as you can use this mechanism to ignore certain errors, you can also use this mechanism to change errors of one type into errors of another type.) Since effectively no error has now occurred, the ExceptionTest sample function continues by executing the third call to the AddCollectionItem function.

The subsequent line of the ExceptionTest sample function attempts to remove an item that is not in the collection, resulting in a collectionItemNotFoundErr error. Again, the Collection Manager responds by calling the exception handler. In this case, however, the error is not the collectionItemLockedErr error, so the exception handler executes this line of code:

```
longjmp(cpuState, status);
```

When you call the longjmp function,

- control is passed to the location of the corresponding call to the setjmp function
- the value passed as the second parameter to the longjmp function becomes the function result of the setjmp function

Therefore, this call to the longjmp function passes control back to the location in the ExceptionTest sample function where setjmp(cpuState) was called earlier. This time, however, the function result returned by the setjmp function is not 0, as it was before, but instead is the value of status, the second parameter in the call to the longjmp function. Therefore, the function result of the setjmp function is set to be the collectionItemNotFoundErr error.

Once again, the `ExceptionTest` sample function assigns this function result to the `result` local variable, and negates it with the `!` operator. This time the negation produces a Boolean value of `false`, and therefore the block of code in the `else` clause begins to execute. In this block of code, you can handle errors not handled in the exception handler, using the `result` local variable to determine which error occurred.

You can find more information about the `SetCollectionExceptionProc` function on page 5-59. You can find more information about exception procedures on page 5-101.

Collection Manager Reference

This section provides reference information about the data types, functions, and resources that allow you to create and manipulate collection objects. It includes

- type definitions of the data types, including enumeration types, that are specific to the Collection Manager
- descriptions of the functions that operate on collection objects and their items
- descriptions of the application-defined callback function used for flattening and unflattening collections and the application-defined callback function used for exception handling
- the definition of the resource type used to store collection objects on disk

Data Types

This section describes the data types that you use to obtain information from and provide information to the Collection Manager functions.

Collection Objects

The Collection Manager provides you with access to a collection object through a `Collection` reference:

```
typedef struct PrivateCollectionRecord *Collection;
```

The `Collection` type defines a reference type that your compiler can type-check; it does not define a pointer to a publicly defined data structure. The contents of the collection object are private; you must use the Collection Manager functions to manipulate collection objects.

Collection Tags

Each item in a collection is uniquely identified by its collection tag and its collection ID. The collection tag is a four-character identifier, similar to the indentifiers used for resources:

```
typedef long CollectionTag;   /* 4-byte identifier ('xxxx') */
```

For more information about collection tags, see “Collection Items” beginning on page 5-8.

Optional Return Value Constants

Many of the Collection Manager functions return multiple pieces of information. For most of these functions, you can specify that you don’t want a specific piece of information to be returned by specifying `nil` for the corresponding parameter when calling the function.

The Collection Manager provides the optional return value constants to make your code easier to read when specifying that you are not interested in obtaining certain types of information:

```
enum {
   dontWantTag = 0L,
   dontWantId = 0L,
   dontWantSize = 0L,
   dontWantAttributes = 0L,
   dontWantIndex = 0L,
   dontWantData = 0L
};
```

You can use these enumeration constants in place of the more generic constant `nil` when specifying that you don’t want to receive certain optional return values from a function.

Attributes Masks

The Collection Manager provides four convenient attributes masks that you can use when specifying attributes for any of the attribute-related Collection Manager functions:

```
enum {
   noCollectionAttributes = 0x00000000,
   allCollectionAttributes = 0xFFFFFFFF,
   userCollectionAttributes = 0x0000FFFF,
   defaultCollectionAttributes = 0x40000000
};
```

Constant descriptions

`noCollectionAttributes`
: Specifies a mask in which all collection attributes are clear. You might use this constant when clearing all the attributes of an item or when testing whether all of an item's attributes are clear.

`allCollectionAttributes`
: Specifies a mask in which all collection attributes are set. You might use this constant as a mask to indicate that you want to edit or test every attribute of an item, or you might use it to set every attribute of an item.

`userCollectionAttributes`
: Specifies a mask in which the user attributes are set and the reserved attributes are clear. You might use this constant as a mask to indicate that you want to edit or test only the user attributes of an item, or you might use it to set every user attribute of an item.

`defaultCollectionAttributes`
: Specifies a mask in which the persistent attribute is set and all other attributes are clear. You might use this constant when testing to see if an item's attributes have been edited.

You can also use the attribute bit masks, described on page 5-52, as masks for individual attributes.

For more information about collection item attributes, see "Collection Items" beginning on page 5-8.-

Attribute Bit Numbers

The Collection Manager provides the attribute bit numbers enumeration to provide constant names for each of the bits in a collection item's attributes.

```
enum {
   collectionUser0Bit = 0,  /* for use by your application */
   collectionUser1Bit = 1,
   collectionUser2Bit = 2,
   collectionUser3Bit = 3,
   collectionUser4Bit = 4,
   collectionUser5Bit = 5,
   collectionUser6Bit = 6,
   collectionUser7Bit = 7,
   collectionUser8Bit = 8,
   collectionUser9Bit = 9,
   collectionUser10Bit = 10,
   collectionUser11Bit = 11,
   collectionUser12Bit = 12,
   collectionUser13Bit = 13,
   collectionUser14Bit = 14,
   collectionUser15Bit = 15,

   collectionReserved0Bit = 16,  /* reserved for use by Apple */
   collectionReserved1Bit = 17,
   collectionReserved2Bit = 18,
   collectionReserved3Bit = 19,
   collectionReserved4Bit = 20,
   collectionReserved5Bit = 21,
   collectionReserved6Bit = 22,
   collectionReserved7Bit = 23,
   collectionReserved8Bit = 24,
   collectionReserved9Bit = 25,
   collectionReserved10Bit = 26,
   collectionReserved11Bit = 27,
   collectionReserved12Bit = 28,
   collectionReserved13Bit = 29,

   collectionPersistenceBit = 30,  /* Currently defined */
   collectionLockBit = 31
};
```

The lower 16 bits of the attributes property of a collection item represent the user-defined attributes. You can use these attributes for any purpose suitable to your application.

The upper 16 bits are reserved for use by Apple Computer, Inc. Currently, the 2 high bits are defined: bit 30 represents the persistence attribute and bit 31 represents the lock attribute.

For more information about collection item attributes, see “Collection Items” beginning on page 5-8.

Attribute Bit Masks

Using the attribute bit numbers, the Collection Manager provides convenient attribute masks for each of the attributes:

```
enum {
   collectionUser0Mask = 1L << collectionUser0Bit,
   collectionUser1Mask = 1L << collectionUser1Bit,
   collectionUser2Mask = 1L << collectionUser2Bit,
   collectionUser3Mask = 1L << collectionUser3Bit,
   collectionUser4Mask = 1L << collectionUser4Bit,
   collectionUser5Mask = 1L << collectionUser5Bit,
   collectionUser6Mask = 1L << collectionUser6Bit,
   collectionUser7Mask = 1L << collectionUser7Bit,
   collectionUser8Mask = 1L << collectionUser8Bit,
   collectionUser9Mask = 1L << collectionUser9Bit,
   collectionUser10Mask = 1L << collectionUser10Bit,
   collectionUser11Mask = 1L << collectionUser11Bit,
   collectionUser12Mask = 1L << collectionUser12Bit,
   collectionUser13Mask = 1L << collectionUser13Bit,
   collectionUser14Mask = 1L << collectionUser14Bit,
   collectionUser15Mask = 1L << collectionUser15Bit,

   collectionReserved0Mask = 1L << collectionReserved0Bit,
   collectionReserved1Mask = 1L << collectionReserved1Bit,
   collectionReserved2Mask = 1L << collectionReserved2Bit,
   collectionReserved3Mask = 1L << collectionReserved3Bit,
   collectionReserved4Mask = 1L << collectionReserved4Bit,
   collectionReserved5Mask = 1L << collectionReserved5Bit,
   collectionReserved6Mask = 1L << collectionReserved6Bit,
   collectionReserved7Mask = 1L << collectionReserved7Bit,
   collectionReserved8Mask = 1L << collectionReserved8Bit,
   collectionReserved9Mask = 1L << collectionReserved9Bit,
   collectionReserved10Mask = 1L << collectionReserved10Bit,
   collectionReserved11Mask = 1L << collectionReserved11Bit,
   collectionReserved12Mask = 1L << collectionReserved12Bit,
   collectionReserved13Mask = 1L << collectionReserved13Bit,

   collectionPersistenceMask = 1L << collectionPersistenceBit,
   collectionLockMask = 1L << collectionLockBit
};
```

You can use these attribute masks when testing or setting a particular collection item attribute.

For more information about collection attributes, see “Collection Attributes” beginning on page 5-9.

For an example using these attributes, see “Getting and Setting the Attributes of an Item” beginning on page 5-24.

Functions

This section describes the Collection Manager functions you can use to

- create and dispose of collection objects
- clone and copy collection objects and determine their owner counts
- get and set the default attributes for a collection object
- add and replace items in a collection
- remove items from a collection
- count items in a collection
- retrieve the variable-length data from a collection item
- get information about an item in a collection (for example, the index of the item, the size of the item’s data, or the item’s attribute flags)
- set the attribute flags of a collection item
- get information about the collection tags associated with the items of a collection
- flatten and unflatten collections
- use Macintosh Memory Manager handles to specify variable-length data

s **WARNING**

Many of the functions in this section require a reference to a collection object (that is, a reference of type `Collection`) as a parameter. When calling any of these functions, you must always provide a valid collection object reference. If you do not, the behavior of the function is undefined. s

Creating and Disposing of Collection Objects

The functions described in this section allow you to work with collections as objects in memory. With the functions in this section, you can create new, empty collection objects and dispose of existing collection objects.

You use the `NewCollection` function to create a new collection object and the `DisposeCollection` function to dispose of a collection object.

NewCollection

You can use the `NewCollection` function to create a new, empty collection object.

```
Collection NewCollection(void);
```

function result A reference to the newly created collection object.

DESCRIPTION

The `NewCollection` function allocates memory for a new collection object, initializes it, and returns a reference to it as the function result. The new collection contains no items and has an owner count of 1.

The `NewCollection` function does not return an error code; it returns `nil` if it cannot create a new collection object.

SPECIAL CONSIDERATIONS

You are responsible for disposing of collection objects that you create with this function when you no longer need them. See the next section, which describes the `DisposeCollection` function, for information about disposing of collection objects.

SEE ALSO

For general information about QuickDraw GX objects, see the chapter "Introduction to QuickDraw GX" in *Inside Macintosh: QuickDraw GX Objects*.

For examples using this function, see "Creating or Disposing of a Collection" beginning on page 5-14 and "Adding Items to a Collection" beginning on page 5-17.

To create a copy of an existing collection object, use the `CopyCollection` function, which is described in the previous section.

To dispose of a collection object, use the `DisposeCollection` function, which is described in the next section.

DisposeCollection

You can use the `DisposeCollection` function to dispose of a collection object.

```
void DisposeCollection(Collection target);
```

`target`	A reference to the collection object you want to dispose of.

DESCRIPTION

The `DisposeCollection` function decrements the owner count of the collection object referenced by the `target` parameter. If the resulting owner count is 0, this function releases the memory occupied by the collection object, and the collection object reference contained in the `target` parameter becomes invalid.

The behavior of this function is undefined if you do not provide a reference to a valid collection object in the `target` parameter.

SEE ALSO

For general information about QuickDraw GX objects, see the chapter “Introduction to QuickDraw GX” in *Inside Macintosh: QuickDraw GX Objects*.

For examples using this function, see “Creating or Disposing of a Collection” beginning on page 5-14.

To create a new collection object, use the `NewCollection` function, which is described on page 5-55.

To increment the owner count of a collection object, use the `CloneCollection` function, which is described in the next section. To determine the owner count of an existing collection object, use the `CountCollectionOwners` function, which is described on page 5-57.

Cloning and Copying Collection Objects

The functions described in this section allow you to examine and manipulate the owner count of a collection object or to make a complete copy of a collection object.

The `CloneCollection` function allows you to increment the owner count of a collection object. Typically, you use this function to signify the creation of a new reference to an existing collection object. The `CountCollectionOwners` function allows you to determine the current owner count of a collection object.

The `CopyCollection` function allows you to create a complete copy of a collection object. The new collection object contains a copy of every item in the original collection object.

CloneCollection

You can use the `CloneCollection` function to clone a collection object—that is, to increment its owner count.

```
Collection CloneCollection (Collection target);
```

`target`	A reference to the collection object you want to clone.
function result	A reference to the cloned collection. (This result is effectively a copy of the reference you provide in the `target` parameter.)

DESCRIPTION

The `CloneCollection` function increments the owner count of the collection object referenced by the `target` parameter, and, as a programming convenience, returns a reference to this collection as the function result.

Typically, you use this function to increment a collection object's owner count to represent a new reference to the collection object. For example, if you want two variables in your application to reference a single collection object, you can use this code to maintain the correct owner count:

```
firstReference = NewCollection();
secondReference = CloneCollection(firstReference);
```

Disposing of either reference (using the `DisposeCollection` function) simply decrements the collection's owner count. Disposing of the remaining reference decrements the owner count again and frees the memory associated with the collection.

The `CloneCollection` function does not return an error code.

SEE ALSO

For general information about QuickDraw GX objects, see the chapter "Introduction to QuickDraw GX" in *Inside Macintosh: QuickDraw GX Objects*.

For examples of this function, see "Cloning or Copying a Collection" beginning on page 5-14.

To decrement the owner count of a collection object, use the `DisposeCollection` function, which is described in the previous section. To determine the owner count of an existing collection object, use the `CountCollectionOwners` function, which is described in the next section.

To copy a collection object, use the `CopyCollection` function, which is described on page 5-57.

CountCollectionOwners

You can use the `CountCollectionOwners` function to determine the number of existing references to a collection object.

```
long CountCollectionOwners(Collection source);
```

`source`	The collection object whose owner count you want to determine.
function result	The owner count of the collection object.

DESCRIPTION

The `CountCollectionOwners` function returns as its function result the owner count of the collection object referenced by the `source` parameter.

SEE ALSO

For general information about QuickDraw GX objects, see the chapter “Introduction to QuickDraw GX Objects” in *Inside Macintosh: QuickDraw GX Objects*.

For examples of this function, see “Cloning or Copying a Collection” on page 5-14.

To increment the owner count of a collection object, use the `CloneCollection` function, which is described on page 5-56. To decrement the owner count of a collection object, use the `DisposeCollection` function, which is described on page 5-55.

CopyCollection

You use the `CopyCollection` function to create a copy of an existing collection.

```
Collection CopyCollection(Collection source, Collection target);
```

`source`	A reference to the collection object you want to copy.
`target`	A reference to a collection object to contain the copied collection items. You may provide `nil` for this parameter to request that the Collection Manager create a new collection object to hold the copied information.
function result	A reference to the collection object containing the copied information.

DESCRIPTION

The `CopyCollection` function copies all of the information (except the owner count and exception procedure) from the collection object referenced by the `source` parameter into the collection object referenced by the `target` parameter.

If you specify `nil` for the `target` parameter, this function creates a new collection object to copy the information into. (This function does not return an error code; it returns `nil` if it cannot create a new collection object.)

In either case, this function returns a reference to the collection object containing the copied information.

SEE ALSO

For general information about QuickDraw GX objects, see the chapter "Introduction to QuickDraw GX Objects" in *Inside Macintosh: QuickDraw GX Objects*.

For examples using this function, see "Cloning or Copying a Collection" on page 5-14.

To clone a collection object, use the `CloneCollection` function, which is described on page 5-56.

Getting and Setting the Exception Procedure for a Collection

The functions described in this section allow you to examine and alter a collection object's exception procedure. You are allowed to specify an exception procedure for any collection object. When the Collection Manager encounters an error while operating on a collection object, it calls that collection's exception procedure, sending it the result code associated with the error.

The `GetCollectionExceptionProc` function allows you to obtain a pointer to the exception procedure intalled in a specified collection.

The `SetCollectionExceptionProc` function allows you to install a new exception procedure into a collection.

You can find a description of exception procedures on page 5-101.

GetCollectionExceptionProc

You use the `GetCollectionExceptionProc` function to obtain a pointer to the exception procedure installed in a specified collection.

```
CollectionExceptionProc GetCollectionExceptionProc
                        (Collection source);
```

`source`	A reference to the collection object whose exception procedure you want to determine.
function result	A pointer to the exception procedure installed in the source collection object.

DESCRIPTION

The `GetCollectionExceptionProc` function returns as its function result a pointer to the exception procedure installed in the collection object referenced by the `source` parameter.

SEE ALSO

To install a new exception procedure in a collection object, use the `SetCollectionExceptionProc` function, which is described in the next section.

For more information about exception procedures, see page 5-101.

SetCollectionExceptionProc

You use the `SetCollectionExceptionProc` function to install an exception procedure in a collection object.

```
void SetCollectionExceptionProc(Collection target,
      CollectionExceptionProc newExceptionProc);
```

`target`	A reference to the collection object whose exception procedure you want to change.
`newExceptionProc`	A pointer to the new exception procedure.

DESCRIPTION

The `SetCollectionExceptionProc` function copies the function pointer from the `newExceptionProc` parameter into the collection object referenced by the `target` parameter.

SEE ALSO

For an example using this function, see “Installing an Exception Procedure” beginning on page 5-45.

To obtain a pointer to an existing exception procedure in a collection object, use the `GetCollectionExceptionProc` function, which is described in the previous section.

For more information about exception procedures, see page 5-101.

Getting and Setting the Default Attributes for a Collection

The functions described in this section allow you to examine and alter a collection object's default attributes. The default attributes of a collection specify the attributes that the Collection Manager assigns to new items added to the collection.

The `GetCollectionDefaultAttributes` function allows you to determine a collection's current default attributes. The `SetCollectionDefaultAttributes` function allows you to change a collection's default attributes.

GetCollectionDefaultAttributes

You use the `GetCollectionDefaultAttributes` function to examine the default attributes of a collection object.

```
long GetCollectionDefaultAttributes(Collection source);
```

`source`	A reference to the collection object whose default attributes you want to determine.
function result	A long word containing the bit flags that make up the collection's default attributes.

DESCRIPTION

The `GetCollectionDefaultAttributes` function returns as its function result the default attributes of the collection object referenced by the `source` parameter.

SEE ALSO

For information about default attributes for collection objects, see "Collection Attributes" beginning on page 5-9.

For information about attribute-related data types and enumerations, see page 5-49 through page 5-53.

To change the attributes of a collection object, use the `SetCollectionDefaultAttributes` function, which is described in the next section.

To examine the attributes of a specific item in a collection, use the functions described in "Getting Information About a Collection Item" beginning on page 5-76.

SetCollectionDefaultAttributes

You use the `SetCollectionDefaultAttributes` function to alter the default attributes of a collection object.

```
void SetCollectionDefaultAttributes(Collection target,
                                    long whichAttributes,
                                    long newAttributes);
```

`target`	A reference to the collection object whose default attributes you want to alter.
`whichAttributes`	A mask indicating which bit flags in the target collection's default attributes you want to alter.
`newAttributes`	A long word containing the new values for the bit flags.

DESCRIPTION

The `SetCollectionDefaultAttributes` function copies the values of bit flags from the `newAttributes` parameter into the default attributes of the target collection.

This function uses the `whichAttributes` parameter to determine which bits to copy. For every bit in the `whichAttributes` parameter, this function takes one of two actions:

- If the bit is set, this function copies the value of the corresponding bit from the `newAttributes` parameter into the corresponding bit of the default attributes of the target collection.
- If the bit is not set, the corresponding bit of the target collection's default attributes remains unchanged.

SEE ALSO

For information about default attributes for collection objects, see "Collection Attributes" beginning on page 5-9.

For information about attribute-related data types and enumerations, see page 5-49 through page 5-53.

For examples of this function, see "Changing the Default Attributes of a Collection" beginning on page 5-15.

To examine the attributes of a collection object, use the `GetCollectionDefaultAttributes` function, which is described in the previous section.

To change the attributes of a specific item in a collection, use the functions described in "Editing Item Attributes" beginning on page 5-82.

Adding and Replacing Items in a Collection

The functions described in this section allow you to add items to a collection and replace items already in a collection.

The `AddCollectionItem` function allows you to add a new item to a collection. You can also use this function to replace a collection item by specifying its collection tag and collection ID.

The `ReplaceIndexedCollectionItem` function allows you to replace a collection item by specifying its collection index.

AddCollectionItem

You use the `AddCollectionItem` function to add a new item to a collection or to replace an existing item in a collection.

```
OSErr AddCollectionItem (Collection target,
                         CollectionTag tag, long id,
                         long itemSize, void *itemData);
```

`target`	A reference to the collection you want to add the item to.
`tag`	The collection tag you want to associate with the new item.
`id`	The collection ID you want to associate with the new item.
`itemSize`	The size in bytes of the item's variable-length data.
`itemData`	A pointer to the item's variable-length data.

DESCRIPTION

The `AddCollectionItem` function adds an item to the collection referenced by the `target` parameter. This new item contains

- the collection tag specified by the `tag` parameter
- the collection ID specified by the `id` parameter
- the attributes specified by the default attributes of the target collection
- the variable-length data specified by the `itemSize` and `itemData` parameters

This function copies the information pointed to by the `itemData` parameter into the new item; after calling this function, you may alter this information or free the memory pointed to by this parameter without affecting the collection.

If the target collection already contains an item with the same collection tag and collection ID as specified in the `tag` and `id` parameters, this function removes the original item and replaces it with the new one, unless the existing item is locked. If it is locked, this function returns a `collectionItemLockedErr` result code.

The `itemSize` parameter determines how many bytes of information this function copies into the new item. If you specify 0 for this parameter, or provide `nil` for the `itemData` parameter, this function copies no information into the variable-length data of the new item, or removes the variable-length data if the item already exists.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.
`collectionItemLockedErr`	–5750	Can't replace locked item.

SEE ALSO

For information about collection items, see "Collection Items" beginning on page 5-8.

For information about locking collection items, see "Getting and Setting the Attributes of an Item" beginning on page 5-24. To lock a collection item, use the functions described in "Editing Item Attributes" beginning on page 5-82.

For examples using this function, see "Adding Items to a Collection" beginning on page 5-17 and "Replacing Items in a Collection" beginning on page 5-28.

To replace a collection item using the item's index (rather than the item's tag and ID), use the `ReplaceIndexedCollectionItem` function, described in the next section.

To remove an item from a collection, use the functions described in "Removing Items From a Collection" beginning on page 5-65.

ReplaceIndexedCollectionItem

You use the `ReplaceIndexedCollectionItem` function to replace the variable-length data of an item in a collection given the item's index.

```
OSErr ReplaceIndexedCollectionItem(Collection target, long index,
                                   long itemSize, void *itemData);
```

`target`	A reference to the collection containing the item you want to replace.
`index`	The collection index associated with the item to replace.
`itemSize`	The item's size.
`itemData`	A pointer to the item's data.

DESCRIPTION

The `ReplaceIndexedCollectionItem` function replaces the variable-length data associated with an item in the target collection. You specify which item to replace using the `index` parameter. If the target collection does not contain an item whose collection index matches the value of the `index` parameter, this function returns a `collectionIndexRangeErr` result code.

If the target collection does contain an item with the specified index, this function replaces that item with a new item (if the existing item is not locked—if it is, this function returns a `collectionItemLockedErr` result code). The new item contains

- the same collection tag as the original item
- the same collection ID as the original item
- the same attributes as the original item
- the variable-length data specified by the `itemSize` and `itemData` parameters

This function copies the information pointed to by the `itemData` parameter into the new item; after calling this function, you may alter this information or free the memory pointed to by this parameter without affecting the collection.

The `itemSize` parameter determines how many bytes of information this function copies into the new item. If you specify 0 for this parameter, or provide `nil` for the `itemData` parameter, this function copies no information into the variable-length data of the new item, or removes the variable-length data if the item already exists.

RESULT CODES

`memFullErr`	–108	Can’t allocate memory.
`collectionItemLockedErr`	–5750	Can’t replace locked item.
`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection items, see “Collection Items” beginning on page 5-8.

For information about locking collection items, see “Getting and Setting the Attributes of an Item” beginning on page 5-24. To lock a collection item, use the functions described in “Editing Item Attributes” beginning on page 5-82.

To replace a collection item using the item’s tag and ID (rather than the item’s index), use the `ReplaceIndexedCollectionItem` function, described on page 5-63.

To remove an item from a collection, use the functions described in the next section.

Removing Items From a Collection

The functions described in this section allow you to remove items from a collection.

The `RemoveCollectionItem` and `RemoveIndexedCollectionItem` functions allow you to remove a single item from a collection. You use the `RemoveCollectionItem` function if you want to specify the item to remove using the item's tag and ID. You use the `RemoveIndexedCollectionItem` function if you want to specify the item to remove using the item's index.

The `PurgeCollection` function allows you to remove from a collection all the items whose attributes match a specified pattern.

The `PurgeCollectionTag` function allows you to remove from a collection all the items with a specified collection tag.

The `EmptyCollection` function allows you to remove every item from a collection.

RemoveCollectionItem

You can use the `RemoveCollectionItem` function to remove an item from a collection given the item's associated collection tag and collection ID.

```
OSErr RemoveCollectionItem (Collection target,
                            CollectionTag tag, long id);
```

`target`	A reference to the collection object from which you want to remove the item.
`tag`	The collection tag associated with the item you want to remove.
`id`	The collection ID associated with the item you want to remove.

DESCRIPTION

The `RemoveCollectionItem` function removes the item specified by the `tag` and `id` parameters from the collection referenced by the `target` parameter. This function removes the specified item even if its lock attribute is set.

If the target collection does not contain an item whose collection tag and collection ID match the values in the `tag` and `id` parameters, this function returns a `collectionItemNotFoundErr` result code.

RESULT CODES

`collectionItemNotFoundErr`	–5751	Can't locate item.

SEE ALSO

For information about collection items, see “Collection Items” beginning on page 5-8.

For examples using this function, see “Removing Items From a Collection” beginning on page 5-30.

To remove a collection item using the item’s index (rather than the item’s tag and ID), use the `RemoveIndexedCollectionItem` function, described in the next section.

To replace an item in a collection, use the functions described in “Adding and Replacing Items in a Collection” beginning on page 5-62.

RemoveIndexedCollectionItem

You can use the `RemoveIndexedCollectionItem` function to remove an item from a collection given the item’s index.

```
OSErr RemoveIndexedCollectionItem(Collection target, long index);
```

`target`	A reference to the collection object from which you want to remove the item.
`index`	The collection index of the item you want to remove.

DESCRIPTION

The `RemoveIndexedCollectionItem` function removes the item specified by the `index` parameter from the collection referenced by the `target` parameter. This function removes the specified item even if its lock attribute is set.

If the target collection does not contain an item whose collection index matches the values in the `index` parameter, this function returns a `collectionIndexRangeErr` result code.

RESULT CODES

`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection items, see “Collection Items” beginning on page 5-8.

For examples using this function, see “Removing Items From a Collection” beginning on page 5-30.

To remove a collection item using the item’s tag and ID (rather than the item’s index), use the `RemoveCollectionItem` function, described in the previous section.

To replace an item in a collection, use the functions described in “Adding and Replacing Items in a Collection” beginning on page 5-62.

PurgeCollection

You use the `PurgeCollection` function to remove all items in a collection whose attributes match a specified pattern.

```
void PurgeCollection(Collection target,
                     long whichAttributes,
                     long matchingAttributes);
```

`target` A reference to the collection object containing the items you want to remove.

`whichAttributes`
A mask indicating which attributes you want to test.

`matchingAttributes`
A long word containing the values of the attributes you want to match.

DESCRIPTION

The `PurgeCollection` function removes from the target collection any items whose attributes match the criteria you specify in the `whichAttributes` and `matchingAttributes` parameters.

The `whichAttributes` parameter allows you to specify which attributes this function examines. You should set the bits of the `whichAttributes` parameter that correspond to the attributes you want to test.

This function compares the specified attributes of each item in the target collection with the corresponding attributes in the `matchingAttributes` parameter. If the values of all the specified attributes match, the function removes the item. To avoid purging locked items, you should clear the lock attribute in the `whichAttributes` and `matchingAttributes` parameters.

SEE ALSO

For information about collection items, see “Collection Items” beginning on page 5-8.

For examples using this function, see “Removing Items From a Collection” beginning on page 5-30.

To remove all of the items in a collection with a specified collection tag, use the `PurgeCollectionTag` function, described in the next section.

To remove every item in a collection, use the `EmptyCollection` function, described on page 5-68.

PurgeCollectionTag

You use the `PurgeCollectionTag` function to remove from a collection all items with a specific collection tag.

```
void PurgeCollectionTag(Collection target,
                        CollectionTag tag);
```

`target`	A reference to the collection object containing the items you want to remove.
`tag`	The collection tag associated with the items to remove.

DESCRIPTION

The `PurgeCollectionTag` function removes from the target collection all items whose collection tag matches the value of the `tag` parameter. This function removes locked and unlocked items.

SEE ALSO

For information about collection items, see "Collection Items" beginning on page 5-8.

For examples using this function, see "Removing Items From a Collection" beginning on page 5-30.

To remove all of the items in a collection whose attributes match a specified pattern, use the `PurgeCollection` function, described in the previous section.

To remove every item in a collection, use the `EmptyCollection` function, described in the next section.

EmptyCollection

You use the `EmptyCollection` function to remove every item in a collection.

```
void EmptyCollection (Collection target);
```

`target`	A reference to the collection object you want to empty.

DESCRIPTION

This function removes every item in the collection referenced by the `target` parameter. This function provides the fastest mechanism for emptying a collection.

SEE ALSO

For information about collection items, see “Collection Items” beginning on page 5-8.

To remove all of the items in a collection whose attributes match a specified pattern, use the `PurgeCollection` function, described on page 5-67.

To remove all of the items in a collection with a specified collection tag, use the `PurgeCollectionTag` function, described in the previous section.

Counting Items in a Collection

The functions described in this section allow you to count items in a collection.

The `CountCollectionItems` function allows you to determine the total number of items in a collection.

The `CountTaggedCollectionItems` function allows you to determine the total number of items in a collection that have a specified collection tag.

CountCollectionItems

You can use the `CountCollectionItems` function to determine the total number of items in a collection.

```
long CountCollectionItems(Collection source);
```

`source`	A reference to the collection object whose items you want to count.
function result	The total number of items in the source collection.

DESCRIPTION

The `CountCollectionItems` function returns as its function result the total number of items in the collection referenced by the `source` parameter.

SEE ALSO

For information about collection items, see “Collection Items” beginning on page 5-8.

For examples using this function, see “Adding Items to a Collection” beginning on page 5-17.

To count the items in a collection that have a specified collection tag, use the `CountTaggedCollectionItems` function, described in the next section.

CountTaggedCollectionItems

You can use the `CountTaggedCollectionItems` function to obtain the total number of items in a collection that have a specified collection tag.

```
long CountTaggedCollectionItems(Collection source,
                                CollectionTag tag);
```

`source`	A reference to the collection object whose items you want to count.
`tag`	The collection tag associated with the items you want to count.
function result	The total number of items in the source collection whose collection tags match the value specified in the `tag` parameter.

DESCRIPTION

The `CountTaggedCollectionItems` function returns as its function result the total number of items in the source collection whose collection tags match the value you specify in the `tag` parameter.

SEE ALSO

For information about collection items, see “Collection Items” beginning on page 5-8.

For examples of this function, see “Adding Items to a Collection” beginning on page 5-17.

To count all of the items in a collection, use the `CountCollectionItems` function, described in the previous section.

Retrieving the Variable-Length Data From an Item

The functions described in this section allow you to obtain a copy of the variable-length data associated with a specified collection item.

The `GetCollectionItem` function allows you to retrieve data from an item given its collection tag and collection ID. The `GetIndexedCollectionItem` function allows you to retrieve data from an item given its collection index.

The `GetTaggedCollectionItem` function provides another way for you to specify the item whose data you want to retrieve. With this function, you specify the item using the item’s collection tag and the item’s tag list position. See “Methods of Identifying Collection Items” beginning on page 5-11 for a discussion of collection tags and tag list positions.

GetCollectionItem

You can use the `GetCollectionItem` function to obtain a copy of the variable-length data associated with a collection item given the item's collection tag and collection ID.

```
OSErr GetCollectionItem(Collection source,
                        CollectionTag tag,
                        long id,
                        long *itemSize,
                        void *itemData);
```

`source`	A reference to the collection object containing the item whose data you want to retrieve.
`tag`	The collection tag associated with the item whose data you want to retrieve.
`id`	The collection ID associated with the item whose data you want to retrieve.
`itemSize`	A pointer to a `long` value indicating the number of bytes of data you want returned in the `itemData` parameter. On return, this value indicates the size in bytes of the variable-length data associated with the specified item. You may specify the constant `dontWantSize` for this parameter to indicate that you want to copy all the specified item's variable-length data and you do not want to determine the size of this data.
`itemData`	A pointer to a block of memory to contain the item's data. On return, this memory contains a copy of the data associated with the specified item. You may specify the constant `dontWantData` for this parameter if you do not want a copy of the item's data.

DESCRIPTION

The `GetCollectionItem` function allows you to obtain a copy of the variable-length data associated with a specific collection item. You specify a collection object using the `source` parameter and you specify an item in that collection using the `tag` and `id` parameters.

You use the `itemSize` parameter to specify how many bytes of data to return in the `itemData` parameter. You may specify the constant `dontWantSize` for this parameter to indicate that you want to copy all of the variable-length data from the specified item into the `itemData` parameter. You may specify a value for the `itemSize` parameter that is greater than the actual number of bytes in the specified item's variable-length data; however, this function never returns in the `itemData` parameter more data than contained in the specified item's variable-length data.

This function returns information in the `itemSize` and `itemData` parameters:

- If you provide a pointer in the `itemSize` parameter, the function uses this parameter to return the size in bytes of the variable-length data associated with the specified collection item.
- If you provide a pointer in the `itemData` parameter, the function uses this parameter to return a copy of the variable-length data associated with the specified collection item.

If you don't know the size of the item you want to retrieve, you typically call this function twice. The first time you provide a pointer in the `itemSize` parameter to determine the size of the specified item's data and you specify `dontWantData` for the `itemData` parameter. Then you allocate a memory block large enough to hold a copy of the item's data. Then you call the function a second time. This time you specify the constant `dontWantSize` for the `itemSize` parameter and provide a pointer to the allocated memory block for the `itemData` parameter. The function then copies the data into the allocated block of memory.

RESULT CODES

`collectionItemNotFoundErr`	–5751	Can't locate item.

SEE ALSO

For information about collection items and their associated collection tags, collection IDs, and variable-length data, see "Collection Items" beginning on page 5-8.

For examples using this function, see "Retrieving the Variable-Length Data From an Item" beginning on page 5-33.

To retrieve the data associated with a collection item given its collection index (rather than its collection tag and ID), use the `GetIndexedCollectionItem` function, described in the next section.

GetIndexedCollectionItem

You can use the `GetIndexedCollectionItem` function to obtain a copy of the variable-length data associated with a collection item given the item's collection index.

```
OSErr GetIndexedCollectionItem(Collection source,
                               long index,
                               long *itemSize,
                               void *itemData);
```

`source`	A reference to the collection object containing the item whose data you want to retrieve.
`index`	The collection index associated with the item whose data you want to retrieve.
`itemSize`	A pointer to a `long` value indicating the number of bytes of data you want returned in the `itemData` parameter. On return, this value indicates the size in bytes of the variable-length data associated with the specified item. You may specify the constant `dontWantSize` for this parameter to indicate that you want to copy all of the specified item's variable-length data and you do not want to determine the size of this data.
`itemData`	A pointer to a block of memory to contain the item's data. On return, this memory contains a copy of the data associated with the specified item. You may specify the constant `dontWantData` for this parameter if you do not want a copy of the item's data.

DESCRIPTION

The `GetIndexedCollectionItem` function allows you to obtain a copy of the variable-length data associated with a specific collection item. You specify a collection object using the `source` parameter and you specify an item in that collection using the `index` parameter.

You use the `itemSize` parameter to specify how many bytes of data to return in the `itemData` parameter. You may specify the constant `dontWantSize` for this parameter to indicate that you want to copy all of the variable-length data from the specified item into the `itemData` parameter. You may specify a value for the `itemSize` parameter that is greater than the actual number of bytes in the specified item's variable-length data; however, this function never returns in the `itemData` parameter more data than contained in the specified item's variable-length data.

This function returns information in the `itemSize` and `itemData` parameters:

- If you provide a pointer in the `itemSize` parameter, the function uses this parameter to return the size in bytes of the variable-length data associated with the specified collection item.
- If you provide a pointer in the `itemData` parameter, the function uses this parameter to return a copy of the variable-length data associated with the specified collection item.

If you don't know the size of the item you want to retrieve, you typically call this function twice. The first time you provide a pointer in the `itemSize` parameter to determine the size of the specified item's data and you specify the constant `dontWantData` for the `itemData` parameter. Then you allocate a memory block large enough to hold a copy of the item's data. Then you call the function a second time. This time you specify the constant `dontWantSize` for the `itemSize` parameter and provide a pointer to the allocated memory block for the `itemData` parameter. The function then copies the data into the allocated block of memory.

RESULT CODES

`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection items and their associated variable-length data, see "Collection Items" beginning on page 5-8. For information about collection indexes, see "Methods of Identifying Collection Items" beginning on page 5-11.

For examples using this function, see "Retrieving the Variable-Length Data From an Item" beginning on page 5-33.

To retrieve the data associated with a collection item given its collection tag and ID (rather than its collection index), use the `GetCollectionItem` function, described in the previous section.

GetTaggedCollectionItem

You can use the `GetTaggedCollectionItem` function to obtain a copy of the variable-length data associated with a collection item given the item's collection tag and tag list position.

```
OSErr GetTaggedCollectionItem(Collection source,
                              CollectionTag tag,
                              long position,
                              long *itemSize,
                              void *itemData);
```

`source`	A reference to the collection object containing the item whose data you want to retrieve.
`tag`	The collection tag associated with the item whose data you want to retrieve.
`position`	The tag list position associated with the specific item.
`itemSize`	A pointer to a `long` value indicating the number of bytes of data you want returned in the `itemData` parameter. On return, this value indicates the size in bytes of the variable-length data associated with the specified item. You may specify the constant `dontWantSize` for this parameter to indicate that you want to copy all of the specified item's variable-length data and you do not want to determine the size of this data.
`itemData`	A pointer to a block of memory to contain the item's data. On return, this memory contains a copy of the data associated with the specified item. You may specify the constant `dontWantData` for this parameter if you do not want a copy of the item's data.

DESCRIPTION

The `GetTaggedCollectionItem` function allows you to obtain a copy of the variable-length data associated with a specific collection item. You specify a collection object using the `source` parameter; you specify the item in that collection using the `tag` and `position` parameters. In the `tag` parameter you specify the collection tag of the desired item and in the `position` parameter you specify the tag list position of the desired item.

Remember that a tag list position is the sequential index that determines an item given a specific collection tag. For example:

- A tag list position of 1 indicates the first item with the specified tag.
- A tag list position of 2 indicates the second item with the specified tag.

By sequentially incrementing the `position` parameter, you can use this function to step through all of the items in a collection without knowing their collection IDs.

This function returns information in the `itemSize` and `itemData` parameters:

- If you provide a pointer in the `itemSize` parameter, the function uses this parameter to return the size in bytes of the variable-length data associated with the specified collection item.
- If you provide a pointer in the `itemData` parameter, the function uses this parameter to return a copy of the variable-length data associated with the specified collection item.

If you don't know the size of the item you want to retrieve, you typically call this function twice. The first time you provide a pointer in the `itemSize` parameter to determine the size of the specified item's data and you specify the constant `dontWantData` for the `itemData` parameter. Then you allocate a memory block large enough to hold a copy of the item's data. Then you call the function a second time. This time you specify the constant `dontWantSize` for the `itemSize` parameter and provide a pointer to the allocated memory block for the `itemData` parameter. The function then copies the data into the allocated block of memory.

RESULT CODES

`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection items and their associated collection tags and variable-length data, see "Collection Items" beginning on page 5-8. For information about tag list positions, see "Methods of Identifying Collection Items" beginning on page 5-11.

For examples of this function, see "Retrieving the Variable-Length Data From an Item" beginning on page 5-33.

To retrieve the data associated with a collection item given its collection tag and ID, use the `GetCollectionItem` function, described on page 5-71.

To retrieve the data associated with a collection item given its collection index, use the `GetIndexedCollectionItem` function, described in the previous section.

Getting Information About a Collection Item

The functions described in this section allow you to determine information about a collection item, such as the item's collection index, the item's size, and the item's attributes.

Each function in this section provides a different way for you to specify which collection item you want to examine:

- The `GetCollectionItemInfo` function requires you to specify the item's collection tag and collection ID.
- The `GetIndexedCollectionItemInfo` function requires you to specify the item's collection index.
- The `GetTaggedCollectionItemInfo` function requires you to specify the item's collection tag and tag list position.

GetCollectionItemInfo

You use the `GetCollectionItemInfo` function to obtain information about a specific collection item given the item's collection tag and collection ID.

```
OSErr GetCollectionItemInfo(Collection source,
                            CollectionTag tag,
                            long id,
                            long *index,
                            long *itemSize,
                            long *attributes);
```

`source`	A reference to the collection object containing the item you want to obtain information about.
`tag`	The collection tag associated with the item you want to obtain information about.
`id`	The collection ID associated with the item you want to obtain information about.

`index`	A pointer to a `long` value. On return, this value represents the collection index of the specified item. You may specify the constant `dontWantIndex` for this parameter if you do not want to determine the specified item's collection index.
`itemSize`	A pointer to a `long` value. On return, this value indicates the size in bytes of the variable-length data associated with the specified item. You may specify the constant `dontWantSize` for this parameter to indicate that you do not want to determine the size of this data.
`attributes`	A pointer to a `long` value. On return, this value contains a copy of the attributes associated with the specified item. You may specify the constant `dontWantAttributes` for this parameter if you do not want a copy of the item's attributes.

DESCRIPTION

The `GetCollectionItemInfo` function allows you to obtain information about a specific collection item in the collection referenced by the `source` parameter. You specify the collection item by specifying the item's collection tag and collection ID in the `tag` and `id` parameters.

This function returns information in the `index`, `itemSize`, and `attributes` parameters:

- If you provide a pointer in the `index` parameter, the function uses this parameter to return the collection index of the specified item. Once you have determined an item's collection index, you can use it to specify the item when calling Collection Manager functions, rather than using the item's collection tag and collection ID. Specifying collection items using their collection index, rather than using the item's collection tag and collection ID, generally results in improved performance.
- If you provide a pointer in the `itemSize` parameter, the function uses this parameter to return the size in bytes of the variable-length data associated with the specified collection item.
- If you provide a pointer in the `attributes` parameter, the function uses this parameter to return a copy of the attributes associated with the specified collection item.

RESULT CODES

`collectionItemNotFoundErr`	–5751	Can't locate item.

SEE ALSO

For information about collection items and their associated collection tags, collection IDs, and variable-length data, see "Collection Items" beginning on page 5-8.

For examples of this function, see "Determining the Collection Index of an Item" beginning on page 5-19, "Determining the Size of an Item's Variable-Length Data" beginning on page 5-22, and "Getting and Setting the Attributes of an Item" beginning on page 5-24.

To obtain information about a collection item using the collection index to specify the item, use the `GetIndexedCollectionItemInfo` function, described in the next section.

To obtain information about a collection item using the collection tag and tag list position to specify the item, use the `GetTaggedCollectionItemInfo` function, described on page 5-80.

GetIndexedCollectionItemInfo

You use the `GetIndexedCollectionItemInfo` function to obtain information about a specific collection item given the item's collection index.

```
OSErr GetIndexedCollectionItemInfo (Collection source,
                                     long index,
                                     CollectionTag *tag,
                                     long *id,
                                     long *itemSize,
                                     long *attributes);
```

`source`	A reference to the collection object containing the item you want to obtain information about.
`index`	The collection index associated with the item you want to obtain information about.
`tag`	A pointer to a collection tag. On return, the collection tag associated with the specified item. You may specify the constant `dontWantTag` for this parameter if you do not want to determine the specified item's collection tag.
`id`	A pointer to a `long` value. On return, the collection ID associated with the specified item. You may specify the constant `dontWantId` for this parameter if you do not want to determine the specified item's collection ID.

`itemSize`	A pointer to a `long` value. On return, this value indicates the size in bytes of the data associated with the specified item. You may specify the constant `dontWantSize` for this parameter if you do not want to determine the specified item's data size.
`attributes`	A pointer to a `long` value. On return, this value contains a copy of the attributes associated with the specified item. You may specify the constant `dontWantAttributes` for this parameter if you do not want a copy of the item's attributes.

DESCRIPTION

The `GetIndexedCollectionItemInfo` function allows you to obtain information about a specific collection item in the collection referenced by the `source` parameter. You specify the collection item by specifying the item's collection index in the `index` parameter.

This function returns information in the `tag`, `id`, `itemSize`, and `attributes` parameters:

- If you provide a pointer in the `tag` parameter, the function uses this parameter to return the collection tag of the specified item.
- If you provide a pointer in the `id` parameter, the function uses this parameter to return the collection ID of the specified item.
- If you provide a pointer in the `itemSize` parameter, the function uses this parameter to return the size in bytes of the variable-length data associated with the specified collection item.
- If you provide a pointer in the `attributes` parameter, the function uses this parameter to return a copy of the attributes associated with the specified collection item.

RESULT CODES

`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection items and their associated collection tags, collection IDs, and variable-length data, see "Collection Items" beginning on page 5-8. For information about collection indexes, see "Methods of Identifying Collection Items" beginning on page 5-11.

For examples of this function, see "Determining the Collection Index of an Item" beginning on page 5-19, "Determining the Size of an Item's Variable-Length Data" beginning on page 5-22, and "Getting and Setting the Attributes of an Item" beginning on page 5-24.

To obtain information about a collection item using the collection tag and collection ID to specify the item, use the `GetCollectionItemInfo` function, described in the previous section.

To obtain information about a collection item using the collection tag and tag list position to specify the item, use the `GetTaggedCollectionItemInfo` function, described in the next section.

GetTaggedCollectionItemInfo

You use the `GetTaggedCollectionItemInfo` function to obtain information about a specific collection item given the item's collection tag and tag list position.

```
OSErr GetTaggedCollectionItemInfo(Collection source,
                                  CollectionTag tag,
                                  long position,
                                  long *id,
                                  long *index,
                                  long *itemSize,
                                  void *attributes);
```

`source`	A reference to the collection object containing the item you want to obtain information about.
`tag`	The collection tag associated with the item you want to obtain information about.
`position`	The tag list position of the item you want to obtain information about.
`id`	A pointer to a `long` value. On return, this value represents the collection ID associated with the specified item. You may specify the constant `dontWantId` for this parameter if you do not want to determine the specified item's collection ID.
`index`	A pointer to a `long` value. On return, this value represents the collection index of the specified item. You may specify the constant `dontWantIndex` for this parameter if you do not want to determine the specified item's collection index.
`itemSize`	A pointer to a `long` value. On return, this value indicates the size in bytes of the data associated with the specified item. You may specify the constant `dontWantSize` for this parameter if you do not want to determine the specified item's data size.
`attributes`	A pointer to a `long` value. On return, this value contains a copy of the attributes associated with the specified item. You may specify the constant `dontWantAttributes` for this parameter if you do not want a copy of the item's attributes.

DESCRIPTION

The `GetTaggedCollectionItemInfo` function allows you to obtain information about a specific collection item in the collection referenced by the source parameter. You specify the item in the source collection using the `tag` and `position` parameters. In the `tag` parameter you specify the collection tag of the desired item and in the `position` parameter you specify the tag list position of the desired item.

Remember that a collection tag and a tag list position uniquely identify a collection item. The tag list position indicates where the collection item would lie in a list made up of all the collection items with the same collection tag. For example:

- A tag list position of 1 indicates the first item with the specified tag.
- A tag list position of 2 indicates the second item with the specified tag.

By sequentially incrementing the `position` parameter, you can use this function to step through all of the items in a collection that share a collection tag without knowing their collection IDs.

The `GetTaggedCollectionItemInfo` function returns information in the `id`, `index`, `itemSize`, and `attributes` parameters:

- If you provide a pointer in the `id` parameter, the function uses this parameter to return the collection ID of the specified item.
- If you provide a pointer in the `index` parameter, the function uses this parameter to return the collection index of the specified item.
- If you provide a pointer in the `itemSize` parameter, the function uses this parameter to return the size in bytes of the variable-length data associated with the specified collection item.
- If you provide a pointer in the `attributes` parameter, the function uses this parameter to return a copy of the attributes associated with the specified collection item.

RESULT CODES

`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection items and their associated collection tags, collection IDs, and variable-length data, see “Collection Items” beginning on page 5-8. For information about tag list positions, see “Methods of Identifying Collection Items” beginning on page 5-11.

For examples of this function, see “Determining the Collection Index of an Item” beginning on page 5-19, “Determining the Size of an Item’s Variable-Length Data” beginning on page 5-22, and “Getting and Setting the Attributes of an Item” beginning on page 5-24.

To obtain information about a collection item using the collection tag and collection ID to specify the item, use the `GetCollectionItemInfo` function, described on page 5-76.

To obtain information about a collection item using the collection index to specify the item, use the `GetIndexedCollectionItemInfo` function, described in the previous section.

Editing Item Attributes

The functions described in this section allow you to edit the attributes of a collection item. Each function in this section provides a different way for you to specify the collection item whose attributes you want to edit:

- The `SetCollectionItemInfo` function requires you to specify the item's collection tag and collection ID.
- The `SetIndexedCollectionItemInfo` function requires you to specify the item's collection index.

SetCollectionItemInfo

You use the `SetCollectionItemInfo` function to edit the attributes of a specific collection item given the item's collection tag and collection ID.

```
OSErr SetCollectionItemInfo(Collection target,
                            CollectionTag tag,
                            long id,
                            long whichAttributes,
                            long newAttributes);
```

`target`	A reference to the collection object containing the item whose attributes you want to edit.
`tag`	The collection tag associated with the item whose attributes you want to edit.
`id`	The collection ID associated with the item whose attributes you want to edit.
`whichAttributes`	A mask indicating which attributes you want to edit.
`newAttributes`	A long word containing the new settings for the attributes.

DESCRIPTION

The `SetCollectionItemInfo` function allows you to edit the attributes of a specific collection item in the collection referenced by the `target` parameter. You specify the collection item by specifying the item's collection tag and collection ID in the `tag` and `id` parameters.

This function copies bit values from the `newAttributes` parameter to the attributes associated with the specified item.

This function uses the `whichAttributes` parameter to determine which bits to copy. For every bit in the `whichAttributes` parameter, this function takes one of two actions:

- If the bit is set, this function copies the value of the corresponding bit from the `newAttributes` parameter into the corresponding bit of the attributes associated with the specified item.
- If the bit is not set, the corresponding bit of the specified item's attributes remains unchanged.

The `whichAttributes` parameter allows you to change the values of specific bits in the specified item's attributes without affecting the values of other bits.

RESULT CODES

`collectionItemNotFoundErr`	–5751	Can't locate item.

SEE ALSO

For information about collection attributes, see "Collection Attributes" beginning on page 5-9.

For attribute-related data types and enumerations, see page 5-49 through page 5-53.

For examples of this function, see "Getting and Setting the Attributes of an Item" beginning on page 5-24.

To obtain information about a collection item using the collection index to specify the item, use the `SetIndexedCollectionItemInfo` function, described in the next section.

SetIndexedCollectionItemInfo

You use the `SetIndexedCollectionItemInfo` function to edit the attributes of a specific collection item given the item's collection index.

```
OSErr SetIndexedCollectionItemInfo(Collection target,
                                   long index,
                                   long whichAttributes,
                                   long newAttributes);
```

`target`	A reference to the collection object containing the item whose attributes you want to edit.
`index`	The collection index of the item whose attributes you want to edit.
`whichAttributes`	A mask indicating which attributes you want to edit.
`newAttributes`	A long word containing the new settings for the attributes.

DESCRIPTION

The `SetIndexedCollectionItemInfo` function allows you to edit the attributes of a specific collection item in the collection referenced by the `target` parameter. You specify the collection item by specifying the item's collection index in the `index` parameter.

This function copies bit values from the `newAttributes` parameter to the attributes associated with the specified item.

This function uses the `whichAttributes` parameter to determine which bits to copy. For every bit in the `whichAttributes` parameter, this function takes one of two actions:

- If the bit is set, this function copies the value of the corresponding bit from the `newAttributes` parameter into the corresponding bit of the attributes associated with the specified item.
- If the bit is not set, the corresponding bit of the specified item's attributes remains unchanged.

The `whichAttributes` parameter allows you to change the values of specific bits in the specified item's attributes without affecting the values of other bits.

RESULT CODES

`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection attributes, see "Collection Attributes" beginning on page 5-9.

For attribute-related data types and enumerations, see page 5-49 through page 5-53.

For examples of this function, see "Getting and Setting the Attributes of an Item" beginning on page 5-24.

To edit the attributes of collection item using the collection tag and collection ID (rather than the collection index) to specify the item, use the `SetCollectionItemInfo` function, described in the previous section.

To examine the attributes of a collection item, use the functions described in "Getting Information About a Collection Item" beginning on page 5-76.

Getting Information About Collection Tags

You use the `CollectionTagExists` function to identify if a specific collection tag exists within a collection. You use the `CountCollectionTags` function to obtain the number of unique collection tags in a collection.

You use the `GetIndexedCollectionTag` function to obtain a specific collection tag from a collection.

CollectionTagExists

You can use the `CollectionTagExists` function to identify if any of the items in a specified collection contain a specified collection tag.

```
Boolean CollectionTagExists(Collection source,
                            CollectionTag tag);
```

`source`	A reference to the collection object you want to search for a specific collection tag.
`tag`	The collection tag to search for in the collection.

function result	A Boolean value indicating whether the source collection contains any items that contain the specified tag.

DESCRIPTION

The `CollectionTagExists` function returns as its function result a Boolean value indicating whether any of the items in the collection referenced by the `source` parameter contain the collection tag specified by the `tag` parameter.

SEE ALSO

For information about collection tags, see “Collection Items” beginning on page 5-8. For information about data types related to collection tags, see the section “Collection Tags” on page 5-49.

CountCollectionTags

You use the `CountCollectionTags` function to determine the number of distinct collection tags contained by the items of a specified collection.

```
long CountCollectionTags(Collection source);
```

`source`	A reference to the collection object whose collection tags you want to count.
function result	The number of distinct collection tags contained by the items of the source collection.

DESCRIPTION

The `CountCollectionTags` function returns as its function result the number of distinct collection tags contained by the items of the collection referenced by the `source` parameter.

SEE ALSO

For information about collection tags, see “Collection Items” beginning on page 5-8. For information about data types related to collection tags, see the section “Collection Tags” on page 5-49.

For an example of this function, see “Examining the Collection Tags of a Collection” beginning on page 5-35.

GetIndexedCollectionTag

Each collection object contains a number of distinct collection tags. You can use the `GetIndexedCollectionTag` function to examine a specific collection tag contained in a collection.

```
OSErr GetIndexedCollectionTag(Collection source,
                              long whichTag,
                              CollectionTag *tag);
```

`source`	The collection from which to obtain a specific collection tag.
`whichTag`	The position of the desired collection tag in the source collection's list of distinct collection tags.
`tag`	A pointer to a collection tag. On return, the collection tag that lies at the specified position in the list of distinct collection tags contained in the source collection.

DESCRIPTION

The `GetIndexedCollectionTag` function returns in the `tag` parameter the collection tag that lies at the position specified by the `whichTag` parameter in the list of distinct collection tags contained in the collection referenced by the `source` parameter.

By sequentially incrementing the value of the `whichTag` parameter from 1 to the result of the `CountCollectionTags` function, you can use this function to determine every collection tag contained in a collection.

RESULT CODES

`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection tags, see "Collection Items" beginning on page 5-8. For information about data types related to collection tags, see the section "Collection Tags" beginning on page 5-49.

For an example of this function, see "Examining the Collection Tags of a Collection" beginning on page 5-35.

To determine the total number of distinct collection tags contained in a collection, use the `CountCollectionTags` function, described in the previous section.

Flattening and Unflattening a Collection

You use the `FlattenCollection` function to flatten a collection into a stream of bytes. You use the `UnflattenCollection` function to unflatten a collection that was flattened using the `FlattenCollection` function.

FlattenCollection

You can use the `FlattenCollection` function to convert a collection object into a stream format suitable for storing and unflattening. For example, you could use this function to copy a collection onto the Clipboard so that it could be pasted into another application.

```
OSErr FlattenCollection(Collection source,
                        CollectionFlattenProc flattenProc,
                        void *refCon);
```

`source`	A reference to the collection that you want to flatten.
`flattenProc`	A pointer to a callback function you provide to process the flattened stream of bytes.
`refCon`	A reference constant that you want the Collection Manager to pass repeatedly to the callback function.

DESCRIPTION

The `FlattenCollection` function flattens into a stream of bytes the collection you specify with the `source` parameter. As this function flattens the collection, it repeatedly calls the callback function you specify using the `flattenProc` parameter. Each time it calls this function, it provides the callback function with a pointer to a block of memory containing flattened data. It continues to call this function until it has flattened the entire collection. Your callback function can process the flattened data in a number of ways: it could copy the flattened data into a handle-based block of memory, it could write the flattened data to disk, and so on.

In the `refCon` parameter, you specify a value that the Collection Manager passes on to your callback function each time it calls your callback function. You can use this parameter as a pointer to a structure containing information your callback function needs to process the blocks of flattened data.

When flattening the source collection, this function includes only the collection items whose persistence attribute is set.

This function can return any error returned by the callback function.

SEE ALSO

For information about the persistence attribute, see “Collection Items” beginning on page 5-8.

For information about the callback function that you provide, see page 5-100.

For examples of this function, see “Flattening and Unflattening a Collection” beginning on page 5-37 and “Reading Collections From and Writing Collections to Disk” beginning on page 5-41.

To create a flattened collection that includes only those collection items whose attributes match a specified pattern, use the `FlattenPartialCollection` function, described in the next section.

To unflatten a flattened collection, use the `UnflattenCollection` function, described on page 5-90.

FlattenPartialCollection

You can use the `FlattenPartialCollection` function to convert a collection object into a stream format suitable for storage and unflattening. With this function, you can include in the flattened collection only those items whose attributes match a specified pattern.

```
OSErr FlattenPartialCollection(Collection source,
                               CollectionFlattenProc flattenProc,
                               void *refCon,
                               long whichAttributes,
                               long matchingAttributes)
```

`source`	The collection that you want to flatten.
`flattenProc`	A pointer to a function to write data.
`refCon`	A reference constant that you want the Collection Manager to pass repeatedly to the flatten procedure.
`whichAttributes`	A mask indicating which attributes you want to test.
`matchingAttributes`	A long word containing the attribute values you want to match.

DESCRIPTION

The `FlattenPartialCollection` function flattens into a stream of bytes the collection you specify with the `source` parameter. It includes only the collection items whose attributes specified by the `whichAttributes` parameter match the values specified by the `matchingAttributes` parameter.

As this function flattens the collection, it repeatedly calls the callback function you specify using the `flattenProc` parameter. Each time it calls this function, it provides the callback function with a pointer to a block of memory containing flattened data. It continues to call this function until it has flattened the entire collection. Your callback function can process the flattened data in a number of ways: it could copy the flattened data into a handle-based block of memory, it could write the flattened data to disk, and so on.

In the `refCon` parameter, you specify a value that the Collection Manager passes on to your callback function each time it calls your callback function. You can use this parameter as a pointer to a structure containing information your callback function needs to process the blocks of flattened data.

When flattening the source collection, this function includes only the collection items whose persistence attribute is set, regardless of the values you provide in the `whichAttributes` and `matchingAttributes` parameters.

This function can return any error returned by the callback function.

SEE ALSO

For information about matching collection item attributes, see “Collection Items” beginning on page 5-8.

For information about the callback function that you provide, see page 5-100.

To create a flattened collection that includes every item in a collection, use the `FlattenCollection` function, described in the previous section.

To unflatten a flattened collection, use the `UnflattenCollection` function, described in the next section.

UnflattenCollection

You use the `UnflattenCollection` function to unflatten a collection that was flattened using the `FlattenCollection` or `FlattenPartialCollection` function.

```
OSErr UnflattenCollection (Collection target,
                           CollectionFlattenProc flattenProc,
                           void *refCon);
```

`target`	A reference to the collection object you want to create from the flattened data.
`flattenProc`	A pointer to a function to read in flattened data.
`refCon`	A reference constant that you want the Collection Manager to pass repeatedly to the callback function.

DESCRIPTION

The `UnflattenCollection` function unflattens a stream of bytes into the collection object you specify with the `target` parameter.

As this function unflattens the collection, it repeatedly calls the callback function you specify using the `flattenProc` parameter. Each time it calls this function, it provides the callback function with a pointer to a block of memory and a requested size. The callback function is responsible for reading the next set of bytes from the flattened byte stream and copying the data into the block of memory.

The Collection Manager continues to call your callback function, requesting more of the flattened stream of bytes each time, until it has unflattened the entire collection. Your callback function can read the flattened data from any source you choose: it could read the flattened data from a handle-based block of memory, it could read the flattened data from disk, and so on.

In the `refCon` parameter, you specify a value that the Collection Manager passes on to your callback function each time it calls your callback function. You can use this parameter as a pointer to a structure containing information your callback function needs when reading the blocks of flattened data.

This function can return any error returned by the callback function.

RESULT CODES

`memFullErr`	–108	Can’t allocate memory.
`collectionVersionErr`	–5753	Unrecognized version / data may be corrupt.

SEE ALSO

For examples of this function, see “Flattening and Unflattening a Collection” beginning on page 5-37 and “Reading Collections From and Writing Collections to Disk” beginning on page 5-41.

For information about the callback function that you provide, see page 5-100.

To create a flattened collection that includes only those collection items whose attributes match a specified pattern, use the `FlattenPartialCollection` function, described in the previous section.

To create a flattened collection that includes every item in a collection, use the `FlattenCollection` function, described on page 5-88.

Working With Macintosh Memory Manager Handles

This section describes a set of utility functions provided by the Collection Manager that allow you to specify a collection item's variable-length data using a Macintosh Memory Manager handle.

AddCollectionItemHdl

You use the `AddCollectionItemHdl` function to add a new item to a collection or to replace an existing item in a collection, specifying the item's variable-length data using a handle rather than a pointer and a data size.

```
OSErr AddCollectionItemHdl (Collection target,
                            CollectionTag tag, long id,
                            Handle itemData);
```

`target`	A reference to the collection you want to add the item to.
`tag`	The collection tag you want to associate with the new item.
`id`	The collection ID you want to associate with the new item.
`itemData`	A Macintosh Memory Manager handle to the item's variable-length data.

DESCRIPTION

The `AddCollectionItemHdl` function adds an item to the collection referenced by the `target` parameter. This new item contains:

- the collection tag specified by the `tag` parameter
- the collection ID specified by the `id` parameter
- the attributes specified by the default attributes of the target collection
- the variable-length data specified by the `itemData` parameter

This function copies the information referenced by the `itemData` parameter into the new item; after calling this function, you may alter this information or free the memory referenced by this parameter without affecting the collection.

If the target collection already contains an item with the same collection tag and collection ID as specified in the `tag` and `id` parameters, this function removes the variable-length data from the original item and replaces it with the new data, unless the existing item is locked. If it is locked, this function returns a `collectionItemLockedErr` result code.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.
`collectionItemLockedErr`	–5750	Can't replace locked item.

SEE ALSO

For information about collection items, see "Collection Items" beginning on page 5-8.

For information about locking collection items, see "Getting and Setting the Attributes of an Item" beginning on page 5-24. To lock a collection item, use the functions described in "Editing Item Attributes" beginning on page 5-82.

To add or replace a collection item using a pointer (rather than a handle) to the item's variable-length data, use the `AddCollectionItem` function, described on page 5-62.

To replace a collection item using the item's collection index (rather than the item's collection tag and collection ID), use the `ReplaceIndexedCollectionItemHdl` function, described in the next section.

ReplaceIndexedCollectionItemHdl

You use the `ReplaceIndexedCollectionItemHdl` function to replace the variable-length data of an item in a collection given the item's collection index, specifying the item's new variable-length data using a handle rather than a pointer and a data size.

```
OSErr ReplaceIndexedCollectionItemHdl(Collection target,
                                      long index,
                                      Handle itemData);
```

`target`	A reference to the collection containing the item you want to replace.
`index`	The collection index associated with the item you want to replace.
`itemData`	A Macintosh Memory Manager handle to the new variable-length data.

DESCRIPTION

The `ReplaceIndexedCollectionItemHdl` function replaces the variable-length data of an item in the `target` collection. You specify which item to replace using the `index` parameter. If the target collection does not contain an item whose collection index matches the value of the `index` parameter, this function returns a `collectionIndexRangeErr` result code.

If the target collection does contain an item with the specified index, this function replaces the data in that item with new data (if the existing item is not locked—if it is, this function returns a `collectionItemLockedErr` result code). The resulting item contains

- the same collection tag as the original item
- the same collection ID as the original item
- the same attributes as the original item
- the variable-length data specified by the `itemData` parameter

This function copies the information referenced by the `itemData` parameter into the collection item; after calling this function, you may alter this information or free the memory referenced by this parameter without affecting the collection.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.
`collectionItemLockedErr`	–5750	Can't replace locked item.
`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection items, see "Collection Items" beginning on page 5-8.

To replace a collection item using a pointer (rather than a handle) to the item's variable-length data, use the `ReplaceIndexedCollectionItem` function, described on page 5-63.

To replace a collection item using the item's collection tag and collection ID (rather than the item's collection index), use the `AddCollectionItemHdl` function, described in the previous section.

GetCollectionItemHdl

You can use the `GetCollectionItemHdl` function to obtain a copy of the variable-length data associated with a collection item given the item's collection tag and collection ID. You must provide a valid Macintosh Memory Manager handle for this function to copy the data into.

```
OSErr GetCollectionItemHdl(Collection source,
                           CollectionTag tag,
                           long id,
                           Handle itemData);
```

`source`	A reference to the collection object containing the item whose data you want to retrieve.
`tag`	The collection tag associated with the item whose data you want to retrieve.
`id`	The collection ID associated with the item whose data you want to retrieve.
`itemData`	A handle to a block of memory to contain the item's data. On return, this memory contains a copy of the data associated with the specified item. You may specify the constant `dontWantData` for this parameter if you do not want a copy of the item's data.

DESCRIPTION

The `GetCollectionItemHdl` function allows you to obtain a copy of the variable-length data associated with a specific collection item. You specify a collection object using the `source` parameter and you specify an item in that collection using the `tag` and `id` parameters. If you provide a valid Macintosh Memory Manager handle in the `itemData` parameter, the function uses this parameter to return a copy of the variable-length data associated with the specified collection item.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.
`collectionItemNotFoundErr`	–5751	Can't locate item.

SEE ALSO

For information about collection items and their associated collection tags, collection IDs, and variable-length data, see "Collection Items" beginning on page 5-8.

For examples using this function, see "Retrieving the Variable-Length Data From an Item" beginning on page 5-33.

To retrieve the data associated with a collection item into a block of memory referenced by a pointer (rather than a handle), use the `GetCollectionItem` function, described on page 5-71.

GetIndexedCollectionItemHdl

You can use the `GetIndexedCollectionItemHdl` function to copy the variable-length data associated with a collection item into a Macintosh Memory Manager handle, given the item's collection index.

```
OSErr GetIndexedCollectionItemHdl(Collection source,
                                  long index,
                                  Handle itemData);
```

`source`	A reference to the collection object containing the item whose data you want to retrieve.
`index`	The collection index associated with the item whose data you want to retrieve.
`itemData`	A handle to a block of memory to contain the item's data. On return, this memory contains a copy of the data associated with the specified item.

DESCRIPTION

The `GetIndexedCollectionItemHdl` function allows you to obtain a copy of the variable-length data associated with a specific collection item. You specify a collection object using the `source` parameter and you specify an item in that collection using the `index` parameter. If you provide a valid Macintosh Memory Manager handle in the `itemData` parameter, the function uses this parameter to return a copy of the variable-length data associated with the specified collection item.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.
`collectionItemNotFoundErr`	–5751	Can't locate item.

SEE ALSO

For information about collection items and their associated collection tags, collection IDs, and variable-length data, see "Collection Items" beginning on page 5-8.

For examples using this function, see "Retrieving the Variable-Length Data From an Item" beginning on page 5-33.

To retrieve the data associated with a collection item into a block of memory referenced by a pointer (rather than a handle), use the `GetCollectionItem` function, described on page 5-71.

FlattenCollectionToHdl

You use the `FlattenCollectionToHdl` utility function to flatten a collection into a Macintosh Memory Manager handle.

```
OSErr FlattenCollectionToHdl(Collection source
                             Handle flattened);
```

`source`	The collection that you want to flatten into a handle.
`flattened`	A handle to contain the flattened data.

DESCRIPTION

This function flattens the collection referenced by the `source` parameter into a block of memory referenced by the handle you provide in the `flattened` parameter.

You must provide a valid collection object reference in the `source` parameter and a valid Macintosh Memory Manager handle in the `flattened` parameter. You may specify a handle of size 0; this function resizes the handle as necessary to hold the flattened data.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.

SEE ALSO

For examples of this function, see "Reading Collections From and Writing Collections to Disk" beginning on page 5-41.

For an example that shows one possible implementation of this function, see "Flattening and Unflattening a Collection" beginning on page 5-37.

To flatten a collection directly to disk, use the `FlattenCollection` function, described on page 5-88.

To unflatten a collection from a block of memory referenced by a handle, use the `UnflattenCollectionFromHdl` function, described in the next section.

UnflattenCollectionFromHdl

You use the `UnflattenCollectionFromHdl` utility function to unflatten a collection that was flattened using the `FlattenCollectionToHdl` utility function.

```
OSErr UnflattenCollectionFromHdl(Collection target
                                 Handle flattened);
```

`target`	A reference to a collection object in which to store the unflattened information.
`flattened`	A handle to the data that was previously flattened.

DESCRIPTION

This function unflattens the information referenced by the handle you provide in the flattened parameter and stores the unflattened collection in the collection object referenced by the `target` parameter. You must provide a reference to a valid collection object in the `target` parameter and a valid Macintosh Memory Manager handle in the `flattened` parameter.

RESULT CODES

`memFullErr`	–108	Can’t allocate memory.
`collectionVersionErr`	–5753	Unrecognized version / data may be corrupt.

SEE ALSO

For examples of this function, see “Reading Collections From and Writing Collections to Disk” beginning on page 5-41.

For an example that shows one possible implementation of this function, see “Flattening and Unflattening a Collection” beginning on page 5-37.

To unflatten a collection directly from disk, use the `UnflattenCollection` function, described on page 5-90.

To flatten a collection to a block of memory referenced by a handle, use the `FlattenCollectionToHdl` function, described in the previous section.

Reading Collections From Resource Files

The function described in this section creates a collection object and initializes it with information stored in a `'cltn'` resource. You can find more information about `'cltn'` resources in “The Collection Resource” beginning on page 5-102.

You should be familiar with the information in the “Resource Manager” chapter of *Inside Macintosh: More Macintosh Toolbox* before using this function.

GetNewCollection

Use the `GetNewCollection` utility function to read a collection in from a collection (`'cltn'`) resource.

```
Collection GetNewCollection(short collectionID);
```

`collectionID`
: The resource ID associated with the collection resource from which you want to create the new collection object.

function result A reference to the new collection object.

DESCRIPTION

This function searches the current resource file path for a collection (`'cltn'`) resource with the resource ID specified by the `collectionID` parameter. If it finds such a resource, this function creates a new collection object, initializes it with the information stored in the resource, and returns a reference to it as the function result.

If this function does not find a collection resource with the specified resource ID, it returns `nil` as the function result.

You can use the `MemError` and `ResError` functions to check for other errors after calling this function.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.
`resNotFound`	–192	Resource not found.

SEE ALSO

For an example using this function, see "Reading a Collection From a Collection Resource" beginning on page 5-44.

For information about collection resources, see "The Collection Resource" beginning on page 5-102.

For more information about resources in general, see the "Resource Manager" chapter of *Inside Macintosh: More Macintosh Toolbox*.

Application-Defined Functions

This section describes two types of functions that you can provide to the Collection Manager:

- the callback function that you provide to the `FlattenCollection`, `FlattenPartialCollection`, and `UnflattenCollection` functions
- the exception procedure that you can provide for any collection object

MyFlattenProc

You provide the `MyFlattenProc` function to read or write flattened collection data.

```
OSErr MyFlattenProc(long size, void *data, void *refCon);
```

`size`	The size of the block of flattened data to read or write.
`data`	A pointer to the block of flattened data. When flattening, this pointer points to the data your callback function should write. When unflattening, your callback function should read flattened data into the memory pointed to by this parameter.
`refCon`	A value you provide to the `FlattenCollection` function or `UnflattenCollection` function that the Collection Manager passes on to your callback function.

DESCRIPTION

You create this function to pass to the `FlattenCollection`, `FlattenPartialCollection`, and `UnflattenCollection` functions when flattening or unflattening a collection.

As the Collection Manager is flattening a collection, it repeatedly calls this callback function to process sequential blocks of flattened data. Each time it calls this function, it provides a pointer to the current block of flattened data in the `data` parameter and the size of the current block in the `size` parameter. You can process this data in a number of ways: appending it to a handle-based block of memory, writing it to disk, and so on.

When unflattening a collection, the Collection Manager repeatedly calls this function to obtain blocks of flattened data. The Collection Manager specifies the size of the requested block in the `size` parameter, and your function should read or copy the requested number of bytes of flattened data into the block of memory pointed to by the `data` parameter.

In either case, the Collection Manager passes in the `refCon` parameter the same value you originally passed as the `refCon` parameter to the `FlattenCollection`, `FlattenPartialCollection`, or `UnflattenCollection` function. You can use this parameter as a pointer to a structure containing relevant state information you need when reading or writing the flattened data.

If the execution of this function results in any fatal error, you should return the error code back to the Collection Manager as the function result. If the function executes successfully, you should return the `noErr` error code as the function result.

SEE ALSO

For more information about the flattening and unflattening functions, see "Flattening and Unflattening a Collection" beginning on page 5-88.

For examples of this function, see "Flattening and Unflattening a Collection" beginning on page 5-37.

MyExceptionProc

You provide the `MyExceptionProc` function (an exception procedure) to handle errors that occur when operating on a collection object.

```
OSErr MyExceptionProc(Collection target, OSErr whichErr);
```

`target`	A reference to the collection object for which the error occurred.
`whichErr`	The result code associated with the error that occurred.

DESCRIPTION

You create this function to install in a collection object using the `SetCollectionExceptionProc` function. Subsequently, whenever the Collection Manager is operating on that collection object and an error occurs, the Collection Manager calls this function, sending it a reference to the collection for which the error occurred and the result code associated with the error. You can use this information to handle the error appropriately for your application.

You can use an exception procedure to respond to an error in a number of ways:

- You can change the error from one result code to another by returning as the function result the new result code.
- You can handle the error and return the `noErr` error code, which indicates that the Collection Manager should return control to the place in your application that generated the error, as if no error had occurred.
- You can use the ANSI C functions `setjmp` and `longjmp` to jump out of the exception procedure into code to handle the error.

SEE ALSO

For an example of an exception procedure see “Installing an Exception Procedure” beginning on page 5-45.

To install an exception procedure in a collection object, use the `SetCollectionExceptionProc` function, which is described on page 5-59.

To obtain a pointer to an existing exception procedure in a collection object, use the `GetCollectionExceptionProc` function, which is described on page 5-58.

Resources

This section describes the structure of the collection resource and the meaning of its fields.

The Collection Resource

The Collection Manager provides the `GetNewCollection` function, described on page 5-99, to create a new collection object and initialize it using information stored in a collection (`'cltn'`) resource. Listing 5-28 shows the structure of the collection resource in Rez format.

Listing 5-28 A Rez template for a `'cltn'` resource

```
type 'cltn' {
   longint = $$CountOf(ItemArray);
   array ItemArray
      {
      longint; /* tag */
      longint; /* id */
         boolean  itemUnlocked = false, /* defined attributes */
                  itemLocked = true;
         boolean  itemNonPersistent = false,
                  itemPersistent = true;
         unsigned bitstring[14] = 0; /* reserved attributes */
         unsigned bitstring[16] userBits; /* user attributes */
      wstring;
      align word;
   };
};
```

The collection resource has two parts:

- a count of the number of items in the resource
- an array of items

Each item in the array specifies

- the collection tag for that item
- the collection ID for the item
- a Boolean value representing the lock attribute for the item
- a Boolean value representing the persistence attribute for the item
- 14 bits representing the 14 reserved attributes for the item
- 16 bits representing the 16 user-defined attributes for the item
- a string containing the variable-length data for the item

Summary of the Collection Manager

Data Types

Optional Return Value Constants

```
enum {
   dontWantTag          = 0L,  /* don’t want collection tag returned */
   dontWantId           = 0L,  /* don’t want collection ID returned */
   dontWantSize         = 0L,  /* don’t want size of data returned */
   dontWantAttributes   = 0L,  /* don’t want attributes returned */
   dontWantIndex        = 0L,  /* don’t want collection index returned */
   dontWantData         = 0L   /* don’t want variable-length data returned */
};
```

Attributes Masks

```
enum {
   noCollectionAttributes      = 0x00000000,  /* no attributes bits set */
   allCollectionAttributes     = 0xFFFFFFFF,  /* all attributes bits set */
   userCollectionAttributes    = 0x0000FFFF,  /* user attributes bits set */
   defaultCollectionAttributes = 0x40000000   /* unlocked, persistent */
};
```

Attribute Bit Numbers

```
enum {
   collectionUser0Bit   = 0,  /* for use by application */
   collectionUser1Bit   = 1,
   collectionUser2Bit   = 2,
   collectionUser3Bit   = 3,
   collectionUser4Bit   = 4,
   collectionUser5Bit   = 5,
   collectionUser6Bit   = 6,
   collectionUser7Bit   = 7,
   collectionUser8Bit   = 8,
   collectionUser9Bit   = 9,
   collectionUser10Bit  = 10,
   collectionUser11Bit  = 11,
   collectionUser12Bit  = 12,
   collectionUser13Bit  = 13,
   collectionUser14Bit  = 14,
   collectionUser15Bit  = 15,

   collectionReserved0Bit  = 16,  /* reserved for use by Apple */
   collectionReserved1Bit  = 17,
   collectionReserved2Bit  = 18,
   collectionReserved3Bit  = 19,
   collectionReserved4Bit  = 20,
   collectionReserved5Bit  = 21,
   collectionReserved6Bit  = 22,
   collectionReserved7Bit  = 23,
   collectionReserved8Bit  = 24,
   collectionReserved9Bit  = 25,
   collectionReserved10Bit = 26,
   collectionReserved11Bit = 27,
   collectionReserved12Bit = 28,
   collectionReserved13Bit = 29,

   collectionPersistenceBit = 30,  /* currently defined by Apple */
   collectionLockBit        = 31
};
```

Attribute Bit Masks

```
enum {
   collectionUser0Mask = 1L << collectionUser0Bit,
   collectionUser1Mask = 1L << collectionUser1Bit,
   collectionUser2Mask = 1L << collectionUser2Bit,
   collectionUser3Mask = 1L << collectionUser3Bit,
   collectionUser4Mask = 1L << collectionUser4Bit,
   collectionUser5Mask = 1L << collectionUser5Bit,
   collectionUser6Mask = 1L << collectionUser6Bit,
   collectionUser7Mask = 1L << collectionUser7Bit,
   collectionUser8Mask = 1L << collectionUser8Bit,
   collectionUser9Mask = 1L << collectionUser9Bit,
   collectionUser10Mask = 1L << collectionUser10Bit,
   collectionUser11Mask = 1L << collectionUser11Bit,
   collectionUser12Mask = 1L << collectionUser12Bit,
   collectionUser13Mask = 1L << collectionUser13Bit,
   collectionUser14Mask = 1L << collectionUser14Bit,
   collectionUser15Mask = 1L << collectionUser15Bit,
   collectionReserved0Mask = 1L << collectionReserved0Bit,
   collectionReserved1Mask = 1L << collectionReserved1Bit,
   collectionReserved2Mask = 1L << collectionReserved2Bit,
   collectionReserved3Mask = 1L << collectionReserved3Bit,
   collectionReserved4Mask = 1L << collectionReserved4Bit,
   collectionReserved5Mask = 1L << collectionReserved5Bit,
   collectionReserved6Mask = 1L << collectionReserved6Bit,
   collectionReserved7Mask = 1L << collectionReserved7Bit,
   collectionReserved8Mask = 1L << collectionReserved8Bit,
   collectionReserved9Mask = 1L << collectionReserved9Bit,
   collectionReserved10Mask = 1L << collectionReserved10Bit,
   collectionReserved11Mask = 1L << collectionReserved11Bit,
   collectionReserved12Mask = 1L << collectionReserved12Bit,
   collectionReserved13Mask = 1L << collectionReserved13Bit,

   collectionPersistenceMask = 1L << collectionPersistenceBit,
   collectionLockMask = 1L << collectionLockBit
};
```

Functions

Creating and Disposing of Collection Objects

```
Collection NewCollection    (void);
void DisposeCollection      (Collection target);
```

Cloning and Copying Collection Objects

```
Collection CloneCollection  (Collection target);
long CountCollectionOwners  (Collection source);
Collection CopyCollection   (Collection source,
                             Collection target);
```

Getting and Setting the Exception Procedure for a Collection

```
CollectionExceptionProc GetCollectionExceptionProc
                            (Collection source);
void SetCollectionExceptionProc
                            (Collection target,
                             CollectionExceptionProc newExceptionProc);
```

Getting and Setting the Default Attributes for a Collection

```
long GetCollectionDefaultAttributes
                            (Collection source);
void SetCollectionDefaultAttributes
                            (Collection target,
                             long whichAttributes,
                             long newAttributes);
```

Adding and Replacing Items in a Collection

```
OSErr AddCollectionItem     (Collection target,
                             CollectionTag tag, long id,
                             long itemSize, void *itemData);
OSErr ReplaceIndexedCollectionItem
                            (Collection target, long index,
                             long itemSize, void *itemData);
```

Removing Items From a Collection

```
OSErr RemoveCollectionItem  (Collection target,
                             CollectionTag tag, long id);
OSErr RemoveIndexedCollectionItem
                            (Collection target, long index);
void PurgeCollection        (Collection target,
                             long whichAttributes,
                             long matchingAttributes);
void PurgeCollectionTag     (Collection target, CollectionTag tag);
void EmptyCollection        (Collection target);
```

Counting Items in a Collection

```
long CountCollectionItems   (Collection source);
long CountTaggedCollectionItems
                            (Collection source, CollectionTag tag);
```

Retrieving the Variable-Length Data From an Item

```
OSErr GetCollectionItem     (Collection source,
                             CollectionTag tag, long id,
                             long *itemSize, void *itemData);
OSErr GetIndexedCollectionItem
                            (Collection source, long index,
                             long *itemSize, void *itemData);
OSErr GetTaggedCollectionItem
                            (Collection source,
                             CollectionTag tag, long position,
                             long *itemSize, void *itemData);
```

Getting Information About a Collection Item

```
OSErr GetCollectionItemInfo (Collection source,
                             CollectionTag tag, long id,
                             long *index, long *itemSize,
                             long *attributes);
OSErr GetIndexedCollectionItemInfo
                            (Collection source, long index,
                             CollectionTag *tag, long *id,
                             long *itemSize, long *attributes);
OSErr GetTaggedCollectionItemInfo
                            (Collection source,
                             CollectionTag tag, long position,
                             long *id, long *index,
                             long *itemSize, void *attributes);
```

Editing Item Attributes

```
OSErr SetCollectionItemInfo
                            (Collection target,
                             CollectionTag tag, long id,
                             long whichAttributes, long newAttributes);
OSErr SetIndexedCollectionItemInfo
                            (Collection target, long index,
                             long whichAttributes, long newAttributes);
```

Getting Information About Collection Tags

```
Boolean CollectionTagExists
                            (Collection source, CollectionTag tag);
long CountCollectionTags    (Collection source);
OSErr GetIndexedCollectionTag
                            (Collection source, long whichTag,
                             CollectionTag *tag);
```

Flattening and Unflattening a Collection

```
OSErr FlattenCollection     (Collection source,
                             CollectionFlattenProc flattenProc,
                             void *refCon);
OSErr FlattenPartialCollection
                            (Collection source,
                             CollectionFlattenProc flattenProc,
                             void *refCon,
                             long whichAttributes,
                             long matchingAttributes);
OSErr UnflattenCollection   (Collection target,
                             CollectionFlattenProc flattenProc,
                             void *refCon);
```

Working With Macintosh Memory Manager Handles

```
OSErr AddCollectionItemHdl  (Collection target,
                             CollectionTag tag, long id,
                             Handle itemData);
OSErr ReplaceIndexedCollectionItemHdl
                            (Collection target, long index,
                             Handle itemData);
OSErr GetCollectionItemHdl  (Collection source,
                             CollectionTag tag, long id,
                             Handle itemData);
OSErr GetIndexedCollectionItemHdl
                            (Collection source, long index,
                             Handle itemData);
OSErr FlattenCollectionToHdl
                            (Collection source, Handle flattened);
OSErr UnflattenCollectionFromHdl
                            (Collection target, Handle flattened);
```

Reading Collections From Resource Files

```
Collection GetNewCollection (short collectionID);
```

Application-Defined Functions

```
OSErr MyFlattenProc          (long size, void *data, void *refCon);
OSErr MyExceptionProc        (Collection target, OSErr whichErr);
```

Resources

The Collection Resource

```
type 'cltn' {
   longint = $$CountOf(ItemArray);
   array ItemArray
      {
      longint; /* tag */
      longint; /* id */
         boolean  itemUnlocked = false, /* defined attributes */
                  itemLocked = true;
         boolean  itemNonPersistent = false,
                  itemPersistent = true;
         unsigned bitstring[14] = 0; /* reserved attributes */
         unsigned bitstring[16] userBits; /* user attributes */
      wstring;
      align word;
   };
};
```

C H A P T E R 6

Message Manager

Contents

Message Manager

The QuickDraw GX Message Manager is a part of the message-passing printing architecture of QuickDraw GX. Read this chapter if you want to use the Message Manager to develop printing extensions or printer drivers.

Because QuickDraw GX uses the Message Manager for printing, you should be familiar with the chapter “Introduction to QuickDraw GX Printing” in *Inside Macintosh: QuickDraw GX Printing* before reading this chapter.

If you want to use the Message Manager to create printing extensions and printer drivers, you should also read *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers.*

This chapter introduces the Message Manager as it is used for printing with QuickDraw GX. It then shows how to use Message Manager functions to

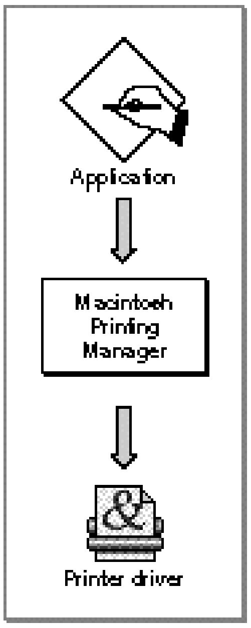
- allocate memory for and dispose of global data
- store global data for a single message handler instance
- store global data for multiple message handler instances
- send and forward messages

This chapter also contains reference information for constants, data types, and functions associated with the Message Manager.

About the Message Manager

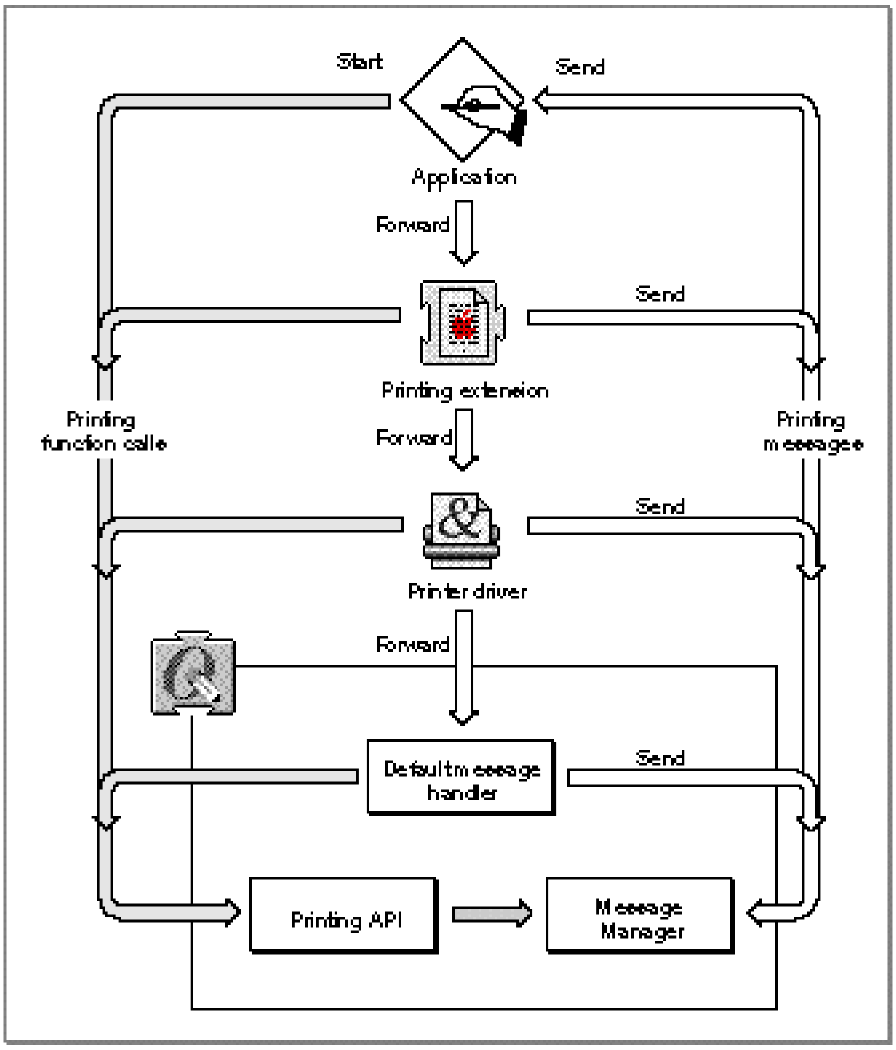
On Macintosh systems in which QuickDraw GX is not installed, the Macintosh Printing Manager calls the printer driver by loading appropriate code resource for the printer driver, as shown in Figure 6-1.

Figure 6-1 Printing with the Macintosh Printing Manager



In contrast, QuickDraw GX provides a low-level software manager called the **Message Manager** to transfer control to the printer driver. Whenever an application makes a printing call, QuickDraw GX interacts with the printer driver by calling the Message Manager to request that the appropriate message be sent to the printer driver. QuickDraw GX printing extensions may be inserted between QuickDraw GX and the printer driver to modify the behavior of printing without changing the printer driver. This approach greatly increases the flexibility of printing and allows printing enhancements to be developed quickly and easily. Figure 6-2 shows the relationship of the QuickDraw GX printing software components.

Figure 6-2 Printing with QuickDraw GX



QuickDraw GX predefines over a hundred messages. An application starts the printing process by calling the QuickDraw GX printing application programming interface (API). QuickDraw GX may perform the task itself or call the Message Manager to send one or more messages to the application to initiate one or more steps in the following sequential message chain: application, printing extensions, printer driver, and the default message handler.

The key to the QuickDraw GX extensible printing architecture is the sequential relationship of the application, printing extensions, printer driver, and default message handler for printing. Applications, printing extensions, and printer drivers are located in the message stream so that they may override messages before the message gets to the default message handler. This is the end of the line for any message that makes it to the end of the chain. QuickDraw GX defines the normal printing characteristics that occur unless modified by an application, printing extensions, or the printer driver. Printing modification may occur when one or more messages are overridden. QuickDraw GX sends a large number of printing messages during the printing process. Since many messages are not normally overridden, QuickDraw GX provides a default printing behavior for most messages via the default message handler.

A partial message override occurs when the application, printing extensions, or printer driver perform one or more tasks in response to a message and then forward the message to the next step in the message chain. A complete message override occurs when the application, printing extensions, or printer driver perform one or more tasks in response to a message and do not forward the message to the the next message handler in the chain. Any message that is not explicitly overridden by a printing extension or printer driver is implicitly forwarded to the next link in the sequential message chain. A complete override of a message prevents the next extension, printer driver, or default implementation in the chain from receiving the overridden message.

The Message Manager is not the only initiator of messages. Applications, printing extensions, printer drivers, and QuickDraw GX not only make printing function calls, but they can also initiate messages.

For additional information about printing with QuickDraw GX, see *Inside Macintosh: QuickDraw GX Printing*. For additional information about how to use the QuickDraw GX Message Manager and messages, see *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers*.

Message Terminology

In working with the Message Manager there are a number of terms that are useful to describe the software components and their interactions.

A **message objec**t is the loose equivalent of an object in a fully object-oriented system. It is the recipient of messages. A message object may also send messages to itself or to another message object.

A **message** is a form of notification passed to a message object in order to have that message object perform some operation.

A **message handler** is a component of a message class. A message class may consist of one or more handlers, each of which overrides zero or more messages. Each message handler may override some portion of the functionality of the handler below it in the message class. Message classes are built up from message handlers, in a manner similar to that in which a class in an object-oriented language is derived from other classes. To **forward** is to invoke the override of the next handler in the chain for the current message.

A **message override** is the loose equivalent of a method. It is the implementation, in actual code, of a given message. The override performs the operation requested by sending a message to a message object.

A **message class** is the loose equivalent of a class in a fully object-oriented system. It defines the set of messages that message objects instantiated from it understand and encapsulates the message handlers that implement the overrides corresponding to those messages. Message classes define the acceptable set of messages for all handlers that they encapsulate.

An **instance** is one copy of a message handler in memory.

Global Data Storage for Printing Extensions and Printer Drivers

Printing extensions and printer drivers are stand-alone code and do not enjoy the full status of an application. When an application is launched, a memory block is automatically allocated for the storage of globals. Unlike applications, stand-alone code is never launched. It is simply loaded, and therefore no memory for globals is allocated.

As a result, if your printing extension or driver requires global data, it must allocate and deallocate memory for this data. Global data can be stored as a constant, a handle, a pointer, or in a so-called A5 world by the use of QuickDraw GX Message Manager functions. QuickDraw GX will not dispose of your globals for you. You must explicitly dispose of them yourself when you are done using them.

Each instance of a message handler can only see its data. If you want to limit access to one instance of your message handler's data, see the section "Setting and Getting Global Data for a Single Handler Instance" beginning on page 6-10.

If you want to use common global data that is accessible to all instances of your handlers, see the section "Setting and Getting Global Data for Multiple Handler Instances" beginning on page 6-12.

To create an A5 world that limits the access of your global data to one copy of your message handler, see the section "Allocating Memory for and Disposing of Global Data" beginning on page 6-8.

For more information about the A5 world, see *Inside Macintosh: Memory*.

Message Sending and Forwarding

QuickDraw GX provides functions that allow you to send a specific message to the top of the message chain (your application), forward a specific message to the next message handler, or forward the current message to the next message handler. For additional information about message sending and forwarding, see the section "Sending and Forwarding Messages" beginning on page 6-15.

Using the Message Manager

This section describes how to

- determine the version of the Message Manager
- allocate and deallocate memory for globals
- create and retrieve global data for a single instance of the message handler
- create and retrieve global data for multiple instances of a message handler
- send and forward messages

Determining the Version of the Message Manager

To determine the current version of the QuickDraw GX Message Manager, you can call the `Gestalt` function with the `gestaltMessageMgrVersion` selector `'mess'`. The `gestaltMessageMgrVersion` selector returns a 2-byte value indicating the version of the QuickDraw GX Message Manager that is currently installed. The high-order byte is the major version number and the low-order byte is the minor revision number.

The selector '`mess`' is defined in the section “Message Manager Gestalt Selector” beginning on page 6-16.

For more information about the `Gestalt` function, see the chapter “Gestalt Manager” in *Inside Macintosh: Operating System Utilities*.

Allocating Memory for and Disposing of Global Data

You can use the `NewMessageGlobals` function to request and allocate memory for globals. You should only call this function while your application is performing a message override.

You should always initialize your global data from a function other than the one in which you call the `NewMessageGlobals` function. Otherwise, your development environment may generate code with bad data references.

Listing 6-1 gives an example of using the `NewMessageGlobals` function to create an A5 world from an MPW programming environment.

Listing 6-1 Creating an A5 world for global data

```
gxShape  gMyShape;
Handle   gMyHandle;

OSErr MyInitGlobalData()
{
   OSErr err;
   gMyShape = nil;
   gMyHandle = TempNewHandle(1024, &err);
   return err;
}

OSErr MyInitialize()
{
   OSErr err;

/*
   Create an A5 world, and initialize the
   global data.
*/
   err = NewMessageGlobals(A5Size(), A5Init);

   if (!err) err = MyInitGlobalData();

   return err;
}
```

The `MyInitalize` function is the override for the `GXInitialize` message. The `MyInitialize` function first sets up an A5 world, as required if an extension is going to use global data. In this case the global data is the `MyShape` structure. Once you create the A5 world by calling the `NewMessageGlobals` function, your global data will be valid whenever your printing extension or printer driver is called. Once the `NewMessageGlobals` function has been called, the extension or driver can initialize its global data. In this example, the code uses a function called `MyInitGlobalData` to do this.

If you have allocated memory for your globals using the `NewMessageGlobals` function, you must use the `DisposeMessageGlobals` function to dispose of the globals and deallocate their memory blocks when they are no longer needed.

Note that `DisposeMessageGlobals` does not dispose of data and handles. These must be disposed of by your code. First, you deallocate any memory that you have allocated and then let QuickDraw GX deallocate memory that it has allocated for your global data.

Listing 6-2 shows how to dispose of global data and deallocate the memory that was allocated in Listing 6-1.

Listing 6-2 Disposing of global data and deallocating memory

```
OSErr MyShutDown()
{/* Dispose of our global data */
   if (gMyHandle != nil)
      DisposHandle(GMyHandle);
/* dispose of the A5 world that was created in MyInitialize */
   DisposeMessageGlobals();
   return noErr;
}
```

The `NewMessageGlobals` function is described on page 6-17. The `DisposeMessageGlobals` function is described on page 6-18.

Global data is discussed in the section “Global Data Storage for Printing Extensions and Printer Drivers” beginning on page 6-7.

Setting and Getting Global Data for a Single Handler Instance

You can use the `SetMessageHandlerInstanceContext` function to store data that can be used by a single instance of a message handler. A new instance of your message handler is created each time a new printing job is created. For example, if four printing jobs are created, four instances of the message handler are created. Each job has a unique context called the instance context.

Listing 6-3 uses this function to store global data whenever a new printing job is initiated. If there are multiple print jobs, this function will be called when each job is started.

Listing 6-3 Storing global data for a single message handler instance

```
typedef struct MyDataRec {
   long something;
   long somethingElse;
} MyDataRec, **MyDataHdl;

OSErr MyInitialize()
{
   OSErr       err;
   MyDataHdl   dataHandle;

/*
   Create a new temporary memory handle, initialize
   it, and store it as the message handler's instance
   context.
*/
   dataHandle = (MyDataHdl) TempNewHandle(sizeof(MyDataRec),
                                                 &err);

   if (err == noErr)
   {
      MyInitDataHandle(dataHandle);
      SetMessageHandlerInstanceContext(dataHandle);
   }

   return err;
}
```

In Listing 6-3, you begin by creating a handle to store global data for the `MyDataRec` structure. Each message handler instance has a unique copy with unique values for the fields of the data structure. If there is insufficient memory to create the handle, an error will be generated. If the handler is successfully created, the handler is initialized. The `SetMessageHandlerInstanceContext` function is then used to store a reference to the handle that can then be used by this message handler's overrides. If you use this code in an extension and four jobs were created for it, each job would have a handle to a unique copy of a record for the structure.

You can use the `GetMessageHandlerInstanceContext` function to retrieve the data that you stored with the `SetMessageHandlerInstanceContext` function. Listing 6-4 uses the `GetMessageHandlerInstanceContext` function to return and dispose of the handle containing the global data that was previously stored in Listing 6-3.

Listing 6-4 Getting and disposing of global data

```
OSErr MyShutDown()
{
   MyDataHdl dataHandle;

/*
   Retrieve the message handler's instance context.  If the
   value returned isn't nil, it's a handle that we stored
   earlier. Dispose of the handle and set the instance
   context to nil to "clear" it.
*/
   dataHandle = (MyDataHdl) GetMessageHandlerInstanceContext();

   if (dataHandle != nil)
   {
      DisposHandle((Handle) dataHandle);
      SetMessageHandlerInstanceContext(nil);
   }

   return noErr;
}
```

In Listing 6-4, the `GetMessageHandlerInstanceContext` function is used to get the previously stored handle containing the global data. If the handle isn't `nil`, it's the `handle` that was previously stored and it is disposed of. Finally, the `SetMessageHandlerInstanceContext` function is used to set the context data to `nil`. If the instance context is `nil`, the handle was previously disposed of.

The `SetMessageHandlerInstanceContext` function is described on page 6-19. The `GetMessageHandlerInstanceContext` function is described on page 6-20.

Global data is discussed in the section "Global Data Storage for Printing Extensions and Printer Drivers" beginning on page 6-7.

Setting and Getting Global Data for Multiple Handler Instances

You can use the `SetMessageHandlerClassContext` function to store data that can be used by multiple copies of your message handler in memory. This common data can be accessed by multiple print jobs and eliminates the need for storing redundant data. Listing 6-5 shows how to use the `SetMessageHandlerClassContext` function to store global data that can be used by multiple handler instances.

Listing 6-5 Storing global data for multiple handler instances

```
typedef struct MySharedDataRec {
   unsigned long  ownerCount;
   long           someData;
   long           someMoreData;
} MySharedDataRec, **MySharedDataHdl;

OSErr MyInitialize()
{
   OSErr             err = noErr;
   MySharedDataHdl   sharedDataHdl;

/*
   Retrieve the message handler's class context.  If the
   value returned is nil, the class context isn’t set up.  In
   that case, create a new handle, initialize
   it, set its owner count to 1, and store it in our class
   context.

   If the class context has been set up, retrieve the data
   handle and increment its owner count.  (We will use the
   owner count in our gxShutDown message override.)
*/
   sharedDataHdl = (MySharedDataHdl)
               GetMessageHandlerClassContext();

   if (sharedDataHdl == nil)
   {
      sharedDataHdl = (MySharedDataHdl)
                  TempNewHandle(sizeof(MySharedDataRec), &err);
      if (!err)
      {
         MyInitSharedDataHandle(sharedDataHdl);
         (*sharedDataHdl)->ownerCount = 1;
         SetMessageHandlerClassContext(sharedDataHdl);
      }
   }
   else
      ++(*sharedDataHdl)->ownerCount;

   return err;
}
```

In contrast to the instance context that is always `nil` as you enter into an initialize routine, with the class context you can't assume that the context is `nil`. For example, you may be the third instance of this message handler. As a result, you need to test to see if the class context is already set up. If it is, you increment the owner count. If it isn't you se tup the context. This ensures that the class context is only set up once Listing 6-5 shows how to use the owner count to set up the class context. If the class context is not `nil`, then you increment the owner count. Otherwise, create the handle, set the owner count to 1, and store the class context.

You can use the `GetMessageHandlerClassContext` function to retrieve data that has been stored by the `SetMessageHandlerClassContext` function. Listing 6-6 shows how to retrieve a message handler's class context and use the information during shutdown.

Listing 6-6 Retrieving a message handler's class context

```
OSErr MyShutDown()
{
   MySharedDataHdl   sharedDataHdl;

/*
   Retrieve the message handler's class context.  If the
   value returned is nil, the class context isn't set up.
   Otherwise, decrement our data's owner count.
   If the owner count falls below 1, dispose of the
   actual data and set our class context to nil to
   "clear" it.
*/
   sharedDataHdl = (MySharedDataHdl)
               GetMessageHandlerClassContext();

   if (sharedDataHdl != nil)
   {
      if (--(*sharedDataHdl)->ownerCount < 1)
      {
         DisposHandle((Handle) sharedDataHdl);
         SetMessageHandlerClassContext(nil);
      }
   }

   return noErr;
}
```

In Listing 6-6, you use the `GetMessageHandlerClassContext` function to obtain and use the class context during shutdown. If the class context is not `nil`, you decrement the owner count. If the owner count is less than 1, there are no other owners and you may then dispose of the data. Using the owner count during shutdown prevents disposing of data more than once.

The `SetMessageHandlerClassContext` function is described on page 6-21. The `GetMessageHandlerClassContext` function is described on page 6-22.

Global data is discussed in the section "Global Data Storage for Printing Extensions and Printer Drivers" beginning on page 6-7.

Sending and Forwarding Messages

Message objects can send a printing message to other clients in the **message chain.** When a message is sent, QuickDraw GX receives it and sends it to the first message handler in the chain. In Figure 6-2 this is the application.

QuickDraw GX provides two methods of sending messages. You can use a statement with the format:

```
anErr = Send_GXMessageName(arguments);
```

A typical example is

```
anErr = Send_GXCompleteSpoolFile(theSpoolFile);
```

Alternatively, you can use the `SendMessage` function to send a specified message to the top of the message chain.

You can use the `ForwardMessage` function to specify the message to be forwarded to the next message handler. This function takes a selector that indicates the message to be forwarded and has parameters that are message-specific.

For example, a four-up printing extension that maps four document pages onto one physical page at print time may require that the `GXCountPages` message be forwarded. The `GXCountPages` message has the following interface:

```
OSErr GXCountPages (gxSpoolFile thePrintFile, long* numPages);
```

You can use the `ForwardThisMessage` function to forward the current message to the next message handler.

```
anErr = ForwardThisMessage(gxCountPages, thePrintFile, &numPages);
```

All the QuickDraw GX `Forward_xxx` functions, where `xxx` is the QuickDraw GX printing message to forward, are in-line aliases to the `ForwardThisMessage` function with the message-specific parameters added for type-checking purposes. An example of the recommended format for forwarding a message is:

```
anErr = Forward_GXCountPages(thePrintFile, &numPages);
```

The `SendMessage` function is described on page 6-23. The `ForwardMessage` function is described on page 6-24. The `ForwardThisMessage` function is described on page 6-25.

Printing messages are described in the "Printing Messages" chapter of *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers*.

Message Manager Reference

This section provides reference information for constants, data types, and functions that allow you to work with the QuickDraw GX Message Manager.

Constants and Data Types

This section describes the constants and data types used by the Message Manager.

Message Manager Gestalt Selector

The Gestalt selector `'mess'` can be used to determine which version, if any, of the Message Manager is installed.

```
enum {
   gestaltMessageMgrVersion = 'mess'
};
```

Message Globals Initiatialization Procedure

You may supply your own initialization procedure for your globals using this type definition.

```
typedef void (*MessageGlobalsInitProc) (void *messageGlobals);
```

To install a message globals initialization procedure, use the `NewMessageGlobals` function described on page 6-17.

For more about initializing your globals, see the section "Allocating Memory for and Disposing of Global Data" beginning on page 6-8.

Functions

This section describes the Message Manager functions you can use to

- allocate memory for and dispose of global data
- define and retrieve global data for a single handler instance
- define and retrieve global data for multiple handler instances
- send and forward messages

Allocating Memory for and Disposing of Global Data

This section describes the functions the QuickDraw GX Message Manager provides for allocating and deallocating memory for your global data.

NewMessageGlobals

You can use the `NewMessageGlobals` function to request and allocate memory for globals.

```
OSErr NewMessageGlobals (long msgGlobalsSize,
                         MessageGlobalsInitProc aProc);
```

`msgGlobalsSize`	The size of the memory requested for global data.
`aProc`	A pointer to an application-defined callback function that initializes and allocates global data memory.
function result	An error of type `OSErr` indicating that the requested memory allocation could not be completed.

DESCRIPTION

The `NewMessageGlobals` function sets up a global world for your printing extension or printer driver. This consists of allocating the specified amount of memory and initializing it with the passed procedure. Once you have created a global world, you can access your data just as you would if your printing extension or printer driver were an application. Whenever your extension or driver is called, your data will be valid.

To establish an A5 world for your globals, the `msgGlobalsSize` parameter is the `A5Size` function and the `aProc` parameter is the `A5Init` function. The `A5Size` and `A5Init` functions are both Macintosh Programming Workshop (MPW) library routines. The `A5Size` function determines how much memory is to be allocated for the A5 world. The `A5Init` function takes a pointer to the A5 globals and initializes them to the appropriate values.

When your extension or printing driver no longer needs the globals, you should release the memory allocated by the `NewMessageGlobals` function by calling the `DisposeMessageGlobals` function.

SEE ALSO

Global data and the A5 world are discussed in the sections "Global Data Storage for Printing Extensions and Printer Drivers" beginning on page 6-7 and "Allocating Memory for and Disposing of Global Data" beginning on page 6-8.

To dispose of printing extension and printer driver globals, use the `DisposeMessageGlobals` function described in the next section.

The prototype for the application-defined callback function for global data initialization is described on page 6-26.

DisposeMessageGlobals

You can use the `DisposeMessageGlobals` function to dispose of globals and deallocate their memory blocks.

```
OSErr DisposeMessageGlobals (void);
```

function result An error of type `OSErr` indicating that the globals are not disposed of.

DESCRIPTION

The `DisposeMessageGlobals` function disposes of all globals and deallocates the memory used by your printing extension or printer driver for globals. You should use this function to free memory whenever your printing extension or printer driver no longer requires globals.

SEE ALSO

Global data and the A5 world are discussed in the sections "Global Data Storage for Printing Extensions and Printer Drivers" beginning on page 6-7 and "Allocating Memory for and Disposing of Global Data" beginning on page 6-8.

To allocate memory for globals, use the `NewMessageGlobals` function described in the previous section.

Setting and Getting Global Data for a Single Handler Instance

This section describes the functions the QuickDraw GX Message Manager provides for defining and retrieving global data for a single handler instance.

SetMessageHandlerInstanceContext

You can use the `SetMessageHandlerInstanceContext` function to store data that can be used by a single handler.

```
void *SetMessageHandlerInstanceContext (void *);
```

DESCRIPTION

The `SetMessageHandlerInstanceContext` function is used to store data that can be used by only a single instance of a message handler. This data is specific to your handler's code and is unique to one copy in memory. The stored data can be in the form of a long word constant, handle, or pointer to other data. The passed data can be accessed only by the instance of the message handler that sets the data.

SEE ALSO

To retrieve the data that has been set by the `SetMessageHandlerInstanceContext` function, use the `GetMessageHandlerInstanceContext` function described in the next section.

To define common data that can be used by multiple instances of a handler, use the `SetMessageHandlerClassContext` function described on page 6-21. To retrieve the common data that has been set, use the `GetMessageHandlerClassContext` function described on page 6-22.

The use of the `SetMessageHandlerInstanceContext` function is described in the section "Setting and Getting Global Data for a Single Handler Instance" beginning on page 6-10.

Global data is discussed in the section "Global Data Storage for Printing Extensions and Printer Drivers" beginning on page 6-7.

GetMessageHandlerInstanceContext

You can use the `GetMessageHandlerInstanceContext` function to retrieve data for a single instance of a handler.

```
void *GetMessageHandlerInstanceContext (void);
```

DESCRIPTION

The `GetMessageHandlerInstanceContext` function returns the data that you stored using the `SetMessageHandlerInstanceContext` function. This function returns the data that was stored by the instance of a handler that is calling the `GetMessageHandlerInstanceContext` function.

If the `SetMessageHandlerInstanceContext` function has not been previously called, the `GetMessageHandlerInstanceContext` function will return `nil`. If a constant, handle, or pointer to other data has been stored, the `GetMessageHandlerInstanceContext` function returns the stored data.

SEE ALSO

The `SetMessageHandlerInstanceContext` function is described in the previous section.

To define common data that can be used by multiple handlers, use the `SetMessageHandlerClassContext` function described on page 6-21. To retrieve the common data that has been set, use the `GetMessageHandlerClassContext` function described on page 6-22.

The use of the `GetMessageHandlerInstanceContext` function is described in the section "Setting and Getting Global Data for a Single Handler Instance" beginning on page 6-10.

Global data is discussed in the section "Global Data Storage for Printing Extensions and Printer Drivers" beginning on page 6-7.

Setting and Getting Global Data for Multiple Handler Instances

This section describes the functions the QuickDraw GX Message Manager provides for defining and retrieving global data for multiple handler instances.

SetMessageHandlerClassContext

You can use the `SetMessageHandlerClassContext` function to store data that can be used by multiple instances of a message handler.

```
void *SetMessageHandlerClassContext (void *);
```

DESCRIPTION

The `SetMessageHandlerClassContext` function is used to store data that can be used by multiple instances of a message handler in one or more print jobs. The parameter passed is a pointer to the `long` data. The stored data can be in the form of a constant, handle, or pointer to additional data. This reference constant can be used by all instances of a message handler.

SEE ALSO

To retrieve the data defined by the `SetMessageHandlerClassContext` function, use the `GetMessageHandlerClassContext` function described in the next section.

The use of the `SetMessageHandlerClassContext` function is described in the section "Setting and Getting Global Data for Multiple Handler Instances" beginning on page 6-12.

To define data that can be used by only one handler, use the `SetMessageHandlerInstanceContext` function described on page 6-19. To retrieve the data that has been set, use the `GetMessageHandlerInstanceContext` function described on page 6-20.

Global data is discussed in the section "Global Data Storage for Printing Extensions and Printer Drivers" beginning on page 6-7.

GetMessageHandlerClassContext

You can use the `GetMessageHandlerClassContext` function to allow multiple instances of your handler to retrieve common global data.

```
void *GetMessageHandlerClassContext (void);
```

DESCRIPTION

The `GetMessageHandlerClassContext` function returns common data that you defined using the `SetMessageHandlerClassContext` function. This function can be called by any instance of your handler.

If the `SetMessageHandlerClassContext` function has not been previously called, the `GetMessageHandlerClassContext` function will return `nil`. If a constant, handle, or pointer has been stored, the `GetMessageHandlerClassContext` function returns the stored data. This function may be used by your handler to allow multiple print jobs to share common global data.

SEE ALSO

To store the data that is retrieved by the `GetMessageHandlerClassContext` function, use the `SetMessageHandlerClassContext` function described in the previous section.

The use of the `GetMessageHandlerClassContext` function is described in the section “Setting and Getting Global Data for Multiple Handler Instances” beginning on page 6-12.

To store data that can be used by only one instance of a handler, use the `SetMessageHandlerInstanceContext` function described on page 6-19. To retrieve the data that has been set, use the `GetMessageHandlerInstanceContext` function described on page 6-20.

Global data is discussed in the section “Global Data Storage for Printing Extensions and Printer Drivers” beginning on page 6-7.

Sending and Forwarding Messages

This section describes the functions the QuickDraw GX Message Manager provides for sending and forwarding messages.

SendMessage

You can use the `SendMessage` function to send a specified message to the current message target.

```
OSErr SendMessage (long messageSelector,…);
```

`messageSelector`
: The number of the message to be sent to the message handler.

additional parameters
: Parameters associated with the message sent.

function result An error of type `OSErr`.

DESCRIPTION

The `SendMessage` function dispatches a message to the topmost handler in the message class that is the parent of the current message target.

The `messageSelector` parameter indicates which message is to be sent.

The ellipsis character at the end of the parameter list indicates that the remaining *additional parameters* are unspecified; the caller must pass whatever parameters are expected by the recipient of the message identified by the `messageSelector` parameter. By definition, all message overrides return a result of type `OSErr`. It is an error to call the `SendMessage` function except from within a message handler. In any other case, behavior is undefined.

The `OSErr` error returned may indicate that the message could not be sent. If no error occurs, the function result is `noErr`. In addition, the receiving message handler may return an error of type `OSErr`.

SEE ALSO

To forward a specified message to the next message handler, use the `ForwardMessage` function described in the next section.

To forward the current message to the next message handler, use the `ForwardThisMessage` function described on page 6-25.

The use of the `SendMessage` function is described in the section “Sending and Forwarding Messages” beginning on page 6-15.

ForwardMessage

You can use the `ForwardMessage` function to specify the message to be forwarded to the next message handler.

```
OSErr ForwardMessage (long messageSelector,…);
```

`messageSelector`
The number of the message to be forwarded.

additional parameters
Parameters associated with the message sent.

function result An error of type `OSErr`.

DESCRIPTION

The `ForwardMessage` function forwards the message specified by the `messageSelector` parameter to the next message handler. This function is like the `ForwardThisMessage` function, except that any message may be forwarded. The `messageSelector` parameter indicates which message is to be forwarded, as in the `SendMessage` function. By definition, all messages return a function result of type `OSErr`.

The ellipsis character at the end of the parameter list indicates that the remaining *additional parameters* are unspecified; the caller must pass whatever parameters are expected by the recipient of the message identified by the `messageSelector` parameter. By definition, all message overrides return a result of type `OSErr`. It is an error to call the `ForwardMessage` function except from within a message handler. In any other case, behavior is undefined.

The `OSErr` error returned may indicate that the message could not be forwarded. If no error occurs, the function result is `noErr`. In addition, the receiving message handler may return an error of type `OSErr`.

SEE ALSO

To send a specified message to the current message target, use the `SendMessage` function described in the previous section.

To forward the current message to the next message handler, use the `ForwardThisMessage` function described in the next section.

The use of the `ForwardMessage` function is described in the section “Sending and Forwarding Messages” beginning on page 6-15.

ForwardThisMessage

You can use the `ForwardThisMessage` function to forward the current message to the next message handler.

```
OSErr ForwardThisMessage (…);
```

parameters	Parameters associated with the message sent.
function result	An error of type `OSErr`.

DESCRIPTION

The `ForwardThisMessage` function explicitly inherits the current message by forwarding it to the next handler in the message class of the current message target. By definition, all message overrides return a function result of type `OSErr`.

The `OSErr` error returned may indicate that the message could not be forwarded. If no error occurs, the function result is `noErr`. In addition, the receiving message handler may return a result of type `OSErr`.

The ellipsis character in the parameter list indicates that the *parameters* are unspecified; the caller must pass whatever parameters are expected by the recipient of the message. It is an error to call the `ForwardThisMessage` function except from within a message handler. In any other case, behavior is undefined.

SEE ALSO

To send a specified message to the current message target, use the `SendMessage` function described on page 6-23.

To forward a specified message to the next message handler, use the `ForwardMessage` function described in the previous section.

The use of the `ForwardThisMessage` function is described in the section “Sending and Forwarding Messages” beginning on page 6-15.

Driver- or Extension-Defined Functions

This section describes the callback function that you must provide for QuickDraw GX to call when initializing global data.

MessageGlobalsInitProc

You can create an initialization function that requests and allocates memory for your global data. The initialization function must have a prototype of this form:

```
typedef void (*MessageGlobalsInitProc) (void *messageGlobals);
```

`messageGlobals`
A pointer to the global data to be initialized.

DESCRIPTION

You must supply the `MessageGlobalsInitProc` function if you use the `NewMessageGlobals` function to allocate memory for your global data. Once this initialization function is installed, QuickDraw GX calls it whenever you use the `NewMessageGlobals` function.

If your programming environment is MPW, you may use the `A5Init` function that MPW provides to establish an A5 world for your global data:

```
void A5Init (void *globalPtr);
```

SEE ALSO

The `NewMessageGlobals` function is described on page 6-17.

For more information on initializing you global data, see the section “Allocating Memory for and Disposing of Global Data” beginning on page 6-8.

Summary of the Message Manager

Constants and Data Types

Message Manager Gestalt Selector

```
#define gestaltMessageMgrVersion 'mess' /* gestalt version selector */
```

Message Globals Inititialization Procedure

```
typedef void (*MessageGlobalsInitProc) (void *messageGlobals);
```

Functions

Allocating Memory for and Disposing of Global Data

```
OSErr NewMessageGlobals     (long msgGlobalsSize,
                             MessageGlobalsInitProc aProc);
OSErr DisposeMessageGlobals (void);
```

Setting and Getting Global Data for Multiple Handler Instances

```
void *SetMessageHandlerClassContext
                            (void *);
void *GetMessageHandlerClassContext
                            (void);
```

Setting and Getting Global Data for a Single Handler Instance

```
void *SetMessageHandlerInstanceContext
                            (void *);
void *GetMessageHandlerInstanceContext
                            (void);
```

Sending and Forwarding Messages

```
OSErr SendMessage           (long messageSelector…);
OSErr ForwardMessage        (long messageSelector, …);
OSErr ForwardThisMessage    (…);
```

Application-DefinedFunctions

Initializing Memory for Global Data

```
typedef void (*MessageGlobalsInitProc) (void *messageGlobals);
```

CHAPTER 7

QuickDraw GX Stream Format

Contents

This chapter describes the format of the compressed data stream that results when the QuickDraw GX `GXFlattenShape` function is used. It also describes the use of such data streams by print files and portable digital documents (PDDs). Read this chapter if you need to uncompress QuickDraw GX stream format data and cannot use the QuickDraw GX `GXUnflattenShape` function.

Before reading this chapter, you should be familiar with the information in the chapters "Introduction to QuickDraw GX Objects" and "Shape Objects" in *Inside Macintosh: QuickDraw GX Objects*.

The `GXFlattenShape` and `GXUnflattenShape` functions and additional information about the objects contained in the data stream are described in *Inside Macintosh: QuickDraw GX Objects*. For more information on graphic shapes, see the shape-specific chapters in *Inside Macintosh: QuickDraw GX Graphics*. For more information on typographic shapes, see the shape-specific chapters in *Inside Macintosh: QuickDraw GX Typography*. For more information on print files and portable digital documents, see the chapter "Advanced Printing Features" of *Inside Macintosh: QuickDraw GX Printing*.

This chapter first describes the QuickDraw GX stream format, print file organization, and portable digital documents. It then shows how you can

- use the GraphicsBug utility to flatten QuickDraw GX shapes
- analyze flattened shape data streams
- obtain information from a print file

About QuickDraw GX Stream Format

A QuickDraw GX **data stream** is a highly structured sequence of bytes that contains all of the information required to store, print, or display QuickDraw GX objects.

QuickDraw GX provides a simple method for creating and interpreting a QuickDraw GX data stream for shape objects. The `GXFlattenShape` function creates the data stream and the `GXUnflattenShape` function reconstructs objects from the data stream that the `GXFlattenShape` function previously created.

When the `GXFlattenShape` function converts shape objects created by your application from their original format to a QuickDraw GX stream format, the **shape** is said to be flattened. When the `GXUnFlattenShape` function interprets the data stream of a flattened shape, the shape is said to be unflattened**.**

If QuickDraw GX is available and you need to **flatten** and **unflatten** QuickDraw GX shapes, you just use the `GXFlattenShape` and `GXUnflattenShape` functions. If QuickDraw GX is not available and you need to unflatten a flattened shape, then you need to create an interpreter for the QuickDraw GX data stream that was created when the shape was flattened. The interpreter must be compatible with your current working environment.

Your interpreter needs to parse the data of the QuickDraw GX data stream to extract the original meaning. The format of the data stream is public. This section describes the data **stream format** and its use in print files and portable digital documents.

In addition to the `GXFlattenShape` and `GXUnflattenShape` functions that create and interpret the QuickDraw GX stream format for shapes, there are other flatten and unflatten functions that perform flattening and unflattening operations on job objects, job objects in a handle, collection objects, and fonts. These functions are not directly related to the stream format.

The `GXFlattenJob` and `GXUnFlattenJob` functions provide your application with a mechanism for flattening and unflattening all information associated with a job object by specifying a pointer to a flattening function. For more information on these functions, see the chapters "QuickDraw GX Printing" and "Core Printing Features" in *Inside Macintosh: QuickDraw GX Printing.*

The `GXFlattenJobToHdl` and `GXUnflattenCollectionFromHdl` functions provides your application with a means of flattening and unflattening all information associated with a job object in a handle. For more information on these functions, see the chapters "Introduction to Printing with QuickDraw GX" and "Core Printing Features" of *Inside Macintosh: QuickDraw GX Printing.*

The `GXFlattenCollection` and `GXUnflattenCollection` functions flatten and unflatten information in a collection object. For more information on this function, see the chapter "Collection Manager" in this book.

The `GXFlattenFont` function flattens a font so that it can be included in a flattened shape. The `GXFlattenFont` function is described in the chapter "Font Objects" in *Inside Macintosh: QuickDraw GX Typography.*

Characteristics

The QuickDraw GX data stream format is used whenever a QuickDraw GX shape is stored to disk or printed. Likewise, the data stream must be interpreted whenever the flattened shape is to be used. The QuickDraw GX stream format is

- **Extensible.** The data stream includes type constants called opcodes that specify the meaning of the data that follows in the data stream and record size values that indicate the number of bytes in the record that follow. The opcode and size are always in the same format. If a reader of a QuickDraw GX data stream doesn't understand the information contained in the stream, the reader can choose to skip to the next opcode. Some opcode constants are reserved for future expansion.
- **Byte oriented.** QuickDraw GX uses a byte-oriented stream format so that it is simple for different processors to interpret the flattened shape information. Multiple byte-oriented data streams, using words (2 bytes) or long words (4 bytes), are larger and therefore are not as efficient for storing, retrieving, and printing shapes.

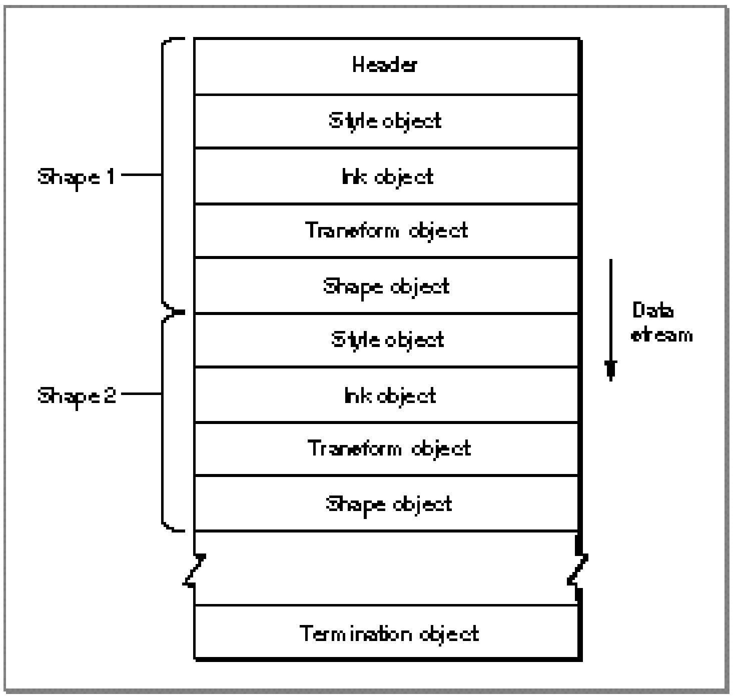
- **Efficient.** The QuickDraw GX data stream format contains a highly structured optimized set of data that minimize the amount of irrelevant information. For example, if your application creates a shape and then moves the shape to another position, the flattened shape stream format describes only the final position of the shape and does not include a description of the intermediate move.
- **Compressed.** The `GXFlattenShape` function always applies a compression algorithm to the flattened shape. The degree of compression that is achieved depends upon the shape and the objects that make up the shape. If applying the compression algorithm results in a data stream that is larger than the original, the original data is adopted as the default. When you call the `GXFlattenShape` function, you are thereby always assured of a data stream format that is equal to or smaller than the original data format. Data in a QuickDraw GX stream format consists of single bits, multiple bits, a byte, multiple bytes, a word, multiple words, a long word, or multiple long words. The QuickDraw GX compression algorithm attempts to minimize the number of bits that are required to represent the data required to describe each object and its properties. For example, the long fixed-point number 125.0, 0x007D0000, requiring 4 bytes may be compressed to the byte 125, 0x7D, requiring only 1 byte. This substitution makes the data stream 3 bytes smaller, while maintaining the integrity of the data value. When the shape is unflattened, the byte must be converted back to its original long value. The QuickDraw GX stream format also compresses the data stream bytes that contain opcodes. These opcode bytes consist of a 2-bit field and a 6-bit field that are packed into1 byte.
- **Shape oriented.** Each QuickDraw GX shape is described by a style object, ink object, transform object, and shape object. When a QuickDraw GX shape is flattened, a new data format is created that contains all of the essential information required to define the original shape. All of the objects and properties that are required to describe all of the QuickDraw GX shapes are included in the data stream.

Stream Design

The data stream includes type constants called *opcodes* that specify the meaning of the data that follows in the data stream and record size values that indicate the number of bytes in the record that follow.

Each QuickDraw GX data stream starts with a header. The header contains the version of QuickDraw GX that produced the stream and flags that describe whether or not a list of fonts and a list of glyphs used by the objects are provided for at the end of the stream. This header is typically followed by the style object, ink object, transform object, and shape object for the shape. This sequence is repeated for all subsequent shapes in the data stream. The data stream is terminated after the last shape by the presence of a termination object, as shown in Figure 7-1.

Figure 7-1 A typical flattened shape data stream sequence



Each header and object type in the data stream is counted. This results in the assignment of **reference** numbers for headers and all object types, such as style, ink, and transform objects. The reference number is the *n*th occurrence of a header or object type.

For example, each data stream always has a header (1), a typically a style object (1), ink object (1), transform object (1), and shape object (1), where the references are given in parentheses. Additional headers and object types in the data stream are assigned the next incremental reference number. Figure 7-1 shows that shape 1 is defined by style object (1), ink object (1), transform object (1), and shape object (1) and that shape 2 is defined by style object (2), ink object (2), transform object (2), and shape object (2). shape 100 in this data stream (not shown) may use the ink object defined in shape 1 by referencing ink object (1).

Besides the style, ink, transform, and shape objects, the data stream may also contain additional objects. The following objects are flattened when referenced by shapes, inks, and transforms:

- tag
- color set
- color profile
- other referenced objects

Examples of other referenced objects are the shapes that represent clips, dashes, and the styles and transforms in text faces.

The following objects are never flattened:

- view ports
- view devices
- view groups

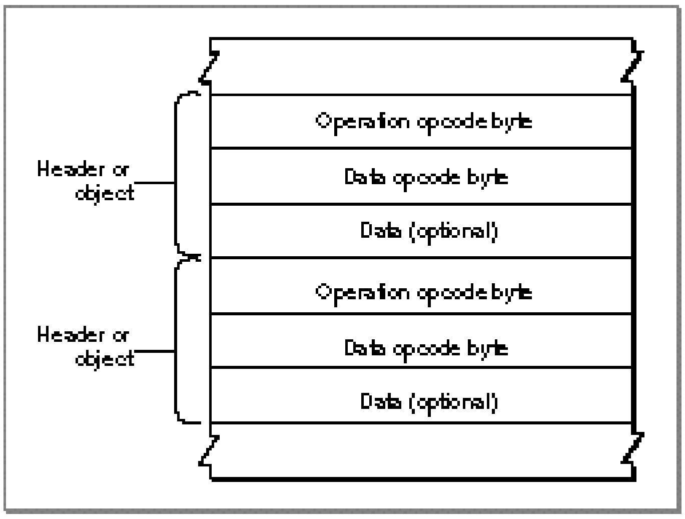
Another rule regarding data stream design requires that all objects and their attributes in the data stream must be defined before they are referenced. QuickDraw GX data streams never forward-reference objects.

For example, the style, ink, and transform objects for a shape must always precede the shape object that they describe in the data stream. In addition, if a style object has a text face property and the text face property has a dash property, then the shape object for the dash property must precede the style object in the data stream.

The data stream design does not require that the order of objects to be style, ink, and transform. Because these objects do not reference each other, they can appear in any order in the data stream, as long as they are defined prior to being referenced.

Each header and object in the data stream consists of an operation opcode byte, a data type opcode byte, and optional data bytes. Figure 7-2 shows these basic data stream format building blocks. This sequence is repeated from the beginning of the stream to the end of the stream. The next sections describe each of these building blocks.

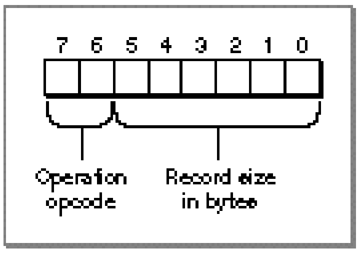
Figure 7-2 Basic components of a stream header or object



Operation Opcode Byte

The first byte of a header or object is always an operation opcode byte. The operation opcode byte contains both an operation opcode and the size in bytes of the record that follows for the current object. The operation opcode either defines a new object, adds data to the current default object, or references a previous object. The record length in bytes includes the data type opcode byte and any data that may follow for the current object. Figure 7-3 shows the format of the operation opcode byte.

Figure 7-3 The format of the operation opcode byte



The operation opcode and record size are always in the same stream format. This enables a reader of the data stream to skip over parts of the data stream that are not understood.

Operation Opcode

Bits 6 and 7 of the operation opcode byte are the operation opcode. Table 7-1 summarizes the 2-bit operation opcodes from the `gxGraphicsOperationOpcode` enumeration.

Table 7-1 Operation opcodes

Type	Value	Description
`gxNewObjectOpcode`	0x00	This opcode type defines a new object.
`gxSetDataOpcode`	0x40	This opcode type adds data to the current object.
`gxSetDefaultOpcode`	0x80	This opcode type replaces the current default with a previously defined object by specifying its reference number.
`gxReservedOpcode`	0xC0	This opcode type is not currently defined and is reserved for future use.

Record Size

The record size defines the number of bytes required to define the header or object record, not including the operation opcode byte. It is always 1 or larger. The record size is given in either bits 0 through 5 of the operation opcode byte or within the bytes that follow the operations opcode byte.

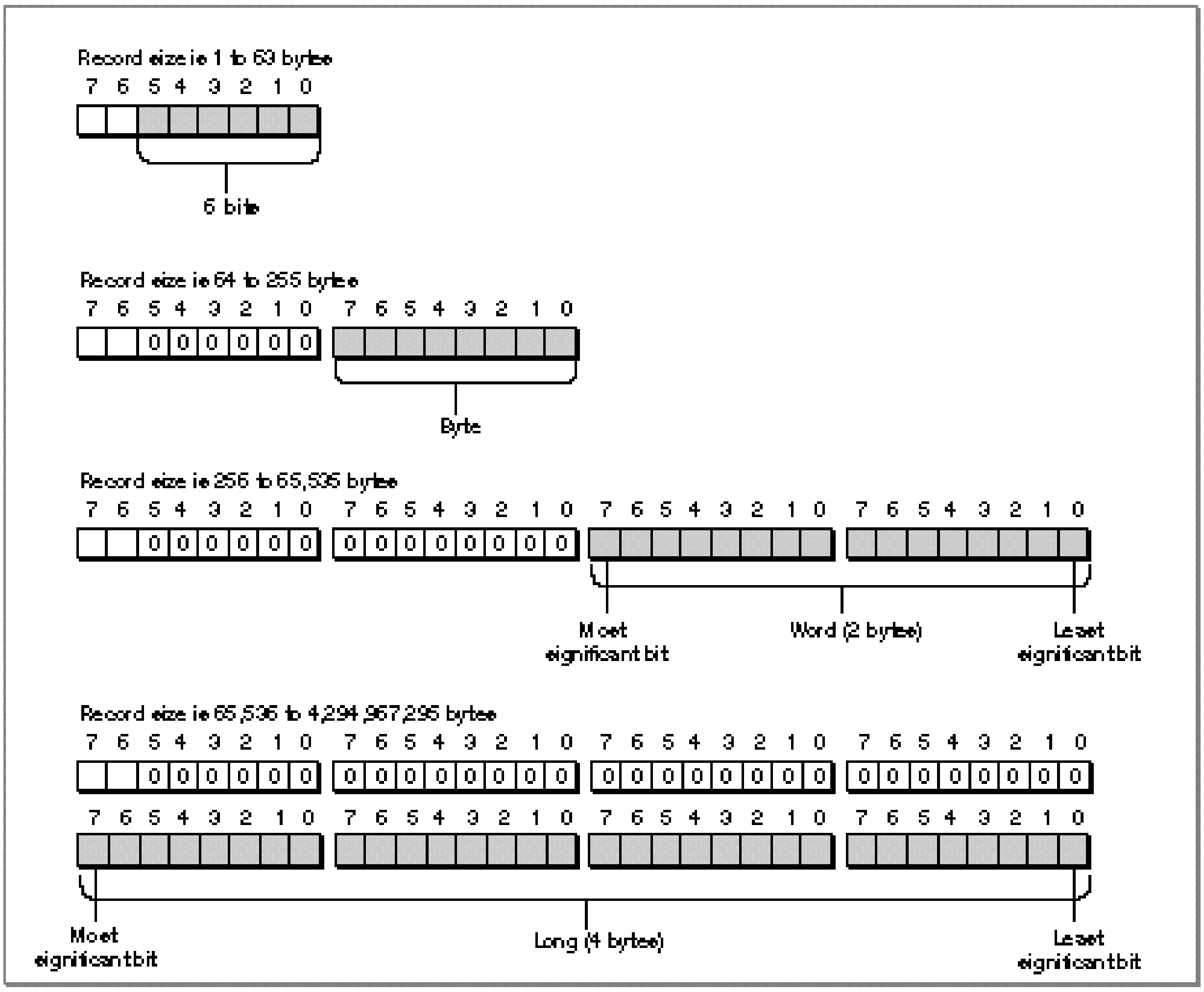
If the record size is larger than the value that can be represented in bits 0 through 5, larger than 63, then a 0 appears in these 6 bits and the next byte in the data stream may contain the record size.

If the record size is larger than the value that can be represented in the next byte, larger than 255, then a 0 appears in this byte and the next word in the stream may contain the record size.

If the record size is larger than the value that can be represented in the next word, larger than 65,535, then a 0 appears in this word and the next long in the stream contains the record size. A long can accommodate a record size up to 4,294,967,295 bytes.

Figure 7-4 shows the operation opcode byte on the left and the subsequent bytes in which the record size is stored in 6-bits, a byte, a `word`, or a `long`. The data stream continues proceeds from left to right.

Figure 7-4 Data format of the record size

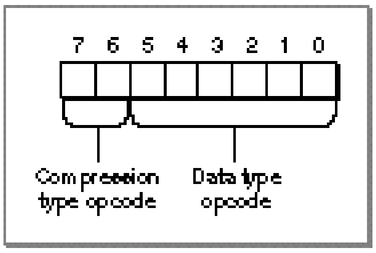


An example of a bit stream in which a long was required to accommodate a record size of 404 bytes is described in the section "Analyzing a Flattened Bitmap Shape" beginning on page 7-81.

Data Type Opcode Byte

A data type opcode byte always follows the record size. This byte contains both a compression type opcode and a data type opcode. Figure 7-5 shows the format of the data type opcode byte.

Figure 7-5 The format of the data type opcode byte



Compression Type Opcode

Bits 6 and 7 of the data type opcode byte contain the compression type opcode. This opcode specifies the type of compression used for the data that follows. The 2-bit compression opcode constants from the `gxTwoBitCompressionValues` enumeration specifies whether the next data are longs, words, bytes, or that no data follows. Table 7-2 lists the compression type opcode values.

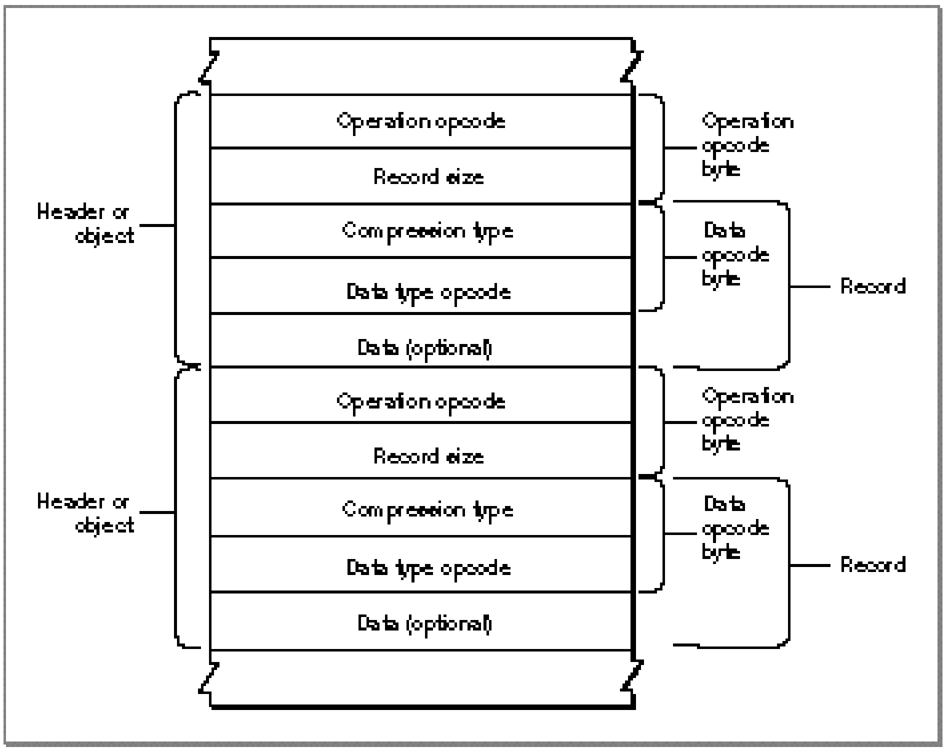
Table 7-2 Compression values

Value	Description
0x00	No compression has been applied. The data that follows are long words.
0x40	Word compression has been applied. The data that follows are words.
0x80	Byte compression has been applied. The data that follows are bytes.
0xC0	Omit compression. No data follows.

The `gxTwoBitCompressionValues` enumeration is also used to interpret the compression in the omit byte. For additional information about the interpretation of omit bytes, see the section “Omit Byte Masks and Omit Byte Shifts” beginning on page 7-22.

The relationship of the operation opcode, record size, compression type opcode, data type opcode, and optional data for a header or object is shown in Figure 7-6.

Figure 7-6 Relationship of stream format components



The appearance or absence of data after the data type opcode byte depends upon the values that appear in the operation opcode byte and the data type opcode byte.

If the `gxNewObjectOpcode` constant appears in the operation opcode byte, a new object follows. The new object copies the default values into the newly created object. The default values may have been changed by the last object created of this type. If the last object and the current object are equal, then the new object requires no additional data for its definition. In this case, the stream following the new opcode byte contains only the compression and data type opcode byte with compression set to no compression.

If the `gxSetDataOpcode` constant appears in the operation opcode byte, the record length is greater than 1 byte and object-specific data follows.

The `gxSetDefaultOpcode` constant appears only after the current object type has been defined. If the `gxSetDefaultOpcode` constant appears in the operation opcode byte, the data type opcode contains the `gxStyleTypeOpcode`, `gxInkTypeOpcode`, or `gxTransformTypeOpcode` constant. The compression type opcode defines the compression of the data of the object reference number that follows. This previously defined object becomes the default styles, ink, or transform for the shapes created subsequently.

The sequence of the object-specific data that follows the data type opcode byte is described in the next section. Subsections are provided for the header, shape data, style, ink, transform, color profile, color set, tag, bit image, font name, and trailer objects.

Data Type Opcode

Bits 0 through 5 of the data type opcode byte contain the data type opcode. This opcode specifies the type of data that follows. The type of data that follows depends upon the current value of the operation opcode. If the operation opcode is `gxNewObjectOpcode`, the data type opcode describes a new object. These data type opcodes are described in the next section. If the operation opcode is `gxSetDataOpcode`, the data type opcode, specifies how the current object will be modified. These data type opcodes are described in the sections "Data Type Opcodes to Modify a Shape Object" beginning on page 7-17, "Data Type Opcodes to Modify a Color Set Object" beginning on page 7-20, "Data Type Opcodes to Modify a Color Profile Object" beginning on page 7-21, and "Data Type Opcodes to Modify a Transform Object" beginning on page 7-21.

Data Type Opcodes for a New Object

When the current operation opcode is the `gxNewObjectOpcode` constant, bits 0 through 5 of the data type opcode byte specify the data type opcode for the new object. Data type opcode constants for header, style, ink, transform, color profile, color set, tag type, bit image, font name, and trailer are defined in the `gxGraphicsNewOpcode` enumeration. Data type opcode constants for empty, point, line, curve, rectangle, polygon, path, bitmap, text, glyph, layout, full, and picture are defined in the `gxShapeTypes` enumeration. Table 7-3 summarizes all of the data type opcodes for a new object.

Table 7-3 Data type opcodes for a new object

Constant	Value	Description
`gxHeaderTypeOpcode`	0x00	The data that follows is the header.
`gxEmptyType`	0x01	The data that follows describes an empty shape object. See the `GXNewShape(gxEmptyType)` function.
`gxPointType`	0x02	The data that follows describes a point object. See the `GXNewPoint` function.
`gxLineType`	0x03	The data that follows describes a line object. See the `GXNewLine` function.
`gxCurveType`	0x04	The data that follows describes a curve object. See the `GXNewCurve` function.

continued

Table 7-3 Data type opcodes for a new object (continued)

Constant	Value	Description
`gxRectangleType`	0x05	The data that follows describes a rectangle object. See the `GXNewRectangle` function.
`gxPolygonType`	0x06	The data that follows describes a polygon object. See the `GXNewPolygons` function.
`gxPathType`	0x07	The data that follows describes a path object. See the `GXNewPaths` function.
`gxBitmapType`	0x08	The data that follows describes a bitmap object. See the `GXNewBitmap` function.
`gxTextType`	0x09	The data that follows describes a text object. See the `GXNewText` function.
`gxGlyphType`	0x10	The data that follows describes a glyph object. See the `GXNewGlyph` function.
`gxLayoutType`	0x11	The data that follows describes a layout object. See the `GXNewLayout` function.
`gxFullType`	0x12	The data that follows describes a full shape object. See the `GXNewShape(gxFullType)` function.
`gxPictureType`	0x13	The data that follows describes a picture object. See the `GXNewPicture` function.
`gxStyleTypeOpcode`	0x28	The data that follows describes a style object. See the `GXNewStyle` function.
`gxInkTypeOpcode`	0x29	The data that follows describes an ink object. See the `GXNewInk` function.
`gxTransformTypeOpcode`	0x2A	The data that follows describes a transform object. See the `GXNewTransform` function.
`gxColorProfileOpcode`	0x2B	The data that follows describes a color profile object. See the `GXNewColorProfile` function.
`gxColorSetOpcode`	0x2C	The data that follows describes a color set object. See the `GXNewColorSet` function.
`gxTagTypeOpcode`	0x2D	The data that follows describes a tag object. See the `GXNewTag` function.
`gxBitImageOpcode`	0x2E	The data that follows describes a bit image, the bits pointed to by a bitmap.
`gxFontNameTypeOpcode`	0x2F	The data that follows describes a font name. See the `GXNewFont` function.
`gxTrailerTypeOpcode`	0x3F	This opcode indicates the end of a data stream.

The omitted numbers are reserved by Apple Computer, Inc. for future use. You should extend the stream format by using tag objects to encapsulate custom data. Tags are described in the “Tag Objects” in *Inside Macintosh:QuickDraw GX Objects*.

Data Type Opcodes to Modify a Shape Object

When the current object is a shape object and the current operation opcode is the `gxSetDataOpcode` constant, bits 0 through 5 of the data type opcode byte specify the data type opcode for the shape object to be modified. Data type opcode constants for attributes, tag, ink, and fill are defined in the `gxShapeDataOpcode` enumeration. Table 7-4 summarizes all of the data type opcodes used to modify a shape object.

Table 7-4 Data type opcodes to modify a shape object

Constant	Value	Description
`gxShapeAttributesOpcode`	0x00	The attributes data that follows is added to the current shape object. See the `GXSetShapeAttributes` function.
`gxTagOpcode`	0x01	The tag data that follows is added to the current shape object. See the `GXSetShapeTags` function.
`gxFillOpcode`	0x02	The fill data that follows is added to the current shape object. See the `GXSetShapeFill` function.

Data Type Opcodes to Modify a Style Object

When the current object is a style object and the current operation opcode is the `gxSetDataOpcode` constant, bits 0 through 5 of the data type opcode byte specify the data type opcode for the style object to be modified. Data type opcode constants for attributes, tag, curve error, pen, join, dash, caps, pattern, text attributes, text size, font, text face, platform, font variations, run controls, run priority justification override, run glyph justification overrides, run glyph substitutions, run features, run kerning adjustments, and justification are defined in the `gxStyleDataOpcode` enumeration. Table 7-5 summarizes all of the data type opcodes used to modify a style object.

Table 7-5 Data type opcodes to modify a style object

Constant	Value	Description
`gxStyleAttributesOpcode`	0x00	The attributes data that follows is added to the current shape object. See the `GXSetStyleAttributes` function.
`gxStyleTagOpcode`	0x01	The tag data that follows is added to the current shape object. See the `GXSetStyleTags` function.
`gxStyleCurveErrorOpcode`	0x02	The curve error data that follows is added to the current style object. See the `GXSetStyleCurveError` function.
`gxStylePenOpcode`	0x03	The pen data that follows is added to the current style object. See the `GXSetStylePen` function.
`gxStyleJoinOpcode`	0x04	The join data that follows is added to the current style object. See the `GXSetStyleJoin` function.
`gxStyleDashOpcode`	0x05	The dash data that follows is added to the current style object. See the `GXSetStyleDash` function.
`gxStyleCapsOpcode`	0x06	The caps data that follows is added to the current style object. See the `GXSetStyleCaps` function.
`gxStylePatternOpcode`	0x07	The pattern data that follows is added to the current style object. See the `GXSetStylePattern` function.
`gxStyleTextAttributesOpcode`	0x08	The text attributes data that follows is added to the current style object. See the `GXSetStyleTextAttributes` function.
`gxStyleTextSizeOpcode`	0x09	The text size data that follows is added to the current style object. See the `GXSetStyleTextSize` function.
`gxStyleFontOpcode`	0x0A	The font data that follows is added to the current style object. See the `GXSetStyleFont` function.

Table 7-5 Data type opcodes to modify a style object (continued)

Constant	Value	Description
`gxStyleTextFaceOpcode`	0x0B	The text face data that follows is added to the current style object. See the `GXSetStyleFace` function.
`gxStylePlatformOpcode`	0x0C	The platform data that follows is added to the current style object. See the `GXSetStyleEncoding` function.
`gxStyleFontVariationsOpcode`	0x0D	The font variations data that follows is added to the current style object. See the `GXSetStyleFontVariations` function.
`gxStyleRunControlsOpcode`	0x0E	The run controls data that follows is added to the current style object. See the `GXSetStyleRunControls` function.
`gxStyleRunPriorityJustOverrideOpcode`	0x1F	The run priority justification override data that follows is added to the current style object. See the `GXSetStyleRunPriorityJustOverride` function.
`gxStyleRunGlyphJustOverridesOpcode`	0x10	The run glyph justification overrides data that follows is added to the current style object. See the `GXStyleRunGlyphJustOverrides` function.
`gxStyleRunGlyphSubstitutionsOpcode`	0x11	The run glyph substitutions data that follows is added to the current style object. See the `GXStyleRunGlyphSubstitutions` function.
`gxStyleRunFeaturesOpcode`	0x12	The run features data that follows is added to the current style object. See the `GXStyleRunFeatures` function.
`gxStyleRunKerningAdjustmentsOpcode`	0x13	The run kerning adjustments data that follows is added to the current style object. See the `GXStyleRunKerningAdjustments` function.
`gxStyleJustificationOpcode`	0x14	The justification data that follows is added to the current style object. See the `GXStyleJustification` function.

Data Type Opcodes to Modify an Ink Object

When the current object is an **ink** object and the current operation opcode is the `gxSetDataOpcode` constant, bits 0 through 5 of the data type opcode byte specify the data type opcode for the ink object to be modified. Data type opcode constants for attributes, tag, color, and transfer mode are defined in the `gxInkDataOpcode` enumeration. Table 7-6 summarizes all of the data type opcodes used to modify an ink object.

Table 7-6 Data type opcodes to modify an ink object

Constant	Value	Description
`gxInkAttributesOpcode`	0x00	The attributes data that follows is added to the current ink object. See the `GXSetInkAttributes` function.
`gxInkTagOpcode`	0x01	The tag data that follows is added to the current ink object. See the `GXSetInkTags` function.
`gxInkColorOpcode`	0x02	The ink color data that follows is added to the current ink object. See the `GXSetInkColor` function.
`gxInkTransferModeOpcode`	0x03	The ink transfer mode data that follows is added to the current ink object. See the `GXSetInkTransfer` function.

Data Type Opcodes to Modify a Color Set Object

When the current object is a color set object and the current operation opcode is the `gxSetDataOpcode` constant, bits 0 through 5 of the data type opcode byte specify the data type opcode for the color set object to be modified. A data type opcode constant for tag is defined in the `gxColorSetDataOpcode` enumeration. The constant 0 is reserved for future use. Table 7-7 summarizes all of the data type opcodes used to modify a color set object.

Table 7-7 Data type opcodes to modify a color set object

Constant	Value	Description
`gxColorSetReservedOpcode`	0x00	This constant is reserved for future assignment.
`gxColorSetTagOpcode`	0x01	The tag data that follows is added to the current color set object. See the `GXSetColorSetTags` function.

Data Type Opcodes to Modify a Color Profile Object

When the current object is a color profile object and the current operation opcode is the `gxSetDataOpcode` constant, bits 0 through 5 of the data type opcode byte specify the data type opcode for the color profile object to be modified. A data type opcode constant for tag is defined in the `gxProfileDataOpcode` enumeration. The constant 0 is reserved for future use. Table 7-8 summarizes the data type opcodes used to modify a color profile object.

Table 7-8 Data type opcodes to modify a color profile object

Constant	Value	Description
`gxColorProfileReservedOpcode`	0x00	This constant is reserved for future assignment.
`gxColorProfileTagOpcode`	0x01	The tag data that follows is added to the current color profile object. See the `GXSetColorProfileTags` function.

Data Type Opcodes to Modify a Transform Object

When the current object is a **transform** object and the current operation opcode is the `gxSetDataOpcode` constant, bits 0 through 5 of the data type opcode byte specify the data type opcode for the transform object to be modified. A data type opcode constant for tag is defined in the `gxTransformDataOpcode` enumeration. The constant 0 is reserved for future use. Table 7-9 summarizes the data type opcodes used to modify a transform object.

Table 7-9 Data type opcodes to modify a transform object

Constant	Value	Description
`gxTransformReservedOpcode`	0x00	This constant is reserved for future assignment.
`gxTransformTagOpcode`	0x01	The tag data that follows is added to the current transform object. See the `GXSetTransformTags` function.
`gxTransformClipOpcode`	0x02	The tag data that follows is added to the current transform object. See the `GXSetTransformClip` function.
`gxTransformMappingOpcode`	0x03	The tag data that follows is added to the current transform object. See the `GXSetTransformMapping` function.

continued

Table 7-9 Data type opcodes to modify a transform object (continued)

Constant	Value	Description
`gxTransformPartMaskOpcode`	0x04	The tag data that follows is added to the current transform object. See the description of the `gxShapePart` mask parameter to the `GXSetTransformHitTest` function.
`gxTransformToleranceOpcode`	0x05	The tag data that follows is added to the current transform object. See the description of the Fixed tolerance parameter to the `GXSetTransformHitTest` function.

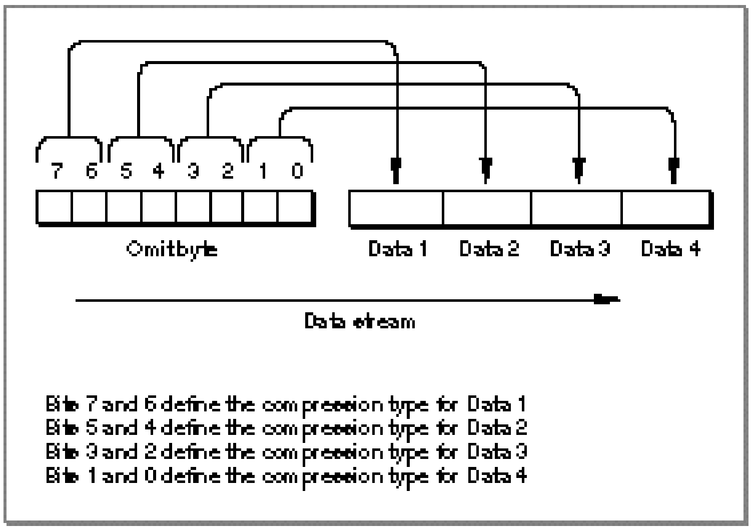
Data

The sequence of the optional object-specific data that follows a data type opcode byte is predetermined and consists of type constants and data. Some data sequences are preceded by an omit byte. An **omit byte** is included in the data stream format to describe the presence or absence, meaning, order, and compression of data that corresponds to the fields of a type or the properties of an object. If an omit byte is not present for an object, then, with the exception of bitmaps and transforms, the compression type opcode in the data type opcode byte defines the data compression.

Omit Byte Masks and Omit Byte Shifts

The omit byte provides an efficient method of assigning different data compressions to type constants and object properties that immediately follow the omit byte. Figure 7-7 shows the relationship of the bits in an omit byte and the four constants or properties that follow.

Figure 7-7 Omit byte relationship with the data that follows


7 6 5 4 3 2 1 0
Omitbyte
Data 1
Data 2
Data 3
Data 4
Data stream
Bits 7 and 6 define the compression type for Data 1
Bits 5 and 4 define the compression type for Data 2
Bits 3 and 2 define the compression type for Data 3
Bits 1 and 0 define the compression type for Data 4

The compression type constants used in the omit byte are defined in the `gxTwoBitCompressionValues` enumeration listed in Table 7-2. Long, word, or byte data compression is applied if the enumeration constants are 0x00, 0x40, 0x80, respectively. If the constant is 0xC0, the compression is "omit compression," then the stream format does not include the field or property. For example, if the omit byte in Figure 7-7 contained 0x0C for bits 3 and 2, Data 3 constant or property would not appear in the stream and Data 4 would follow Data 2.

Some omit byte enumerations provide multiple bytes of mask constants and shift constants to accommodate the description of all of the properties of an object or all of the fields of a structure. For example, the description of a layout shape requires three omit bytes to specify the compression of all of the properties. The data corresponding to each omit byte mask follows the mask. For multiple masks, the sequence is omit mask1, data, omit mask2, data, omit mask3, data, and so on.

You can use an **omit byte mask** and its corresponding **omit byte shift** to interpret the meaning of each of the bits in the omit byte. Each entry in an omit mask enumeration has a name and a value. The name describes the property. The hexadecimal value of the mask is given in the enumeration. The binary equivalent is the mask.

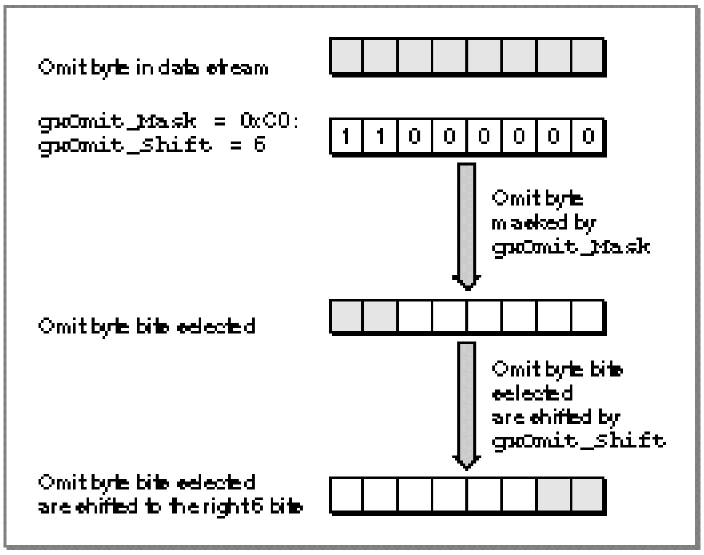
Table 7-10 shows a typical omit byte mask enumeration and its corresponding omit byte shift enumeration values. The example shows the `gxOmitTextMask` enumeration binary mask values and the bit shift from the corresponding `gxOmitTextShift` enumeration.

Table 7-10 Constants from the `gxOmitTextMask` and the `gxOmitTextShift` enumerations

gxOmitTextMask enumeration	**Enumeration value**	**Binary mask value**	**Bit shift constant**
`gxOmitTextCharacterMask`	`0xC0`	11000000	6
`gxOmitTextPositionXMask`	`0x30`	00110000	4
`gxOmitTextPositionYMask`	`0x0C`	00001100	2
`gxOmitTextDataMask`	`0x02`	00000010	1

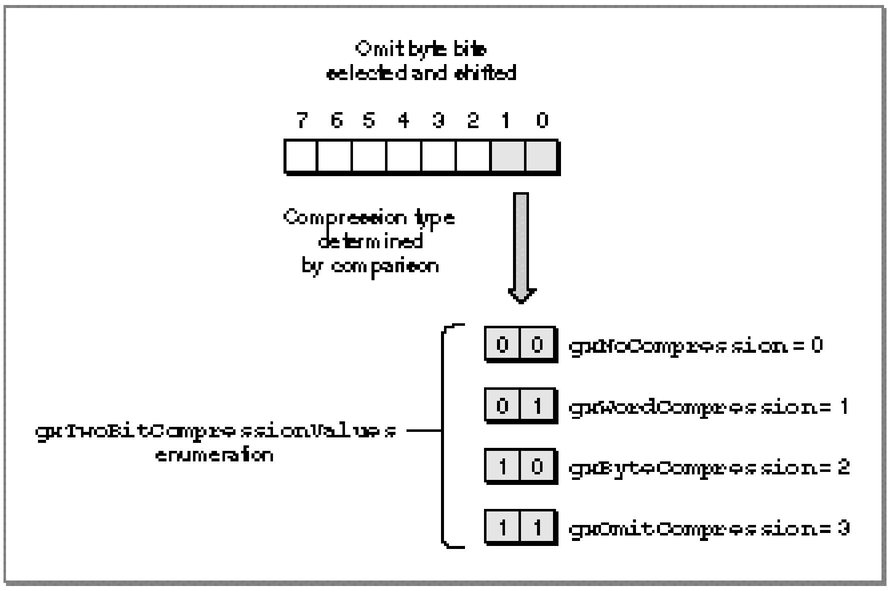
Figure 7-8 shows how you can use an omit mask and corresponding omit shift to analyze an omit byte in the data stream.

Figure 7-8 Select the bits from the omit byte


Omit byte in data stream
gxOmit_Mask = 0xC0:
gxOmit_Shift = 6
1 1 0 0 0 0 0 0
Omit byte masked by gxOmit_Mask
Omit byte bits selected
Omit byte bits selected are shifted by gxOmit_Shift
Omit byte bits selected are shifted to the right 6 bits

First, the bits in the omit byte are masked with the gxOmit_Mask enumeration with a value of 0xC0 and a binary value 11000000. This mask selects the first two high-order bits of the omit byte. In order to interpret the two bits selected, shift the bits to the right by the number of bits indicated by the gxOmit_Shift enumeration value. Once the bits are selected and shifted, determine the compression of the data that follows by comparing these bits with the gxTwoBitCompressionValues enumeration, as shown in Figure 7-9. The values of the gxTwoBitCompressionValues enumeration are given in Table 7-2.

Figure 7-9 Compare the bits selected and shifted with the compression enumeration


Omit byte bits selected and shifted
7 6 5 4 3 2 1 0
Compression type determined by comparison
gxTwoBitCompressionValues enumeration
0 0 gxNoCompression = 0
0 1 gxWordCompression = 1
1 0 gxByteCompression = 2
1 1 gxOmitCompression = 3

Here is an example of how this works with an omit byte describing the shape object for a text shape. First you need to correlate the names of the constants in the omit mask enumeration with the structures, enumerations, or properties of the object that they describe. For more information on correlating omit bytes, see the appropriate object-specific heading in the section“Data” beginning on page 7-22.

Table 7-11 shows the correlation between the `gxOmitTextMask` names and the parameters of the `GXNewTextFunction`.

Table 7-11 Correlation between `gxOmitTextMask` and the `GXNewText` function

Constants in the `gxOmitTextMask` enumeration	Text shape property
`gxOmitTextCharacterMask`	`charCount`
`gxOmitTextPositionXMask`	`position .x`
`gxOmitTextPositionYMask`	`position .y`
`gxOmitTextDataMask`	`text`

A summary of these constants is provided in Table 7-10. The `gxOmitTextMask` enumeration constants correlate with the properties of the text shape. The text shape is described in the text shape chapter of *Inside Macintosh: QuickDraw GX Typography.*

The order of the `gxOmitTextMask` enumeration tells us that the data to follow will be in the sequence `charCount`, `position .x`, `position .y`, and `text`.

For instance, suppose the omit byte is 0xA4 or binary 10100100.

The binary mask value for the `gxOmitTextCharacterMask`, 11000000, selects the high order 2 bits, 10. The `gxTwoBitCompressionValues` enumeration with value 2 is `gxByteCompression`. The data for `charCount` is therefore byte compressed.

The binary mask value for the `gxOmitPositionXMask`, 00110000, selects the next 2 bits, 10. The `gxTwoBitCompressionValues` enumeration with value 2 is again `gxByteCompression`. The data for `position.x` is therefore byte compressed.

The binary mask value for the `gxOmitPositionYMask`, 00001100, selects the next 2 bits, 01. The `gxTwoBitCompressionValues` enumeration with value 1 is `gxWordCompression`. The data for `position.y` is therefore word compressed.

The binary mask value for the `gxOmitTextDataMask`, 10, selects the next bit, 0. The `gxTwoBitCompressionValues` enumeration with value 0 is `gxNoCompression`. The `text` data is therefore not compressed.

The above example is from the analysis of a data stream of a flattened text shape. For additional information about this example see the section “Analyzing a Flattened Text Shape” beginning on page 7-72.

One or more omit mask bytes are included in the data stream whenever specific enumeration or structure data is required to describe a specific object.

Omit mask and omit shift enumerations can be used to analyze QuickDraw GX omit bytes and compare the masked bits to other values.

An omit byte is first masked to obtain the bits desired. The bits are then shifted using the omit shift enumeration that corresponds to the omit byte. The resulting bits can then be compared to other data in your application to obtain information about the data stream.

Listing 7-1 shows how to determine if the x-coordinate of the position field in a flattened shape data stream is compressed.

Listing 7-1 Determining if `position(x)` is byte compressed

```
unsigned char a = ReadByte();
if ((a & (gxOmitTextPositionXMask >> gxOmitTextPositionXShift)) ==
    gxByteCompression
{

/* perform an action */

}
```

The function reads the byte, masks it with `gxOmitTextPositionXMask` to obtain the desired two bits, and then shifts it by the amount given by the `gxOmitTextPositionShift`. The resulting 2 bits can now be compared to the 2 bits of `gxByteCompression.`

Header Data

The header marks the beginning of a new flattened shape in the data stream. The `gxHeaderTypeOpcode` constant indicates that the version of QuickDraw GX that generated the data stream follows. As new versions become available, older software may not be able to interpret the newer portions of a data stream. The interpreter can then look at the version number and skip over versions that it doesn't understand. For example, if an interpreter that understands only QuickDraw GX version 1.0 encounters version 2.0 or if the interpreter finds a version 1.0 opcode, but doesn't recognize the data, an error is posted.

The byte after the version byte contains the `gxFontListFlatten` and `gxFontGlyphsFlatten` flags. These flags are functional only if the shape contains text.

The `gxFontListFlatten` flag instructs the `GXFlattenShape` function to attach a tag object to the flattened shape containing a list of the fonts referenced in the shape. A list of all of the fonts used in the data stream are included at the end of the data stream.

The `gxFontGlyphsFlatten` flag instructs the `GXFlattenShape` function to attach a tag to the flattened shape containing a list of the specific glyphs used from each font referenced by the shape. A list of all of the glyph codes used by all of the fonts referenced in a data stream is then included at the end of that data stream.

For more information about the font and glyph list flags, see the chapter "Shape Objects" in *Inside Macintosh: QuickDraw GX Objects.*

The font list and glyph list are combined to form a tag that is of type `gxFlatFontListItem` and designated `'flst'`. During printing, only the fonts and glyphs used in the stream are loaded to the printing device.

The gxFlatFontList structure includes the gxFlatFontListItem structure. The gxFlatFontListItem contains two arrays. The first is the array of font names. The second is the array of glyphs that are used. The array of glyphs is obtained by setting a bit in an array for each glyph that is used. If you ask only for the font names, the glyph array will be omitted. The glyphs array cannot be selected without the font array selected. In other words, you may specify either a list of fonts or specify a list of fonts and glyphs to be listed at the end of the data stream.

The fonts and glyphs included in the flattened list, 'flst', are used in the print file for the QuickDraw GX portable digital document. For more information on the QuickDraw GX portable digital document see the section "Portable Digital Documents" beginning on page 7-53.

For more information on the QuickDraw GX print file, see the section "About Print Files and Portable Digital Documents" beginning on page 7-51. For more information about how to use the print file data, see the section "Obtaining Data From a Print File" beginning on page 7-89.

For more information on the gxFlatFontName, gxFlatFontListItemTag, and gxFlatFontList structures see the chapter "Fonts" in *Inside Macintosh: QuickDraw GX Typography.*

New Shape Object Data

A new shape object always follows the style, ink, transform, and any other objects that have been built for the shape object in the data stream. New shape data follows an operation opcode gxNewObjectOpcode constant and a data type opcode containing one of the constants in the gxGraphicsNewOpcode enumeration. Values 1 (gxEmptyType) through 13 (gxPictureType) are the constants from the gxShapeTypes enumeration.

This opcode creates a new shape object with all of the properties of the previous shape object in the data stream. If the current shape object is the first shape object in the stream, then it is created with default properties.

The values of the constants for all of the shape objects are summarized in Table 7-3. Shape types are described in the chapter "Shape Objects" in *Inside Macintosh: QuickDraw GX Objects.*

Empty Shape Data

The data type opcode with a value 1 is the gxEmptyType constant. Empty shapes store no information in their geometries. For the current shape object, the gxEmptyType means that the current shape is an empty shape. No data follows.

The gxEmptyTypes constant is described in the chapter "Geometric Shapes" in *Inside Macintosh: QuickDraw GX Graphics.*

Point Shape Data

The data type opcode with a value 2 is the `gxPointType` constant. The data for the fields of a `gxPoint` structure follows. The data sequence is `x` (`Fixed`), `y` (`Fixed`).

Compression data: `gxNoCompression` - read 2 `longs` per point; `gxWordCompression` - read 2 shorts per point or treat each short as a signed integer (120 = 120.0 and –171 = –171.0); `gxByteCompression` - read 2 bytes per point and treat each byte as a signed integer (7 = 7.0 and –13 = –13.0).

The `gxPoint` structure is described in the chapter "Geometric Shapes" in *Inside Macintosh: QuickDraw GX Graphics.*

Line Shape Data

The data type opcode with a value 3 is the `gxLineType` constant. The data for the fields of the `gxLine` structure follows. The data sequence is `first.x`, `first.y` `last.x`, `last.y`.

Compression data: `gxNoCompression` - read 2 `longs` per point; `gxWordCompression` - read 2 shorts per point or treat each short as a signed integer (120 = 120.0 and –171 = –171.0); `gxByteCompression` - read 2 bytes per point and treat each byte as a signed integer (7 = 7.0 and –13 = –13.0).

The `gxLine` structure is described in the chapter "Geometric Shapes" in *Inside Macintosh: QuickDraw GX Graphics.*

Curve Shape Data

The data type opcode with a value 4 is the `gxCurveType` constant. The data for the fields of the `gx`Curve structure follows. The fields in the structure correspond to the parameters in the `GXNew`Curve function. The data sequence is `x` (first point), `y` (first point), `x` (control point), `y` (control point), `x` (last point), and `y` (last point).

Compression data: `gxNoCompression` - read 2 `longs` per point; `gxWordCompression` - read 2 shorts per point or treat each short as a signed integer (120 = 120.0 and –171 = –171.0); `gxByteCompression` - read 2 bytes per point and treat each byte as a signed integer (7 = 7.0 and –13 = –13.0).

The `gxCurve` structure is described in the chapter "Geometric Shapes" in *Inside Macintosh: QuickDraw GX Graphics.*

Rectangle Shape Data

The data type opcode with a value 5 is the `gxRectangleType` constant. The data for the fields of the `gxRectangle` structure follows. The data sequence is left, top, right, bottom. Typically, the first corner is left-top and the second corner is right-bottom; but this order is not required. They need only be opposite corners of a rectangle.

Compression data: `gxNoCompression` - read 2 `longs` per point; `gxWordCompression` - read 2 shorts per point or treat each short as a signed integer (120 = 120.0 and –171 = –171.0); `gxByteCompression` - read 2 bytes per point and treat each byte as a signed integer (7 = 7.0 and –13 = –13.0).

The `gxRectangle` structure is described in the chapter “Geometric Shapes” in *Inside Macintosh: QuickDraw GX Graphics.*

Polygon Shape Data

The data type opcode with a value 6 is the `gxPolygonType` constant. The data for the fields of the `gxPolygons` structure follows. The `gxPolygons` structure includes the `gxPolygon` structure.

The data sequence is `contours`, `vectors`, omit byte, `x` (first point), `y` (first point), `x` (second point), `y` (second point), `x` (third point), `y` (third point), and so on. The numbers are compressed as fixed-point numbers.

The point array for polygons and paths stream is stored as relative positions, not absolute positions (as is the case for the point arrays in polygon and path shapes.)

The omit byte is interpreted by the `gxOmitPathMask` and `gxOmitPathShift` enumerations.

The first two entries of the omit byte describe the compression for the first two points of the polygon shape, which are absolute. The numbers are compressed as fixed-point numbers: `gxNoCompression` means 1 `long` for each fixed number; `gxWordCompression` means 1 `short` for each fixed number treated as an integer (17 = 17.0); `gxByteCompression` means 1 byte per fixed number. Thus a byte compressed value can represent an integer fixed point number from –128.0 to 127.0; a word compression value can represent any integer fixed-point number.

The second two entries in the omit byte describe the compression for the second through the last points in the contour. The coordinates of these points are relative to the first absolute points and appear in the stream as differences. The relative values are stored as differences. Thus each x value in the stream is subtracted from the prior value to reconstruct the original value. Conversely, each value in the shape is subtracted from the prior value to compute the delta to be written to the stream. The x and y coordinate values are considered separately. Each may be independently byte, word, or long compressed, using the same fixed-point compression as the absolute values. Each subsequent contour has its own omit byte to describe the absolute initial point values and the subsequent relative point values.

The compression bits in the data type opcode byte control the compression of the contour counts and all vector counts. Compression data: `gxNoCompression` - read 1 long for contour and each vector count; `gxWordCompression` - read 1 word for contour count and each vector count; `gxByteCompression` - read 1 byte for contour count and each vector count.

The `gxPolygons` structure is described in the chapter “Geometric Shapes” in *Inside Macintosh: QuickDraw GX Graphics.*

Path Shape Data

The data type opcode with a value 7 is the `gxPathType` constant. The data for the fields of the `gxPaths` structure follows. The `gxPaths` structure includes the `gxPath` structure.

The data sequence is `contours` (number of contours), `vectors` (number of points in the contour), control bytes, omit byte, `x` (absolute coordinate of first point), `y` (absolute coordinate of first point), `x` (relative coordinate of second point), `y` (relative coordinate of second point), `x` (relative coordinate of third point), `y` (relative coordinate of third point), and so on.

A control byte contains control bits for each point off or on the path. Each point is assigned a bit. Bits with value 1 are off the path; bits with value 0 are on the path. If the number of points exceeds 8, multiple control bytes are used. If the number of points is not an even multiple of 8, the final unused bits are ignored.

The omit byte is interpreted by the `gxOmitPathMask` and `gxOmitPathShift` enumerations.

The first two entries of the omit byte describe the compression for the first two points of the path shape, which are absolute coordinates. The numbers are compressed as fixed-point numbers: `gxNoCompression` means 1 `long` for each fixed number; `gxWordCompression` means 1 `short` for each fixed number treated as an integer (17 = 17.0); `gxByteCompression` means 1 byte per fixed number. Thus a byte compressed value can represent an integer fixed point number from –128.0 to 127.0; a word compression value can represent any integer fixed-point number.

The second two entries in the omit byte describe the compression for the second through the last relative points in the contour. The coordinates of these points are relative to the first absolute points and appear in the stream as differences. Thus each x value in the stream is subtracted from the prior value to reconstruct the original value. Conversely, each value in the shape is subtracted from the prior value to compute the delta to be written to the stream. The x and y coordinate values are considered separately. Each may be independently byte, word, or long compressed, using the same fixed-point compression as the absolute values. Each subsequent contour has its own omit byte to describe the absolute initial point values and the subsequent relative point values.

The compression bits in the data type opcode byte control the compression of the contour counts and all vector counts. Compression data: `gxNoCompression` - read 1 long for contour and each vector count; `gxWordCompression` - read 1 word for contour count and each vector count; `gxByteCompression` - read 1 byte for contour count and each vector count.

The `gxPaths` structure is described in the chapter “Geometric Shapes” in *Inside Macintosh: QuickDraw GX Graphics.*

Bitmap Shape Data

The data type opcode with a value 8 is the `gxBitmapType` constant. The data for the fields of the `gxBitmap` and `gxPoint` structures follow. The `gxBitmap` structure includes the `gxColorSpace` enumeration and the references to the `gxColorSet` and `gxColorProfile` structures.

The data sequence is omit byte 1, `image` reference, `width`, `height`, `rowBytes`, omit byte 2, `pixelSize`, `space` (color space), `set` (color set), `profile` (color profile), omit byte 3, `x` (position), `y` (position).

Omit byte 1 is interpreted by the `gxOmitBitmapMask1` and `gxOmitBitmapShift1` enumerations. Omit byte 2 is interpreted by the `gxOmitBitmapMask2` and `gxOmitBitmapShift2` enumerations. Omit byte 3 is interpreted by the `gxOmitBitmapMask3` and `gxOmitBitmapShift3` enumerations.

Data compression: The value may be a byte, word, or long. The value references a previous bit image: a value of 1 references the first bit image, a value of 2 references the second bit image, etc. A value of 0 indicates that the bitmap references a bit image through a file alias. The bitmap shape must reference a tag containing the file alias and offset as described in the chapter "Tag Objects" in *Inside Macintosh: QuickDraw GX Objects.* All bitmap values are compressed as integers (see polygon coutour compression above) except for the x and y coordinate positions. These are compressed as `Fixed` (see polygon first absolute position). Unlike prior shape types in this section, bitmaps and shape types described below can also have fields with the `gxOmitCompression` bits set. In this case, the value 0 or `nil` is used wherever the omit compression bits are set.

The `gxBitmap` structure is described in the chapter "Bitmap Shapes" in *Inside Macintosh: QuickDraw GX Graphics.*

Text Shape Data

The data type opcode with a value 9 is the `gxTextType` constant. The data that follows corresponds to the parameters of the `GXNewText` function.

The data sequence is omit byte, byte length (of text), `x` (position), `y` (position), `charCount` (number of characters), data (character text).

The data is the character stream or glyph indexes for the text. For nonRoman scripts, the actual byte length may be more than the number of characters.

The omit byte is interpreted by the `gxOmitTextDataMask` and `gxOmitTextDataShift` enumerations.

Data compression: The byte length is compressed as a long. The x and y coordinates are compressed as a fixed number. The data stream may contain bytes or shorts. If the stream contains shorts and all values are less than 255, then the stream may be compressed. It is an error to specify a character count of zero (omit compression) and to set the text data omit bit.

The `GXNewText` function is described in the chapter "Text Shapes" in *Inside Macintosh: QuickDraw GX Typography*.

Glyph Shape Data

The data type opcode with a value 10 is the gxGlyphType constant. The data correspond to the parameters in the GXNewGlyphs function and include the gxPoint and gxStyle structures.

The data sequence is omit byte 1, charCount (number of characters), byte length (of text), runNumber (number of runs), data (glyph character), omit byte 2, positions, advance, tangents, styleRuns, glyphStyles.

Omit byte 1 is interpreted by the gxOmitGlyphDataMask1 and gxOmitGlyphDataShift1 enumerations. Omit byte 2 is interpreted by the gxOmitGlyphDataMask2 and gxOmitGlyphDataShift2 enumerations.

Data compression: charCount, byte length, and runNumber are compressed as longs. If charCount is 0, the data, positions, advance, and tangents are not read. If the gxOmitGlyphOnePosition bit is set in the first byte, then the glyph shape contains 1 absolute position or as many positions as there are in the stream. In either case, all are compressed as fixed point values, as bytes, words, or longs. Unlike polygon positions, the x and y values do not have separate compression bits, nor are the positions stored in the relative manner of polygons or paths.

The advances in the glyph shape are read after the positions, if the gxOmitGlyphAdvance bits are not gxOmitCompression constant. The character count determines the number of bytes read, as is the case with the control bits in a path shape.

If the gxOmitGlyphTangent bits in the second omit byte are not equal to the gxOmitCompression constant, the tangents parameter follows. The tangent values are stored and compressed identically to the positions. If the number of runs (runNumber) is greater than zero, then 1 bit in the second omit byte interprets the runs as shorts or shorts compressed to bytes (like the text character compression). If runNumber is greater than 0, then the style array is compressed into an array of bytes, words, or longs. The values are references to previous styles in the stream: a value of 1 references the 1st style in the stream, and so on.

The GXNewGlyphs function is described in the chapter “Glyph Shapes” in *Inside Macintosh: QuickDraw GX Typography*.

Layout Shape Data

The data type opcode with a value 11 is the gxGlyphType constant. The data correspond to the parameters in the GXNewLayout function.

Layouts are compressed in a way that is similar to glyphs. Like all types that are greater than or equal to bitmap type, all fields default to zero and omit compression is allowed. If the length is greater than 0, the data is read as shorts compressed as bytes or as an uncompressed stream (like text and glyphs). If the style run number is greater than 0, the style run array and style array are present identically to the glyph format. If the omitLayoutHasBaseline bit is set in omit byte 3, uncompressed data is read the size of the `gxLineBaselineRecord`. If the level run number is greater than zero, the 4th omit byte (read regardless) specifies the compression of the `levelRunLength` and level arrays as an optionally compressed array of shorts.

The data sequence is omit byte 1, `length`, `x` (position), `y` (position), data, omit byte 2, `width`, `flush`, `set`, `just`, `options`, omit byte 3, `style`, run number, level run number, `hasBaseline`, style runs, styles, omit byte 4, level runs, levels.

Omit byte 1 is interpreted by the `gxOmitLayoutMask1` and `gxOmitLayoutShift1` enumerations. Omit byte 2 is interpreted by the `gxOmitLayoutMask2` and `gxOmitLayoutShift2` enumerations. Omit byte 3 is interpreted by the `gxOmitLayoutMask3` and `gxOmitLayoutShift3` enumerations. Omit byte 4 is interpreted by the `gxOmitLayoutMask4` and `gxOmitLayoutShift4` enumerations.

The `GXNewLayout` function is described in the chapter “Layout Shapes” in *Inside Macintosh: QuickDraw GX Typography*.

Full Shape Data

The data type opcode with a value 12 is the `gxFullType` constant. Full shapes store no information in their geometries. For the current shape object, the `gxFullType` constant is a parameter in the `GXNewShape` function. No data follows.

The `gxFullType` constant is described in the chapter “Geometric Shapes” in *Inside Macintosh: QuickDraw GX Graphics.*

Picture Shape Data

The data type opcode with a value 13 is the `gxPictureType` constant. The data corresponds to the parameters in the `GXNewPicture` function. The data sequence is omit byte 1, the number of items (compressed as long as specified by the data type opcode), followed by an array of shapes and optional arrays of styles, inks, and transforms. The shape array must exist and may not contain nil (zero) references. The styles, inks and transform array references may be omitted entirely.

The `gxPicture` structure is described in the chapter “Picture Shapes” in *Inside Macintosh: QuickDraw GX Graphics.*

Modified Shape Object Data

Once a shape object is defined, it can be modified. Modified shape data follow a `gxSetDataOpcode` operation opcode and a data type opcode containing one of the constants from the `gxShapeDataOpcode` enumeration. Table 7-4 summarizes the values of the constants for all of the modified shape objects.

Attributes Data

An attribute is added to the current shape object if the data type opcode has value 0. This is the `gxShapeAttributesOpcode` constant.

The data for the fields of the `gxShapeAttributes` structure follow and are compressed as long. That data may be 1, 2, or 4 bytes depending on the compression bits.

The `gxShapeAttributes` enumeration is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects.*

Tag Data

A tag is added to the current shape if the data type opcode has value 1. This is the `gxShapeTagOpcode` constant. The data for the parameters of the `GXSetShapeTags` function follows.

The size of the opcode specifies the number of tags; the compression specifies whether the data is in bytes, words, or longs. For instance, if the size is 4 and the compression is `gxShortCompression` (2 bytes), then the stream contains 4/2 == 2 tags. The equivalent operation would be `GXSetShapeTags` (shape, nil, 1, 0, 2, tag array).

The `GXSetShapeTags` function is described in the chapter “Shape Objects” of *Inside Macintosh: QuickDraw GX Objects.*

Fill Data

A **shape fill,** compressed as long, is added to the current shape if the data type opcode has value 2. This is the `gxShapeFillOpcode`. A constant from the `gxShapeFill` enumeration follows.

The `gxShapeFills` enumeration is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects.*

New Style Object Data

Data for a new style object follows a `gxNewObjectOpcode` operation opcode and a data type opcode with a value 28. This is the `gxStyleTypeOpcode` constant from the `gxGraphicsNewOpcode` enumeration.

This opcode creates a new style object with all of the properties of the previous style object in the data stream. If the current style object is the first style object in the stream, then it is created with default properties. No data follows for the new style object.

The style object is described in the chapter “Style Objects” in *Inside Macintosh: QuickDraw GX Objects.*

Modified Style Object Data

Once a style object is defined, it can be modified by the addition of style data. Modified style data follows a `gxSetDataOpcode` operation opcode and a data type opcode containing one of the constants from the `gxStyleDataOpcode` enumeration. Table 7-5 summarizes the values of the constants for all of the modified style objects. For all style data, the opcodes described in the following subsections change the default style.

Attributes Data

An attribute is added to the current style object if the data type opcode has value 0. This is the `gxStyleAttributesOpcode` constant.

The data, compressed as long, for the fields of the `gxStyleAttribute` structure follow and may be byte, short, or long.

The `gxStyleAttributes` enumeration is described in the chapter "Geometric Styles" in *Inside Macintosh: QuickDraw GX Graphics.*

Tag Data

A tag is added to the current style if the data type opcode has value 1. This is the `gxStyleTagOpcode` constant. The data for the parameters of the `GXSetStyleTags` function follows.

The size of the opcode specifies the number of tags; the compression specifies whether the data is in bytes, words, or longs. For instance, if the size is 4 and the compression is `gxShortCompression` (2 bytes), then the stream contains 4/2 == 2 tags. The equivalent operation would be `GXSetShapeTags` (shape, nil, 1, 0, 2, tag array);

The `GXSetStyleTags` function is described in the chapter "Style Objects" in *Inside Macintosh: QuickDraw GX Objects.*

Curve Error Data

A curve error, compressed as fixed-point, is added to the current style if the data type opcode has value 2. This is the `gxStyleCurveErrorOpcode` constant. The data for the `error` (`Fixed`) parameter of the `GXSetStyleCurveError` function follows.

For fixed point compression `gxNoCompression` means 1 `long` for each fixed number; `gxWordCompression` means 1 `short` for each fixed number treated as an integer (17 = 17.0); `gxByteCompression` means 1 byte per fixed number. Thus a byte compressed value can represent an integer fixed point number from –128.0 to 127.0; a word compression value can represent any integer fixed-point number.

The `GXSetStyleCurveError` function is described in the chapter "Geometric Styles" in *Inside Macintosh: QuickDraw GX Graphics.*

Pen Data

A pen, compressed as fixed point, is added to the current style object if the data type opcode has value 3. This is the `gxStylePenOpcode` constant. The data for the `pen` (`Fixed`) parameter of the `GXSetStylePen` function follows.

For fixed-point compression `gxNoCompression` means 1 `long` for each fixed number; `gxWordCompression` means 1 `short` for each fixed number treated as an integer (17 = 17.0); `gxByteCompression` means 1 byte per fixed number. Thus a byte compressed value can represent an integer fixed point number from –128.0 to 127.0; a word compression value can represent any integer fixed-point number.

The `GXSetStylePen` function is described in the chapter "Geometric Styles" in *Inside Macintosh: QuickDraw GX Objects.*

Join Data

A join is added to the current style object if the data type opcode has value 4. This is the `gxStyleJoinOpcode`. The data for the fields of the `gxJoinRecord` structure follows. The `gxJoinRecord` structure includes the `gxShape` and `gxJoinAttribute` structures.

The data sequence is omit byte, `attributes` (modifier flags) compressed as long, `join` (corner shape) compressed as long (reference), `miter` (size limit) compressed as fixed point.

The omit byte is interpreted by the `gxOmitJoinMask` and `gxOmitJoinShift` enumerations.

The `gxJoinAttribute` structure is described in the chapter "Geometric Styles" in *Inside Macintosh: QuickDraw GX Graphics.*

Dash Data

A dash is added to the current style object if the data type opcode has value 5. This is the `gxStyleDashOpcode` constant. The data for the fields of the `gxDashRecord` structure follows. The `gxShape` and `gxDashAttribute` enumerations are included in the `gxDashRecord` structure.

The data sequence is omit byte 1, `attributes` (modifier flags) compressed as long, `dash` (shape used for dash) compressed as long (reference), `advance` (distance between dashes) compressed as long, `phase` (start offset) compressed as fract, omit byte 2, and `scale` (height of dash) compressed as fixed`.`

In fract compression a long means a full fract; a word means that 16 bits are read followed by 16 bits of zeros; a byte means that 8 bits are read followed by 24 bits of zeros. Thus numbers like 1.0, -1.0, or fract 0.5 fit into a compressed byte.

Omit byte 1 is interpreted by the `gxOmitDashMask1` and `gxOmitDashShift1` enumerations. Omit byte 2 is interpreted by the `gxOmitDashMask2` and `gxOmitDashShift2` enumerations.

The `gxDashRecord` structure is described in the chapter "Geometric Styles" of *Inside Macintosh: QuickDraw GX Graphics.*

Caps Data

A cap is added to the current style object if the data type opcode has value 6. This is the `gxStyleCapsOpcode`. The data for the fields of the `gxCapRecord` structure follows. The `gxShape` and `gxCapAttribute` enumerations are included in the `gxCapRecord` structure.

The data sequence is omit byte, `attributes` (modifier flags) compressed as long, `startCap` (shape used at start of contours) compressed as long (reference), `endCap` (shape used at end of contours) compressed as long (reference).

The omit byte is interpreted by the `gxOmitCapMask` and `gxOmitCapShift` enumerations.

The `gxCapRecord` structure is described in the chapter "Geometric Styles" in *Inside Macintosh: QuickDraw GX Graphics.*

Pattern Data

A pattern is added to the current style object if the data type opcode has value 7. This is the `gxStylePatternOpcode` constant. The data for the fields of the `gxPatternRecord` structure follows. The `gxShape`, `gxPatternAttribute`, and `gxPoint` enumerations are included in the `gxPatternRecord` structure.

The data sequence is omit byte 1, `attributes` (modifier flags) compressed as long, `pattern` (shape to use as pattern) compressed as long (reference), `x` (x-coordinate of vector `u` for pattern grid) compressed as fixed, `y` (y-coordinate of vector `u` for pattern grid) compressed as fixed, omit byte 2, `x` (x coordinate of vector `v` for pattern grid) compressed as fixed, and `y` (y-coordinate of vector `v` for pattern grid) compressed as fixed. Note that for all of these, omit (zero) values are permitted.

Omit byte 1 is interpreted by the `gxOmitPatternMask1` and `gxOmitPatternShift1` enumerations. Omit byte 2 is interpreted by the `gxOmitPatternMask2` and `gxOmitPatternShift2` enumerations.

The `gxPatternRecord` structure is described in the chapter "Geometric Styles" in *Inside Macintosh: QuickDraw GX Graphics.*

Text Attributes Data

A text attribute compressed as long is added to the current style object if the data type opcode has value 8. This is the `gxStyleTextAttributesOpcode` constant. The data may be byte, word, or long.

The `gxTextAttribute` enumeration is described in the chapter "Typographic Styles" in *Inside Macintosh: QuickDraw GX Typography.*

Text Size Data

The text size, compressed as long, for the current style object is specified if the data type opcode has value 9. This is the `gxStyleTextSizeOpcode` constant. The data for the `size` (fixed point size of text) parameter of the `GXSetStyleTextSize` function follows.

The `GXSetStyleTextSize` function is described in the chapter "Typographic Styles" in *Inside Macintosh: QuickDraw GX Typography*.

Font Data

A font is added to the current style object if the data type opcode has value 10. This is the `gxStyleFontOpcode` constant. The attribute data for the `GXSetStyleFont` function follows. It is compressed as long (reference); the reference is to a font name defined earlier in the stream

The `GXSetStyleFont` function is described in the chapter "Typographic Styles" in *Inside Macintosh: QuickDraw GX Typography*.

Text Face Data

A text face is added to the current style object if the data type opcode has value 11. This is the `gxStyleTextFaceOpcode` constant. The data for the fields of the `gxTextFace` structure follows.

The data sequence is omit byte, `faceLayers` compressed as long, mapping size and `advanceMapping.`

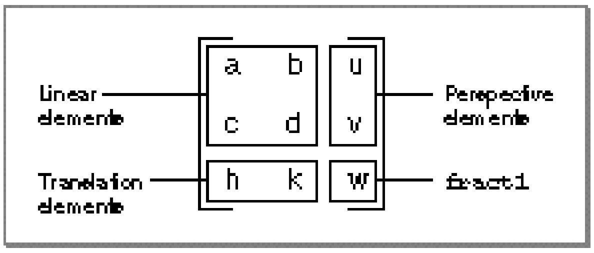
The `advanceMapping` in text face and transform mapping is reordered so that common mappings can be stored in fewer bytes. The omit byte and number of layers is followed by an optional byte (whose compression is described by `omitFaceMapping).`

The value of the byte may be one of the following:

Byte	Value
2	Mapping contains identity plus elements h and k.
4	Same as byte 2, plus elements a and d.
6	Same as byte 4, plus elements b and c.
9	Same as byte 6 plus elements u, v, and w.

The meaning of the elements mentioned in the previous table are shown in Figure 7-10.

Figure 7-10 Mapping matrix elements



The byte value is multiplied by the compression level to specify the length of the mapping data that follows. Byte compression multiplies by 1; word compression multiplies by 2; long compression multiplies by 4. The values in the left and middle columns are compressed as fixed values. The values in the right column are compressed as fract values. All elements whether the stream contains 2, 4, 6, or 9 numbers, have the same level of compression.

If the `faceLayers` value is greater than 0, then following the mapping data is an omit byte as described by `gxOmitFaceLayer` Mask 1. The omit byte is followed by the `outlineFill` compressed as a long, the flags comrpessed as a long, the `outlineStyle` and reference compressed as a long, and the `outlineTransform`, also comrpessed as a long. The second omit byte describes the bold x and bold y, compressed as fixed values. This sequence is repeated for the second and all remaining layers.

The omit byte is interpreted by the `gxOmitFaceMask` and `gxOmitFaceShift` enumerations.

The `gxTextFace` structure is described in the chapter “Typographic Styles” in *Inside Macintosh: QuickDraw GX Typography*.

Platform Data

The platform, script, and language is defined for the current object if the data type opcode has value 12. This is the `gxStylePlatformOpcode` constant. Data from the `gxFontPlatform`, `gxFontScript`, and `gxFontLanguage` enumerations follow.

The platform, script, and language are combined into a long and then that value is compressed as a long that is equal to

```
(platform << 16) | (script << 8) | language
```

The `gxFontPlatform`, `gxFontScript`, and `gxFontLanguage` enumerations are described in the chapter “Font Objects” in *Inside Macintosh: QuickDraw GX Typography*.

Font Variations Data

Font variations are added to the current style object if the data type opcode has value 13. The data is uncompressed. This is the `gxStyleFontVariationsOpcode` constant. The data for the fields of the `gxFontVariation` structure follows. The `gxFontVariationTag` structure is included in the `gxFontVariations` structure.

The data sequence is an array `[name` (variation tag), `value` (`Fixed`)`].`The opcode size specifies the number of variations in the stream.

The `gxFontVariation` structure is described in the chapter “Fonts” in *Inside Macintosh: QuickDraw GX Typography*.

Run Controls Data

Run controls are added to the current style object if the data type opcode has value 14. The data is uncompressed. This is the `gxStyleRunControlsOpcode` constant. The data for the fields of the `gxRunControls` structure follows. The opcode size specifies the size in bytes of the run control stream.

The `gxRunControls` structure is described in the chapter “Layout Line Controls” in *Inside Macintosh: QuickDraw GX Typography*.

Run Priority Justification Override Data

A run priority justification override is added to the current style object if the data type opcode has value 15. The data is uncompressed. This is the `gxStyleRunPriorityJustOverrideOpcode` constant. The data for the fields of the `gxPriorityJustificationOverride` structure follows. The opcode size specifies the size in bytes of the run control stream.

The data sequence is an array of delta. The opcode specifies the byte size.

The `gxPriorityJustificationOverride` structure is described in the chapter “Layout Line Controls” in *Inside Macintosh: QuickDraw GX Typography*.

Run Glyph Justification Overrides Data

A run glyph justification override is added to the current style object if the data type opcode has value 16. The data is uncompressed. This is the `gxStyleRunGlyphJustOverrideOpcode` constant. The data for the fields of the `gxGlyphJustificationOverride` structure follows. The `gxGlyphJustificationOverride` structure includes the `gxGlyphcode` and `gxWidthDeltaRecord` enumerations. The opcode specifies the byte size.

The data sequence is `count`, `glyphJustificationOverrides.`

The `gxGlyphJustificationOverride` structure is described in the chapter “Layout Line Controls” in *Inside Macintosh: QuickDraw GX Typography*.

Run Glyph Substitutions Data

A run glyph substitution is added to the current style object if the data type opcode has value 17. The data is uncompressed. This is the `gxStyleRunGlyphSubstitutionsOpcode` constant. The data for the fields of the `gxGlyphSubstitution` structure follows.

The data sequence is `count`, `glyphsubstitutions[].`

The `GXSetStyleRunGlyphSubstitutions` structure is described in the chapter "Layout Line Controls" in *Inside Macintosh: QuickDraw GX Typography*.

Run Features Data

A run feature is added to the current style object if the data type opcode has value 18. The data is uncompressed. This is the `gxStyleRunFeaturesOpcode` constant. The data for the fields of the `gxRunFeature` structure follows.

The data sequence is `count`, `runFeatures[].`

The `gxRunFeature` structure is described in the chapter "Layout Line Controls" in *Inside Macintosh: QuickDraw GX Typography*.

Run Kerning Adjustments Data

Run kerning adjustment is added to the current style object if the data type opcode has value 19. The data is uncompressed. This is the `gxStyleRunKerningAdjustmentsOpcode` constant. The data for the fields of the `gxKerningAdjustment` structure follows.

The data sequence is `count`, `kerningAdjustments[].`

The `gxKerningAdjustment` structure is described in the chapter "Layout Line Controls" in *Inside Macintosh: QuickDraw GX Typography.*

Style Justification Data

Style justification is added to the current style object if the data type opcode has value 20. The data is compressed as fract. This is the `gxStyleJustificationOpcode` constant. The data for the justify parameter of the `GXSetStyleJustification` function follows.

In fract compression a long means a full fract; a word means that 16 bits are read followed by 16 bits of zeros; a byte means that 8 bits are read followed by 24 bits of zeros. Thus numbers like 1.0, -1.0, or fract 0.5 fit into a compressed byte.

The `GXSetStyleJustification` function is described in the chapter "Typographic Styles" in *Inside Macintosh: QuickDraw GX Typography*.

New Ink Object Data

Data for a new ink object follows a `gxNewObjectOpcode` operation opcode and a data type opcode with a value 29. This is the `gxInkTypeOpcode` constant from the `gxGraphicsNewOpcode` enumeration.

This opcode creates a new ink object with all of the properties of the previous ink object in the data stream. If the current ink object is the first ink object in the stream, then it is created with default properties. No data follows for the new ink object.

The ink object is described in the chapter "Ink Objects" in *Inside Macintosh: QuickDraw GX Objects.*

Modified Ink Object Data

Once an ink object is defined, it can be modified by the addition of ink data. Modified style data follows a `gxSetDataOpcode` operation opcode and a data type opcode containing one of the constants from the `gxInkDataOpcode` enumeration. Table 7-6 summarizes the values of the constants for all of the modified ink objects.

Attributes Data

An attribute, compressed as long, is added to the current ink object if the data type opcode has value 0. This is the `gxInkAttributes`Opcode constant.

The data for the fields of the `gxInkAttributes` structure follow. The next two bytes contain the ink attribute flags.

The `gxInkAttributes` enumeration is described in the chapter "Ink Objects" in *Inside Macintosh: QuickDraw GX Objects.*

Tag Data

A tag is added to the current ink object if the data type opcode has value 1. This is the `gxInkTagOpcode` constant. The data for the parameters of the `GXSetInkTags` function follows.

The size of the opcode specifies the number of tags; the compression specifies whether the data is in bytes, words, or longs. For instance, if the size is 4 and the compression is `gxShortCompression` (2 bytes), then the stream contains 4/2 == 2 tags. The equivalent operation would be `GXSetShapeTags` (shape, nil, 1, 0, 2, tag array).

The sequence is `tagType`, `index`, `oldCount`, `newCount`, `items`.

The `GXSetInkTags` function is described in the chapter "Ink Objects" in *Inside Macintosh: QuickDraw GX Objects.*

Color Data

A color is added to the current ink object if the data type opcode has value 2. This is the `gxInkColorOpcode` constant. The data for the fields of the `gxInkAttributes` structure follow. The data for the fields of the `gxColor` structure follows.

The data sequence is omit byte, `space` (long), `profile` (long). The value of the omit byte may be omit compression.

The omit byte is interpreted by the `gxOmitColorsMask` and `gxOmitColorsShift` enumerations.

If space is indexed space, `gxOmitColoursIndex` is used to determine index compression (compressed as long), which is read first, followed by color set (compressed as long), with the compression determined by `gxOmitColorsIndexSet`.

If space is not indexed space, the color space determines the number of elements read from the stream as shown in Table 7-12.

Table 7-12 Color space and words read

16-bit	1
32-bit	2
gray, index	1
gray alpha	2
RGB, HSV, HLS, YXY, XYZ, LUV, LAB, YIQ	3
RGBA, CYMK	4

The bits in the omit byte determine whether a word is read from the stream for each word in the component or whether the byte is repeated twice for each word. For example, if the byte contains 0x3A, the word contains 0X3A3A. The `gxOmitColorsComponentsMask` sets 1 bit for up to 4 components.

The `gxColor` enumeration is described in the chapter "Ink Objects" in *Inside Macintosh: QuickDraw GX Objects.*

Transfer Mode Data

A transfer mode is added to the current ink object if the data type opcode has value 3. This is the `gxInkTransferModeOpcode` constant. The data for the fields of the `gxTransferMode` structure follow.

The data sequence is omit byte 1, `space`, compressed as long, `set`, compresssed as long, `profile`, compressed as long; omits are allowed. Omit byte 2 follows and then `sourceMatrix`, `deviceMatrix`, `resultMatrix`, `flags`, and `component`; omits are allowed.

The `sourceMatrix`, `deviceMatrix`, and `resultMatrix` are compressed as arrays of Fixed values. The color space determines the number of transfer components that follow, as shown in Table 7-12.

Each transfer component is preceded by an omit byte (`gxomitTransferComponentMask1`) that describes the first 4 fields of the structure. Omit byte one is followed by `gxOmitTransferComponentModeMask`, compressed as byte, `gxOmitTransferComponentFlagsMask`, compressed as byte, `gxOmitTransferComponentSourceMinimumShift`, compressed as color, `gxOmitTransferComponentSourceMaximumMask`, compressed as color, and `gxOmitTransferComponentDeviceMinimumMask`, compressed as color. Omit byte 2 follows which describes `gxOmitTransferComponentDeviceMaximumMask`, `gxOmitTransferComponentClampMinimumMask`, `gxOmitTransferComponentClampMaximumMask`, and `gxOmitTransferComponentOperandMask`; all these are compressed as color. The color compression specifies that the field may be omitted (inherits value from default), or is represented by a repeated byte (for example, `0X7A ==0X7A7A`), or is represented as a word.

Note that the mode and flags in the first omit byte have a single bit

The `gxTransferMode` structure is described in the chapter "Ink Objects" in *Inside Macintosh: QuickDraw GX Objects.*

New Object Transform Data

Data for a new transform object follows a `gxNewObjectOpcode` operation opcode and a data type opcode with a value 0x2A. This is the `gxTransformTypeOpcode` constant from the `gxGraphicsNewOpcode` enumeration.

This opcode creates a new transform object with all of the properties of the previous transform object in the data stream. If the current transform object is the first transform object in the stream, then it is created with default properties. No data follows for the new transform object.

The transform object is described in the chapter "Transform Objects" in *Inside Macintosh: QuickDraw GX Objects.* For additional information about transform mapping, see "Mapping Data" on page 7-46.

Modified Transform Object Data

Once a transform object is defined in the data stream, it can then be modified. Modified transform object data follows a `gxSetDataOpcode` operation opcode and a data type opcode containing one of the constants from the `gxTransformDataOpcode` enumeration. Table 7-9 summarizes the values of the constants for all of the modified transform objects.

Reserved Opcode for Modified Transform Data

The data type opcode with value 0 is reserved for future expansion.

Tag Data

A tag is added to the current transform object if the data type opcode has value 1. This is the `gxTransformTagOpcode` constant. The data for the parameters of the `GXSetTransformTags` function follows.

The data stream sequence is `tagType`, `index`, `oldCount`, `newCount`, `items[]`.

The `GXSetTransformTags` function is described in the chapter "Transform Objects" of *Inside Macintosh: QuickDraw GX Objects.*

Clip Data

A clip, compressed as long (reference) is added to the current transform object if the data type opcode has value 2. This is the `gxTransformClipOpcode` constant. The data for the `clip` parameter of the `GXSetTransformClip` function follows.

The `GXSetTransformClip` function is described in the chapter "Transform Objects" in *Inside Macintosh: QuickDraw GX Objects.*

Mapping Data

A mapping is added to the current transform object if the data type opcode has value 3. This is the `gxTransformMappingOpcode` constant. The data for the `map` parameter of the `GXSetTransformMapping` function follows.

A transform mapping is initiated by the sequential appearance of the `gxSetDataOpcode`, and `gxTransformDataOpcode` constants in the data stream.

The bytes following the appearance in the data stream of the `gxTransformMapping` constant from the `gxTransformDataOpcode` enumeration have a special format. The interpretation of the bytes that follow require the determination of a size constant. The size to be used for a specific transform depends upon the compression and the size of the transform data specified by the byte containing the previous `gxGraphicsOperationOpcode` constant. The size is the number of bytes, words, or longs, depending upon the type of compression.

If the size obtained from the `gxGraphicsOperationOpcode` byte indicated that there are 24 bytes of transform data and the byte containing the `gxTransformMappingOpcode` constant indicated that there was no compression, then the size of each transform attribute would be 4 bytes (longs) and the size constant for our transformation bytes format would be size 24/4 = 6. The interpretation of the mapping that occurs for each mapping size is summarized in the section "Text Face Data" on page 7-39.

The `GXSetTransformMapping` function is described in the chapter "Transform Objects" in *Inside Macintosh: QuickDraw GX Objects.*

Part Mask Data

A part mask, compressed as a long, is added to the current transform object if the data type opcode has value 4. This is the `gxTransformPartMaskOpcode` constant. The data for the `mask` parameter of the `GXSetTransformHitTest` function follows.

The `GXSetTransformHitTest` function is described in the chapter "Transform Objects" in *Inside Macintosh: QuickDraw GX Objects.*

Tolerance Data

Tolerance, compressed as long, is added to the current transform object if the data type opcode has value 5. This is the `gxTransformToleranceOpcode` constant. The data for the tolerance parameter of the `GXSetTransformHitTest` function follows.

The `GXSetTransformHitTest` function is described in the chapter "Transform Objects" in *Inside Macintosh: QuickDraw GX Objects.*

New Color Profile Object Data

Data for a new color profile object follows a `gxNewObjectOpcode` operation opcode and a data type opcode with a value 0x2B. This is the `gxColorProfileTypeOpcode` constant from the `gxGraphicsNewOpcode` enumeration.

This opcode creates a new color profile object with all of the properties of the previous color profile object in the data stream. If the current color profile object is the first color profile object in the stream, then it is created with default properties. The data that follows is uncompressed; the opcode size specifies the size of the stream.

The color profile object is described in the chapter "Color Objects" in *Inside Macintosh: QuickDraw GX Objects.*

Modified Color Profile Object Data

Once a color profile object is defined in the data stream, it can be modified. Modified color set object data follows a `gxSetDataOpcode` operation opcode and a data type opcode containing one of the constants from the `gxColorProfileDataOpcode` enumeration. Table 7-8 summarizes the values of the constants for all of the modified color profile objects.

Reserved Opcode for Modified Color Profile Data

The data type opcode with value 0 is reserved for future expansion.

Color Profile Tag Data

A tag for the current color profile object is added if the data type opcode has value 1. This is the `gxColorProfileTagOpcode` constant. The data for the parameters of the `GXSetColorProfileTags` function follows.

The size of the opcode specifies the number of tags; the compression specifies whether the data is in bytes, words, or longs. For instance, if the size is 4 and the compression is `gxShortCompression` (2 bytes), then the stream contains 4/2 == 2 tags. The equivalent operation would be `GXSetShapeTags` (shape, nil, 1, 0, 2, tag array).

The `GXSetColorProfileTags` function is described in the chapter "Color Objects" in *Inside Macintosh: QuickDraw GX Objects.*

New Color Set Object Data

Data for a new color set object follows a `gxNewObjectOpcode` operation opcode and a data type opcode with a value 0x2C. This is the `gxColorSetTypeOpcode` constant from the `gxGraphicsNewOpcode` enumeration.

This opcode creates a new color set object with all of the properties of the previous color set object in the data stream. If the current color set object is the first color set object in the stream, then it is created with default properties.

The color set object is described in the chapter "Color Objects" in *Inside Macintosh: QuickDraw GX Objects.*

Modified Color Set Object Data

Once a color set object is defined in the data stream, it can be modified. Modified color set object data follows an operation opcode `gxSetDataOpcode` constant from the `gxGraphicsOperationOpcode` enumeration and a data type opcode containing one of the constants from the `gxColorSetDataOpcode` enumeration. Table 7-7 summarizes the values of the constants for modified color set objects.

The first byte or two is space, space and specifies the number of components. The remaining stream is colors. The compression for the color set can be byte or word. To determine the number of colors in the stream use the following formula:

```
(size - colorSpaceByte * compression) / componentsInColorSpace *
                                                    compression
```

For instance, if the space is `gxRGBSpace`, the compression is `gxByteCompression`, and the size is 7, the number of colors would be (7 - 1 * 1)/3*1, which evaluates to 2. If the stream continued with 0, 0, 0, 0XFF, 0XFF, 0XFF, then the color set would contain black (0X0000, 0X0000,0X0000) and white 0XFFFF, 0XFFFF, 0XFFFF). As the example shows, the color set entries are compressed as colors. See section "Transfer Mode Data" on page 7-44 for information on color compression.

Reserved Opcode for Modified Color Set Data

The data type opcode with value 0 is reserved for future expansion.

Color Set Tag Data

A tag is added to the current color set object if the data type opcode has value 1. This is the `gxColorSetTagOpcode` constant. The data for the parameters of the `GXSetColorSetTags` function follows.

The size of the opcode specifies the number of tags; the compression specifies whether the data is in bytes, words, or longs. For instance, if the size is 4 and the compression is `gxShortCompression` (2 bytes), then the stream contains 4/2 == 2 tags. The equivalent operation would be `GXSetShapeTags` (shape, nil, 1, 0, 2, tag array).

The `GXSetColorSetTags` function is described in the chapter “Color Objects” of *Inside Macintosh: QuickDraw GX Objects.*

New Tag Object Data

Data for a new tag object follows a `gxNewObjectOpcode` operation opcode and a data type opcode with a value 0x2D. This is the `gxTagTypeOpcode` constant from the `gxGraphicsNewOpcode` enumeration.

This opcode creates a new tag object with all of the properties of the previous tag object in the data stream. If the current tag object is the first tag object in the stream, then it is created with default properties. For tag data is uncompressed. The first parameter is tag type (long), followed by data computed from opcode length - sizeof (long).

The `GXNewtag` function is described in the chapter “Tag Objects” in *Inside Macintosh: QuickDraw GX Objects.*

New Bit Image Object Data

Data for a bit image object follows a `gxNewObjectOpcode` operation opcode and a data type opcode with a value 0x2E. This is the `gxBitImageTypeOpcode` constant from the `gxGraphicsNewOpcode` enumeration.

The data sequence is omit byte (gxOmitBitImage), followed by the fields described by omit byte: `rowBytes`, compressed as long, `height`, compressed as long, and `data` compressed in the custom format described ahead. The bit image is compressed only if it makes the data stream smaller.

The `GXNewBitmap` function is described in the chapter “Bitmap Shapes” in *Inside Macintosh: QuickDraw GX Graphics.*

The bit image compression byte appears only in data streams containing a bitmap shape. This byte describes how each section of a bit image is compressed. The bit image compression byte follows the bytes containing the bit image attributes described by the `gxOmitBitImageMask` constant.

Bit images are described in the “Bitmap Shapes” chapter of *Inside Macintosh: QuickDraw GX Graphics.*

The bit image compression byte has the format *xx yyyyyy.*

The *xx* bits describe which of the bit image compression type opcodes is used for the next part of the bit image. The bit image compression opcode values are either 0, 1, 2, or 3.

The *yyyyyy* bits describe the number of times, z, that the action defined by the bit image compression opcode is replicated. The number of replications, *z*, can vary range from 0 to 63. Table 7-13 summarizes the four compression opcodes.

Table 7-13 Bit image compression opcodes

Bit image compression opcode	Bit image compression description
0	Add the *z* bytes of bit image that follow to the current row. *Z* Bytes of data follow.
1	Repeat 1 byte *z* times and add the bits to the current row. One byte of data follows.
2	Copy *z* bytes of the previous row and add the bits to the current row. No data follows.
3	Copy the previous row of bits *z* times and add the bits to the next z rows. No data follows.

The analysis of a bit image compression byte in a stream format is described in the section "Analyzing a Flattened Bitmap Shape" beginning on page 7-81.

New Font Name Data

Data for a font name follows a `gxNewObjectOpcode` operation opcode and a data type opcode with a value 0x2F. This is the `gxFontNameTypeOpcode` constant from the `gxGraphicsNewOpcode` enumeration.

The fields in the `gxFlatFontName` structure follow. This structure includes the `gxFontName`, `gxFontPlatform`, `gxFontScript`, `gxFontLanguage`, and `gxFontName` structures, the byte length of the name and the name itself.

The stream exactly mirrors the sequence and size of the fields in the `gxFlatFontName` structure.

New Trailer Object Data

Data for a trailer object follows a `gxNewObjectOpcode` operation opcode and a data type opcode with a value 0x3F. This is the `gxTrailerTypeOpcode` constant from the `gxGraphicsNewOpcode` enumeration. This is the termination (last) object in the stream. No data follows.

The last two bytes of a stream are always 0x01 and 0x3F. The next to the last byte in a data stream contains a `gxNewObjectOpcode` constant with a record size of 1 byte. The last byte in a data stream contains a `gxTrailerTypeOpcode` constant with a `gxTwoBitCompression` value of 0, indicating the `gxNoCompression` constant.

About Print Files and Portable Digital Documents

QuickDraw GX printing performs background printing to all devices, allowing users continued access to the application. The printing process includes the creation of a specialized print file called a portable digital document.

Print Files

When an application prints, QuickDraw GX collects the printing information sent by the application and writes it to a file. This process is called *spooling* and the file that is created is called a **print file.** QuickDraw GX then reads the print file and prints it to the appropriate device. The read and interpretation process is called *despooling* and the printing process is called *imaging*.

A print file can be duplicated, dragged onto desktop printers, manipulated by print queues, and redirected to other printer devices without re-spooling. Print files also provide a device-independent information interchange format.

The QuickDraw GX spooling process consists of creating a print file and writing a stream of flattened shape data to that file. This data is unflattened during the unspooling process. Additional information must be provided in the print files. This includes job, formatting, and optimization information.

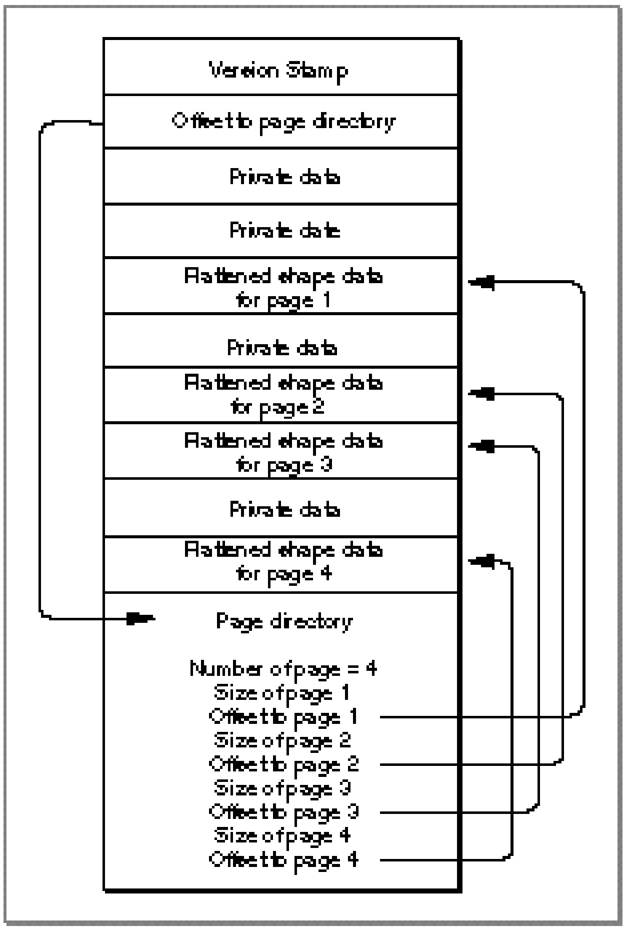
The job-related information includes the name of the job, the destination device, quality, and the number of copies. The formatting information includes the page sizes and orientations. The optimization information includes the font database.

The print file consists of two forks, a data fork and a resource fork. The data fork contains all the core data necessary to print a document. This consists of the flattened job data, the flattened shape data for each page, and the flattened format data for each page.

The print file begins with a 32-bit QuickDraw GX version followed by a 32-bit offset that describes the number of bytes from the beginning of the file to the start of the page directory located at the end of the file.

The page directory contains a 32-bit number indicating the number of pages in the document, an array of page sizes, and offsets to the start of the flattened shape data for each page. The format of a print file for a four-page document is shown in Figure 7-11.

Figure 7-11 Print file format



QuickDraw Picture Data in Print Files

When creating a print file from a document that contains QuickDraw drawing commands, QuickDraw GX by default saves the QuickDraw data for each page in a tag object of tag type `'pict'` attached to a rectangle shape. Therefore, if you are examining the data stream of a print file, you should note that a rectangle shape with an attached tag object of type `'pict'` indicates the presence of QuickDraw data. For more information about this tag object and QuickDraw data, see the discussion of the `'pict'` tag object in the advanced printing features chapter of *Inside Macintosh: QuickDraw GX Printing.* u

Portable Digital Documents

QuickDraw GX provides document portability that is independent of fonts, applications, and output devices. The users of your application can create and save their results in the form of a **portable digital document** or **PDD.**

A portable digital document consists of the print file containing flattened shapes described in the previous section. These files provide all of the information necessary to view and print the document, including the fonts that are used and other information necessary to render the text and graphics. A portable digital document can be sent to other Macintosh users and viewed or printed simply by opening the documents with a viewer that can interpret them.

For more information on print files and portable digital documents, see the chapters “Introduction to QuickDraw GX Printing” and “Core Printing Features” of *Inside Macintosh: QuickDraw GX Printing.*

Using QuickDraw GX Stream Format

This section describes the use of the GraphicsBug utility to analyze flattened data streams. Sample code is provided that draws a QuickDraw GX picture containing seven shapes. GraphicsBug is used to flatten each shape. The resulting data stream for each flattened shape is then analyzed.

This section describes how you can

- flatten shapes using GraphicsBug
- interpret the GraphicsBug flattened shape output format
- analyze flattened shape data streams

Flattening Shapes With GraphicsBug

GraphicsBug is not just a QuickDraw GX debugging tool. It also allows you to evaluate the data at specific memory locations. You can use GraphicsBug to look at the data describing a QuickDraw GX shape both before and after you invoke the `GXFlattenShape` function. This allows you to compare the original data and the stream format after the `GXFlattenShape` function has been called.

For more information concerning GraphicsBug, see the chapter "QuickDraw GX Debugging " in this book.

You can use GraphicsBug to analyze a data stream by using the following procedure:

1. Create a QuickDraw GX shape.
2. Use the GraphicsBug heap dump `HD` command to determine the memory location of the QuickDraw GX shape to be flattened.
3. Copy the memory location of the shape to the clipboard.
4. Type `FL` and paste the memory address. The command line should look like this:

```
fl <memory address>
```

For example: `fl 41d788`

5. The command `FL` applies `the GXFlattenShape` function to the shape located at the specified memory address. This results in a flattened shape. An annotated version of the QuickDraw GX data stream appears in the GraphicsBug window. GraphicsBug does not alter the graphics memory in any way.

To create a flattened file, you can use the command

```
fl <memory address> "filename"
```

To view the contents of a file, such as a print file generated by printing a document, you can use the command

```
uf "filename"
```

To view the stream associated with a particular page of a document, you can use the command

```
uf <page number> "filename"
```

Here are some guidelines for using GraphicsBug to analyze data streams:

- The data in parentheses in the GraphicsBug window are the compressed byte codes that were generated when the original shape was flattened. The data not in parenthesis is GraphicsBug's brief annotation of the data stream. The annotation usually describes the shape data in its original format. The data in parentheses always relates to the immediately previous data that is not in parentheses.
- Sometimes GraphicsBug will not give the name of the font. This is because GraphicsBug reads only the information contained in memory. GraphicsBug cannot make a call to get the information. If GraphicsBug is used to flatten shapes that were generated by a client call, the required data will always already be in memory and will therefore be available. In this case, the GraphicsBug annotation will always provide the name of the font.
- If part of an object is compressed and another part of the object is not compressed, GraphicsBug reports that there is "no compression."
- Bracketed numbers are references. When `gxSetData` or `gxSetReference` opcodes are encountered, they can't generate pointers to other objects. They have to generate references. The first object is given reference 1. Subsequent objects are given references 2, 3, and so on.

Listing 7-2 shows an example of the information provided by GraphicsBug for a flattened line.

Listing 7-2 A GraphicsBug annotation of the data stream of a flattened shape

```
fl 0c79090
owners           1)
newObject; size: #2 (03)
headerType; byte compression (80)
version == 1.0; flags == fontListFlatten | fontGlyphsFlatten
(01 03)
newObject; size: #6 (07) [1]
fontNameType; no compression (2f)
(04 02 01 01 00 00)
```

Listing 7-2 shows only the beginning of a data stream. For more examples of GraphicsBug annotation of flattened shape data streams, see the next section.

Analyzing the Data Streams of Flattened Shapes

This section first uses sample code to generate a picture with seven shapes. Each of the seven shapes is then flattened using the procedure described in the section “Flattening Shapes With GraphicsBug” beginning on page 7-54. The section “Analyzing the Data Streams of Flattened Shapes” beginning on page 7-56 describes how to use GraphicsBug to interpret the data for each of the seven shapes. The GraphicsBug data stream output is provided for each flattened shape in Listing 7-4 through Listing 7-10. The byte-by-byte analysis of the data stream for each flattened shape is provided in Table 7-14 through Table 7-20.

Creating a Picture With Seven Shapes

Listing 7-3 creates seven primitive shapes and adds them to a window’s page shape to form the picture shown in Figure 7-12. This picture contains (from left to right and top to bottom) a line, rectangle, curve, path, text, polygon and bitmap shape.

Listing 7-3 A picture with seven shapes

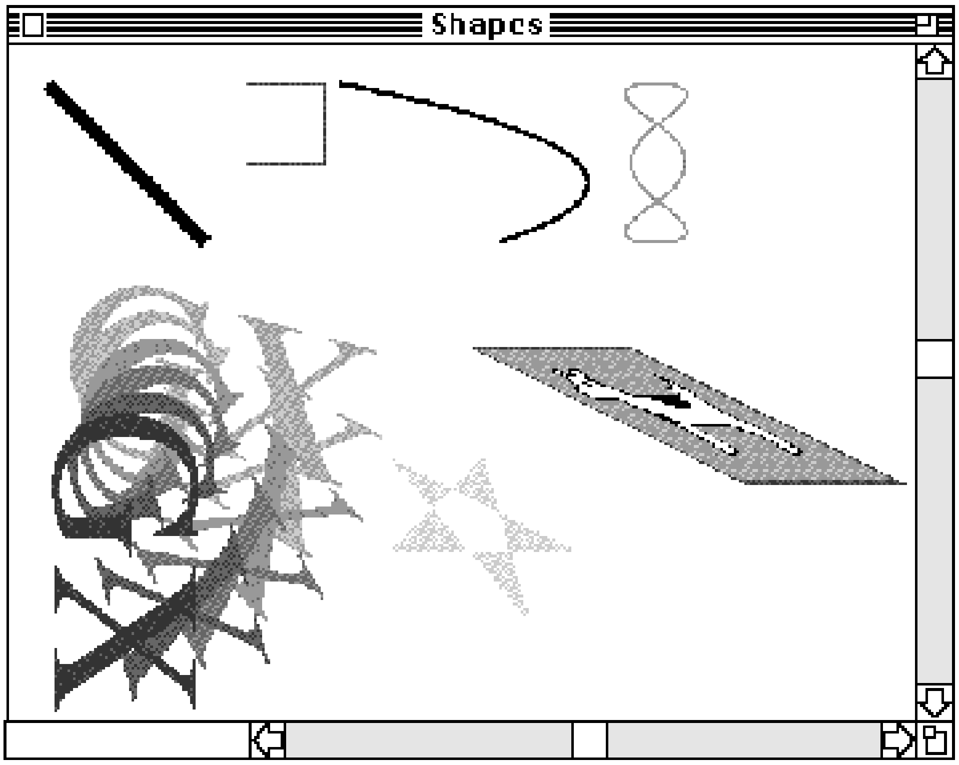
```
void CreateSampleImage(WindowPtr wind)
{
gxShape thePage;
gxShape theLine;
line lineData = {ff(25), ff(25), ff(125), ff(125)};
gxShape theRect;
gxRectangle rectData = {ff(25), ff(25), ff(75), ff(75)};
gxShape theCurve;
gxCurve curveData = {ff(25), ff(25), ff(275), ff(75), ff(125),
ff(125)};
gxShape thePath;
long tripleEightData[] = {1/* # of contours */, 6 /* # of points
*/, 0xff000000,
   0, 0,
   ff(75),  0,
   ff(5), ff(50),
   ff(75), ff(100),
   0, ff(100),
   ff(75), ff(50)};
gxShape theText;
gxRectangle theTextBounds;
gxColor textColor;
fixed x,y;
short loop;
gxShape thePolygon;
```

```
long starData[] = {1,  /* number of contours */ 5, /* number of
points */
   ff(60), 0, ff(90), ff(90),  ff(0), ff(30),  ff(120), ff(30),
   ff(0), ff(90)}; /* the points */
   gxShape theBitmap;

/* retrieve the page shape so we can add to it */
thePage = GetDocShape(wind);

/* Create a line shape*/

   theLine = GXNewLine (&lineData);
   GXSetShapePen(theLine, ff(9));
   GXAddToShape(thePage, theLine);
   GXDisposeShape(theLine);

/* create a rectangle; the color of the rectangle is red */

theRect = GXNewRectangle(&rectData);
   {gxColor redColor =
      {gxRGBSpace, nil,{
         0xFFFF,0,0}};
GXSetShapeColor(theRect, &redColor);
   }
GXSetShapeFill (theRect, closedFrameFill);
GXMoveShapeTo (theRect,  ff(150), ff(25));
GXAddToShape(thePage, theRect);
GXDisposeShape(theRect);

/* create a curve shape; the shape has a pen thickness of 3.25 */

theCurve = GXNewCurve(&curveData);
GXSetShapePen(theCurve, fl(3.25));
GXMoveShapeTo (theCurve,  ff(210), ff(25));
GXAddToShape(thePage, theCurve);
GXDisposeShape(theCurve);

/* create a path shape; the shape’s color is green and the pen
thickness is 2 */

thePath = GXNewPaths((paths *) tripleEightData);
GXSetShapeFill (thePath, closedFrameFill);
   GXSetShapePen(thePath, ff(2));
GXSetShapeCommonColor (thePath, green);
```

```
GXMoveShapeTo (thePath,  ff(390), ff(25));
GXAddToShape(thePage, thePath);
GXDisposeShape(thePath);

/* create a text shape; the shape is the characters GX colored
in hsv space and rotated 90 degrees */

/* create the text, set the font size, and set the font name */

theText = NewText(2,(unsigned char*)"GX",  nil);
GXSetShapeCommonFont(theText, timesFont);
GXSetShapeTextSize(theText, ff(135));
GXMoveShapeTo (theText,  ff(25), ff(230));
GXSetShapeAttributes (theText,  gxMapTransformShape);

/* create an hsv color space and set up the initial colors */

textColor.space = hsvSpace;
textColor.profile = nil;
textColor.element.hsv.hue = 0x7400;
textColor.element.hsv.saturation = 0xFFFF;
textColor.element.hsv.value = 0xFFFF;

/* get the bounds of "theText" and determine the coordinates of
the bottom left corner */
GXGetShapeBounds(theText, 0L, &theTextBounds);
x = theTextBounds.left;
y = theTextBounds.bottom;

/* rotate "theText"; add each letter to the picture */
for (loop = 0; loop < 6; loop++) {
   GXSetShapeColor(theText, &textColor);
   GXRotateShape(theText, ff(90), x, y);
   GXAddToShape(thePage, theText);
   textColor.element.hsv.hue += 0x0940;
}
GXDisposeShape(theText);

/* create a polygon shape; the shape’s color is yellow, the pen
size is 3, and it is skewed in the vertical direction by a
factor of 0.5 */

thePolygon = GXNewPolygons((gxPolygons *) starData);
GXSetShapeFill(thePolygon, gxEvenOddFill);
```

```
GXSetShapePen (thePolygon, ff(3));
GXSetShapeCommonColor (thePolygon, yellow);
GXMoveShapeTo (thePolygon,  ff(240), ff(110));
GXSkewShape(thePolygon, 0, fl(0.5), 0, 0);

GXAddToShape(thePage, thePolygon);
GXDisposeShape(thePolygon);

/* create a bitmap by retrieving a bitmap from the resource fork
and skewing it in the horizontal direction by a factor of .*/

   theBitmap = GXGetPixMapShape(128);
   GXValidateShape (theBitmap);

   GXSkewShape(theBitmap, ff(2), 0, 0, 0);
   GXMoveShapeTo (theBitmap,  ff(290), ff(190));

   GXAddToShape(thePage, theBitmap);
   GXDisposeShape(theBitmap);
```

Figure 7-12 A picture with seven shapes

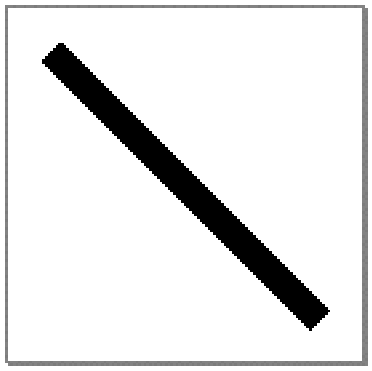


Analyzing a Flattened Line Shape

The function described in the section “Creating a Picture With Seven Shapes” beginning on page 7-56 was first used to draw the picture shown in Figure 7-12 containing the line shape shown in Figure 7-13.

The line shape is created with a pen size of 9 and a default color of black. The pen is moved from the point (25.0, 25.0) to point (125.0, 125.0).

Figure 7-13 The line shape drawn



The procedure described in the section “Flattening Shapes With GraphicsBug” beginning on page 7-54 was then used to generate the GraphicsBug output shown in Listing 7-4. The first line of the output shows the use of the `fl` command on the memory address that contained the line shape. The flattened line shape data stream is the sequential byte data that appears in parentheses. For example, the first four bytes of the data stream in Listing 7-4 are (06) (80) (01 03). All other annotation is provided by GraphicsBug.

Since the flattened line shape is the first shape in the data stream, this first part of the GraphicsBug output shows the data stream header. The GraphicsBug output for the other flattened shapes described in this section correspond to the data stream that describes that specific shape. These shape-specific sections are presented in QuickDraw GX drawing order.

Listing 7-4 GraphicsBug analysis of a flattened line

```
fl 0c79090
owners            1)
newObject; size: #2 (03)
headerType; byte compression (80)
version == 1.0; flags == fontListFlatten | fontGlyphsFlatten
(01 03)
newObject; size: #6 (07) [1]
fontNameType; no compression (2f)
```

```
(04 02 01 01 00 00)
newObject; size: #0 (01) [1]
styleType; no compression (28)
setData; size: #1 (42)
stylePen; byte compression (83)
(09)
newObject; size: #0 (01) [1]
inkType; no compression (29)
newObject; size: #0 (01) [1]
transformType; no compression (2a)
newObject; size: #4 (05)
lineType; byte compression (83)
(19 19 7d 7d)
newObject; size: #0 (01)
trailerType; no compression (3f)
```

Table 7-14 shows the data stream analysis of the flattened line shape. The stream data is obtained from the GraphicsBug output in Listing 7-4. This table provides a description of each byte of the data stream for this shape.

Table 7-14 Analysis of the data stream of a flattened line shape

Values in data stream (binary)	**Type of information**	**Value**	**Description**
New header			
`0x03 (00 000011)`	Operation opcode	0	New object
	Record size	3	Record size is 3 bytes
`0x80 (10 000000)`	Compression type opcode	2	Byte compression
	Data type opcode	0	Header
`0x01 (00000001)`	Data	1.0	QuickDraw GX Version 1.0
`0x03 (00000011)`	Data	3	`gxFontListFlatten` constant from the `gxFlattenFlags` enumeration is 0x01
			`gxFontGlyphsFlatten` constant from the `gxFlattenFlags` enumeration is 0x02

continued

Table 7-14 Analysis of the data stream of a flattened line shape (continued)

Values in data stream (binary)	Type of information	Value	Description
New font name for the style object			
`0x07` `(00 000111)`	Operation opcode	0	New object
	Record size	7	Record size is 7 bytes
`0x2F` `(00 101111)`	Compression type opcode	2	No compression
	Data type opcode	0x2F	Font name
`0x04`	Data	4	The `gxUniqueFontName` constant of the `gxFontName` enumeration
`0x02`	Data	2	The `gxMacintoshPlatform` constant of the `gxFontPlatform` enumeration
`0x01`	Data	1	The `gxMacintoshRomanScript` constant of the `gxMacintoshScripts` enumeration
`0x01`	Data	1	The `gxEnglishLanguage` constant of the `gxFontLanguage` enumeration
`0x0001A`	Data	26	The `length` field (short) of the `gxFontName` is 26 bytes.
0x41 70 70 6C 65 20 43 6F 6D 70 75 74 65 72 20 54 69 6D 65 73 20 52 6F 6D 61 6E			
	Data		Each of the 26 bytes is one glyph code. The font name is “Apple Computer Times ROman.”
New style object			
`0x01` `(00 000001)`	Operation opcode	0	New object
	Record size	1	Record size is 1 byte
`0x28` `(00 101000)`	Compression type opcode	2	No compression
	Data type opcode	0x28	New style
`0x42` `(01 000010)`	Operation opcode	1	Set data
	Record size	1	Record size is 1 byte

Table 7-14 Analysis of the data stream of a flattened line shape (continued)

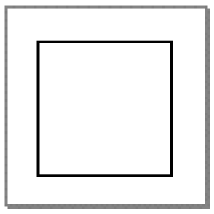
Values in data stream (binary)	Type of information	Value	Description
0x83 (10 000011)	Compression type opcode	2	No compression
	Data type opcode	3	`gxStylePenOpcode` constant of the `gxStyleDataOpcode` enumeration
0x09	Data	9.0	The pen width parameter for the `GXSetShapePen` function is 9.0
New ink object			
0x01 (00 000001)	Operation opcode	0	New object
	Record size	1	Record size is 1 byte
0x29 (00 101001)	Compression type opcode	0	No compression
	Data type opcode	0x29	New ink
New transform			
0x01 (00 000001)	Operation opcode	0	New object
	Record size	1	Record size is 1 byte
0x2A (00 101001)	Compression type opcode	0	No compression
	Data type opcode	0x2A	New transform
New shape object			
0x05 (00 000101)	Operation opcode	0	New object
	Record size	5	Record size is 5 bytes
0x83 (10 000011)	Compression type opcode	2	Byte compression
	Data type opcode	3	`gxLineType` constant of the `gxShapeTypes` enumeration
0x19	Data	25.0	x coordinate of the first point is 25.0
0x19	Data	25.0	y coordinate of the first point is 25.0
0x7D	Data	125.0	x coordinate of the last point is 125.0
0x7D	Data	125.0	y coordinate of the last point is 125.0

Analyzing a Flattened Rectangle Shape

The function described in section “Creating a Picture With Seven Shapes” beginning on page 7-56 was first used to draw the picture shown in Figure 7-12 containing the rectangle shape shown in Figure 7-14.

The red rectangle shape is created with its frame. The size and shape of the rectangle is defined by its upper-left boundary point (25.0, 25.0) and its lower-right boundary point (75.0, 75.0). The fill type is closed-frame. Once the rectangle is drawn, it is moved to the point (150.0, 25.0) to position it in the picture.

Figure 7-14 The rectangle shape drawn



The procedure described in the section “Flattening Shapes With GraphicsBug” beginning on page 7-54 was then used to generate the GraphicsBug output shown in Listing 7-5. The flattened rectangle shape data stream is the sequential data that appears in parentheses.

Listing 7-5 GraphicsBug analysis of a flattened rectangle shape

```
inkType; no compression (29)
  space        gxRGBSpace
  profile           nil
  value(s)       1.0000 (ffff)  0.0000 0x0000  0.0000 0x0000
setData; size: #4 (45)
inkColor; no compression (02)
(fe ff 00 00)
newObject; size: #8 (09)
rectangleType; word compression (45)
(00 96 00 19 00 c8 00 4b)
setData; size: #1 (42)
shapeFill; byte compression (82)
(02)
```

Table 7-15 shows the data stream analysis of the flattened rectangle shape. The stream data is obtained from the GraphicsBug output in Listing 7-5. This table provides a description of each byte of the data stream for this shape. Data format sequences that are identical to previously described data sequences in the stream are not shown..

Table 7-15 Analysis of the data stream of a flattened rectangle shape

Values in data stream (binary)	Type of information	Value	Description
New ink object			
`0x01` `(00 000001)`	Operation opcode	0	New object
	Record size	1	Record size is 1 byte.
`0x29` `(00 101001)`	Compression type opcode	0	No compression
	Data type opcode	0x29	New ink
Set data for ink color			
`0x45` `(01 000101)`	Operation opcode	1	Set data
	Record size	5	Record size is 5 bytes.
`0x02` `(00 000010)`	Compression type opcode	0	No compression
	Data type opcode	2	`gxInkColorOpcode` constant of the `gxInkDataOpcode` enumeration
`0xFE` `(11 11 1110)`	Omit byte	–	The `gxOmitColorsMask` and `gxOmitColorsShift` enumerations are used to interpret this byte. Data1, color space, is omitted so the default RGB color space properties are applied to the current object. Data2, color profile, is omitted so the default color profile is applied to the current object. Data3, color components, uses only bits 3, 2, and 1 for RGB. The compression for each of the red, green, and blue color components is byte compression.
`0xFF`	Data	0xFFFF	Since color components are 2-byte values, the byte is replicated to the value 0xFFFF or 65,535. The RGB value for the `red` field of the `gxRgbColor` structure is 65,535.

continued

Table 7-15 Analysis of the data stream of a flattened rectangle shape (continued)

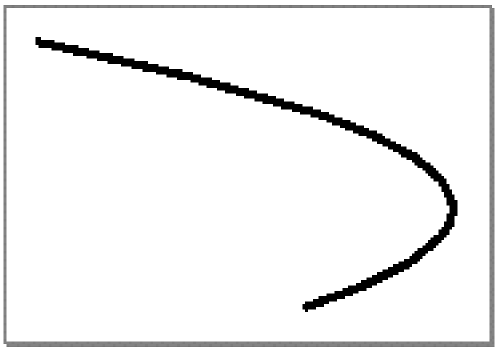
Values in data stream (binary)	Type of information	Value	Description
`0x00`	Data	0x0000	Since color components are 2-byte values, the byte is replicated to the value 0x0000 or 0. The RGB value for the `green` field of the `gxRgbColor` structure is 0.
`0x00`	Data	0x0000	Since color components are 2-byte values, the byte is replicated to the value 0x0000 or 0. The RGB value for the `blue` field of the `gxRgbColor` structure is 0.
New rectangle object			
`0x09` `(00 001001)`	Operation opcode	0	New object
	Record size	5	Record size is 9 bytes.
`0x45` `(01 000101)`	Compression type opcode	1	Word compression
	Data type opcode	5	`gxRectangleType` constant of the `gxShapeTypes` enumeration
`0x00 96`	Data	150.0	x-coordinate of the left top corner point is 150.0
`0x00 19`	Data	25.0	y-coordinate of the left top corner point is 25.0
`0x00 C8`	Data	200.0	x-coordinate of the right bottom corner point is 200.0
`0x00 4B`	Data	125.0	y-coordinate of the right bottom corner point is 75.0
Set data for shape fill			
`0x42` `(01 000010)`	Operation opcode	1	Set data
	Record size	2	Record size is 2 bytes.
`0x82` `(10 000010)`	Compression type opcode	2	Byte compression
	Data type opcode	2	`gxShapeFillOpcode` constant of the `gxShapeDataOpcode` enumeration
`0x02`	Data	2	`gxClosedFrameFill` constant of the `gxShapeFills` enumeration. The shape fill constant is a long number so the byte is expanded to a long.

Analyzing a Flattened Curve Shape

The function described in the section “Creating a Picture With Seven Shapes” beginning on page 7-56 was first used to draw the picture shown in Figure 7-12 containing the curve shape shown in Figure 7-15.

The curve has a pen thickness of 3.25. The size and shape of the curve are defined by its first point (210.0), control point (460.0, 75.0), and last point (310.0, 125.0). Once the curve is drawn, it is moved to the point (210.0, 25.0) to position it in the picture.

Figure 7-15 The curve shape drawn



The procedure described in the section “Flattening Shapes With GraphicsBug” beginning on page 7-54 was then used to generate the GraphicsBug output shown in Listing 7-6. The flattened curve shape data stream is the sequential data that appears in parentheses.

Listing 7-6 GraphicsBug analysis of a flattened curve shape

```
.
.
.
newObject; size: #6 (07) [1]
fontNameType; no compression (2f)
(04 02 01 01 00 00)
newObject; size: #0 (01) [1]
styleType; no compression (28)
setData; size: #4 (45)
stylePen; no compression (03)
(00 03 40 00)
.
.
.
```

```
newObject; size: #12 (0d)
curveType; word compression (44)
(00 d2 00 19 01 cc 00 4b 01 36 00 7d)
newObject; size: #0 (01)
trailerType; no compression (3f)
```

Table 7-16 shows the data stream analysis of the flattened rectangle shape. The stream data is obtained from the GraphicsBug output in Listing 7-6. This table provides a description of each byte of the data stream for this shape. Data format sequences that are identical to previously described data sequences in the stream are not shown and are not analyzed here.

Table 7-16 Analysis of the data stream of a flattened curve shape

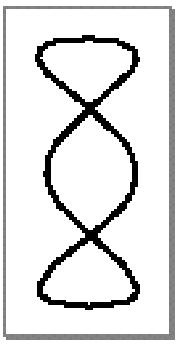
Values in data stream (binary)	Type of information	Value	Description
`0x45 (01 000101)`	Operation opcode	1	Set data.
	Record size	5	Record size is 5 bytes.
`0x03 (00 000011)`	Compression type opcode	0	No compression
	Data type opcode	3	`gxStylePenOpcode` constant of the `gxStyleDataOpcode` enumeration
`0x00034000`	Data	3.25	The pen width parameter for the `GXSetPen` function is 3.25.
.			
.			
.			
`0x0D (00 01101)`	Operation opcode	0	New object
	Record size	13	Record size is 13 bytes.
`0x44 (01 000100)`	Compression type opcode	1	Word compression
	Data type opcode	4	`gxCurveType` constant of the `gxShapeTypes` enumeration
`0x00 D2`	Data	210.0	x-coordinate of the first point is 210.0.
`0x00 19`	Data	25.0	y-coordinate of the first point is 25.0.
`0x00 CC`	Data	460.0	x-coordinate of the control point is 460.0.
`0x00 4B`	Data	75.0	y-coordinate of the control point is 75.0.
`0x00 36`	Data	310.0	x-coordinate of the last point is 310.0.
`0x00 7D`	Data	125.0	x-coordinate of the last point is 125.0.

Analyzing a Flattened Path Shape

The function described in the section “Creating a Picture With Seven Shapes” beginning on page 7-56 was first used to draw the picture shown in Figure 7-12 containing the path shape shown in Figure 7-16.

A path is created with a pen thickness of 2.0 and a color of green. The size and shape of the curve are defined by the points (0.0, 0.0), (75.0, 0.0), (5.0, 50.0), (75.0, 100.0), (0.0, 100.0), and (75.0, 50.0). Once the path is drawn, it is moved to the point (290.0, 25.0) to position it in the picture. The line is not on any of the points.

Figure 7-16 The path shape drawn



The procedure described in the section “Flattening Shapes With GraphicsBug” beginning on page 7-54 was then used to generate the GraphicsBug output shown in Listing 7-7. The flattened path shape data stream is the sequential data that appears in parentheses.

Listing 7-7 GraphicsBug analysis of a flattened path shape

```
newObject; size: #0 (01) [1]
transformType; no compression (2a)
newObject; size: #19 (14)
pathType; byte compression (87)
(01 06 ff 2a 01 73 40 00 19 b5 00 46 ce ba ce 4b 00 b5 32)
setData; size: #1 (42)
shapeFill; byte compression (82)
(02)
```

Table 7-17 shows the data stream analysis of the flattened path shape. The stream data is obtained from the GraphicsBug output in Listing 7-7. This table provides a description of each byte of the data stream for this shape. Data format sequences that are identical to previously described data sequences in the stream are not shown and are not analyzed here.

Table 7-17 Analysis of the data stream of a flattened path shape

Values in data stream (binary)	Type of information	Value	Description
New path object			
`0x14` `(00 010100)`	Operation opcode	0	New object
	Record size	14	Record size is 14 bytes.
`0x87` `(10 000111)`	Compression type opcode	2	Byte compression
	Data type opcode	7	`gxPathType` constant of the `gxShapeTypes` enumeration
`0x01`	Data	1	The number of contours is 1.
`0x06`	Data	6	The number of points in the contour is 6.
`0xFF` `(11111111)`	Control byte	–	Each of the 6 points is assigned a control bit from the control byte. Points having a 0 bit are on the line. Points having a 1 bit are off the line. All 6 points are off the line. The final 2 bits are unused.
`0x2A` `(00 10 10 10)`	Omit byte	–	The `gxOmitPathMask` and `gxOmitPathShift` enumerations are used to interpret this byte. No compression is used for data1, x coordinate of first point. Byte compression is used for data2, y coordinate of first point. Byte compression is used for data3, all x relative coordinate deltas. Byte compression is used for data4, all y relative coordinate deltas.
`0x01734000`	Data1	371.25	Absolute x-coordinate of the first point is 371.25.

Table 7-17 Analysis of the data stream of a flattened path shape (continued)

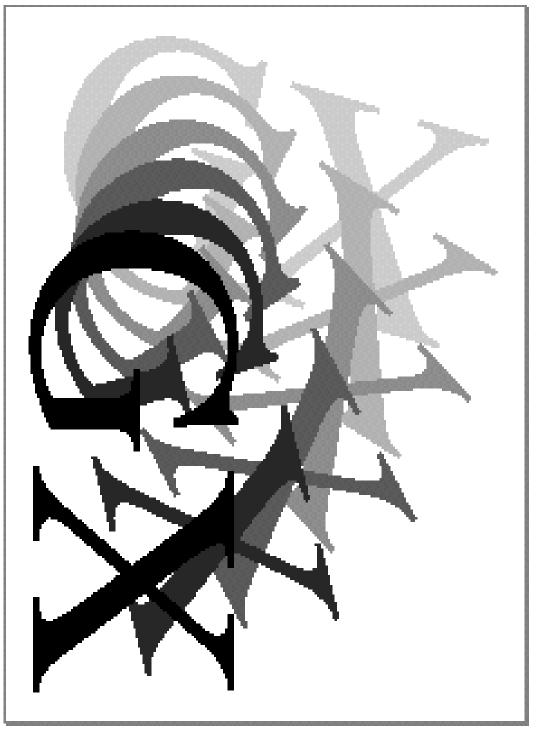
Values in data stream (binary)	Type of information	Value	Description
`0x0x19`	Data2	25.0	Absolute y-coordinate of the first point is 25.0.
`0xB5`	Data3	-75.0	Relative x-coordinate of the second point is -75.0. Absolute x coordinate is 371.25 – (–75.0) = 446.25.
`0x00`	Data4	0.0	Relative y-coordinate of the second point is 0. Absolute y coordinate is 25.0 – (0.0) = 25.0.
`0x46`	Data3	70.0	Relative x-coordinate of the third point is 70.0. Absolute x coordinate is 371.25 – (70.0) = 301.25.
`0xCE`	Data4	–50.0	Relative y-coordinate of the third point is –50.0. Absolute y coordinate is 25.0 – (–50.0) = 75.0.
`0xBA`	Data3	–70.0	Relative x coordinate of the fourth point is -70.0. Absolute x-coordinate is 371.25 – (–70.0) = 441.25.
`0xCE`	Data4	–50.0	Relative y coordinate of the fourth point –50.0. Absolute y-coordinate is 25.0 – (–50.0) = 75.0.
`0x4B`	Data3	75.0	Relative x coordinate of the fifth point is 75.0. Absolute x-coordinate is 371.25 – (75.0) = 296.25.
`0x00`	Data4	0.0	Relative y coordinate of the fifth point is 0.0. Absolute y-coordinate is 25.0 – (0.0) = 25.0.
`0xB5`	Data3	–75.0	Relative x coordinate of the sixth point is –75.0. Absolute x-coordinate is 371.25 – (–75.0) = 446.25.
`0x32`	Data4	50.0	Relative y coordinate of the sixth point is 50.0. Absolute y-coordinate is 25.0 – (50.0) = –25.0.

Analyzing a Flattened Text Shape

The function described in the section “Creating a Picture With Seven Shapes” beginning on page 7-56 was first used to draw the picture shown in Figure 7-12 containing the path shape shown in Figure 7-17.

A text shape with glyphs G and X is colored in hsv space. The glyphs are rotated six times by 90 degrees about the left bottom corner. Once the text is drawn, it is moved to the point (25.0, 230.0) to position it in the picture.

Figure 7-17 The text shape drawn



The procedure described in the section “Flattening Shapes With GraphicsBug” beginning on page 7-54 was then used to generate the GraphicsBug output shown in Listing 7-8. The flattened text shape data stream is the sequential data that appears in parentheses.

Listing 7-8 GraphicsBug analysis of a flattened text shape

```
newObject; size: #32 (21) [1]
fontNameType; no compression (2f)
(04 02 01 01 00 1a)
Apple Computer Times Roman
(41 70 70 6c 65 20 43 6f 6d 70 75 74 65 72 20 54 69 6d 65 73 20 52
6f 6d 61 6e)
newObject; size: #0 (01) [1]
styleType; no compression (28)
setData; size: #1 (42)
styleFont; byte compression (8a)
(01)
setData; size: #2 (43)
styleTextSize; word compression (49)
(00 87)
newObject; size: #0 (01) [1]
inkType; no compression (29)
  space        hsvSpace
  profile           nil
  value(s)       0.4531 0x7400  1.0000 (ffff)  1.0000 (ffff)
setData; size: #6 (47)
inkColor; no compression (02)
(b6 03 74 00 ff ff)
newObject; size: #0 (01) [1]
transformType; no compression (2a)
setData; size: #24 (59)
transformMapping; no compression (03)
(00 3d 02 12 00 00 98 fe 00 00 f7 47 00 00 f7 47 00 00 42 42 ff ff
bd be)
newObject; size: #8 (09)
textType; no compression (09)
  byteLength        2
  position                 {    25.0000,   230.0000}
Displaying memory from 00c7a116
 00c7a116  4758                                    GX
(a4)
bytes (02)
position.x (19)
position.y (00 e6 02 47 58)
setData; size: #1 (42)
shapeAttributes; byte compression (80)
(20)
```

Table 7-18 shows the data stream analysis of the flattened rectangle shape. The stream data is obtained from the GraphicsBug output in Listing 7-8. This table provides a description of each byte of the data stream for this shape. Data format sequences that are identical to previously described data sequences in the stream are not shown and are not analyzed here.

Table 7-18 Analysis of the data stream of a flattened text shape

Values in data stream (binary)	Type of information	Value	Description
New font name for the style object			
`0x21` `(00 100001)`	Operation opcode	0	New object
	Record size	21	Record size is 21 bytes
`0x2F` `(0010111)`	Compression type opcode	0	No compression
	Data type opcode	7	`gxFontNameOpcode` constant of the `gxGraphicsNewOpcode` enumeration
`0x04`	Data	4	The `gxUniqueFontName` constant of the `gxFontName` enumeration
`0x02`	Data	2	The `gxMacintoshPlatform` constant of the `gxFontPlatform` enumeration
`0x01`	Data	1	The `gxMacintoshRomanScript` constant of the `gxMacintoshScripts` enumeration
`0x01`	Data	1	The `gxEnglishLanguage` constant of the `gxFontLanguage` enumeration
`0x0001A`	Data	26	The `length` field (short) of the `gxFontName` structure is 26 bytes.
`0x41 70 70 6C 65 20 43 6F 6D 70 75 74 65 72 20 54 69 6D 65 73 20 52 6F 6D 61 6E`	Data		Each of the 26 bytes is one glyph code. The font name is “Apple Computer Times Roman.”

Table 7-18 Analysis of the data stream of a flattened text shape (continued)

Values in data stream (binary)	Type of information	Value	Description
New style object			
0x01 (00 000001)	Operation opcode	0	New object
	Record size	1	Record size is 1 byte
0x28 (00 101000)	Compression type opcode	0	No compression
	Data type opcode	0x28	`gxStyleTypeOpcode` constant of the `gxGraphicsNewOpcode` enumeration
Set data for style object			
0x42 (01 000010)	Operation opcode	1	Set data.
	Record size	2	Record size is 2 bytes.
0x8A (10 001010)	Compression type opcode	2	Byte compression
	Data type opcode	10	`gxStyleFontOpcode` constant of the `gxStyleDataOpcode` enumeration
0x01	Data	1	A reference to font name object 1.
Set data for the text size of the style object			
0x43 (01 000011)	Operation opcode	1	Set data.
	Record size	3	Record size is 3 bytes.
0x49 (01 001001)	Compression type opcode	1	Word compression
	Data type opcode	9	`gxStyleTextSizeOpcode` constant of the `gxStyleDataOpcode` enumeration
0x00 87	Data	135.0	The size parameter for the `GXSetShapeTextSize` function is 135.0 points.
New ink object			
0x01 (00 000001)	Operation opcode	0	New object
	Record size	1	Record size is 1 byte.
0x29 (00 101001)	Compression type opcode	0	No compression
	Data type opcode	0x29	`gxInkTypeOpcode` constant of the `gxGraphicsNewOpcode` enumeration

continued

Table 7-18 Analysis of the data stream of a flattened text shape (continued)

Values in data stream (binary)	Type of information	Value	Description
Set data for ink color of the ink object			
0x47 (01 000111)	Operation opcode	1	Set data.
	Record size	7	Record size is 7 bytes.
0x02 (00 000010)	Compression type opcode	0	No compression
	Data type opcode	2	`gxInkColorOpcode` constant of the `gxInkDataOpcode` enumeration
0xB6 (10 11 0110)	Omit byte	–	The `gxOmitColorsMask` and `gxOmitColorsShift` enumerations are used to interpret this omit byte. Data1, color space, is byte compressed. Data2, color profile, is omitted so the default color profile is applied to the current object. Data3, color components, uses bits 3, 2, 1, and 0 for color space. The compression for each of the red, green and blue color components is byte compression.
0x03	Data1	3	`gxHSVSpace` constant of the `gxColorSpaces` enumeration
0x74 00	Data2	0.453	The hue of the `gxHSVColor` structure is 0.453.
0xFF	Data	0xFFFF	Since color components are 2-byte values, the byte is replicated to the value 0xFFFF. The saturation of the `gxHSVColor` structure is 1.0000.
0xFF	Data	0xFFFF	Since color components are 2-byte values, the byte is replicated to the value 0xFFFF. The value of the `gxHSVColor` structure is 1.0000.
New transform object			

Bytes 0x01 and 0x2A define the new transform object. This data sequence is identical to the previous line shape example.

Table 7-18 Analysis of the data stream of a flattened text shape (continued)

Values in data stream (binary)	Type of information	Value	Description
Set data for mapping of the transform object			
0x59 (01 011001)	Operation opcode	1	Set data.
	Record size	7	Record size is 25 bytes. The transform data size is 25 – 1 (data type opcode byte) = 24 bytes. Since each mapping requires 8 bytes, there are 24/8 = 3 mappings. This indicates that there is a translate, scale, and skew mapping.
0x03 (00 000011)	Compression type opcode	0	No compression
	Data type opcode	3	`gxTransformMapping` constant of the `gxTransformDataOpcode` enumeration
`0x003D0212`	Data	61.12	The `deltaY` parameter for the `GXSetTransformMapping` function
`0x000098FE`	Data	0.60	The `deltaX` parameter for the `GXSetTransformMapping` function
`0x0000F747`	Data	0.97	The `hScale` parameter for the `GXSetTransformMapping` function
`0x0000F747`	Data	0.97	The `scale` parameter for the `GXSetTransformMapping` function
`0x00004242`	Data	0.26	The `hSkew` parameter for the `GXSetTransformMapping` function
`0xFFFFBDBE`	Data	–0.4242	The `vSkew` parameter for the `GXSetTransformMapping` function
New shape object			
0x09 (00 001001)	Operation opcode	0	New object
	Record size	9	Record size is 9 bytes.
0x09 (00 001001)	Compression type opcode	0	No compression
	Data type opcode	9	`gxTextType` constant of the `gxShapeTypes` enumeration

continued

Table 7-18 Analysis of the data stream of a flattened text shape (continued)

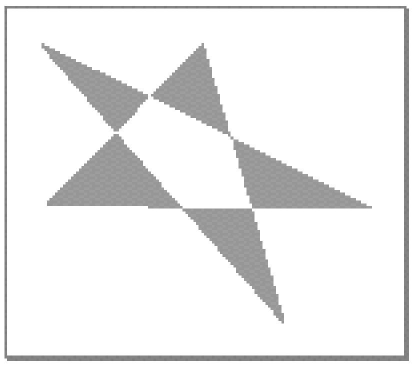
Values in data stream (binary)	Type of information	Value	Description
`0x2A` `(00 10 10 10)`	Omit byte	–	The `gxOmitTextMask` and `gxOmitTextShift` enumerations are used to interpret this omit byte. Byte compression is used for data1, and byte length. Byte compression is used for data2, and the x coordinate of the position. Word ?? compression is used for data3, y coordinate of position point. Byte compression is used for data4, number of characters and text.
`0x02`	Data1	2	The byte length is 2.
`0x19`	Data2	25.0000	The x-coordinate of the text position is 25.0000.
`0x00 E6`	Data3	230.0000	The y-coordinate of the text position is 230.0000.
`0x02`	Data4	2	The number of characters is 2.
`0x47`	Data4	0x47	Roman capital G
`0x58`	Data4	0x58	Roman capital X
Set data for attributes of the text object			
`0x42` `(01 000010)`	Operation opcode	2	Set data
	Record size	2	Record size is 2 bytes.
`0x80` `(10 000010)`	Compression type opcode	2	Byte compression
	Data type opcode	3	`gxShapeAttributes` constant of the `gxShapeDataOpcode` enumeration
`0x20`	Data	32	`gxMapTransformShape` constant of the `gxShapeAttributes` enumeration

Analyzing a Flattened Polygon Shape

The function described in the section “Creating a Picture With Seven Shapes” beginning on page 7-56 was first used to draw the picture shown in Figure 7-12 containing the polygon shape shown in Figure 7-18.

The yellow polygon shape is drawn with a pen thickness of 3.0 and skewed in the vertical direction by 0.5. Its size and shape is controlled by the vectors defined by the points (60.0, 0.0), (90.0, 90.0), (0.0, 30.0), (120.0, 30.0), (0.0, 90.0). The fill is even-odd. Once the polygon is drawn, it is moved to the point (240.0, 110.0) to position it in the picture.

Figure 7-18 The polygon shape drawn



The procedure described in the section “Flattening Shapes With GraphicsBug” beginning on page 7-54 was then used to generate the GraphicsBug output shown in Listing 7-9. The flattened polygon shape data stream is the sequential data that appears in parentheses.

Listing 7-9 GraphicsBug analysis of a flattened polygon shape

```
polygonType; byte compression (86)
(01 05 5a 01 2c 01 04 e2 97 5a 69 88 c4 78 00)
```

Table 7-19 shows the data stream analysis of the flattened polygon shape. The stream data is obtained from the GraphicsBug output in Listing 7-9. This table provides a description of each byte of the data stream for this shape. Data format sequences that are identical to previously described data sequences in the stream are not shown and are not analyzed here.

Table 7-19 Analysis of the data stream of a flattened polygon shape

Values in data stream (binary)	**Type of information**	**Value**	**Description**
New shape object			
`0x10`	Operation opcode	0	New object
`(00 010000)`	Record size	10	Record size is 10 bytes.
`0x86`	Compression type opcode	2	Byte compression
`(10 000110)`	Data type opcode	6	`gxPolygonType` constant of the `gxShapeTypes` enumeration
`0x01`	Data	1	The number of contours is 1.
`0x05`	Data	5	The number of vectors in the contour is 5.
`0x5A (01 01 10 10)`	Omit byte	–	The `gxOmitPathMask` and `gxOmitPathShift` enumerations are used to interpret this byte. Word compression is used for data1, and x coordinate of first point. Word compression is used for data2, and y coordinate of first point. Byte compression is used for data3, and all x relative coordinate deltas. Byte compression is used for data4, and all y relative coordinate deltas.
`0x01 2C`	Data1	290.0	Absolute x-coordinate of the first point is 290.0
`0x01 04`	Data2	260.0	Absolute y-coordinate of the first point is 260.0
`0xE2`	Data3	–30.0	The x-coordinate distance of the second point from the first point is –75.0. Absolute x coordinate of the second point is 290.0 – (–30.0) = 320.0
`0x97`	Data4	–105.0	The y-coordinate distance of the second point from the first point is –105.0. Absolute y-coordinate of the second point is 260.0 – (–105.0) = 365.0.
`0x5A`	Data3	90.0	The x-coordinate distance of the third point from the first point is 90.0. Absolute x-coordinate of the third point is 290.0 – (90.0) = 200.0.

Table 7-19 Analysis of the data stream of a flattened polygon shape (continued)

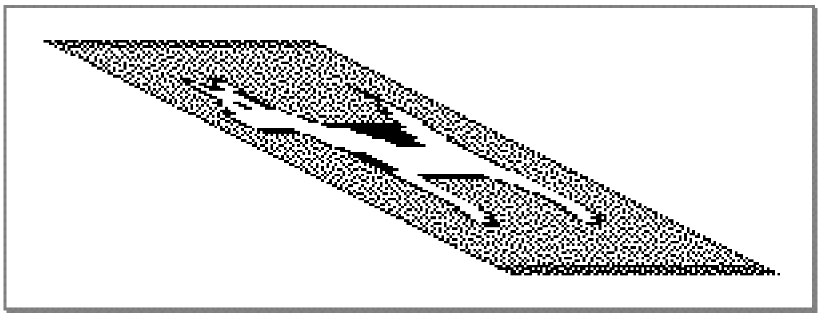
Values in data stream (binary)	Type of information	Value	Description
`0x69`	Data4	151.0	The y-coordinate distance of the third point from the first point is 151.0. Absolute y-coordinate of the third point is 260.0 – (151.0) = 109.0.
`0x88`	Data3	136.0	The x-coordinate distance of the fourth point from the first point is 70.0. Absolute x-coordinate of the fourth point is 290.0 – (70.0) = 220.0
`0xC4`	Data4	–60.0	The y-coordinate distance of the fourth point from the first point is –60.0. Absolute y-coordinate of the fourth point is 260.0 – (–60.0) = 320.0.
`0x78`	Data3	120.0	The x-coordinate distance of the fifth point from the first point is 70.0. Absolute x-coordinate of the fifth point is 290.0 – (120.0) = 170.0.
`0x00`	Data4	0.0	The y-coordinate distance of the fifth point from the first point is –50.0. Absolute y-coordinate of the fifth is 260.0 – (0.0) = 260.0.

Analyzing a Flattened Bitmap Shape

The function described in the section “Creating a Picture With Seven Shapes” beginning on page 7-56 was first used to draw the picture shown in Figure 7-12 containing the polygon shape shown in Figure 7-19.

The bitmap was retrieved from the resource fork and skewed in the horizontal direction by a factor of 2.0. Once the bitmap is drawn, it is moved to the point (200.0, 190.0) to position it in the picture.

Figure 7-19 The bitmap shape drawn



The procedure described in the section “Flattening Shapes With GraphicsBug” beginning on page 7-54 was then used to generate the GraphicsBug output shown in Listing 7-10. The flattened bitmap shape data stream is the sequential data that appears in parentheses.

Listing 7-10 GraphicsBug analysis of a flattened bitmap shape

```
newObject; size: #0 (01) [1]
transformType; no compression (2a)
setData; size: #12 (4d)
transformMapping; word compression (43)
(01 22 00 be 00 01 00 01 00 00 00 02)
newObject; size: #403 (00 00 01 94) [1]
bitImage; no compression (2e)
(a8 34 58 73 11 01 01 c2 81 70 22 01 21 82 ca ... )
newObject; size: #49 (32) [1]
colorSetType; byte compression (ac)
(01 ff ff ff ff 00 00 33 ff 00 33 cc 00 00 ...)
newObject; size: #10 (0b)
bitmapType; no compression (08)
(aa)
image (01)
width (66)
height (58)
rowBytes (34 ab)
pixelSize (04)
space (0b)
set (01 f0)
```

Table 7-20 shows the data stream analysis of the flattened bitmap shape. The stream data is obtained from the GraphicsBug output in Listing 7-10. This table provides a description of each byte of the data stream for this shape. Data format sequences that are identical to previously described data sequences in the stream are not shown and are not analyzed here.

Table 7-20 Analysis of the data stream of a bitmap shape

Values in data stream (binary)	Type of information	Value	Description
New transform object			
Bytes 0x01 and 0x2A define the new transform object. This data sequence is identical to the previous line shape example.			
Set data for mapping of the transform object			
`0x4D`	Operation opcode	1	Set data
`(01 001101)`	Record size	13	Record size is 13 bytes. The transform data size is13 – 1 (data type opcode byte) = 12 bytes. Since each mapping requires 8 bytes, there are 12/2 =6 mappings. This indicates that there is a translate, scale, and skew mapping.
`0x43`	Compression type opcode	1	Word compression
`(01 000011)`	Data type opcode	3	`gxTransformMapping` constant of the `gxTransformDataOpcode` enumeration
`0x0122`	Data	290.0	The `deltaX` parameter for the `GXSetTransformMapping` function is 290.0.
`0x00BE`	Data	190.0	The `deltaY` parameter for the `GXSetTransformMapping` function is 190.0.
`0x0001`	Data	1.0	The `hScale` parameter for the `GXSetTransformMapping` function is 1.0.
`0x0001`	Data	1.0	The `vScale` parameter for the `GXSetTransformMapping` function is 1.0.
`0x0000`	Data	0.0	The `hSkew` parameter for the `GXSetTransformMapping` function is 0.0.
`0x0002`	Data	2.0	The `vSkew` parameter for the `GXSetTransformMapping` function is 2.0.

continued

Table 7-20 Analysis of the data stream of a bitmap shape (continued)

Values in data stream (binary)	Type of information	Value	Description
New bitmap image			
`0x00` (`00 000000`)	Operation opcode	0	New object
	Record size	0	Record size is > 64 bytes.
`0x00`	Record size (continued)	0	Record size is > 256 bytes.
`0x01 94`	Record size (continued)	404	Record size is 404 bytes. For additional information about the stream format for the record size, see the section “Record Size” beginning on page 7-11.
`0x2E` (`00 101110`)	Compression type opcode	0	No compression
	Data type opcode	0x2E	`gxBitImageOpcode` constant of the `gxGraphicsNewOpcode` enumeration
`0xA8` (`10 10 1 000`)	Omit byte	–	The `gxOmitBitImageMask` and `gxOmitBitImageShift` enumerations are used to interpret this omit byte. Data1, `width`, is byte compressed. Data2, `height`, is byte compressed. Data3, indicates that the bit image data is compressed. The last3 bits are not used and are reserved.
`0x34`	Data1	52	The bit image row width is 52 bytes.
`0x58`	Data2	88	The bit image column height is 88 bytes.
Row 1 of the bit image follows			
`0x73` (`01 110011`)	Bit image compression byte	1	Bits 6 and 7 are 1. This is the `gxRepeatBitImageBytesOpcode` constant of the `gxBitImageCompression` enumeration.
		51	The bits that follow are to be repeated 51 times.
`0x11`	Data	11	The bits “11” are to be repeated 51 times

Table 7-20 Analysis of the data stream of a bitmap shape (continued)

Values in data stream (binary)	Type of information	Value	Description
`0x01 (00 000001)`	Data	0	This is the `gxCopyBitImageBytesOpcode` constant of the `gxBitImageCompression` enumeration. The bits in the next byte are added to the first row *x* number of times.
		1	The value of *x* is 1.
`0x01`	Data	"01"	The bits "01" are added to row 1
Rows 2 through 11 of the bit image follow			
`0xC2 (11 000010)`	Bit image compression byte	3	This is the `gxRepeatBitImageScanOpcode` constant of the `gxBitImageCompression` enumeration. The previous scan line is repeated *x* times.
	Previous row repeat number	2	The value of *x* is 2. The first row of bits is repeated 2 times.
Row 12			
`0x81 (10 000001)`	Bit image compression byte	2	This is the `gxLookupBitImageBytesOpcode` constant of the `gxBitImageCompression` enumeration. Repeat *x* bytes from the previous row and add them to the current row.
		1	The value of *x* is 1. One byte of data is to be repeated from the previous scan line.
`0x70 01 110000`	Bit image compression byte	1	Bits 6 and 7 are 1. This is the `gxRepeatBitImageBytesOpcode` constant of the `gxBitImageCompression` enumeration.
		48	The bits in the byte that follow are to be repeated 48 times.
`0x22`	Data	"100010"	The bits "100010" are to be repeated 48 times

continued

Table 7-20 Analysis of the data stream of a bitmap shape (continued)

Values in data stream (binary)	Type of information	Value	Description
`0x01 (00 000001)`	Bit image compression byte	0	This is the `gxCopyBitImageBytesOpcode` constant of the `gxBitImageCompression` enumeration. Repeat *x* bytes from the previous row and add them to the current row.
		1	The value of x is 1. One byte of data is to be repeated from the previous scan line.
`0x21`	Data	"100001"	The bits "100001" are to be repeated 1 time on the second row.
`0x82 (10 000010)`	Bit image compression byte	2	This is the `gxLookupBitImageBytesOpcode` constant of the `gxBitImageCompression` enumeration. Repeat x bytes from the previous row and add them to the current row.
		2	The value of *x* is 2. Two bytes of data is to be repeated from the previous scan line.
`0xCA (11 001010)`	Bit image compression byte	3	This is the `gxRepeatBitImageScanOpcode` constant of the `gxBitImageCompression` enumeration. The previous scan line is repeated *x* times.
		10	The value of *x* is 10. The first row of bits is repeated 10 times.
The remaining bytes of the bit image are not shown here.			
New color set object			
`0x32 (00 110010)`	Operation opcode	0	New object
	Record size	50	Record size is 50 bytes.
`0xAC (10 101100)`	Compression type opcode	2	Byte compression
	Data type opcode	3	`gxColorSetTypeOpcode` constant of the `gxGraphicsNewOpcode` enumeration
`0x01`	Data	1	`gxRGBSpace` constant of the `gxColorSpaces` enumeration

Table 7-20 Analysis of the data stream of a bitmap shape (continued)

Values in data stream (binary)	Type of information	Value	Description
White color for the bitmap object			
`0xFF`	Data	0xFFFF	Since color components are 2-byte values, the byte is replicated to the value 0xFFFF or 65,535. The RGB value for the `red` field of the `gxRgbColor` structure is 65,535.
`0xFF`	Data	0xFFFF	Since color components are 2-byte values, the byte is replicated to the value 0xFFFF or 65,535. The RGB value for the `green` field of the `gxRgbColor` structure is 65,535.
`0xFF`	Data	0xFFFF	Since color components are 2-byte values, the byte is replicated to the value 0xFFFF or 65,535. The RGB value for the `blue` field of the `gxRgbColor` structure is 65,535.
Dark blue color for the bitmap object			
`0x00`	Data	0x0000	Since color components are 2-byte values, the byte is replicated to the value 0x0000 or 0. The RGB value for the `red` field of the `gxRgbColor` structure is 0.
`0x00`	Data	0x0000	Since color components are 2-byte values, the byte is replicated to the value 0x0000 or 0. The RGB value for the `green` field of the `gxRgbColor` structure is 0.
`0x33`	Data	0x0000	Since color components are 2-byte values, the byte is replicated to the value 0x3333 or 0. The RGB value for the `blue` field of the `gxRgbColor` structure is 0x3333.
Cherry red color for the bitmap object			
`0xFF`	Data	0xFFFF	Since color components are 2-byte values, the byte is replicated to the value 0xFFFF or 65,535. The RGB value for the `red` field of the `gxRgbColor` structure is 65,535.

continued

Table 7-20 Analysis of the data stream of a bitmap shape (continued)

Values in data stream (binary)	Type of information	Value	Description
`0x00`	Data	0x0000	Since color components are 2-byte values, the byte is replicated to the value 0x0000 or 0. The RGB value for the `green` field of the `gxRgbColor` structure is 0.
`0x33`	Data	0x3333	Since color components are 2-byte values, the byte is replicated to the value 0x3333. The RGB value for the `blue` field of the `gxRgbColor` structure is 0x3333.
Dull red color for the bitmap object			
`0xCC`	Data	0xCCCC	Since color components are 2-byte values, the byte is replicated to the value 0xCCCC or 52,428. The RGB value for the `red` field of the `gxRgbColor` structure is 52,428.
`0x00`	Data	0x0000	Since color components are 2-byte values, the byte is replicated to the value 0x0000 or 0. The RGB value for the `green` field of the `gxRgbColor` structure is 0.
`0x00`	Data	0x0000	Since color components are 2-byte values, the byte is replicated to the value 0x0000 or 0. The RGB value for the `blue` field of the `gxRgbColor` structure is 0x0000.
The remaining 35 bytes of the color set are not shown here.			
New shape object			
`0x10` `(00 010000)`	Operation opcode	0	New object
	Record size	11	Record size is 11 bytes.
`0x08` `(00 001000)`	Compression type opcode	0	Byte compression
	Data type opcode	8	`gxBitmapType` constant of the `gxShapeTypes` enumeration
`0xAA` `(10 10 10 10)`	Omit byte	–	The `gxOmitBitmapMask1` and `gxOmitBitmapShift1` enumerations are used to interpret this byte. Byte compression is used for data1, data2, data3, and data4.
`0x01`	Data1	1	A pointer to the pixels located at 1.
`0x66`	Data2	102	The row width is 102 pixels.

Table 7-20 Analysis of the data stream of a bitmap shape (continued)

Values in data stream (binary)	Type of information	Value	Description
`0x58`	Data3	88	The column height is 88 pixels.
`0x34`	Data4	52	The row width is 52 bytes.
`0xAB (10 10 10 11)`	Omit byte	–	The `gxOmitBitmapMask2` and `gxOmitBitmapShift2` enumerations are used to interpret this byte. Byte compression is used for data1, data2, and data3. Data4 is omitted.
`0x04`	Data1	4	The number of bits per pixel is 1.
`0x0B`	Data2	11	`gxIndexedSpace` constant of the `gxColorSpaces` enumeration
`0x01`	Data3	1	The first set of bitmaps is used.
`0xF0 (11 11 00 00)`	Omit byte	–	The `gxOmitBitmapMask3` and `gxOmitBitmapShift3` enumerations are used to interpret this byte. Data1 and data2 are omitted. These are the x and y positions of the bitmap. The position is therefore at point (0, 0). The other bits are reserved.

Obtaining Data From a Print File

Any suitably equipped Macintosh computer with QuickDraw GX installed can read and print portable digital document print files created by your application. You may want to use the public data in a QuickDraw GX print file for other purposes. Listing 7-11 reads a portable digital document print file and returns the page count. For more information on print files and portable digital documents, see the chapters “Introduction to QuickDraw GX Printing” and “Core Printing Features” of *Inside Macintosh: QuickDraw GX Printing*.

Listing 7-11 Obtaining the page count from a portable digital document print file

```
#define nrequire( x, LABEL ) if((x)) goto LABEL

/* Returns the page count from an open print file */

   Parameters:-> short dataRefNum:reference to the spool file
                 <- long *pageCount:returns page count
   Returns:    OSErr
   Preconditions:dataRefNum != NULL
   Postconditions:none */
```

```
OSErr DespoolPageCount (short dataRefNum, long *pageCount);
OSErr DespoolPageCount (short dataRefNum, long *pageCount) {

   register OSErr anErr;
            long pageDirOffset, numPages;
            long dataLen;
/* position to read offset to page directory */

   anErr = SetFPos(dataRefNum, fsFromStart, (long) (kHeaderSize +
sizeof(long)));
   nrequire (anErr, SetPageDirOffsetPos);

   /* read offset to page directory */
   dataLen = sizeof(pageDirOffset);
   anErr = FSRead(dataRefNum, &dataLen, &pageDirOffset);
   nrequire (anErr, ReadPageDirOffsetPos);

/* move to page directory */
   anErr = SetFPos(dataRefNum, fsFromStart, (long) (pageDirOffset));
   nrequire (anErr, SetPageDirPos);

/* read number of pages */
   dataLen = sizeof(numPages);
   anErr = FSRead(dataRefNum, &dataLen, &numPages);
   nrequire (anErr, ReadNumPages);

   *pageCount = numPages;/* Return the result */

   ncheck (anErr);
   return anErr;

/* exception handling*/

ReadNumPages:
SetPageDirPos:
ReadPageDirOffsetPos:
SetPageDirOffsetPos:
   return anErr;
}
```

QuickDraw GX Stream Format Reference

This section provides reference information to the data structures and enumerations that are used in the stream format of a flattened shape.

Opcode Constants and Data Types

This section describes the constants and data types that describe the opcodes used in the data streams of flattened shapes.

Operation Opcode Byte

Bits 6 and 7 of the operation opcode byte are the operation opcode. This opcode provides a description of the data record that follows. Each operation opcode is defined in the `gxGraphicsOperationOpcode` enumeration.

```
enum gxGraphicsOperationOpcode {
   gxNewObjectOpcode = 0x00,
   gxSetDataOpcode   = 0x40,
   gxSetDefaultOpcode= 0x80,
   gxReservedOpcode  = 0xC0,
   gxNextOpcode      = 0xFF,
};
```

Constant descriptions

`gxNewObjectOpcode`	Data for a new object follows.
`gxSetDataOpcode`	Attributes for the current object follow.
`gxSetDefaultOpcode`	Replace current default with the object that follows.
`gxReservedOpcode`	This opcode is reserved for future expansion.
`gxNextOpcode`	This constant is used by the current operand field to indicate that an opcode is coming.

Bits 0 through 5 of the operation opcode byte are the record size in bytes (1 to 63 bytes). The `gxObjectSizeMask` constant, binary 111111, masks bits 0 through 5 to select the record size. For additional information about the stream format for the record size, see the section “Record Size” beginning on page 7-11.

```
#define gxObjectSizeMask   0x3F
```

The `gxOpcodeShift` constant allows you to compare `gxGraphicsOperationOpcode` constants with other values.

```
#define gxOpcodeShift   6
```

Data Type Opcode Byte

Bits 6 and 7 of the data type opcode byte are the compression type opcode. The compression of the data to follow is given by the `gxTwoBitCompressionValues` enumeration in Table 7-3. The `gxCompressionMask` constant, binary 11, masks the constant defined by the `gxTwoBitCompressionValue` enumeration.

```
#define gxCompressionMask  0x03
```

The `gxCompressionShift` constant defines the number of bits to be shifted to the right so that the masked value of the compression type opcode can be compared to other values.

```
#define gxCompressionShift 6
```

Bits 0 through 5 of the data type opcode byte are the data type opcode. These opcodes describe the data that follows in the stream. The `gxObjectTypeMask` constant, binary 111111, masks bits 0 through 5 of the data type opcode byte to select the data type opcode. No shift is required to compare the data type opcode with other values.

```
#define gxObjectTypeMask   0x3F
```

Generic Data Opcode

The current operand uses a constant from the `gxGenericDataOpcode` enumeration when the current operand is the `gxNextOpcode` constant.

```
enum gxGenericDataOpcode {
   gxTypeOpcode,
   gxSizeOpcode
};
```

Constant descriptions

`gxTypeOpcode`	The next opcode is a type opcode.
`gxSizeOpcode`	The next opcode is a size opcode.

Bit Image Compression Opcode Byte

Bits 6 and 7 of the bit image compression opcode byte contain the compression type opcode that describes the data compression used for a region of a a bit image.The `gxBitimageOpcodeMask` constant, binary 11000000, masks bits 6 and 7 of the bit image compression opcode byte to select the bit image opcode.

```
#define gxBitimageOpcodeMask 0xC0
```

Once the `gxBitimageOpcodeMask` constant has been used to select the compression type opcode, a bit shift given by the `gxBitimageOpcodeShift` constant can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the compression type opcode so that it can be compared to other values.

```
#define gxBitimageOpcodeShift 6
```

Bits 0 through 5 of the bit image compression opcode byte contain the bit image count. This is the number of times that a binary sequence is repeated. The `gxBitimageCountMask` constant, binary 111111, masks bits 0 through 5 of the bit image compression opcode byte to select the bit image count. No shift is required to compare the bit image count with other values.

```
#define gxBitimageCountMask0x3F
```

Table 7-13 gives the bit image compression opcode constants. For additional information about the use of the bit image compression opcode byte, see the section “New Bit Image Object Data” beginning on page 7-49.

Modified Shape Data Opcodes

A constant from the `gxShapeDataOpcode` enumeration follows a `gxSetDataOpcode` operation opcode if shape data follows. The data stream bytes describe one of the fields specified in this enumeration.

```
enum gxShapeDataOpcode {
   gxShapeAttributesOpcode,
   gxShapeTagOpcode,
   gxShapeFillOpcode
};
```

Constant descriptions

`gxShapeAttributesOpcode`
: An attribute from the `gxShapeAttributes` enumeration is added to the current shape object.

`gxShapeTagOpcode`
: A tag is added to the current shape object.

`gxShapeFillOpcode`
: A fill is added to the current shape object.

Modified Style Data Opcodes

A constant from the `gxStyleDataOpcode` enumeration follows a `gxSetDataOpcode` if style data follows. The data stream bytes that follow describe one of the attributes specified in this enumeration.

```
enum gxStyleDataOpcode {
   gxStyleAttributesOpcode,
   gxStyleTagOpcode,
   gxStyleCurveErrorOpcode,
   gxStylePenOpcode,
   gxStyleJoinOpcode,
   gxStyleDashOpcode,
   gxStyleCapsOpcode,
   gxStylePatternOpcode,
   gxStyleTextAttributesOpcode,
   gxStyleTextSizeOpcode,
   gxStyleFontOpcode,
   gxStyleTextFaceOpcode,
   gxStylePlatformOpcode,
   gxStyleFontVariationsOpcode,
   gxStyleRunControlsOpcode,
   gxStyleRunPriorityJustOverrideOpcode,
   gxStyleRunGlyphJustOverridesOpcode,
   gxStyleRunGlyphSubstitutionsOpcode,
   gxStyleRunFeaturesOpcode,
   gxStyleRunKerningAdjustmentsOpcode,
   gxStyleJustificationOpcode
};
```

Constant descriptions

`gxStyleAttributesOpcode`
: The style attributes flags from the `gxStyleAttributes` enumeration follow.

`gxStyleTagOpcode`
: The parameters of the `GXSetStyleTags` function follow.

`gxStyleCurveErrorOpcode`
: Data for the `error` parameter of the `GXSetStyleCurveError` function follows.

`gxStylePenOpcode`
: The data for the `pen` parameter of the `GXSetStylePen` function follows.

`gxStyleJoinOpcode`
: The data for the fields of the `gxJoinRecord` structure follows.

`gxStyleDashOpcode`
: The data for the fields of the `gxDashRecord` structure follows.

`gxStyleCapsOpcode`
: The data for the fields of the `gxCapRecord` structure follows.

`gxStylePatternOpcode`
: The data for the fields of the `gxPatternRecord` structure follows.

`gxStyleTextAttributesOpcode`
: The data from the `gxTextAttributes` enumeration follows.

`gxStyleTextSizeOpcode`
: The data for the `size` parameter of the `GXSetStyleTextSize` function follows.

`gxStyleFontOpcode`
: The data for the `font` parameter of the `GXSetStyleFont` function follows.

`gxStyleTextFaceOpcode`
: The data for the fields of the `gxTextFace` structure follows.

`gxStylePlatformOpcode`
: The data for the parameters of the `GXStyleEncoding` function follows.

`gxStyleFontVariationsOpcode`
: The data for the fields of the `gxFontVariations` structure follows.

`gxStyleRunControlsOpcode`
: The data for the fields of the `gxRunControls` structure follows.

`gxStyleRunPriorityJustOverrideOpcode`
: The data for the fields of the `gxPriorityJustificationOverride` structure follows.

`gxStyleRunGlyphJustOverridesOpcode`
: The data for the fields of the `gxGlyphJustificationOverride` structure follows.

`gxStyleRunGlyphSubstitutionsOpcode`
: The data for the fields of the `gxGlyphSubstitutionOverride` structure follows.

`gxStyleRunFeaturesOpcode`
: The data for the fields of the `gxRunFeature` structure follows.

`gxStyleRunKerningAdjustmentsOpcode`
: The data for the fields of the `gxKerningAdjustment` structure follows.

`gxStyleJustificationOpcode`
: The data for the `justify` parameter of the `GXSetStyleJustification` function follows.

Modified Ink Data Opcodes

A constant from the `gxInkDataOpcode` enumeration follows a `gxSetDataOpcode` operation opcode if ink data follows. The data stream bytes that follow describe one of the attributes specified in this enumeration.

```
enum gxInkDataOpcode {
   gxInkAttributesOpcode,
   gxInkTagOpcode,
   gxInkColorOpcode,
   gxInkTransferModeOpcode
};
```

Constant descriptions

`gxInkAttributesOpcode`	The parameters of the `GXSetInkAttributes` function follow.
`gxInkTagOpcode`	The parameters of the `GXSetInkTags` function follow.
`gxInkColorOpcode`	The parameters of the `GXSetInkColor` function follow.
`gxInkTransferModeOpcode`	The parameters of the `GXSetInkTransfer` function follow.

Modified Color Set Data Opcodes

A constant from the `gxColorSetDataOpcode` enumeration follows a `gxSetDataOpcode` operation opcode if color set data follows. The bytes that follow describe one of the attributes specified in this enumeration.

```
enum gxColorSetDataOpcode {
   gxColorSetReservedOpcode,
   gxColorSetTagOpcode
};
```

Constant descriptions

`gxColorSetReservedOpcode`	This opcode is reserved for future expansion.
`gxColorSetTagOpcode`	The data parameters for the `GXSetColorSetTags` function follows.

Modified Color Profile Data Opcodes

A constant from the `gxProfileDataOpcode` `enumeration` follows a `gxSetDataOpcode` operation opcode if profile data follows. The data stream bytes that follow describe one of the attributes specified in this enumeration.

```
enum gxProfileDataOpcode {
   gxColorProfileAttributesOpcode,
   gxColorProfileTagOpcode
};
```

Constant descriptions

`gxColorProfileAttributesOpcode`
: This opcode is reserved for future expansion.

`gxColorProfileTagOpcode`
: The data parameters for the `GXSetColorProfileTags` function follow.

Modified Transform Data Opcodes

A constant from the `gxTransformDataOpcode` enumeration follows a `gxSetDataOpcode` operation opcode if transform data follows. The data stream bytes that follow describe one of the attributes specified in this enumeration.

```
enum  gxTransformDataOpcode{
   gxTransformReservedOpcode,
   gxTransformTagOpcode,
   gxTransformClipOpcode,
   gxTransformMappingOpcode,
   gxTransformPartMaskOpcode,
   gxTransformToleranceOpcode
};
```

Constant descriptions

`gxTransformReservedOpcode`
: This opcode is reserved for future expansion.

`gxTransformTagOpcode`
: The data parameters for the `GXSetTransformTags` function follow.

`gxTransformClipOpcode`
: The data for the `clip` parameter of the `GXSetTransformClip` function follows.

`gxTransformMappingOpcode`
: The data for the `map` parameter of the `GXSetTransformMapping` function follows.

`gxTransformPartMaskOpcode`
: The data for the `mask` parameter of the `GXSetTransformHitTest` function follows.

`gxTransformToleranceOpcode`
: The data for the `gxProfileRecord` structure and `gxProfileResponse` enumeration follows.

Bit Image Compression Opcodes

Bits 6 and 7 of the bit image compression opcode byte contain the bit image compression opcode. A constant from the `gxBitImageCompression` enumeration defines the compression of the bit image data sequence to immediately follow.

```
enum gxBitImageCompression {
   gxCopyBitImageBytesOpcode  = 0x00,
   gxRepeatBitImageBytesOpcode= 0x40,
   gxLookupBitImageBytesOpcode= 0x80,
   gxRepeatBitImageScanOpcode = 0xC0
};
```

Constant descriptions

`gxCopyBitImageBytesOpcode`
: Bit image compression opcode 0.

`gxRepeatBitImageBytesOpcode`
: Bit image compression opcode 1.

`gxLookupBitImageBytesOpcode`
: Bit image compression opcode 2.

`gxRepeatBitImageScanOpcode`
: Bit image compression opcode 3.

The bit image compression opcode is described in the section “New Bit Image Object Data” beginning on page 7-49.

Flatten Header Bytes

The two bytes following the byte containing the `gxHeaderTypeOpcode` contain the version of QuickDraw GX that generated the stream of data that follows and two flags that are defined by the `gxFlattenFlags` enumeration.

```
struct gxFlattenHeader {
   fixed          version;
   unsigned char  flatFlags;
};
```

Field descriptions

`version`	The version of QuickDraw GX that was used to create the stream.
`flatFlags`	The `gxFontListFlatten` and `gxFontGlyphsFlatten` flags.

The QuickDraw GX version and the flatten flags are described in the section "Header Data" beginning on page 7-27.

Style Object Omit Byte Constants and Data Types

This section describes the constants and data types that are used to interpret omit bytes that are used with style object data. The use of omit bytes is described in the section "Omit Byte Masks and Omit Byte Shifts" beginning on page 7-22.

Dash Style Omit Byte Masks and Shifts

The `gxOmitDashMask1` enumeration defines which bits in an omit byte correspond to the a data compression opcode for the field descriptors in the `gxDashRecord` structure. The sequence of data is also defined. The omit byte and its related data sequence are given in the section "Dash Data" beginning on page 7-37.

```
enum gxOmitDashMask1 {
   gxOmitDashAttributesMask   = 0xC0,
   gxOmitDashShapeMask        = 0x30,
   gxOmitDashAdvanceMask      = 0x0C,
   gxOmitDashPhaseMask        = 0x03
};
```

Constant descriptions

`gxOmitDashAttributesMask`	The mask to select the data compression bits for the `attributes` field descriptor.
`gxOmitDashShapeMask`	The mask to select the data compression bits for the `dash` field descriptor.
`gxOmitDashAdvanceMask`	The mask to select the data compression bits for the `advance` field descriptor.
`gxOmitDashPhaseMask`	The mask to select the data compression bits for the `phase` field descriptor.

Once one of the `gxOmitDashMask1` enumeration masks has been used to select data compression bits for a field descriptor in the `gxDashRecord` structure, the corresponding bit shift from the `gxOmitDashShift1` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression bits so that they can be compared to other values.

```
enum gxOmitDashShift1 {
   gxOmitDashAttributesShift  = 6,
   gxOmitDashShapeShift       = 4,
   gxOmitDashAdvanceShift     = 2,
   gxOmitDashPhaseShift       = 0
};
```

Constant descriptions

`gxOmitDashAttributesShift`
: The bit shift required to isolate the compression bits for the `attributes` field descriptor.

`gxOmitDashShapehift`
: The bit shift required to isolate the compression bits for the `dash` field descriptor.

`gxOmitDashAdvanceShift`
: The bit shift required to isolate the compression bits for the `advance` field descriptor.

`gxOmitDashPhaseShift`
: The bit shift required to isolate the compression bits for the `phase` field descriptor.

The `gxOmitDashMask2` enumeration defines which bits in a second omit byte correspond to the data compression bits for additional field descriptors in the `gxDashRecord` structure. The sequence of data is also continued. The use of this mask and shift are described in the section “Dash Data” beginning on page 7-37.

```
enum gxOmitDashMask2 {
   gxOmitDashScaleMask = 0xC0
};
```

Constant descriptions

`gxOmitDashScaleMask`
: The mask for the data compression bits for the `scale` field descriptor.

Once one of the `gxOmitDashMask2` enumeration masks has been used to select data compression bits for a field descriptor in the `gxDashRecord` structure, the corresponding bit shift from the `gxOmitDashShift2` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression bits so that they can be compared to other values.

```
enum gxOmitDashShift2{
      gxOmitDashScaleShift = 6
};
```

Constant descriptions

`gxOmitDashScaleShift`
: The bit shift required to isolate the compression bits for the `scale` field descriptor.

Pattern Style Omit Byte Masks and Shifts

The `gxOmitPatternMask1` enumeration defines which bits in an omit byte correspond to the data compression opcodes for the field descriptors in the `gxPatternRecord` structure. The sequence of data is also defined. The omit byte and its related data sequence is given in the section “Pattern Data” beginning on page 7-38.

```
enum gxOmitPatternMask1{

      gxOmitPatternAttributesMask   = 0xC0,
      gxOmitPatternShapeMask        = 0x30,
      gxOmitPatternUXMask           = 0x0C,
      gxOmitPatternUYMask           = 0x03
   };
```

Constant descriptions

`gxOmitPatternAttributesMask`
: The mask used to select the data compression bits for the `attributes` field descriptor.

`gxOmitPatternShapeMask`
: The mask used to select the data compression bits for the `pattern` field descriptor.

`gxOmitPatternUXMask`
: The mask used to select the data compression bits for the `ux` field descriptor.

`gxOmitPatternUYMask`
: The mask used to select the data compression bits for the `uy` field descriptor.

Once one of the `gxOmitPatternMask1` enumeration masks has been used to select data compression bits for one of the field descriptors in the `gxPatternRecord` structure, the corresponding bit shift from the `gxOmitPatternShift1` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression bits so that they can be compared to other values.

```
enum gxOmitPatternShift1 {
   gxOmitPatternAttributesShift  = 6,
   gxOmitPatternShapeShift       = 4,
   gxOmitPatternUXShift          = 2,
   gxOmitPatternUYShift          = 0
   };
```

Constant descriptions

`gxOmitPatternAttributesShift`
: The bit shift required to isolate the compression bits for the `attributes` field descriptor.

`gxOmitPatternShapeShift`
: The bit shift required to isolate the compression bits for the `pattern` field descriptor.

`gxOmitPatternUXShift`
: The bit shift required to isolate the compression bits for the `ux` field descriptor.

`gxOmitPatternUYShift`
: The bit shift required to isolate the compression bits for the `uy` field descriptor.

The `gxOmitPatternMask2` enumeration defines which bits in a second omit byte correspond to the data compression opcode for additional field descriptors in the `gxPatternRecord` structure. The sequence of data is also continued. The omit byte and its related data sequence is given in the section “Pattern Data” beginning on page 7-38.

```
enum gxOmitPatternMask2 {
   gxOmitPatternVXMask = 0xC0,
   gxOmitPatternVYMask = 0x30
};
```

Constant descriptions

`gxOmitPatternVXMask`
: The mask used to select the data compression bits for the u.x field descriptor.

`gxOmitPatternVYMask`
: The mask to select the data compression bits for the u.y field descriptor.

Once one of the `gxOmitPatternMask2` enumeration masks has been used to select a data compression opcode for one of the field descriptors in the `gxPatternRecord` structure, the corresponding bit shift from the `gxOmitPatternShift2` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitPatternShift2 {
   gxOmitPatternVXShift= 6,
   gxOmitPatternVYShift= 4
};
```

Constant descriptions

`gxOmitPatternVXShift`
: The bit shift required to isolate the compression bits for the u.x field descriptor.

`gxOmitPatternVYShift`
: The bit shift required to isolate the compression bits for the u.y field descriptor.

Join Style Omit Byte Masks and Shifts

The `gxOmitJoinMask` enumeration defines which bits in an omit byte correspond to the data compression opcode for the field descriptors in the `gxJoinRecord` structure. The sequence of data is also defined. The omit byte and its related data sequence is given in the section "Join Data" beginning on page 7-37.

```
enum gxOmitJoinMask {
   gxOmitJoinAttributesMask= 0xC0,
   gxOmitJoinShapeMask     = 0x30,
   gxOmitJoinMiterMask     = 0x0C
};
```

Constant descriptions

`gxOmitJoinAttributesMask`
: The mask used to select the data compression bits for the `attributes` field descriptor.

`gxOmitJoinShapeMask`
: The mask used to select the data compression bits for the `join` field descriptor.

`gxOmitJoinMiterMask`
: The mask used to select the data compression bits for the `miter` field descriptor.

Once one of the `gxOmitJoinMask` enumeration masks has been used to select a data compression opcode for a field descriptor in the `gxJoinRecord` structure, the corresponding bit shift from the `gxOmitJoinShift` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitJoinShift {
   gxOmitJoinAttributesShift  = 6,
   gxOmitJoinShapeShift       = 4,
   gxOmitJoinMiterShift       = 2
};
```

Constant descriptions

`gxOmitJoinAttributesShift`
: The bit shift required to isolate the compression bits for the `attributes` field descriptor.

`gxOmitJoinShapeShift`
: The bit shift required to isolate the compression bits for the `join` field descriptor.

`gxOmitJoinMiterShift`
: The bit shift required to isolate the compression bits for the `miter` field descriptor.

Cap Style Omit Byte Masks and Shifts

The `gxOmitCapMask` enumeration defines which bits in an omit byte correspond to the data compression opcode for the field descriptors in the `gxCapRecord` structure. The sequence of data is also defined. The omit byte and its related data sequence is given in the section “Caps Data” beginning on page 7-38.

```
enum gxOmitCapMask {
   gxOmitCapAttributesMask = 0xC0,
   gxOmitCapStartShapeMask = 0x30,
   gxOmitCapEndShapeMask   = 0x0C
};
```

Constant descriptions

`gxOmitCapAttributesMask`
: The mask used to select the data compression bits for the `attributes` field descriptor.

`gxOmitCapStartShapeMask`
: The mask used to select the data compression bits for the `startCap` field descriptor.

`gxOmitCapEndShapeMask`
: The mask used to select the data compression bits for the `endCap` field descriptor.

Once one of the `gxOmitCapMask` enumeration masks has been used to select a data compression opcode for a field descriptor in the `gxCapRecord` structure, the corresponding bit shift from the `gxOmitCapShift` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitCapShift {
   gxOmitCapAttributesShift= 6,
   gxOmitCapStartShapeShift= 4,
   gxOmitCapEndShapeShift  = 2
};
```

Constant descriptions

`gxOmitCapAttributesShift`
: The bit shift required to isolate the compression bits for the `attributes` field descriptor.

`gxOmitCapStartShapeShift`
: The bit shift required to isolate the compression bits for the `startCap` field descriptor.

`gxOmitCapEndShapeShift`
: The bit shift required to isolate the compression bits for the `endCap` field descriptor.

Text Face Style Omit Byte Masks and Shifts

The `gxOmitFaceMask` enumeration defines which bits in an omit byte correspond to the data compression opcode for the field descriptors in the `gxTextFace` structure. The sequence of data is also defined. The omit byte and its related data sequence is given in the section “Text Face Data” beginning on page 7-39.

```
enum gxOmitFaceMask {
   gxOmitFaceLayersMask = 0xC0,
   gxOmitFaceMappingMask= 0x30
};
```

Constant descriptions

`gxOmitFaceLayersMask`
: The mask used to select the data compression bits for the `faceLayers` field descriptor.

`gxOmitFaceMappingMask`
: The mask used to select the data compression bits for the `advanceMapping` field descriptor.

Once one of the `gxOmitFaceMask` enumeration masks has been used to select a data compression opcode for a field descriptor in the `gxTextFace` structure, the corresponding bit shift from the `gxOmitFaceShift` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitFaceShift {
   gxOmitFaceLayersShift = 6,
   gxOmitFaceMappingShift= 4
};
```

Constant descriptions

`gxOmitFaceLayersShift`
: The bit shift required to isolate the compression bits for the `faceLayers` field descriptor.

`gxOmitFaceMappingShift`
: The bit shift required to isolate the compression bits for the `advanceMapping` field descriptor.

SEE ALSO

The section "Text Face Data" beginning on page 7-39 provides a full descrtiption of the `gxTextFace` structure.

Face Layer Omit Byte Masks and Shifts

The `gxOmitFaceLayerMask1` enumeration defines which bits in an omit byte correspond to the data compression opcode for the field descriptors in the `gxFaceLayer` structure. The sequence of data is also defined. The omit byte and its related data sequence is given in the section "Text Face Data" on page 7-39.

```
enum gxOmitFaceLayerMask1 {
   gxOmitFaceLayerFillMask       = 0xC0,
   gxOmitFaceLayerFlagsMask      = 0x30,
   gxOmitFaceLayerStyleMask      = 0x0C,
   gxOmitFaceLayerTransformMask  = 0x03
};
```

Constant descriptions

`gxOmitFaceLayerFillMask`
: The mask used to select the data compression bits for the `outlineFill` field descriptor.

`gxOmitFaceLayerFlagsMask`
: The mask used to select the data compression bits for the `flags` field descriptor.

`gxOmitFaceLayerStyleMask`
: The mask used to select the data compression bits for the `outlineStyle` field descriptor.

`gxOmitFaceLayerTransformMask`
: The mask used to select the data compression bits for the `outlineTransform` field descriptor.

Once one of the `gxOmitFaceLayerMask1` enumeration masks has been used to select a data compression opcode for a field descriptor in the `gxFaceLayer` structure, the corresponding bit shift from the `gxOmitFaceLayerShift1` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitFaceLayerShift1 {
   gxOmitFaceLayerFillShift      = 6,
   gxOmitFaceLayerFlagsShift     = 4,
   gxOmitFaceLayerStyleShift     = 2,
   gxOmitFaceLayerTransformShift = 0
};
```

Constant descriptions

`gxOmitFaceLayerFillShift`
: The bit shift required to isolate the compression bits for the `outlineFill` field descriptor.

`gxOmitFaceLayerFlagsShift`
: The bit shift required to isolate the compression bits for the `flags` field descriptor.

`gxOmitFaceLayerStyleShift`
: The bit shift required to isolate the compression bits for the `outlineStyle` field descriptor.

`gxOmitFaceLayerTransformShift`
: The bit shift required to isolate the compression bits for the `outlineTransform` field descriptor.

The `gxOmitFaceLayerMask2` enumeration defines which bits in a second omit byte correspond to the data compression bits for additional field descriptors in the `gxFaceLayer` structure. The sequence of data is also defined. The use of this mask and shift are described in the section “Text Face Data” on page 7-39.

```
enum gxOmitFaceLayerMask2 {
   gxOmitFaceLayerBoldXMask   = 0xC0,
   gxOmitFaceLayerBoldYMask   = 0x30
};
```

Constant descriptions

`gxOmitFaceLayerBoldXMask`
: The mask used to select the data compression bits for the `boldOutset.X` field descriptor.

`gxOmitFaceLayerBoldYMask`
: The mask used to select the data compression bits for the `boldOutset.Y` field descriptor.

Once one of the `gxOmitFaceLayerMask2` enumeration masks has been used to select a data compression opcode for a field descriptor in the `gxFaceLayer` structure, the corresponding bit shift from the `gxOmitFaceLayerShift2` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitFaceLayerShift2 {
   gxOmitFaceLayerBoldXShift  = 6,
   gxOmitFaceLayerBoldYShift  = 4
};
```

Constant descriptions

`gxOmitFaceLayerBoldXShift`
The bit shift required to isolate the compression bits for the `boldOutset.X` field descriptor.

`gxOmitFaceLayerBoldYShift`
The bit shift required to isolate the compression bits for the `boldOutset.Y` field descriptor.

Ink Object Omit Byte Constants and Data Types

This section describes the constants and data types that are used to interpret omit bytes that are used with ink object data. The use of omit bytes is described in the section "Omit Byte Masks and Omit Byte Shifts" beginning on page 7-22.

Colors Omit Byte Masks and Shifts

The `gxOmitColorsMask` enumeration defines which bits in an omit byte correspond to the data compression opcode for the field descriptors in the `gxColor` structure. The sequence of data is also defined. The omit byte and its related data sequence is given in the section "Color Data" beginning on page 7-44.

```
enum gxOmitColorsMask {
   gxOmitColorsSpaceMask      = 0xC0,
   gxOmitColorsProfileMask    = 0x30,
   gxOmitColorsComponentsMask = 0x0F,
   gxOmitColorsIndexMask      = 0x0C,
   gxOmitColorsIndexSetMask   = 0x03
};
```

Constant descriptions

`gxOmitColorsSpaceMask`
: The mask used to select the data compression bits for the `space` field descriptor.

`gxOmitColorsProfileMask`
: The mask used to select the data compression bits for the `profile` field descriptor.

`gxOmitColorsComponentsMask`
: The mask used to select the data compression bits for the `element.component[4]` field descriptor.

`gxOmitColorsIndexMask`
: The mask used to select the data compression bits for the `element.indexed.index` field descriptor.

`gxOmitColorsIndexSetMask`
: The mask used to select the data compression bits for the `element.index.Set` field descriptor.

Once one of the `gxOmitColorsMask` enumeration masks has been used to select a data compression opcode for a field descriptor in the `gxColor` structure, the corresponding bit shift from the `gxOmitColorsShift` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitColorsShift {
   gxOmitColorsSpaceShift        = 6,
   gxOmitColorsProfileShift      = 4,
   gxOmitColorsComponentsShift   = 0,
   gxOmitColorsIndexShift        = 2,
   gxOmitColorsIndexSetShift     = 0
};
```

Constant descriptions

`gxOmitColorsSpaceShift`
: The bit shift required to isolate the compression bits for the `space` field descriptor.

`gxOmitColorsProfileShift`
: The bit shift required to isolate the compression bits for the `profile` field descriptor.

`gxOmitColorsComponentsShift`
: The bit shift required to isolate the compression bits for the `element.component`[4] field descriptor.

`gxOmitColorsIndexShift`
: The bit shift required to isolate the compression bits for the `element.indexed.index` field descriptor.

`gxOmitColorsIndexSetShift`
: The bit shift required to isolate the compression bits for the `element.indexed.set` field descriptor.

Transfer Omit Byte Masks and Shifts

The `gxOmitTransferMask1` enumeration defines which bits in an omit byte correspond to the data compression opcode for the field descriptors in the `gxTransferMode` structure. The sequence of data is also defined. The omit byte and its related data sequence is given in the section "Transfer Mode Data" beginning on page 7-44.

```
enum gxOmitTransferMask1 {
   gxOmitTransferSpaceMask    = 0xC0,
   gxOmitTransferSetMask      = 0x30,
   gxOmitTransferProfileMask  = 0x0C
};
```

Constant descriptions

`gxOmitTransferSpaceMask`
: The mask used to select the data compression bits for the `space` field descriptor.

`gxOmitTransferSetMask`
: The mask used to select the data compression bits for the `set` field descriptor.

`gxOmitTransferProfileMask`
: The mask used to select the data compression bits for the `profile` field descriptor.

Once one of the `gxOmitTransferMask1` enumeration masks has been used to select a data compression opcode for a field descriptor in the `gxTransferMode` structure, the corresponding bit shift from the `gxOmitTransferShift1` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitTransferShift1 {
   gxOmitTransferSpaceShift   = 6,
   gxOmitTransferSetShift     = 4,
   gxOmitTransferProfileShift = 2
};
```

Constant descriptions

`gxOmitTransferSpaceShift`
: The bit shift required to isolate the compression bits for the `space` field descriptor.

`gxOmitTransferSetShift`
: The bit shift required to isolate the compression bits for the `set` field descriptor.

`gxOmitTransferProfileShift`
: The bit shift required to isolate the compression bits for the `profile` field descriptor.

The gxOmitTransferMask2 enumeration defines which bits in a second omit byte correspond to the data compression opcode for additional field descriptors in the gxTransferMode structure. The sequence of data is also defined. The omit byte and its related data sequence is given in the section "Transfer Mode Data" beginning on page 7-44.

```
enum gxOmitTransferMask2 {
   gxOmitTransferSourceMatrixMask= 0xC0,
   gxOmitTransferDeviceMatrixMask= 0x30,
   gxOmitTransferResultMatrixMask= 0x0C,
   gxOmitTransferFlagsMask       = 0x03
};
```

Constant descriptions

gxOmitTransferSourceMatrixMask
: The mask used to select the data compression bits for the sourceMatrix field descriptor.

gxOmitTransferDeviceMatrixMask
: The mask used to select the data compression bits for the deviceMatrix field descriptor.

gxOmitTransferResultMatrixMask
: The mask used to select the data compression bits for the resultMatrix field descriptor.

gxOmitTransferFlagsMask
: The mask used to select the data compression bits for the flags field descriptor.

Once one of the gxOmitTransferMask2 enumeration masks has been used to select a data compression opcode for a field descriptor in the gxTransferMode structure, the corresponding bit shift from the gxOmitTransferShift2 enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitTransferShift2 {

   gxOmitTransferSourceMatrixShift  = 6,
   gxOmitTransferDeviceMatrixShift  = 4,
   gxOmitTransferResultMatrixShift  = 2,
   gxOmitTransferFlagsShift         = 0
};
```

Constant descriptions

`gxOmitTransferSourceMatrixShift`
: The bit shift required to isolate the compression bits for the `sourceMatrix` field descriptor.

`gxOmitTransferDeviceMatrixShift`
: The bit shift required to isolate the compression bits for the `deviceMatrix` field descriptor.

`gxOmitTransferResultMatrixShift`
: The bit shift required to isolate the compression bits for the `resultMatrix` field descriptor.

`gxOmitTransferFlagsShift`
: The bit shift required to isolate the compression bits for the `flags` field descriptor.

Transfer Component Omit Byte Masks and Shifts

The `gxOmitTransferComponentMask1` enumeration defines which bits in an omit byte correspond to the data compression opcode for the field descriptors in the `gxTransferComponent` structure. The sequence of data is also defined. The omit byte and its related data sequence is given in the section “Transfer Mode Data” beginning on page 7-44.

```
enum gxOmitTransferComponentMask1 {
   gxOmitTransferComponentModeMask           = 0x80,
   gxOmitTransferComponentFlagsMask          = 0x40,
   gxOmitTransferComponentSourceMinimumMask  = 0x30,
   gxOmitTransferComponentSourceMaximumMask  = 0x0C,
   gxOmitTransferComponentDeviceMinimumMask  = 0x03
};
```

Constant descriptions

`gxOmitTransferComponentModeMask`
: The mask used to select the data compression bits for the `mode` field descriptor.

`gxOmitTransferComponentFlagsMask`
: The mask used to select the data compression bits for the `flags` field descriptor.

`gxOmitTransferComponentSourceMinimumMask`
: The mask used to select the data compression bits for the `sourceMinimum` field descriptor.

`gxOmitTransferComponentSourceMaximumMask`
: The mask used to select the data compression bits for the `sourceMaximum` field descriptor.

`gxOmitTransferComponentDeviceMinimumMask`
: The mask used to select the data compression bits for the `deviceMinimum` field descriptor.

Once one of the `gxOmitTransferComponentMask1` enumeration masks has been used to select a data compression opcode for a field descriptor in the `gxTransferComponent` structure, the corresponding bit shift from the `gxOmitTransferComponentShift1` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitTransferComponentShift1 {
   gxOmitTransferComponentModeShift          = 7,
   gxOmitTransferComponentFlagsShift         = 6,
   gxOmitTransferComponentSourceMinimumShift = 4,
   gxOmitTransferComponentSourceMaximumShift = 2,
   gxOmitTransferComponentDeviceMinimumShift = 0
};
```

Constant descriptions

`gxOmitTransferComponentModeShift`	The bit shift required to isolate the compression bits for the `mode` field descriptor.
`gxOmitTransferComponentFlagsShift`	The bit shift required to isolate the compression bits for the `flags` field descriptor.
`gxOmitTransferComponentSourceMinimumShift`	The bit shift required to isolate the compression bits for the `sourceMinimum` field descriptor.
`gxOmitTransferComponentSourceMaximumShift`	The bit shift required to isolate the compression bits for the `sourceMaximum` field descriptor.
`gxOmitTransferComponentDeviceMinimumShift`	The bit shift required to isolate the compression bits for the `deviceMinimum` field descriptor.

The `gxOmitTransferComponentMask2` enumeration defines which bits in a second omit byte correspond to the data compression opcode for additional field descriptors in the `gxTransferComponent` structure. The sequence of data is also continued. The omit byte and its related data sequence is given in the section “Transfer Mode Data” beginning on page 7-44.

```
enum gxOmitTransferComponentMask2 {
   gxOmitTransferComponentDeviceMaximumMask  = 0xC0,
   gxOmitTransferComponentClampMinimumMask   = 0x30,
   gxOmitTransferComponentClampMaximumMask   = 0x0C,
   gxOmitTransferComponentOperandMask        = 0x03
};
```

Constant descriptions

`gxOmitTransferComponentDeviceMaximumMask`
: The mask used to select the data compression bits for the `deviceMaximum` field descriptor.

`gxOmitTransferComponentClampMinimumMask`
: The mask used to select the data compression bits for the `clampMinimum` field descriptor.

`gxOmitTransferComponentClampMaximumMask`
: The mask used to select the data compression bits for the `clampMaximum` field descriptor.

`gxOmitTransferComponentOperandMask`
: The mask used to select the data compression bits for the `operand` field descriptor.

Once one of the `gxOmitTransferComponentMask2` enumeration masks has been used to select a data compression opcode for a field descriptor in the `gxTransferComponent` structure, the corresponding bit shift from the `gxOmitTransferComponentShift2` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitTransferComponentShift2 {
   gxOmitTransferComponentDeviceMaximumShift = 6,
   gxOmitTransferComponentClampMinimumShift  = 4,
   gxOmitTransferComponentClampMaximumShift  = 2,
   gxOmitTransferComponentOperandShift       = 0
};
```

Constant descriptions

`gxOmitTransferComponentDeviceMaximumShift`
: The bit shift required to isolate the compression bits for the `deviceMaximum` field descriptor.

`gxOmitTransferComponentClampMinimumShift`
: The bit shift required to isolate the compression bits for the `clampMinimum` field descriptor.

`gxOmitTransferComponentClampMaximumShift`
: The bit shift required to isolate the compression bits for the `clampMaximum` field descriptor.

`gxOmitTransferComponentOperandShift`
: The bit shift required to isolate the compression bits for the `operand` field descriptor.

Shape Object Omit Byte Constants and Data Types

This section describes the constants and data types that are used to interpret omit bytes that are used with shape object data. The use of omit bytes is described in the section "Omit Byte Masks and Omit Byte Shifts" beginning on page 7-22.

Path Shape Omit Byte Masks and Shifts

The `gxOmitPathMask` enumeration defines which bits in an omit byte correspond to the data compression opcode for the field descriptors in the `gxPaths` structure. The sequence of data is also defined. The omit byte and its related data sequence is given in the section "Path Shape Data" beginning on page 7-31.

```
enum gxOmitPathMask {
   gxOmitPathPositionXMask = 0xC0,
   gxOmitPathPositionYMask = 0x30,
   gxOmitPathDeltaXMask    = 0x0C,
   gxOmitPathDeltaYMask    = 0x03
};
```

Constant descriptions

`gxOmitPathPositionXMask`
: The mask used to select the data compression bits for the `vector[0].x` field descriptor.

`gxOmitPathPositionYMask`
: The mask used to select the data compression bits for the `vector[0].y` field descriptor.

`gxOmitPathDeltaXMask`
: The mask used to select the data compression bits for the `vector[`*n*`].x` field descriptor where *n* is greater than zero, represented as a delta from the previous value.

`gxOmitPathDeltaYMask`
: The mask used to select the data compression bits for the `vector[`*n*`].y` field descriptor where *n* is greater than zero, represented as a delta from the previous value.

Once one of the `gxOmitPathMask` enumeration masks has been used to select a data compression opcode for a field descriptor in the `gxPaths??` structure, the corresponding bit shift from the `gxOmitPathShift` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitPathShift {
   gxOmitPathPositionXShift   = 6,
   gxOmitPathPositionYShift   = 4,
   gxOmitPathDeltaXShift      = 2,
   gxOmitPathDeltaYShift      = 0
};
```

Constant descriptions

`gxOmitPathPositionXShift`
: The bit shift required to isolate the compression bits for the`vector[0].x` field descriptor.

`gxOmitPathPositionYShift`
: The bit shift required to isolate the compression bits for the`vector[0].y` field descriptor.

`gxOmitPathDeltaXShift`
: The bit shift required to isolate the compression bits for the `vector[`*n*`].x` field descriptor where *n* is greater than zero, represented as a delta from the previous value.

`gxOmitPathDeltaYShift`
: The bit shift required to isolate the compression bits for the `vector[`*n*`].y` field descriptor where *n* is greater than zero, represented as a delta from the previous value.

Bitmap Shape Omit Byte Masks and Shifts

The `gxOmitBitmapMask1` enumeration defines which bits in an omit byte correspond to the data compression opcode for the field descriptors in the `gxBitmap` structure. The sequence of data is also defined. The omit byte and its related data sequence is given in the section “Bitmap Shape Data” beginning on page 7-32.

```
enum gxOmitBitmapMask1 {
   gxOmitBitmapImageMask      = 0xC0,
   gxOmitBitmapWidthMask      = 0x30,
   gxOmitBitmapHeightMask     = 0x0C,
   gxOmitBitmapRowBytesMask   = 0x03
};
```

Constant descriptions

`gxOmitBitmapImageMask`
: The mask used to select the data compression bits for the `image` field descriptor.

`gxOmitBitmapWidthMask`
: The mask used to select the data compression bits for the `width` field descriptor.

`gxOmitBitmapHeightMask`
: The mask used to select the data compression bits for the `height` field descriptor.

`gxOmitBitmapRowBytesMask`
: The mask used to select the data compression bits for the `rowBytes` field descriptor.

Once one of the `gxOmitBitmapMask1` enumeration masks has been used to select a data compression opcode for a field descriptor in the `gxBitmap` structure, the corresponding bit shift from the `gxOmitBitmapShift1` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitBitmapShift1 {
   gxOmitBitmapImageShift     = 6,
   gxOmitBitmapWidthShift     = 4,
   gxOmitBitmapHeightShift    = 2,
   gxOmitBitmapRowBytesShift  = 0
};
```

Constant descriptions

`gxOmitBitmapImageShift`
: The bit shift required to isolate the compression bits for the `image` field descriptor.

`gxOmitBitmapWidthShift`
: The bit shift required to isolate the compression bits for the `width` field descriptor.

`gxOmitBitmapHeightShift`
: The bit shift required to isolate the compression bits for the `height` field descriptor.

`gxOmitBitmapRowBytesShift`
: The bit shift required to isolate the compression bits for the `rowBytes` field descriptor.

The `gxOmitBitmapMask2` enumeration defines which bits in a second omit byte correspond to the data compression opcode for additional field descriptors in the `gxBitmap` structure. The sequence of data is also defined. The omit byte and its related data sequence is given in the section “Bitmap Shape Data” beginning on page 7-32.

```
enum gxOmitBitmapMask2 {
   gxOmitBitmapPixelSizeMask  = 0xC0,
   gxOmitBitmapSpaceMask      = 0x30,
   gxOmitBitmapSetMask        = 0x0C,
   gxOmitBitmapProfileMask    = 0x03
};
```

Constant descriptions

`gxOmitBitmapPixelSizeMask`
The mask used to select the data compression bits for the `pixelSize` field descriptor.

`gxOmitBitmapSpaceMask`
The mask used to select the data compression bits for the `space` field descriptor.

`gxOmitBitmapSetMask`
The mask used to select the data compression bits for the `set` field descriptor.

`gxOmitBitmapProfileMask`
The mask used to select the data compression bits for the `profile` field descriptor.

Once one of the `gxOmitBitmapMask2` enumeration masks has been used to select a data compression opcode for a field descriptor in the `gxBitmap` structure, the corresponding bit shift from the `gxOmitBitmapShift2` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitBitmapShift2 {
   gxOmitBitmapPixelSizeShift = 6,
   gxOmitBitmapSpaceShift     = 4,
   gxOmitBitmapSetShift       = 2,
   gxOmitBitmapProfileShift   = 0
};
```

Constant descriptions

`gxOmitBitmapPixelSizeShift`
The bit shift required to isolate the compression bits for the `pixelSize` field descriptor.

`gxOmitBitmapSpaceShift`
The bit shift required to isolate the compression bits for the `space` field descriptor.

`gxOmitBitmapSetShift`
: The bit shift required to isolate the compression bits for the `set` field descriptor.

`gxOmitBitmapProfileShift`
: The bit shift required to isolate the compression bits for the `profile` field descriptor.

The `gxOmitBitmapMask3` enumeration defines which bits in a third omit byte correspond to the data compression opcode for additional field descriptors in the `gxBitmap` structure. The sequence of data is also defined. The omit byte and its related data sequence is given in the section "Bitmap Shape Data" beginning on page 7-32.

```
enum gxOmitBitmapMask3 {
   gxOmitBitmapPositionXMask = 0xC0,
   gxOmitBitmapPositionYMask = 0x30
};
```

Constant descriptions

`gxOmitBitmapPositionXMask`
: The mask used to select the data compression bits for the `positionX` field descriptor.

`gxOmitBitmapPositionYMask`
: The mask used to select the data compression bits for the `positionY` field descriptor.

Once one of the `gxOmitBitmapMask3` enumeration masks has been used to select a data compression opcode for a field descriptor in the `gxBitmap` structure, the corresponding bit shift from the `gxOmitBitmapShift3` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitBitmapShift3 {
   gxOmitBitmapPositionXShift = 6,
   gxOmitBitmapPositionYShift = 4
};
```

Constant descriptions

`gxOmitBitmapPositionXShift`
: The bit shift required to isolate the compression bits for the `positionX` field descriptor.

`gxOmitBitmapPositionYShift`
: The bit shift required to isolate the compression bits for the `positionY` field descriptor.

Bit Image Omit Byte Masks and Shifts

The `gxOmitBitImageMask` enumeration defines which bits in an omit byte correspond to the data compression opcode for additional field descriptors. The sequence of data is also defined. The omit byte and its related data sequence is given in the section “New Bit Image Object Data” on page 7-49.

```
enum gxOmitBitImageMask {
   gxOmitBitImageRowBytesMask = 0xC0,
   gxOmitBitImageHeightMask   = 0x30,
   gxOmitBitImageDataMask     = 0x08
};
```

Constant descriptions

`gxOmitBitImageRowBytesMask`
: The mask used to select the data compression bits for the `rowBytes` field descriptor.

`gxOmitBitImageHeightMask`
: The mask used to select the data compression bits for the height.

`gxOmitBitImageDataMask`
: The mask used to select the data compression bits for the image.

Once one of the `gxOmitBitImageMask` enumeration masks has been used to select a data compression opcode for a field descriptor in the `gxBitmap` structure, the corresponding bit shift from the `gxOmitBitImageShift` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitBitImageShift {
   gxOmitBitImageRowBytesShift   = 6,
   gxOmitBitImageHeightShift     = 4,
   gxOmitBitImageDataShift       = 3
};
```

Constant descriptions

`gxOmitBitImageRowBytesShift`
: The bit shift required to isolate the compression bits for the `rowBytes` field descriptor.

`gxOmitBitImageHeightShift`
: The bit shift required to isolate the compression bits for the height.

`gxOmitBitImageDataShift`
: The bit shift required to isolate the compression bits for the image.

Text Shape Omit Byte Masks and Shifts

The `gxOmitTextMask` enumeration defines which bits in an omit byte correspond to the data compression opcode for parameters of the `GXNewText` function. The sequence of data is also defined. The omit byte and its related data sequence is given in the section "Text Shape Data" beginning on page 7-32.

```
enum gxOmitTextMask {
   gxOmitTextCharactersMask   = 0xC0,
   gxOmitTextPositionXMask    = 0x30,
   gxOmitTextPositionYMask    = 0x0C,
   gxOmitTextDataMask         = 0x02
};
```

Constant descriptions

`gxOmitTextCharactersMask`
: The mask used to select the data compression bits for the `charCount` parameter.

`gxOmitTextPositionXMask`
: The mask used to select the data compression bits for the `position.X` parameter.

`gxOmitTextPositionYMask`
: The mask used to select the data compression bits for the `position` .Y parameter.

`gxOmitTextDataMask`
: The mask used to select the data compression bits for the `text` parameter.

Once one of the `gxOmitText`Mask enumeration masks has been used to select a data compression opcode for the parameters of the `GXNewText` function, the corresponding bit shift from the `gxOmitTextShift` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitTextShift {
   gxOmitTextCharactersShift  = 6,
   gxOmitTextPositionXShift   = 4,
   gxOmitTextPositionYShift   = 2,
   gxOmitTextDataShift        = 1
};
```

Constant descriptions

`gxOmitTextCharactersShift`
: The bit shift required to isolate the compression bits for the `charCount` field descriptor.

`gxOmitTextPositionXShift`
: The bit shift required to isolate the compression bits for the `position.X` field descriptor.

`gxOmitTextPositionYShift`
: The bit shift required to isolate the compression bits for the `position`.Y field descriptor.

`gxOmitTextDataShift`
: The bit shift required to isolate the compression bits for the `text` field descriptor.

Glyph Shape Omit Byte Masks and Shifts

The `gxOmitGlyphMask1` enumeration defines which bits in an omit byte correspond to the data compression opcode for additional field descriptors in the `gx NewGlyphs` structure. The sequence of data is also defined. The omit byte and its related data sequence is given in the section “Glyph Shape Data” beginning on page 7-33.

```
enum gxOmitGlyphMask1 {
   gxOmitGlyphCharactersMask  = 0xC0,
   gxOmitGlyphLengthMask      = 0x30,
   gxOmitGlyphRunNumberMask   = 0x0C,
   gxOmitGlyphOnePositionMask = 0x02,
   gxOmitGlyphDataMask        = 0x01
};
```

Constant descriptions

`gxOmitGlyphCharactersMask`
: The mask used to select the data compression bits for the `charCount` function parameter.

`gxOmitGlyphLengthMask`
: The mask used to select the data compression bits for the length in bytes of the data.

`gxOmitGlyphRunNumberMask`
: The mask used to select the data compression bits for the number of `styleRuns`.

`gxOmitGlyphOnePositionMask`
: The mask used to specify that the position can be represented with one point.

`gxOmitGlyphDataMask`
: The mask used to select the data compression bits for the `text` function parameter.

Once one of the gxOmitGlyphMask1 enumeration masks has been used to select a data compression opcode for the parameters to GXNewGlyphs function, the corresponding bit shift from the gxOmitGlyphShift1 enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitGlyphShift1 {
   gxOmitGlyphCharactersShift    = 6,
   gxOmitGlyphLengthShift        = 4,
   gxOmitGlyphRunNumberShift     = 2,
   gxOmitGlyphOnePositionShift   = 1,
   gxOmitGlyphDataShift          = 0
};
```

Constant descriptions

gxOmitGlyphCharactersShift
: The bit shift required to isolate the compression bits for the charCount function parameter.

gxOmitGlyphLengthShift
: The bit shift required to isolate the compression bits for the length in bytes of the data.

gxOmitGlyphRunNumberShift
: The bit shift required to isolate the compression bits for the number of styleRuns.

gxOmitGlyphOnePositionShift
: The bit shift required to specify that the position can be represented with 1 point.

gxOmitGlyphDataShift
: The bit shift required to isolate the compression bits for the text function parameter.

The gxOmitGlyphMask2 enumeration defines which bits in an omit byte correspond to the data compression opcode for the parameters of the GXNewGlyphs function. The sequence of data is also defined. The omit byte and its related data sequence is given in the section “Glyph Shape Data” beginning on page 7-33.

```
enum gxOmitGlyphMask2 {
   gxOmitGlyphPositionsMask   = 0xC0,
   gxOmitGlyphAdvancesMask    = 0x20,
   gxOmitGlyphTangentsMask    = 0x18,
   gxOmitGlyphRunsMask        = 0x04,
   gxOmitGlyphStylesMask      = 0x03
};
```

Constant descriptions

`gxOmitGlyphPositionsMask`
: The mask used to select the data compression bits for the `positions` function parameter.

`gxOmitGlyphAdvancesMask`
: The mask used to select the data compression bits for the `advance` function parameter.

`gxOmitGlyphTangentsMask`
: The mask used to select the data compression bits for the `tangents` function parameter.

`gxOmitGlyphRunsMask`
: The mask used to select the data compression bits for the `styleRuns` function parameter.

`gxOmitGlyphStylesMask`
: The mask used to select the data compression bits for the `glyphStyles` function parameter.

Once one of the `gxOmitGlyphMask2` enumeration masks has been used to select a data compression opcode for the parameters to the `GXNewGlyph` function, the corresponding bit shift from the `gxOmitGlyphShift2` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitGlyphShift2 {
   gxOmitGlyphPositionsShift  = 6,
   gxOmitGlyphAdvancesShift   = 5,
   gxOmitGlyphTangentsShift   = 3,
   gxOmitGlyphRunsShift       = 2,
   gxOmitGlyphStylesShift     = 0
};
```

Constant descriptions

`gxOmitGlyphPositionsShift`
: The bit shift required to isolate the compression bits for the `positions` function parameter.

`gxOmitGlyphAdvancesShift`
: The bit shift required to isolate the compression bits for the `advance` function parameter.

`gxOmitGlyphTangentsShift`
: The bit shift required to isolate the compression bits for the `tangents` function parameter.

`gxOmitGlyphRunsShift`
: The bit shift required to isolate the compression bits for the `styleRuns` function parameter.

`gxOmitGlyphStylesShift`
: The bit shift required to isolate the compression bits for the `glyphStyles` function parameter.

Layout Shape Omit Byte Masks and Shifts

The `gxOmitLayoutMask1` enumeration defines which bits in an omit byte correspond to the data compression opcode for parameters for the `GXNewLayout` function. The sequence of data is also defined. The omit byte and its related data sequence is given in the section "Layout Shape Data" beginning on page 7-33.

```
enum gxOmitLayoutMask1 {
   gxOmitLayoutLengthMask     = 0xC0,
   gxOmitLayoutPositionXMask  = 0x30,
   gxOmitLayoutPositionYMask  = 0x0C,
   gxOmitLayoutDataMask       = 0x02
};
```

Constant descriptions

`gxOmitLayoutLengthMask`
: The mask used to select the data compression bits for the `textRunLength` parameter.

`gxOmitLayoutPositionXMask`
: The mask used to select the data compression bits for the `position.X` parameter.

`gxOmitLayoutPositionYMask`
: The mask used to select the data compression bits for the `position.Y` parameters.

`gxOmitLayoutDataMask`
: The mask used to select the data compression bits for the `text` parameter.

Once one of the `gxOmitLayoutMask1` enumeration masks has been used to select a data compression opcode for the parameters for the `GXNewLayout` function, the corresponding bit shift from the `gxOmitLayoutShift1` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitLayoutShift1 {
   gxOmitLayoutLengthShift    = 6,
   gxOmitLayoutPositionXShift = 4,
   gxOmitLayoutPositionYShift = 2,
   gxOmitLayoutDataShift      = 1
};
```

Constant descriptions

`gxOmitLayoutLengthShift`
: The bit shift required to isolate the compression bits for the `textRunLength` parameter.

`gxOmitLayoutPositionXShift`
: The bit shift required to isolate the compression bits for the `position.X` parameter.

`gxOmitLayoutPositionYShift`
: The bit shift required to isolate the compression bits for the `position.Y` parameter.

`gxOmitLayoutDataShift`
: The bit shift required to isolate the compression bits for the `text` parameter descriptor.

The `gxOmitLayoutMask2` enumeration defines which bits in a second omit byte correspond to the data compression opcode for additional parameters for the `GXNewLayout` function. The sequence of data is also defined. The omit byte and its related data sequence is given in the section "Layout Shape Data" beginning on page 7-33.

```
enum gxOmitLayoutMask2 {
   gxOmitLayoutWidthMask   = 0xC0,
   gxOmitLayoutFlushMask   = 0x30,
   gxOmitLayoutJustMask    = 0x0C,
   gxOmitLayoutOptionsMask = 0x03
};
```

Constant descriptions

`gxOmitLayoutWidthMask`
: The mask used to select the data compression bits for the `width` field descriptor.

`gxOmitLayoutFlushMask`
: The mask used to select the data compression bits for the `flush` field descriptor.

`gxOmitLayoutJustMask`
: The mask used to select the data compression bits for the `just` field descriptor.

`gxOmitLayoutOptionsMask`
: The mask used to select the data compression bits for the `flags` field descriptor.

Once one of the gxOmitLayoutMask2 enumeration masks has been used to select a data compression opcode for the parameters for the GXNewLayout function, the corresponding bit shift from the gxOmitLayoutShift2 enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitLayoutShift2 {
   gxOmitLayoutWidthShift     = 6,
   gxOmitLayoutFlushShift     = 4,
   gxOmitLayoutJustShift      = 2,
   gxOmitLayoutOptionsShift   = 0
};
```

Constant descriptions

gxOmitLayoutWidthShift
: The bit shift required to isolate the compression bits for the width field descriptor.

gxOmitLayoutFlushShift
: The bit shift required to isolate the compression bits for the flush field descriptor.

gxOmitLayoutJustShift
: The bit shift required to isolate the compression bits for the just field descriptor.

gxOmitLayoutOptionsShift
: The bit shift required to isolate the compression bits for the flags field descriptor.

The gxOmitLayoutMask3 enumeration defines which bits in a third omit byte correspond to the data compression opcode for additional parameters for the GXNewLayout function. The sequence of data is also defined. The omit byte and its related data sequence is given in the section “Layout Shape Data” beginning on page 7-33.

```
enum gxOmitLayoutMask3 {
   gxOmitLayoutStyleRunNumberMask= 0xC0,
   gxOmitLayoutLevelRunNumberMask= 0x30,
   gxOmitLayoutHasBaselineMask   = 0x08,
   gxOmitLayoutStyleRunsMask     = 0x04,
   gxOmitLayoutStylesMask        = 0x03
};
```

Constant descriptions

`gxOmitLayoutStyleRunNumberMask`
: The mask used to select the data compression bits for the `styleRunCount` field descriptor.

`gxOmitLayoutLevelRunNumberMask`
: The mask used to select the data compression bits for the `levelRunCount` field descriptor.

`gxOmitLayoutHasBaselineMask`
: The mask used to select the data compression bits for the `hasBaseline` field descriptor.

`gxOmitLayoutStyleRunsMask`
: The mask used to select the data compression bits for the `styleRunLengths` field descriptor.

`gxOmitLayoutStylesMask`
: The mask used to select the data compression bits for the ??? field descriptor.

Once one of the `gxOmitLayoutMask3` enumeration masks has been used to select a data compression opcode for the parameters for the `GXNewLayout` function, the corresponding bit shift from the `gxOmitLayoutShift3` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitLayoutShift3 {
   gxOmitLayoutStyleRunNumberShift  = 6,
   gxOmitLayoutLevelRunNumberShift  = 4,
   gxOmitLayoutHasBaselineShift     = 3,
   gxOmitLayoutStyleRunsShift       = 2,
   gxOmitLayoutStylesShift          = 0
};
```

Constant descriptions

`gxOmitLayoutStyleRunNumberShift`
: The bit shift required to isolate the compression bits for the `styleRunCount` field descriptor.

`gxOmitLayoutLevelRunNumberShift`
: The bit shift required to isolate the compression bits for the `levelRunCount` field descriptor.

`gxOmitLayoutHasBaselineShift`
: The bit shift required to isolate the compression bits for the `hasBaseline` field descriptor.

`gxOmitLayoutStyleRunsShift`
: The bit shift required to isolate the compression bits for the `styleRunLengths` field descriptor.

`gxOmitLayoutStylesShift`
: The bit shift required to isolate the compression bits for the ??? field descriptor.

The `gxOmitLayoutMask4` enumeration defines which bits in a fourth omit byte correspond to the data compression opcode for additional parameters for the `GXNewLayout` function. The sequence of data is also defined. The omit byte and its related data sequence is given in the section "Layout Shape Data" beginning on page 7-33.

```
enum gxOmitLayoutMask4 {

   gxOmitLayoutLevelRunsMask      = 0x80,
   gxOmitLayoutLevelsMask         = 0x40
};
```

Constant descriptions

`gxOmitLayoutLevelRunsMask`
: The mask used to select the data compression bits for the `levelRunLengths` parameter.

`gxOmitLayoutLevelsMask`
: The mask used to select the data compression bits for the `levels` parameter.

Once one of the `gxOmitLayoutMask4` enumeration masks has been used to a select data compression opcode for the parameters for the `GXNewLayout` function, the corresponding bit shift from the `gxOmitLayoutShift4` enumeration can be applied to the selected bits. The selected bits must be moved to the right by the indicated number of bits to isolate the data compression opcode so that it can be compared to other values.

```
enum gxOmitLayoutShift4 {
   gxOmitLayoutLevelRunsShift = 7,
   gxOmitLayoutLevelsShift    = 6
};
```

Constant descriptions

`gxOmitLayoutLevelRunsShift`
: The bit shift required to isolate the compression bits for the `levelRunLengths` parameter.

`gxOmitLayoutLevelsShift`
: The bit shift required to isolate the compression bits for the `levels` parameter.

Picture Shape Omit Byte Masks and Shifts

The `gxOmitPictureParametersMask` enumeration defines which bits in an omit byte correspond to the data compression opcode for parameters of the `GXDrawPicture` function. The sequence of data is also defined. The omit byte and its related data sequence is given in the section "Picture Shape Data" beginning on page 7-34.

```
enum gxOmitPictureParametersMask {
   gxOmitPictureShapeMask        = 0xC0,
   gxOmitOverrideStyleMask       = 0x30,
   gxOmitOverrideInkMask         = 0x0C,
   gxOmitOverrideTransformMask   = 0x03
};
```

Constant descriptions

`gxOmitPictureShapeMask`
: The mask used to select the data compression bits for the `shapes` parameter.

`gxOmitOverrideStyleMask`
: The mask used to select the data compression bits for the `styles` parameter.

`gxOmitOverrideInkMask`
: The mask used to select the data compression bits for the `inks` parameter.

`gxOmitOverrideTransformMask`
: The mask used to select the data compression bits for the `transforms` parameter.

```
enum gxOmitPictureParametersShift {
   gxOmitPictureShapeShift       = 0x6,
   gxOmitOverrideStyleShift      = 0x4,
   gxOmitOverrideInkShift        = 0x2,
   gxOmitOverrideTransformShift  = 0x0
};
```

Constant descriptions

`gxOmitPictureShapeShift`
: The bit shift required to isolate the compression bits for the `shapes` parameter.

`gxOmitOverrideStyleShift`
: The bit shift required to isolate the compression bits for the `styles` parameter.

`gxOmitOverrideInkShift`
: The bit shift required to isolate the compression bits for the `inks` parameter.

`gxOmitOverrideTransformShift`
: The bit shift required to isolate the compression bits for the `transforms` parameter.

QuickDraw GX Stream Format Summary

Opcode Constants and Data Types

Operation Opcode Byte

```
enum gxGraphicsOperationOpcode {
   gxNewObjectOpcode = 0x00,
   gxSetDataOpcode   = 0x40,
   gxSetDefaultOpcode= 0x80,
   gxReservedOpcode  = 0xC0,
   gxNextOpcode      = 0xFF,
};
```

Data Type Opcode Byte

```
enum gxGraphicsNewOpcode {
   gxHeaderTypeOpcode      = 0x00,
   gxStyleTypeOpcode    = 0x28,
   gxInkTypeOpcode,
   gxTransformTypeOpcode,
   gxColorProfileTypeOpcode,
   gxColorSetTypeOpcode,
   gxTagTypeOpcode,
   gxBitImageOpcode,
   gxFontNameTypeOpcode,
   gxTrailerTypeOpcode,
};
```

Generic Data Opcode

```
enum gxGenericDataOpcode {
   gxTypeOpcode,
   gxSizeOpcode
};                      /* constants used by current operand when
                         current operation is gxNextOpcode */
#define gxCompressionShift       6
#define gxObjectTypeMask      0x3F
```

```
#define gxBitImageOpcodeMask  0xC0
#define gxBitImageCountMask   0x3F
#define gxBitImageOpcodeShift    6
```

Modified Shape Data Opcodes

```
enum gxShapeDataOpcode {
   gxShapeAttributesOpcode,
   gxShapeTagOpcode,
   gxShapeFillOpcode
};
```

Modified Style Data Opcodes

```
enum gxStyleDataOpcode {
   gxStyleAttributesOpcode,
   gxStyleTagOpcode,
   gxStyleCurveErrorOpcode,
   gxStylePenOpcode,
   gxStyleJoinOpcode,
   gxStyleDashOpcode,
   gxStyleCapsOpcode,
   gxStylePatternOpcode,
   gxStyleTextAttributesOpcode,
   gxStyleTextSizeOpcode,
   gxStyleFontOpcode,
   gxStyleTextFaceOpcode,
   gxStylePlatformOpcode,
   gxStyleFontVariationsOpcode,
   #ifdef gxLayoutStyleRuns
   gxStyleRunControlsOpcode,
   gxStyleRunPriorityJustOverrideOpcode,
   gxStyleRunGlyphJustOverridesOpcode,
   gxStyleRunGlyphSubstitutionsOpcode,
   gxStyleRunFeaturesOpcode,
   gxStyleRunKerningAdjustmentsOpcode,
   gxStyleLayoutInfoOpcode,
   gxStyleJustificationOpcode
};
```

Modified Ink Data Opcodes

```
enum gxInkDataOpcode {
   gxInkAttributesOpcode,
   gxInkTagOpcode,
   gxInkColorOpcode,
   gxInkTransferModeOpcode
};
```

Modified Color Set Data Opcodes

```
enum gxColorSetDataOpcode {
   gxColorSetReservedOpcode,
   gxColorSetTagOpcode
};
```

Modified Color Profile Data Opcodes

```
enum gxProfileDataOpcode {
   gxColorProfileAttributesOpcode,
   gxColorProfileTagOpcode
};
```

Modified Transform Data Opcodes

```
enum gxTransformDataOpcode {
   gxTransformReservedOpcode,
   gxTransformTagOpcode,
   gxTransformClipOpcode,
   gxTransformMappingOpcode,
   gxTransformPartMaskOpcode,
   gxTransformToleranceOpcode
};
```

Bit Image Compression Opcodes

```
enum gxBitImageCompression {
   gxCopyBitImageBytesOpcode  = 0x00,
   gxRepeatBitImageBytesOpcode= 0x40,
   gxLookupBitImageBytesOpcode= 0x80,
   gxRepeatBitImageScanOpcode = 0xC0
};
```

Two Bit Compression Values

```
enum gxTwoBitCompressionValues {
   gxNoCompression,     = 0x00
   gxWordCompression,   = 0x40
   gxByteCompression,   = 0x80
   gxOmitCompression =  = 0x??
};
```

Flatten Header Bytes

```
struct gxFlattenHeader {
   fixed          version;
   unsigned char  flatFlags;
};
```

Style Object Omit Byte Constants and Data Types

Dash Style Omit Byte Masks and Shifts

```
enum gxOmitDashMask1 {
   gxOmitDashAttributesMask   = 0xC0,
   gxOmitDashShapeMask        = 0x30,
   gxOmitDashAdvanceMask      = 0x0C,
   gxOmitDashPhaseMask        = 0x03
};

enum gxOmitDashShift1 {
   gxOmitDashAttributesShift  = 6,
   gxOmitDashShapeShift       = 4,
   gxOmitDashAdvanceShift     = 2,
   gxOmitDashPhaseShift       = 0
};

enum gxOmitDashMask2 {
   gxOmitDashScaleMask        =  0xC0
};

enum gxOmitDashShift2 {
      gxOmitDashScaleShift    = 6
   };
```

Pattern Style Omit Byte Masks and Shifts

```
enum gxOmitPatternMask1 {

      gxOmitPatternAttributesMask   = 0xC0,
      gxOmitPatternShapeMask        = 0x30,
      gxOmitPatternUXMask           = 0x0C,
      gxOmitPatternUYMask           = 0x03
   };

enum gxOmitPatternShift1 {
   gxOmitPatternAttributesShift  = 6,
   gxOmitPatternShapeShift       = 4,
   gxOmitPatternUXShift          = 2,
   gxOmitPatternUYShift          = 0
   };

enum gxOmitPatternMask2 {
   gxOmitPatternVXMask = 0xC0,
   gxOmitPatternVYMask = 0x30
};

enum gxOmitPatternShift2 {
   gxOmitPatternVXShift= 6,
   gxOmitPatternVYShift= 4
};
```

Join Style Omit Byte Masks and Shifts

```
enum gxOmitJoinMask {
   gxOmitJoinAttributesMask= 0xC0,
   gxOmitJoinShapeMask     = 0x30,
   gxOmitJoinMiterMask     = 0x0C
};

enum gxOmitJoinShift {
   gxOmitJoinAttributesShift  = 6,
   gxOmitJoinShapeShift       = 4,
   gxOmitJoinMiterShift       = 2
};
```

Cap Style Omit Byte Masks and Shifts

```
enum gxOmitCapMask {
   gxOmitCapAttributesMask = 0xC0,
   gxOmitCapStartShapeMask = 0x30,
   gxOmitCapEndShapeMask   = 0x0C
};

enum gxOmitCapShift {
   gxOmitCapAttributesShift= 6,
   gxOmitCapStartShapeShift= 4,
   gxOmitCapEndShapeShift  = 2
};
```

Text Face Style Omit Byte Masks and Shifts

```
enum gxOmitFaceMask {
   gxOmitFaceLayersMask = 0xC0,
   gxOmitFaceMappingMask= 0x30
};

enum gxOmitFaceShift {
   gxOmitFaceLayersShift = 6,
   gxOmitFaceMappingShift= 4
};
```

Face Layer Omit Byte Masks and Shifts

```
enum gxOmitFaceLayerMask1 {
   gxOmitFaceLayerFillMask       = 0xC0,
   gxOmitFaceLayerFlagsMask      = 0x30,
   gxOmitFaceLayerStyleMask      = 0x0C,
   gxOmitFaceLayerTransformMask  = 0x03
};

enum gxOmitFaceLayerShift1 {
   gxOmitFaceLayerFillShift      = 6,
   gxOmitFaceLayerFlagsShift     = 4,
   gxOmitFaceLayerStyleShift     = 2,
   gxOmitFaceLayerTransformShift = 0
};
```

```
enum gxOmitFaceLayerMask2 {
   gxOmitFaceLayerBoldXMask   = 0xC0,
   gxOmitFaceLayerBoldYMask   = 0x30
};

enum gxOmitFaceLayerShift2 {
   gxOmitFaceLayerBoldXShift  = 6,
   gxOmitFaceLayerBoldYShift  = 4
};
```

Ink Object Omit Byte Constants and Data Types

Colors Omit Byte Masks and Shifts

```
enum gxOmitColorsMask {

   gxOmitColorsSpaceMask      = 0xC0,
   gxOmitColorsProfileMask    = 0x30,
   gxOmitColorsComponentsMask = 0x0F,
   gxOmitColorsIndexMask      = 0x0C,
   gxOmitColorsIndexSetMask   = 0x03
};

enum gxOmitColorsShift {
   gxOmitColorsSpaceShift        = 6,
   gxOmitColorsProfileShift      = 4,
   gxOmitColorsComponentsShift   = 0,
   gxOmitColorsIndexShift        = 2,
   gxOmitColorsIndexSetShift     = 0
};
```

Transfer Omit Byte Masks and Shifts

```
enum gxOmitTransferMask1 {
   gxOmitTransferSpaceMask    = 0xC0,
   gxOmitTransferSetMask      = 0x30,
   gxOmitTransferProfileMask  = 0x0C
};

enum gxOmitTransferShift1 {
   gxOmitTransferSpaceShift   = 6,
   gxOmitTransferSetShift     = 4,
   gxOmitTransferProfileShift = 2
};
```

```
enum gxOmitTransferMask2 {
   gxOmitTransferSourceMatrixMask= 0xC0,
   gxOmitTransferDeviceMatrixMask= 0x30,
   gxOmitTransferResultMatrixMask= 0x0C,
   gxOmitTransferFlagsMask       = 0x03
};

enum gxOmitTransferShift2 {

   gxOmitTransferSourceMatrixShift  = 6,
   gxOmitTransferDeviceMatrixShift  = 4,
   gxOmitTransferResultMatrixShift  = 2,
   gxOmitTransferFlagsShift         = 0
};
```

Transfer Component Omit Byte Masks and Shifts

```
enum gxOmitTransferComponentMask1{
   gxOmitTransferComponentModeMask           = 0x80,
   gxOmitTransferComponentFlagsMask          = 0x40,
   gxOmitTransferComponentSourceMinimumMask  = 0x30,
   gxOmitTransferComponentSourceMaximumMask  = 0x0C,
   gxOmitTransferComponentDeviceMinimumMask  = 0x03
} ;

enum gxOmitTransferComponentShift1 {
   gxOmitTransferComponentModeShift          = 7,
   gxOmitTransferComponentFlagsShift         = 6,
   gxOmitTransferComponentSourceMinimumShift = 4,
   gxOmitTransferComponentSourceMaximumShift = 2,
   gxOmitTransferComponentDeviceMinimumShift = 0
};

enum gxOmitTransferComponentMask2 {
   gxOmitTransferComponentDeviceMaximumMask  = 0xC0,
   gxOmitTransferComponentClampMinimumMask   = 0x30,
   gxOmitTransferComponentClampMaximumMask   = 0x0C,
   gxOmitTransferComponentOperandMask        = 0x03
};
```

```
enum gxOmitTransferComponentShift2 {
   gxOmitTransferComponentDeviceMaximumShift = 6,
   gxOmitTransferComponentClampMinimumShift  = 4,
   gxOmitTransferComponentClampMaximumShift  = 2,
   gxOmitTransferComponentOperandShift       = 0
};
```

Shape Object Omit Byte Constants and Data Types

Path Shape Omit Byte Masks and Shifts

```
enum gxOmitPathMask {
   gxOmitPathPositionXMask = 0xC0,
   gxOmitPathPositionYMask = 0x30,
   gxOmitPathDeltaXMask    = 0x0C,
   gxOmitPathDeltaYMask    = 0x03
};

enum gxOmitPathShift {
   gxOmitPathPositionXShift   = 6,
   gxOmitPathPositionYShift   = 4,
   gxOmitPathDeltaXShift      = 2,
   gxOmitPathDeltaYShift      = 0
};
```

Bitmap Shape Omit Byte Masks and Shifts

```
enum gxOmitBitmapMask1 {
   gxOmitBitmapImageMask      = 0xC0,
   gxOmitBitmapWidthMask      = 0x30,
   gxOmitBitmapHeightMask     = 0x0C,
   gxOmitBitmapRowBytesMask   = 0x03
};

enum gxOmitBitmapShift1 {
   gxOmitBitmapImageShift     = 6,
   gxOmitBitmapWidthShift     = 4,
   gxOmitBitmapHeightShift    = 2,
   gxOmitBitmapRowBytesShift  = 0
};
```

```
enum gxOmitBitmapMask2 {
   gxOmitBitmapPixelSizeMask  = 0xC0,
   gxOmitBitmapSpaceMask      = 0x30,
   gxOmitBitmapSetMask        = 0x0C,
   gxOmitBitmapProfileMask    = 0x03
};

enum gxOmitBitmapShift2 {
   gxOmitBitmapPixelSizeShift = 6,
   gxOmitBitmapSpaceShift     = 4,
   gxOmitBitmapSetShift       = 2,
   gxOmitBitmapProfileShift   = 0
};

enum gxOmitBitmapMask3 {
   gxOmitBitmapPositionXMask = 0xC0,
   gxOmitBitmapPositionYMask = 0x30
};

enum gxOmitBitmapShift3 {
   gxOmitBitmapPositionXShift = 6,
   gxOmitBitmapPositionYShift = 4
};
```

Bit Image Omit Byte Masks and Shifts

```
enum gxOmitBitImageMask {
   gxOmitBitImageRowBytesMask = 0xC0,
   gxOmitBitImageHeightMask   = 0x30,
   gxOmitBitImageDataMask     = 0x08
};

enum gxOmitBitImageShift {
   gxOmitBitImageRowBytesShift   = 6,
   gxOmitBitImageHeightShift     = 4,
   gxOmitBitImageDataShift       = 3
};
```

Text Shape Omit Byte Masks and Shifts

```
enum gxOmitTextMask {
   gxOmitTextCharactersMask   = 0xC0,
   gxOmitTextPositionXMask    = 0x30,
   gxOmitTextPositionYMask    = 0x0C,
   gxOmitTextDataMask         = 0x02
};

enum gxOmitTextShift {
   gxOmitTextCharactersShift  = 6,
   gxOmitTextPositionXShift   = 4,
   gxOmitTextPositionYShift   = 2,
   gxOmitTextDataShift        = 1
};
```

Glyph Shape Omit Byte Masks and Shifts

```
enum gxOmitGlyphMask1 {
   gxOmitGlyphCharactersMask  = 0xC0,
   gxOmitGlyphLengthMask      = 0x30,
   gxOmitGlyphRunNumberMask   = 0x0C,
   gxOmitGlyphOnePositionMask = 0x02,
   gxOmitGlyphDataMask        = 0x01
};

enum gxOmitGlyphShift1 {
   gxOmitGlyphCharactersShift    = 6,
   gxOmitGlyphLengthShift        = 4,
   gxOmitGlyphRunNumberShift     = 2,
   gxOmitGlyphOnePositionShift   = 1,
   gxOmitGlyphDataShift          = 0
};

enum gxOmitGlyphMask2 {
   gxOmitGlyphPositionsMask   = 0xC0,
   gxOmitGlyphAdvancesMask    = 0x20,
   gxOmitGlyphTangentsMask    = 0x18,
   gxOmitGlyphRunsMask        = 0x04,
   gxOmitGlyphStylesMask      = 0x03
};
```

```
enum gxOmitGlyphShift2 {
   gxOmitGlyphPositionsShift  = 6,
   gxOmitGlyphAdvancesShift   = 5,
   gxOmitGlyphTangentsShift   = 3,
   gxOmitGlyphRunsShift       = 2,
   gxOmitGlyphStylesShift     = 0
};
```

Layout Shape Omit Byte Masks and Shifts

```
enum gxOmitLayoutMask1 {
   gxOmitLayoutLengthMask     = 0xC0,
   gxOmitLayoutPositionXMask  = 0x30,
   gxOmitLayoutPositionYMask  = 0x0C,
   gxOmitLayoutDataMask       = 0x02
};

enum gxOmitLayoutShift1 {
   gxOmitLayoutLengthShift    = 6,
   gxOmitLayoutPositionXShift = 4,
   gxOmitLayoutPositionYShift = 2,
   gxOmitLayoutDataShift      = 1
};

enum gxOmitLayoutMask2 {
   gxOmitLayoutWidthMask   = 0xC0,
   gxOmitLayoutFlushMask   = 0x30,
   gxOmitLayoutJustMask    = 0x0C,
   gxOmitLayoutOptionsMask = 0x03
};

enum gxOmitLayoutShift2 {
   gxOmitLayoutWidthShift     = 6,
   gxOmitLayoutFlushShift     = 4,
   gxOmitLayoutJustShift      = 2,
   gxOmitLayoutOptionsShift   = 0
};
```

```
enum gxOmitLayoutMask3 {
   gxOmitLayoutStyleRunNumberMask= 0xC0,
   gxOmitLayoutLevelRunNumberMask= 0x30,
   gxOmitLayoutHasBaselineMask   = 0x08,
   gxOmitLayoutStyleRunsMask     = 0x04,
   gxOmitLayoutStylesMask        = 0x03
};

enum gxOmitLayoutShift3 {
   gxOmitLayoutStyleRunNumberShift  = 6,
   gxOmitLayoutLevelRunNumberShift  = 4,
   gxOmitLayoutHasBaselineShift     = 3,
   gxOmitLayoutStyleRunsShift       = 2,
   gxOmitLayoutStylesShift          = 0
};

enum gxOmitLayoutMask4 {
   gxOmitLayoutLevelRunsMask     = 0x80,
   gxOmitLayoutLevelsMask        = 0x40
};

enum gxOmitLayoutShift4 {
   gxOmitLayoutLevelRunsShift = 7,
   gxOmitLayoutLevelsShift    = 6
};
```

Picture Shape Omit Byte Masks and Shifts

```
enum gxOmitPictureParametersMask {
   gxOmitPictureShapeMask        = 0xC0,
   gxOmitOverrideStyleMask       = 0x30,
   gxOmitOverrideInkMask         = 0x0C,
   gxOmitOverrideTransformMask   = 0x03
};

enum gxOmitPictureParametersShift {
   gxOmitPictureShapeShift       = 0x6,
   gxOmitOverrideStyleShift      = 0x4,
   gxOmitOverrideInkShift        = 0x2,
   gxOmitOverrideTransformShift  = 0x0
};
```

C H A P T E R 8

QuickDraw GX Mathematics

Contents

This chapter describes QuickDraw GX number formats, number-format conversions, mathematical functions, and functions that operate on mappings (transformation matrices). Read this chapter if your application requires the explicit use of any of the mathematical capabilities of QuickDraw GX.

Related information on how QuickDraw GX uses mappings can be found in the chapter "Transform Objects" and the chapter "View-Related Objects" in *Inside Macintosh: QuickDraw GX Objects*.

This chapter first describes the number formats used in QuickDraw GX. It then describes the number-format conversion macros and mathematical functions that are provided by QuickDraw GX. It then shows how to use QuickDraw GX macros and functions to provide

- fixed-point number conversions
- fixed-point operations
- operations on 64-bit numbers
- vector operations
- Cartesian and polar coordinate conversions
- random number generation
- roots of linear and quadratic equations
- bit analysis
- mapping operations

About QuickDraw GX Mathematics

QuickDraw GX supports 16-bit, 32-bit, and 64-bit fixed-point number formats. You can use QuickDraw GX macros for efficient number-format conversions. QuickDraw GX mathematical functions provide a full spectrum of operations. QuickDraw GX mapping functions allow you to manipulate the matrices that transform shapes.

Number Formats

QuickDraw GX accepts standard integer and floating-point number formats, and defines several fixed-point number formats.

Integer Formats

Some Quickdraw GX functions and data structures may make use of the standard C language integer formats `short`, `unsigned short`, `long`, and `unsigned long`. The **short number** format is a 16-bit signed or unsigned integer; the **long number** format is a 32-bit signed or unsigned integer. Numbers in these formats have the following ranges of values:

Format	Range
`short`	–32, 768 to 32,767
`unsigned short`	0 to 65,535
`long`	–2,147,483,648 to 2,147,483,647
`unsigned long`	0 to 4,294,967,295

Floating-Point Formats

QuickDraw GX supports conversion to and from the C language single precision floating-point format `float`; double precision floating-point format `double`; and extra precision floating-point format `extended`. QuickDraw GX macros that convert between floating-point numbers and `Fixed` or `fract` numbers can handle all three floating-point formats.

Fixed-Point Formats

QuickDraw GX defines 16-bit, 32-bit, and 64-bit **fixed-point number** formats. Fixed-point number formats are integers that are interpreted as real numbers. The conversion between integer number format and a fixed-point number format is described by bias. A **bias** is a number (commonly expressed as a power of 2) by which an integer is divided in order to obtain the real number it represents. For example, the bias for the `Fixed` number format is 16 bits, or 2^{16}. In this case, the integer must be divided by 2^{16} to obtain the real number represented. Therefore, `Fixed` 0x10000 = $65{,}536 / 2^{16}$, or 1.0.

There are one 16-bit, two 32-bit, and one 64-bit number formats:

- The **`gxColorValue`** format for fixed-point numbers is a 16-bit unsigned integer. The values range from 0 to 65,535 to represent numbers from 0 to 1. This fixed-point number is described by a bias of 65,535. The integer must be divided by 65,535 to obtain the real number represented. (Its name derives from the fact that it is used to describe color-component values in a Quickdraw GX color structure; see the chapter “Colors and Color-Related Objects” in *Inside Macintosh: QuickDraw GX Objects* for more information.)
- The **`Fixed`** format for fixed-point numbers has 16 bits to the left and 16 bits to the right of the binary point. This corresponds to a fixed-point bias of 16 bits. `Fixed` format numbers range from –32,768 to [32,767 + (65,535 / 65,536)], or approximately 32,768.

- The **`fract`** format for fixed-point numbers has 2 bits to the left and 30 bits to the right of the binary point. This corresponds to a fixed-point bias of 30 bits. Numbers in `fract` format range from –2 to [2 – (2^{-30})] or –2 to [1 + (1,073,741,823 / 1,073,741,824)], or approximately 2.0.
- The **`wide`** format is a signed integer data type that has 64 bits. It can be given a bias, just as any other integer type can. With a bias of 0 bits, a `wide` format number represents an integer and can range from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. With a bias of 16 bits, a `wide` format number represents an extended version of the `Fixed` format (that is, it has the same precision but a larger range) and can range from –140,737,488,355,328 to [140,737,488,355,327 + (65,535 / 65,536)].

All of the fixed-point number formats except for `gxColorValue` are two's complement signed integers.

The `wide` data type is defined as a structure that contains an unsigned long integer as its low-order half and a signed long integer as its high-order half. You can convert a `long` into a `wide` in either of two ways:

- First, assign the `long` to the low half of the `wide`. Then, if the `long` is not negative, assign 0 to the high half of the `wide`; if the `long` is negative, assign –1 to the high half of the `wide`.
- Assign the `long` to the high half of the `wide`. Use the `WideShift` function to shift the bits of the `wide` rightward by 32 bits.

The WideShift function is described on page 8-51. The `wide` structure is described on page 8-35.

Working With Bias in Fixed-Point Operations

`Fixed` numbers have a bias of 16; `fract` numbers have a bias of 30; and `long` and `wide` numbers have a bias of 0. Unless stated otherwise, all biases will be powers of 2. For brevity, we use the convention of describing a bias by the exponent of 2; for example, we say "a bias of 16" instead of "a bias of 16 bits."Operations that are designed to work on a specific number format (such as `FixedMultiply` or `FractDivide` or `WideMultiply`) apply a bias to the result of their operations that reflects the number format they expect. If you understand how the bias is applied, you can use (and even mix) any of several different fixed-point number formats in these functions, and know what bias to use when interpreting the result:

- Operations on `Fixed` numbers (such as `FixedMultiply` and `FixedDivide`) use a bias of 16; operations on `fract` numbers (such as `FractMultiply` and `FractDivide`) use a bias of 30; operations on `long` and `wide` numbers (such as `MultiplyDivide` and `WideDivide`) use a bias of 0.
- When multiplying two fixed-point numbers, the bias of the result is the sum of the biases of the input numbers, minus the bias of the operation. Thus, the result of using `FixedMultiply` to multiply two `Fixed` numbers is a `Fixed` (= (16 + 16) – 16), as expected. On the other hand, the result of using `FixedMultiply` to multiply a `fract` and a `Fixed` is a `fract` (= (30 + 16) – 16), and the result of `FractMultiply` on a `fract` and a `Fixed` is a `Fixed` (= (30 + 16) – 30).

- When dividing two fixed-point numbers, the bias of the result is the operation bias plus the difference between the biases of the input numbers. Thus, as expected, the result of using `FixedDivide` to divide one `Fixed` number by another is a `Fixed` (= 16 + (16 – 16)). The result of `FixedDivide` on a `fract` divided by a `Fixed` is a `Fract` (= 16 + (30 – 16)), and the result of `FractDivide` on a `Fixed` divided by a `Fract` is a `Fixed` (= 30 + (16 – 30)).
- For operations that have no bias, the result is simply the sum or difference of the input biases. For example, if you use `MultiplyDivide` to multiply a `Fixed` by a `fract` and divide the result by a `Fixed`, the result will be a `fract` (= 16 + 30 – 16). If you use `WideMultiply` to multiply two `Fixed` numbers, the result will have a bias of 32 bits (= 16 + 16).

Remember also that using the standard C operators + and – to add or subtract fixed-point numbers is meaningful only if the numbers have the same bias. Thus, if you wish to add together a long integer and a `Fixed`, for example, you must first convert one format to the other, or convert both to a common format.

The functions referred to in this section are described in the section "Fixed-Point Operations" beginning on page 8-42, and "Operations on wide Numbers" beginning on page 8-49.

Number-Conversion Macros

QuickDraw GX provides a set of predefined **macros** for the conversion between different fixed-point number formats. This allows you the convenience of accessing the number-conversion formulas as if they were function calls.

Table 8-1 summarizes the number-format conversions that are supported.

Table 8-1 Macro number-format conversions

From number format	To number format
`Fixed`	`fract`
`Fixed`	floating-point
`Fixed`	integer
`fract`	floating-point
`fract`	`Fixed`
`fract`	`gxColorValue`
floating-point	`fract`
floating-point	`Fixed`
integer	`Fixed`
`gxColorValue`	`fract`

QuickDraw GX also provides macros that

- round a `Fixed` number to its nearest integer
- determine the greatest integer that is not greater than a given `Fixed` number
- use a function to determine the square root of a `Fixed` number

The use of QuickDraw GX macros is described in the section “Converting Number Formats” beginning on page 8-26. Each macro is described in the section “Number-Conversion Macros” beginning on page 8-36.

Mathematical Functions

QuickDraw GX provides mathematical functions for

- fixed-point operations on `Fixed`, `long`, and `fract` number formats
- fixed-point operations on a `wide` number format
- vector operations
- Cartesian and polar coordinate point conversions
- random number generation
- linear and quadratic roots
- bit analysis

A description of each QuickDraw GX mathematics function is provided in the section “Mathematical Functions” beginning on page 8-42.

Operations on Fixed, long, and fract Numbers

QuickDraw GX provides functions that perform operations on `Fixed`, `long`, and `fract` number formats. Functions are provided that

- determine the product of two numbers ($a \times b$)
- determine the quotient of two numbers (a / b)
- determine the product of two numbers and the quotient of a third number ($(a \times b) / c$)
- determine both the sine and cosine of an angle measured in degrees [sine(angle) and cosine(angle)]
- determine the square root of a number $(a)^{1/2}$
- determine the cube root of a number $(a)^{1/3}$
- determine the magnitude of a two-dimensional vector

The functions that perform operations on `Fixed`, `long`, and `fract` number formats are described in the section “Fixed-Point Operations” beginning on page 8-42.

Operations on wide Numbers

QuickDraw GX provides functions for operations on `wide` numbers. Functions are provided that

- determine the sum of two `wide` numbers (a + b)
- determine the difference between two `wide` numbers (a – b)
- determine the product, as a `wide` number, of two `long` numbers (a × b)
- determine the quotient, as a `long` number (without remainder), of a `wide` number divided by a `long` number (a / b)
- determine the result, as a `long` quotient and a `long` remainder, of dividing a `wide` number by a `long` number (a / b + remainder)
- determine the square root of a `wide` number $(a)^{1/2}$
- negate a `wide` number (–a)
- shift bits in a `wide` number to the right or left
- determine the highest order bit in the absolute value of a `wide` number
- compare two `wide` numbers

The functions that perform operations on `wide` number formats are described in the section "Operations on wide Numbers" beginning on page 8-49.

Vector Operations

QuickDraw GX provides vector operation functions that

- determine the dot product of two vectors $(v_1 \cdot v_2)$
- determine the dot product of two vectors and divide by a number $(v_1 \cdot v_2)/a$

The use of QuickDraw GX vector operation functions is described in the section "Performing Vector Operations" beginning on page 8-29. These functions are described in the section "Vector Operations" beginning on page 8-54.

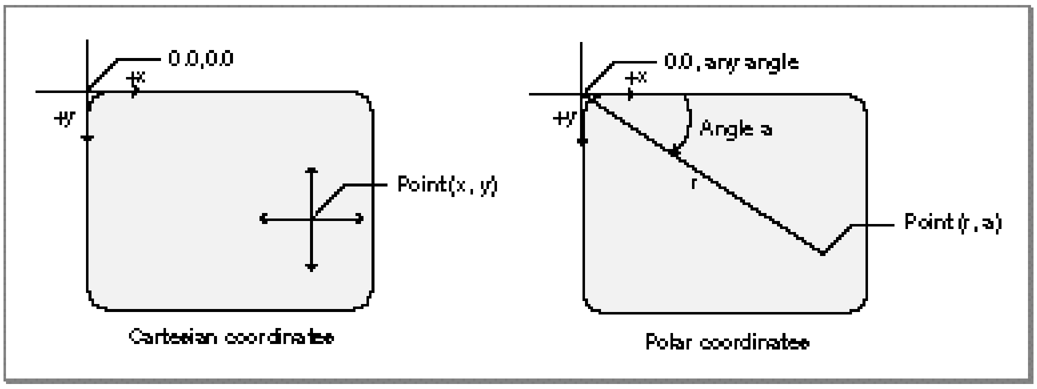
Cartesian and Polar Coordinate Conversion

You use Cartesian coordinates to specify points with QuickDraw GX. Some shapes, such as rectangles, are more easily drawn using Cartesian coordinates; however, some shapes that have symmetry about a point are more easily drawn with polar coordinates. For that reason, QuickDraw GX provides conversion routines so that you can work in either coordinate system.

For QuickDraw GX, **Cartesian coordinates** have a positive *x* direction to the right and a positive *y* direction downward (not upward, as in many other Cartesian coordinate systems). Cartesian coordinates are written in the order (*x*, *y*). The origin is at (0, 0). The `gxPoint` structure describes points using Cartesian coordinates.

Polar coordinates have the same origin point as Cartesian coordinates, but locations are specified differently. The polar coordinate of a point is specified by the length of the radius vector *r* from the origin to the point and the direction of the vector is specified by polar angle *a*. Angles in QuickDraw GX are measured clockwise in degrees from the Cartesian coordinate positive x-axis. The polar coordinate of a point specified by a vector of length *r* and direction *a* degrees from the x-axis is written as point (*r*, *a*). The polar origin point has the coordinates (0, *a*), where *a* is any angle. Points having polar coordinates are defined by the `gxPolar` structure. The `gxPolar` structure is described in the section "Constants and Data Types" beginning on page 8-35. The relationship of the Cartesian and polar coordinates is shown in Figure 8-1.

Figure 8-1 Cartesian and polar coordinates



The `gxPolar` location (*r*, *a*) corresponds to the `gxPoint` location ($r \times \cos(a)$, $r \times \sin(a)$). The mathematical relationship between the two coordinate systems is given by the expressions $r^2 = x^2 + y^2$ and $\tan(a / 2) = y / (r + x)$. The angle can also be defined by the more familiar term $\tan(a) = y / x$.

The use of the polar-to-Cartesian and Cartesian-to-polar coordinates functions are described in the section "Converting Between Cartesian and Polar Coordinates" beginning on page 8-29. These functions are described in the section "Cartesian and Polar Coordinate Point Conversions" beginning on page 8-56.

Random Number Generation

The QuickDraw GX random-number algorithm generates random integers in the range of 0 to $2^{count} - 1$, where *count* is the number of bits to be generated by the random number generator.

The sequence of values that the random number generator produces is dependent upon the initialization value called the **seed**. The algorithm uses the seed to calculate the next random number and a new seed. If no seed is provided, QuickDraw GX uses a default seed value of 0. To repeat a sequence of random numbers, you can use the same seed value.

QuickDraw GX provides random number generation functions that

- generate a sequence of random bits
- change the seed used by the random number algorithm
- determine the current seed for the random number algorithm

The use of the random number generation functions is described in the section "Generating Random Numbers" beginning on page 8-33. These functions are described in the section "Random Number Generation" beginning on page 8-58.

Roots of Linear and Quadratic Equations

QuickDraw GX provides mathematical functions that

- determine the root of a linear equation
- determine the roots of a quadratic equation

The linear and quadratic equation solving functions are described in the section "Linear and Quadratic Roots" beginning on page 8-60.

Bit Analysis

QuickDraw GX provides a mathematical function that allows you to determine the highest bit number that is set in a number.

The `FirstBit` function is described in the section "Bit Analysis" beginning on page 8-62.

Transformation Operations With Mappings

A **mapping** is a 3 × 3 perspective matrix that performs transformations of spatial locations in two dimensions. You can apply a mapping operation to a set of points either directly (as when directly modifying the geometry of a shape), or indirectly, by multiplying a mapping with another mapping (as when altering the mapping in the transform object associated with a shape).

QuickDraw GX uses mappings to perform the following transformations on shapes or other two-dimensional data:

- **Translation** shifts the position of a shape by the amount specified in the mapping. Translation functions allow you to specify either a relative shift along either coordinate axis, or an absolute shift to a new specified location.
- **Scaling** changes the size of a shape by the factor specified in the mapping. Scaling functions allow you to change size along either axis, and can also result in reflection about the coordinate axes.
- **Rotation** changes the angle of rotation of a shape by the amount specified in the mapping, rotating all points around a given point.
- **Skewing** changes the slant applied to a shape by the amount specified in the mapping. Skewing functions allow you to apply slant along either coordinate axis, relative to a given point. The term *shearing* is synonymous with skewing.
- **Perspective** modifies the positions of points to give a three-dimensional effect.

When you multiply two or more matrices to obtain a cumulative result, you **concatenate,** or accumulate the transformations of, both mappings. Matrix multiplication is not commutative. This means that $[A] \times [B] \neq [B] \times [A]$. As a result, the order that you concatenate is important. [A] is **postmultiplied** by [B] if [A] is replaced by $[A] \times [B]$. Conversely, [A] is **premultiplied** by [B] if [A] is replaced by $[B] \times [A]$. A mapping is applied to a point via postmultiplication (which is to say that points are row vectors); therefore, the default for applying one mapping to another is also postmultiplication.

Multiple concatenations can occur in QuickDraw GX, such as when drawing picture shapes or when drawing any shape through a hierarchy of view ports. If you are going to apply several mappings to a relatively large bitmap or other shape, it is advantageous to concatenate the mappings first (with the `MapMapping` function) and then apply the resultant mapping to the shape (with the `GXMapShape` function).

The motivation is speed. It is much faster to concatenate mappings than to apply a mapping to a large number of points. For bitmaps, an additional motivation is accuracy. Each time a shape is transformed, a certain amount of roundoff error is introduced. Because the pixels of a bitmap are at integral coordinates, the roundoff error is on the average of a quarter pixel, compared with thousandths of a pixel for fixed-point coordinates.

QuickDraw GX provides two groups of mapping functions. The first group allows you to copy and perform standard matrix operations on mappings. With these functions, you can

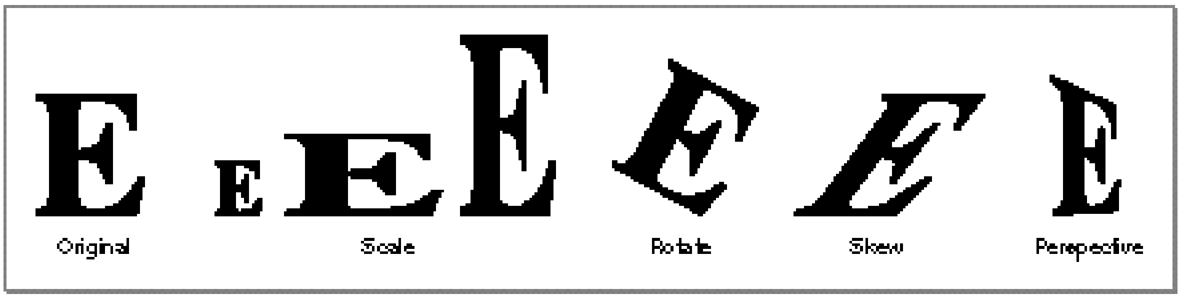
- make a copy of a mapping
- normalize a mapping
- reset a mapping to identity
- invert a mapping
- concatenate (postmultiply) a mapping to another mapping
- apply a mapping to each of a given set of points

The second group allows you to modify how a mapping transforms the objects or coordinate space it is applied to. With these functions, you can

- add translation to mapping
- modify a mapping to specify translation to an absolute location
- add horizontal and vertical scaling to a mapping
- add rotation to a mapping
- add horizontal and vertical skew to a mapping

Figure 8-2 shows an example of how modifying a mapping can modify the scaling, rotation, skewing, and perspective of a shape.

Figure 8-2 Transformation operations with a mapping matrix



Characteristics of a Mapping

QuickDraw GX achieves these two-dimensional transformations of shapes and points on a plane by matrix multiplication of each Cartesian point P by the mapping matrix [T] to generate a transformed point P´.

P(x, y) [T] = P´(x´, y´)

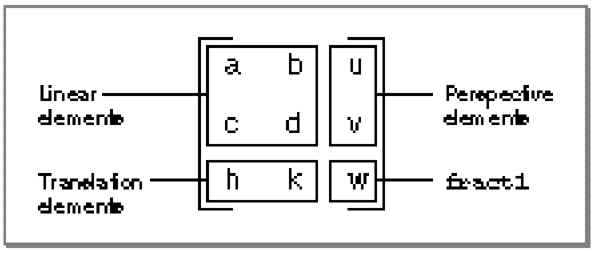
To multiply a two-dimensional point by a three-dimensional matrix, we first expand it to a three-dimensional point (x, y, 1). After multiplication, the resulting point is (x´, y´, z´), which normalizes to (x´/z´, y´/z, 1) or, in two dimensions, (x´/z´, y´/z).

The QuickDraw GX mapping is defined as

```
struct gxMapping { Fixed map[3][3];};
```

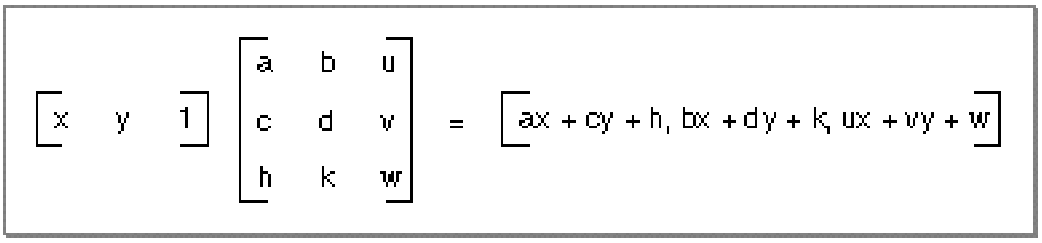
The mapping consists of linear elements a, b, c, and d; perspective elements u and v; translation elements h and k; and the scale factor w, which is commonly set to `fract1`. Although defined as containing only `Fixed` numbers, the rightmost column of the matrix—containing elements u, v, and w—consists of `fract` numbers. Figure 8-3 shows the elements of the matrix in place.

Figure 8-3 Mapping matrix elements



Point P(x, y) is transformed to point P´(x´, y´) by matrix multiplication of the row vector [x y 1] by the mapping matrix to yield the expanded general expression shown in Figure 8-4.

Figure 8-4 Applying a mapping matrix to a point



$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} a & b & u \\ c & d & v \\ h & k & w \end{bmatrix} = \begin{bmatrix} ax + cy + h, & bx + dy + k, & ux + vy + w \end{bmatrix}$$

The x and y elements of the transformed vector can be mapped back to the x and y-coordinates by dividing each element by the term ux + vy + w. The resulting general expression for the transformation of point P(x, y) to P´(x´, y´) is shown in Figure 8-5.

Figure 8-5 The point (x, y) as transformed by the mapping matrix

$$\begin{bmatrix} x & y \end{bmatrix} \longrightarrow \begin{bmatrix} \dfrac{ax + cy + h}{ux + vy + w} \,, & \dfrac{bx + dy + k}{ux + vy + w} \end{bmatrix}$$

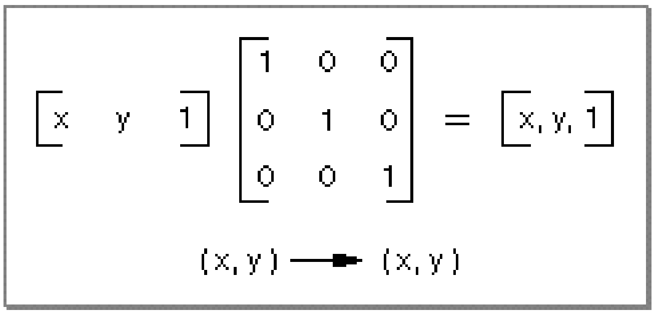
A mapping is **normalized** whenever the transformation matrix element w has the value 1. Most QuickDraw GX mapping operations will be automatically normalized. However, mappings that an application generates itself might not be normalized. Subsequent operations with that mapping may be slow.

If a mapping does not specify perspective (that is, if its perspective elements u and v are zero), normalization of the transformation involves dividing the map by the absolute value of w, if possible. If this division is not possible (due to overflow) or if the mapping specifies perspective, normalization involves bit-shifting each element of the mapping to the left. The amount of shift provided by the minimum of the following two operations is selected:

- shift the minimum number of bits so that the absolute value of some element of the mapping is >= `fract1` (compared as `long` values).
- shift the maximum number of bits so that the sum of the absolute values of u and v is <= `fract1 – fixed1` (compared as `long` values).

The identity mapping, or **identity matrix,** has the unique characteristic that it maps points to the same point. The identity matrix has all diagonal elements equal to 1 and all other matrix elements have the value 0. The identity matrix is shown in Figure 8-6.

Figure 8-6 The identity matrix



The **inverse of a mapping** is the mathematical inverse of the matrix. This means that if you concatenate a mapping with its inverse, you will get the identity matrix.

The rest of this section discusses the use of the mapping functions in modifying the translation, scaling, rotation, and skewing factors in a mapping. It ends with a discussion of how to modify the perspective factors in a mapping. For additional information about the use of mappings in the transform object and in view port and view device objects, see the chapters "Transform Objects" and "View-Related Objects," respectively, in *Inside Macintosh: QuickDraw GX Objects.*

Translation by a Relative Amount

You can use the `MoveMapping` function to make a relative change (in both x and y) to the translation specified by a mapping. Matrix elements h and k control the amount of the translation. Figure 8-7 shows what happens to a mapping *M* when you call `MoveMapping` and specify horizontal and vertical offsets of `hOffset` and `vOffset`. A purely translational matrix is applied to the target mapping, so that the resultant mapping's translation is increased by the specified offsets.

Figure 8-7 Changing the translation specified by a mapping

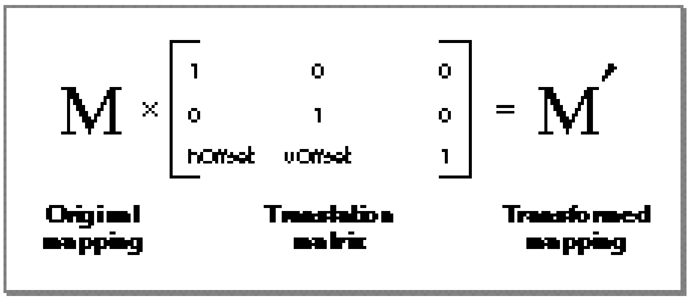
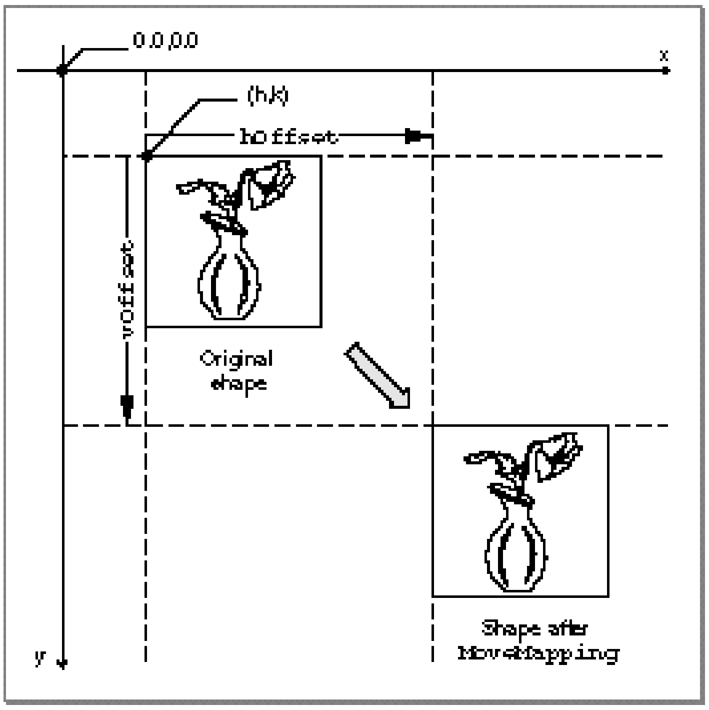


Figure 8-8 shows the use of the `MoveMapping` function to provide translation of a mapping by the increments given by the `hOffset` and `vOffset` parameters. The `MoveMapping` function is described on page 8-67.

Figure 8-8 Translation by a relative amount with `MoveMapping`



Translation to a Specified Point

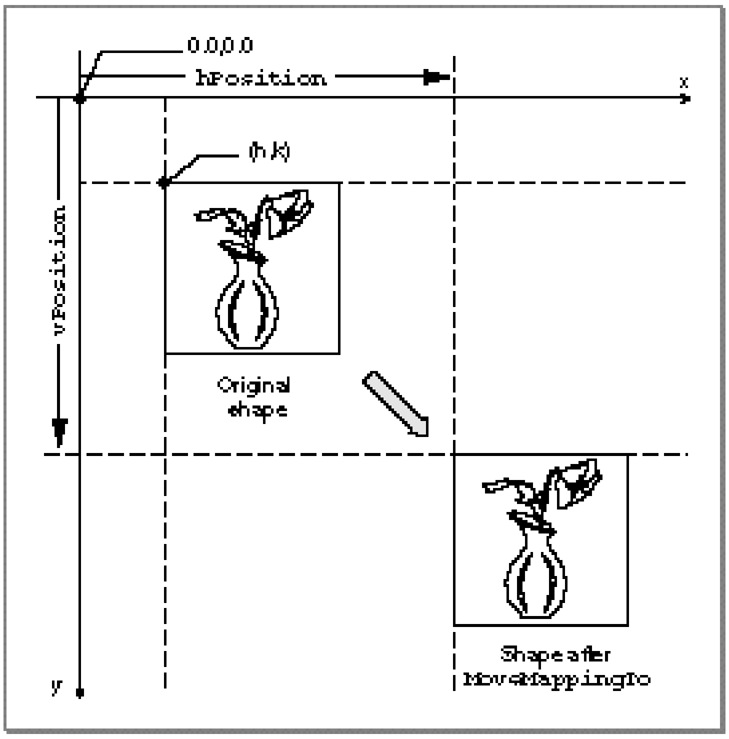
You can specify translation of the origin to a given point by using the `MoveMappingTo` function. Moving the origin means that the point (0, 0) will become the point (h, k) after the mapping is applied to it. Matrix elements h and k again control the amount of translation. Figure 8-9 shows what happens to a mapping *M* when you call `MoveMappingTo` and specify the desired location (`hPosition`, `vPosition`). A relative translation of (–h / w, –k / w) is applied to the target mapping to bring its origin to (0, 0), and then a relative translation of (`hPosition`, `vPosition`) is applied. The resultant mapping ends up with translational values of `hPosition` and `vPosition`.

Figure 8-9 Setting the origin specified by a mapping

$$\underset{\text{Original mapping}}{M} \times \underset{\text{Translation matrix}}{\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ (hPosition-\frac{h}{w}) & (vPosition-\frac{k}{w}) & 1\end{bmatrix}} = \underset{\text{Transformed mapping}}{M'}$$

Figure 8-10 shows the use of the `MoveMappingTo` function to move the origin to a specific location. Note that this figure assumes that the origin of the shape—point (0.0, 0.0) in its geometry—is at its upper left corner. The `MoveMappingTo` function is described on page 8-68.

Figure 8-10 Translation to a specific origin location



Scaling

You can use the `ScaleMapping` function to modify the scaling factors in a mapping. Matrix elements a and d in the mapping matrix control the degree of the scaling in the horizontal and vertical directions, respectively. Figure 8-11 shows what happens to a mapping *M* when you call `ScaleMapping` with horizontal and vertical scaling factors of `hFactor` and `vFactor` and a center of scaling at (`xCenter`, `yCenter`). First, a relative translation of –`xCenter` and –`yCenter` moves the center of scaling to (0, 0); then a purely scaling matrix multiplies the scaling by `hFactor` and `vFactor`; finally, another relative translation moves the center of scaling by +`xCenter` and +`yCenter`. In effect, the center of scaling is moved to (0, 0), the scaling is applied, and the scaling center is then moved back to where it was.

Figure 8-11 Changing the amount of scaling specified by a mapping

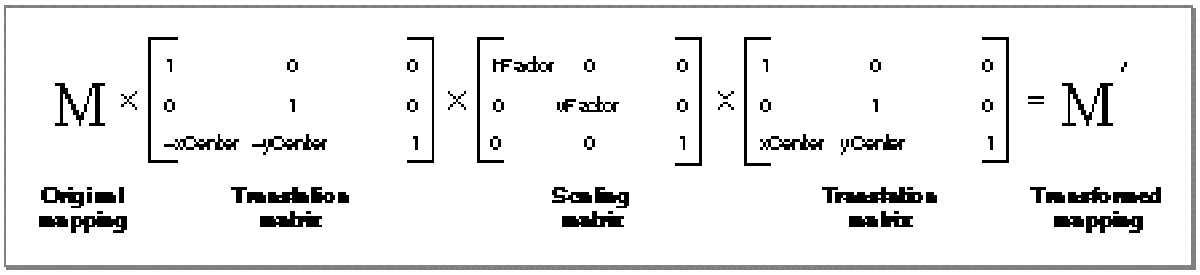
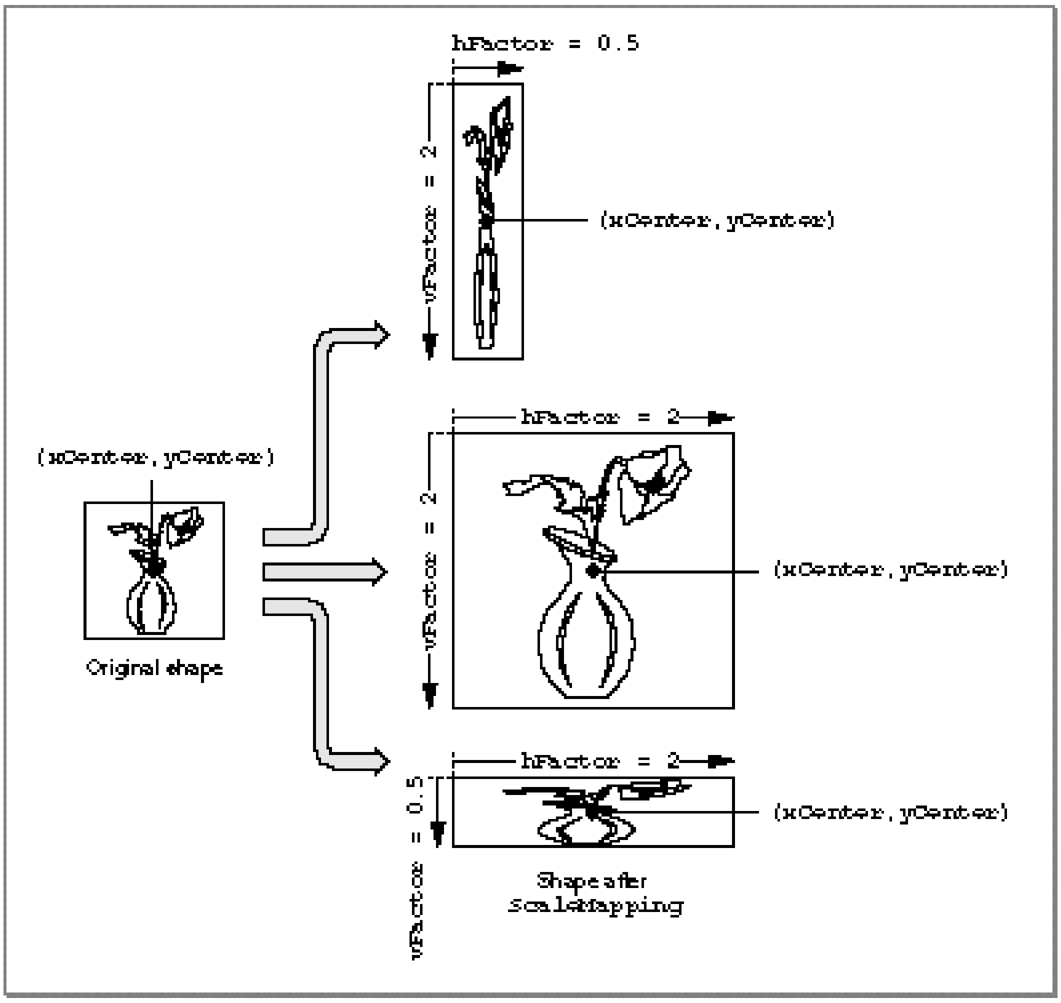


Figure 8-12 shows the use of the `ScaleMapping` function scale for various horizontal and vertical factors, in which the center of scaling corresponds to the center of the shape. The `ScaleMapping` function is described on page 8-69.

Figure 8-12 Scaling horizontally and vertically



Note that if `vFactor` equals `hFactor`, scaling is uniform in both directions. If `vFactor` is not equal to `hFactor`, distortion of the image occurs, as shown in Figure 8-12.

The mapping matrix also accommodates **reflection** transformations. If `hFactor` is negative, a reflection about the vertical axis occurs. If `vFactor` is negative, a reflection about the horizontal axis occurs. If both `vFactor` and `hFactor` are negative, a 180° rotation occurs.

Rotation

You can use the `RotateMapping` function to modify the rotation specified by a mapping. Matrix elements a, b, c, and d together specify the angle of rotation. Figure 8-13 shows what happens to a mapping *M* when you call `RotateMapping` to rotate by an angle β about a rotational origin of `xCenter` and `yCenter`. First, a relative translation of –`xCenter` and –`yCenter` moves the center of rotation to (0, 0); then a purely rotational matrix adds β to the amount of rotation already specified in the mapping; finally, another relative translation moves the center of rotation by +`xCenter` and +`yCenter`, back to where it was.

Figure 8-13 Changing the degree of rotation specified by a mapping

$$\underbrace{M}_{\text{Original mapping}} \times \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -xCenter & -yCenter & 1 \end{bmatrix}}_{\text{Translation matrix}} \times \underbrace{\begin{bmatrix} \cos(\beta) & \sin(\beta) & 0 \\ -\sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Rotation matrix}} \times \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ xCenter & yCenter & 1 \end{bmatrix}}_{\text{Translation matrix}} = \underbrace{M'}_{\text{Transformed mapping}}$$

Figure 8-14 shows the use of the `RotateMapping` function to change the rotation of a mapping. Note that positive values of the angle parameter cause clockwise rotation (consistent with y values increasing downward), and note also that changing the center of rotation can significantly change the final position of the rotated objects. The `RotateMapping` function is described on page 8-70.

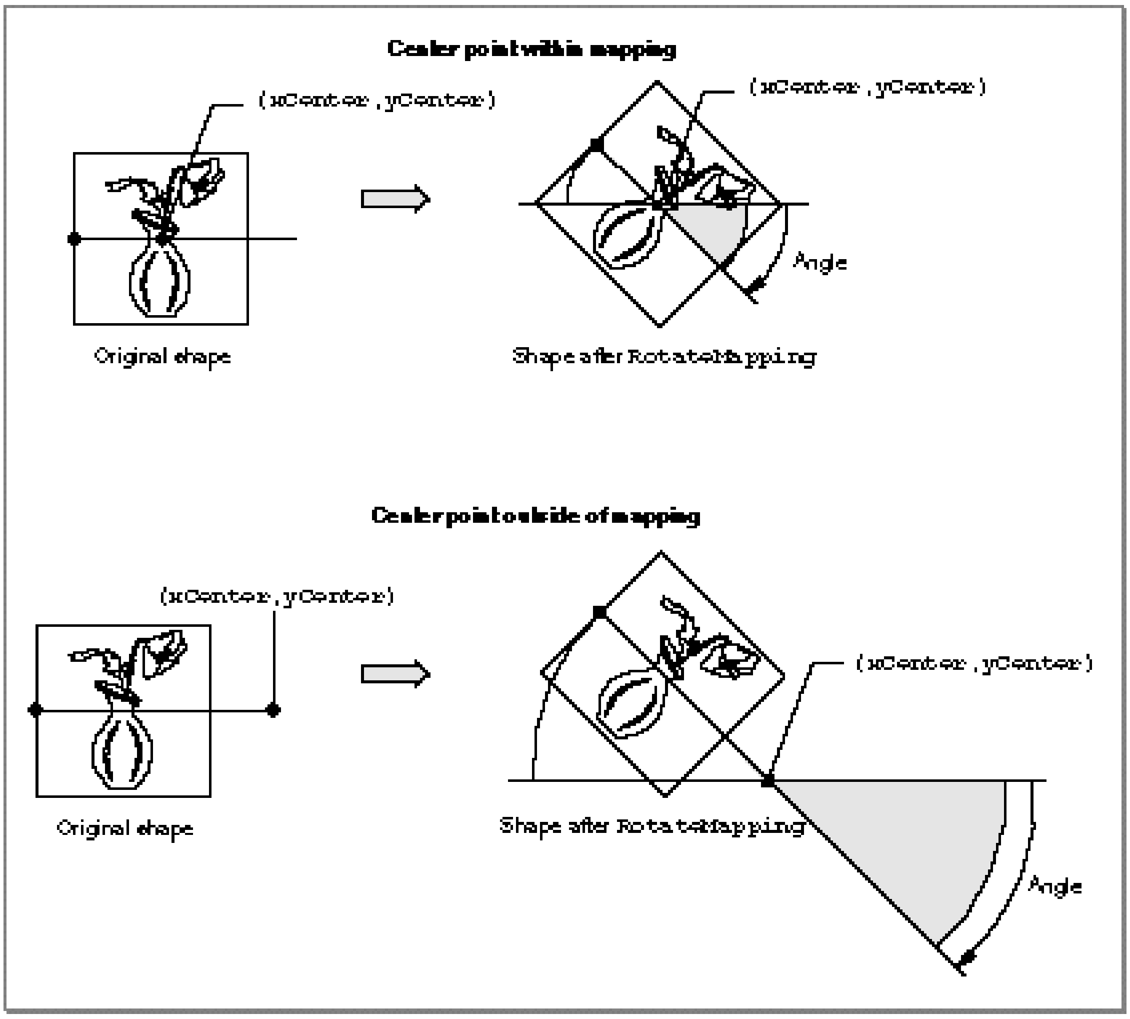
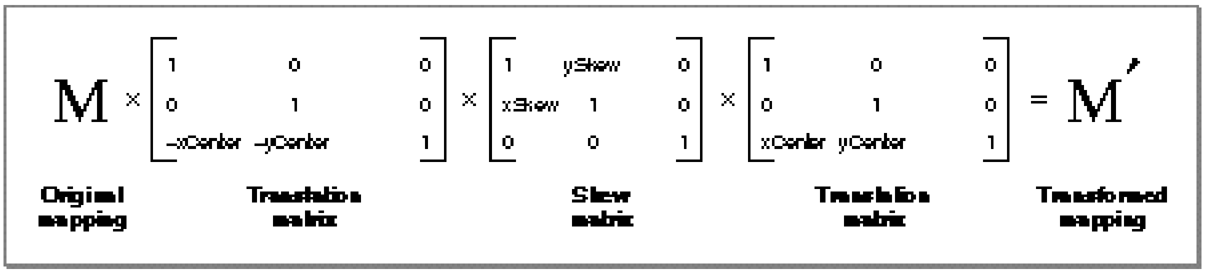


Figure 8-14 Rotating about different center points

Skewing

You can use the SkewMapping function to modify the skewing imposed by a mapping. Matrix elements b and c control the amount of the skew. Element b controls skew in the y direction and element c controls skew in the x direction. Figure 8-15 shows what happens to a mapping *M* when you call SkewMapping with x and y skew factors of xSkew and ySkew, and a skew origin (the point at which no shearing takes place) of xCenter and yCenter. First, a relative translation of –xCenter and –yCenter moves the center of skewing to (0, 0); then a purely skewing matrix modifies the amount of skew already specified in the mapping; finally, another relative translation moves the center of skewing by +xCenter and +yCenter, back to where it was.

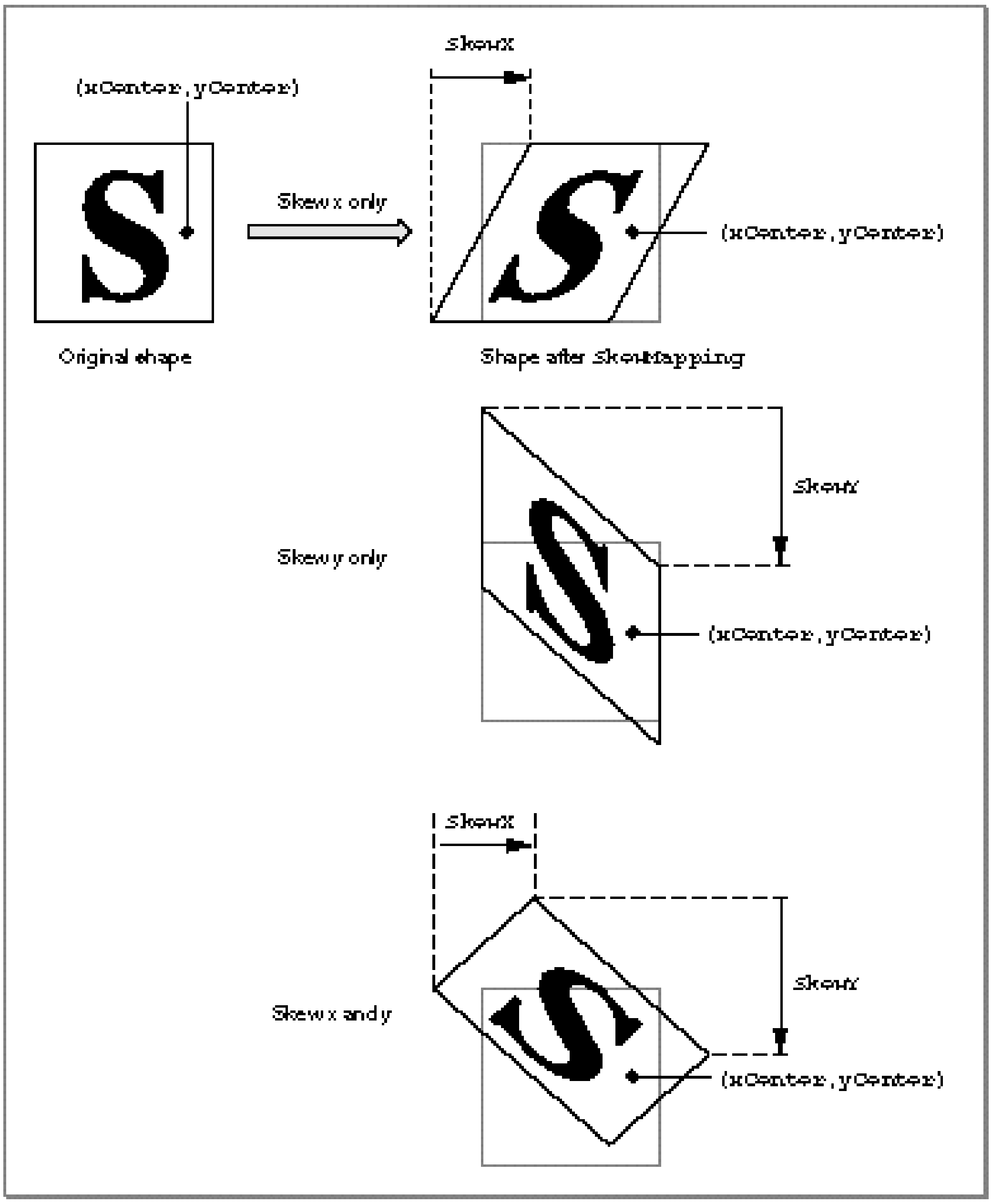
Figure 8-15 Changing the amount of skew specified by a mapping



$$M \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -xCenter & -yCenter & 1 \end{bmatrix} \times \begin{bmatrix} 1 & ySkew & 0 \\ xSkew & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ xCenter & yCenter & 1 \end{bmatrix} = M'$$

Figure 8-16 shows the use of the SkewMapping function to change the skew specified by a mapping. (Note that the skew in the x direction in Figure 8-16 is negative; as y decreases—upward—the amount of shear in the x direction increases.) The SkewMapping function is described on page 8-71.

Figure 8-16 Skewing a shape both horizontally and vertically



Perspective

You can manipulate the elements of a mapping to modify its specification of perspective. The matrix elements u, v, and w determine how the perspective will appear when the mapping is applied. The action performed on a point by a mapping whose perspective elements are nonzero is shown in Figure 8-18.

Figure 8-17 Changing the perspective specified by a mapping

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & u \\ 0 & 1 & v \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x, y, xu + yv + 1 \end{bmatrix}$$

$$(x, y) \longrightarrow \left(\frac{x}{xu + yv + 1} \;,\; \frac{y}{xu + yv + 1} \right)$$

There is currently no QuickDraw GX function that modifies the perspective-controlling elements of a mapping for you. If you wish to create perspective, you need to modify the individual matrix elements directly.

Using QuickDraw GX Mathematics

This section describes how you can use QuickDraw GX number formats, macros, and functions in your application.

Converting Number Formats

You can use QuickDraw GX macros to convert between `Fixed`, `fract`, integer, floating-point, and `gxColorValue` number formats. Macros are also provided to round, truncate, and compute the square root of a fixed-point number.

For example, you can use the `IntToFixed` macro to convert an integer to a `Fixed` format and you can use the `FloatToFixed` macro to convert from a floating-point format to a `Fixed` format. The functionality of the `FloatToFixed` macro is also provided as the shortened `fl` macro. The functionality of the `IntToFixed` macro is also provided as the shortened `ff` macro.

The `ff` macro is especially useful when you are coding specific points in your application. For example, it's easier to define a line in your application using the `ff` macro:

```
gxLine lineData = {ff(25), ff(25) , ff(125), ff(125)};
```

than to use the equivalent, but much longer `IntToFixed` macro:

```
gxLine lineData = {IntToFixed(25), IntToFixed(25),
                   IntToFixed(125), IntToFixed(125)};
```

For constants, using `ff` is faster and more efficient than using `fl`, because `ff` is evaluated at compile time, whereas `fl` is evaluated at run time.

The `IntToFixed` macro is described on page 8-37. The `FloatToFixed` macro is described on page 8-39. The `fl` macro is described on page 8-39. The `ff` macro is described on page 8-38.

Performing Fixed-Point Operations

You can use QuickDraw GX functions to provide operations on `Fixed`, `long`, `fract` and `wide` numbers. The equivalent QuickDraw GX fixed-point functions for functions in the Macintosh Mathematical Utilities is shown in Table 8-2.

Table 8-2 QuickDraw GX and Macintosh Toolbox fixed-point functions

QuickDraw GX	Macintosh Mathematical Utilities
`FractDivide`	`FracDiv`
`FractMultiply`	`FracMul`
`FractSquareRoot`	`FracSqrt`
`FixedDivide`	`FixDiv`
`FixedMultiply`	`FixMul`
`WideMultiply`	`LongMul`

The Macintosh Mathematical Utilities are described in *Inside Macintosh: Operating System Utilities.*

Some functions combine multiple functions into a single function to increase calculation speed over that obtained using sequential function calls. For example, the `FractSineCosine` function returns both the sine and cosine of an angle.

Some functions support the use of 64-bit numbers to increase the accuracy of calculations. For example, the `WideAdd` function returns the 64-bit sum of two 64-bit numbers, and the `WideDivide` function returns the quotient of a 64-bit number and a 32-bit number. The `MultiplyDivide` function uses a 64-bit intermediate result to increase accuracy of the calculation and to prevent premature overflow.

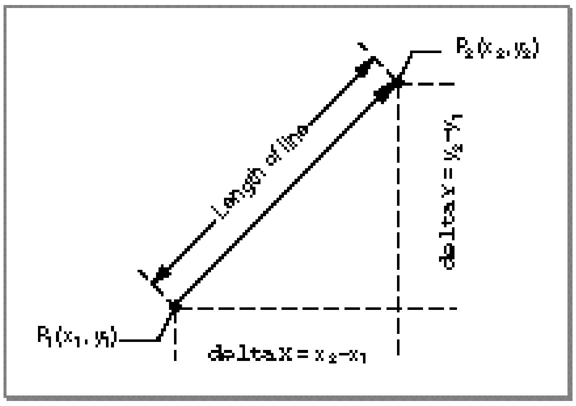
The `MultiplyDivide`, `Magnitude`, and `VectorMultiplyDivide` functions are derivatives of other functions. For example, `MultiplyDivide (x, y, z)` is the same as:

```
wide temp;
WideDivide (WideMultiply(x, y, &temp), z, 0)
```

The final argument of 0 specifies that the returned number will be rounded with no remainder.

You can use the `Magnitude` function to determine the magnitude (length) of a two-dimensional vector, or the distance between two points on a plane. Figure 8-18 shows the use of function parameters `deltaX` and `deltaY`.

Figure 8-18 Determining the length of a line with the `Magnitude` function

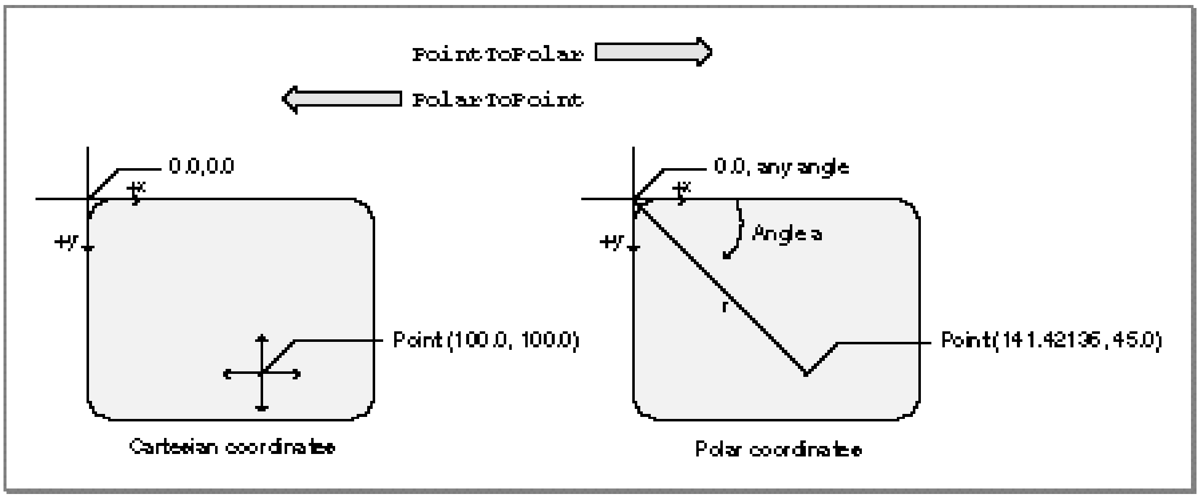


Functions that provide arithmetic operations on fixed-point numbers are described in the section “Fixed-Point Operations” beginning on page 8-42. Functions that provide operations on `wide` numbers are described in the section “Operations on wide Numbers” beginning on page 8-49. The `Magnitude` function is described on page 8-45.

Converting Between Cartesian and Polar Coordinates

You can use QuickDraw GX functions to convert between Cartesian and polar coordinates. The `PolarToPoint` function converts a point in polar coordinates to Cartesian coordinates, (r, a) to (x, y). The `PointToPolar` function converts a point in Cartesian coordinates to polar coordinates, (x, y) to (r, a). The `gxPolar` point (r, a) corresponds to the `gxPoint` point $(r \times \cos(a), r \times \sin(a))$. Since $r^2 = x^2 + y^2$ and $\tan(a) = y / x$, the `gxPoint` structure (100, 100) corresponds to the `gxPolar` structure (141.42136, 45). Figure 8-19 shows the Cartesian coordinate of point (100, 100) and the polar coordinate of identical point (141.42136, 45).

Figure 8-19 Converting between Cartesian and polar coordinates



The Cartesian and polar coordinate systems are described in the section "Cartesian and Polar Coordinate Conversion" beginning on page 8-10. The `PolarToPoint` function is described on page 8-56. The `PointToPolar` function is described on page 8-57.

Performing Vector Operations

You can use the `VectorMultiply` function to obtain the dot product of two vectors with 64-bit accuracy. The function takes six parameters: the first parameter specifies the number of long numbers to multiply, and the third and fifth parameters specify the step size to use when walking the arrays to which the second and fourth parameters point.

For example:

`VectorMultiply(4,a,1,b,2,&c)` sets the `wide` number pointed to by the parameter `c` to the following value:

```
a[0] * b[0] +a [1] * b[2] +a[2] * b[4] + a[3] * b[6]
```

If the count is negative, the sign of the terms in the dot product are alternated.

`VectorMultiply(–4,a,1,b,2,&c)` sets the `wide` parameter `c` to the following value and the result is returned in `c`:

```
a[0] * b[0] – a[1] * b[2] + a[2] * b[4] – a[3] * b[6]
```

You can also use `VectorMultiply` to determine the cross-product of a pair of vectors, as in Listing 8-1.

Listing 8-1 Calculating a cross-product with `VectorMultiply`

```
gxPoint *CrossProduct(const gxPoint *a, gxPoint *b, )
{
   wide temp;
   WideShift(VectorMultiply(-2, &a->x, 1, &b->y, -1, &temp), 16);
}
```

You can also use `VectorMultiply` to work with mappings. Listing 8-2 is a sample function that applies a mapping to a single point.

Listing 8-2 Applying a mapping to one point

```
gxPoint *MapPoint(const gxMapping *map, gxPoint *pt)
{
   fixed temp[3] = { 0, 0, fixed1 };
   *(gxPoint *)temp = *pt;
   wide dot;
   fixed p = WideShift(VectorMultiply(3, temp, 1, &map[0][2],
                         3, &dot), 30);
   pt->x = WideDivide(VectorMultiply(3, temp, 1, &map[0][0],
                         3, &dot), p, nil);
   pt->y = WideDivide(VectorMultiply(3, temp, 1, &map[0][1],
                         3, &dot), p, nil);
   return pt;
}
```

The `VectorMultiply` function is described on page 8-54. Functions that perform vector operations are described in the section “Vector Operations” beginning on page 8-54.

Shifting the Bits of a wide Number

You can use the `WideShift` function to shift bits in a `wide` format number. Listing 8-3 shows how to use the `WideShift` function to provide a fixed-point version of the `VectorMultiply` function.

Listing 8-3 Using the `WideShift` function to create a fixed-point `VectorMultiply` function

```
Fixed VectorFixMul(long count, Fixed *vector1, long step1,
                     Fixed *vector2, long step2)
{
   wide temp;
   return WideShift(VectorMultiply(count, vector1, step1,
                     vector2, step2, &temp), 16)->lo;
}
```

Listing 8-4 shows how to use the `WideShift` function in a multiplication function for a fixed-point number with a fixed-point bias of 6 bits.

Listing 8-4 Using the `WideShift` function in a fixed-point multiplication function

```
long MultiplyDot6(long a, long b)
{
   wide temp;
   return (long)WideShift(WideMultiply(a, b, &temp), 6)->lo;
}
```

Listing 8-5 shows how to use the `WideShift` function in a division function for a fixed-point number with a fixed-point bias of 6 bits. Listing 8-6 gives an alternative, but equivalent, approach.

Listing 8-5 Using the `WideShift` function to create a fixed-point division function

```
long DivideDot6(long a, long b)
{
   wide temp;
   temp.hi = (temp.lo = a) < 0 ? -1 : 0;  /* sign extend a */
   return WideDivide(WideShift(&temp, -6), b, 0);
}
```

Listing 8-6 shows how to use the `WideShift` function for a second fixed-point division function with a fixed-point bias of 6 bits. Listing 8-5 gives an alternative, but equivalent, approach.

Listing 8-6 Using the `WideShift` function to create a second fixed-point division function

```
long DivideDot6(long a, long b)
{
   wide temp;
   temp.hi = a;
   temp.lo = 0;
   return WideDivide(WideShift(&temp, 26), b, 0);
}
```

Determining the Highest Order Bit of a wide Number

You can use the `WideScale` function to obtain the bit number of the highest order bit in the absolute value of a `wide` number. Listing 8-3 shows how to use the `WideScale` function in a function that multiplies two numbers in `long` format. If the product is too big to fit in a `long`, the function shifts the product so that it fits into a `long` and returns the bit shift. This operation can be termed *pseudo-floating-point*.

Listing 8-7 Using the `WideScale` function to create a pseudo-floating-point function

```
long FloatMul(long a, long b, long *product)
{
   wide temp;
   long shift = WideScale(WideMultiply(a, b, &temp)) - 30;
   if (shift > 0)
      WideShift(&temp, shift);
   else
      shift = 0;
   if (product) *product = temp.lo;
   return shift;
}
```

The `WideScale` function is described on page 8-53.

Generating Random Numbers

You can use the QuickDraw GX random number functions to return a sequence of random numbers. The `RandomBits` function generates random integers in the range of 0 to $2^{count} - 1$, where *count* is the number of bits in the integer to be generated by the random number generator.

The `SetRandomSeed` function allows you to use a seed other than the default seed. If the `SetRandomSeed` function is not used, the initial seed will always be 0. You can use the `GetRandomSeed` function to return the value of the current seed.

Listing 8-8 is a sample function that generates an unsigned random number between zero and the value passed in the `limit` parameter. It uses the `RandomBits` function, and it also uses the `WideMultiply` function, correcting for the fact that `WideMultiply` works with signed long integers whereas this random generator uses unsigned longs.

Listing 8-8 A random number generator

```
unsigned long RandomLong(unsigned long limit)
{
   wide temp;
   unsigned long random = RandomBits(32, 0);

   /* This treats random and limit as signed */
   WideMultiply(random, limit, &temp);
   if ((long)limit < 0)
      temp.hi += random;   /* correct for the “sign” of limit */
   if ((long)random < 0)
      temp.hi += limit;    /* correct for the “sign” of random */
   return temp.hi;
}
```

The general topic of random numbers and the functions you use to generate them generation are discussed in the section “Random Number Generation” beginning on page 8-58.

Analyzing the Bits in a Number

You can use the `FirstBit` function to determine the highest bit number that is set in a 32-bit number. The following examples demonstrate the use of this function with the parameter `x`:

If `x` is 1, the highest order bit that is set is bit number 0,
so `FirstBit(1)` = 0, as shown below.

```
FirstBit(00000000000000000000000000000001) = 0x0000
```

If x is 2, the highest order bit that is set is bit number 1, so `FirstBit(2)` = 1, as shown below.

```
FirstBit(00000000000000000000000000000010) = 0x0001
```

If x is 3, the highest order bit that is set is bit 1, so `FirstBit(3)` = 1, as shown below.

```
FirstBit(00000000000000000000000000000011) = 0x0001
```

If no bits in the number are set, `FirstBit` returns a value of –1.

You can also use `FirstBit` to find the *last* (= lowest-order) bit that is set in a number. Listing 8-9 is an example of such a function.

Listing 8-9 Determining the lowest bit of a number

```
short LastBit(unsigned long x)
{
   if (x == 0)
      return 32;
   return FirstBit(x & -x);
}
```

The `FirstBit` function is described on page 8-62.

Resetting a Mapping

You can use the `ResetMapping` function to reset a mapping. The following code example first uses the `ResetMapping` function to initialize the destination to the identity matrix, and then uses `RotateMapping` to calculate a resultant mapping that rotates by a given angle about a specified center.

```
gxMapping *RotationMap(gxMapping *dest, Fixed angle,
                        gxPoint *center)
{
    return RotateMapping(ResetMapping(dest), angle,
                           center->x,center->y);
}
```

The `ResetMapping` function is described on page 8-64.

QuickDraw GX Mathematics Reference

This section describes the constants, data types, structures, macros, and functions that relate to QuickDraw GX mathematics.

Constants and Data Types

This section describes the constants and data types that are used to define QuickDraw GX mathematical number formats and the transformation matrix.

Number Formats and Constants

QuickDraw GX provides `Fixed`, `fract`, and `gxColorValue` number formats. Polar coordinates are defined by the `gxPolar` structure. A structure consisting of two `long` values defines the `wide` number format.

```
typedef long fract;

typedef unsigned short gxColorValue;

struct gxPolar {
   Fixed radius;
   Fixed angle;
};

struct wide {
   long hi;
   unsigned long lo;
};
```

For convenience, QuickDraw GX provides constants for the value 1.0 for `Fixed`, `fract`, and `gxColorValue` types:

```
#define fixed1        ((Fixed) 0x00010000)
#define fract1        ((fract) 0x40000000)
#define gxColorValue1 ((gxColorValue) 0xFFFF)
```

QuickDraw GX also provides constants for the largest and smallest possible values for `Fixed` and `fract` numbers:

```
#define gxPositiveInfinity ((Fixed) 0x7FFFFFFF)
#define gxNegativeInfinity ((Fixed) 0x80000000)
```

The Mapping Structure

QuickDraw GX defines a transformation matrix with the `gxMapping` structure:

```
struct gxMapping {
   Fixed map[3][3];
};
```

Field descriptions

`map`	A 3 × 3 array of `Fixed` numbers whose values determine the translation, scaling, rotation, skewing, and perspective operations that can be applied to two-dimensional data. Although defined as containing only `Fixed` numbers, the rightmost column of the matrix consists of `fract` numbers. Furthermore, element `[3][3]` is commonly set to `fract1`.

The use of the mapping matrix is described further in the section "Transformation Operations With Mappings" beginning on page 8-12.

Number-Conversion Macros

QuickDraw GX defines macros for conversion between fixed-point number formats. It also provides macros to round and truncate numbers, as well as a macro that uses the `FractSquareRoot` function to compute the square root of a `Fixed` number.

Format Conversions

The macros in this section convert between `Fixed`, `fract`, integer, floating-point, and `gxColorValue` numbers.

FixedToFract

You can use the `FixedToFract` macro to convert a fixed number to a `fract` number.

```
#define FixedToFract(a) ((fract) (a) << 14)
```

`a`	A `Fixed` number to be converted to a `fract` number, $-2 \le a < 2$.
macro result	A `fract` number having the same value as the fixed number.

FractToFixed

You can use the `FractToFixed` macro to convert a `fract` number to a `Fixed` number.

```
#define FractToFixed (a) ((Fixed) (a) + 8192L >> 14)
```

`a`	A `fract` number to be converted to a `Fixed` number.
macro result	A `Fixed` number having the closest value to the `fract` number.

FixedToInt

You can use the `FixedToInt` macro to convert a `Fixed` number to an integer.

```
#define FixedToInt(a) ((short) ((Fixed) (a) + fixed1/2 >> 16))
```

`a`	A `Fixed` number to be converted to an integer.
macro result	An integer having the closest value to the `Fixed` number.

IntToFixed

You can use the `IntToFixed` macro to convert an integer to a `Fixed` number.

```
#define IntToFixed(a) ((Fixed)(a) << 16)
```

`a`	An integer to be converted to a `Fixed` number.
macro result	A `Fixed` number having the same value as the integer.

SPECIAL CONSIDERATIONS

QuickDraw GX also defines a shorthand version of this macro. `IntToFixed(a)` can also be coded as `ff(a)`.

SEE ALSO

The `ff` macro is described next.

ff

You can use the `ff` macro to convert an integer to a `Fixed` number.

```
#define ff(a) ((Fixed)(a) << 16)
```

`a`	An integer to be converted to a `Fixed` number.
macro result	A `Fixed` number having the same value as the integer.

DESCRIPTION

The `ff` macro converts an integer `a` to a `Fixed` number. This macro name is shorthand notation for the `IntToFixed` macro, and provides identical functionality.

SEE ALSO

For an example of how to use the `ff` macro, see the section "Converting Number Formats" beginning on page 8-26.

The `IntToFixed` macro is described in the previous section.

FixedToFloat

You can use the `FixedToFloat` macro to convert a `Fixed` number to a floating-point number.

```
#define FixedToFloat(a) ((float)(a) / fixed1)
```

`a`	A `Fixed` number to be converted to a floating-point number.
macro result	A floating-point number having the same value as the `Fixed` number.

FloatToFixed

You can use the `FloatToFixed` macro to convert a floating-point number to a `Fixed` number.

```
#define FloatToFixed(a) ((Fixed)((float) (a) * fixed1))
```

`a`	A floating-point number to be converted to a `Fixed` number.
macro result	The closest `Fixed` number to the floating-point number.

SPECIAL CONSIDERATIONS

QuickDraw GX also defines a shorthand version of this macro. The `FloatToFixed` macro can also be coded as `fl(a)`.

SEE ALSO

The `fl` macro is described next.

fl

You can use the `fl` macro to convert a floating-point number to a `Fixed` number.

```
#define fl(a) ((Fixed)((float) (a) * fixed1))
```

`a`	A floating-point number to be converted to a `Fixed` number.
macro result	The closest `Fixed` number to the floating-point number.

DESCRIPTION

The `fl` macro converts a floating-point number `a` to a `Fixed` number. This macro name is shorthand notation for the `FloatToFixed` macro, and provides identical functionality.

SEE ALSO

The `FloatToFixed` macro is described in the previous section.

FractToFloat

You can use the `FractToFloat` macro to convert a `fract` number to a floating-point number.

```
define FractToFloat(a) ((float)(a)/fract1)
```

`a`	A `fract` number to be converted to a floating-point number.
macro result	A floating-point number having the closest value to the `fract` number.

FloatToFract

You can use the `FloatToFract` macro to convert a floating-point number to a `fract` number.

```
define FloatToFract(a) ((fract)((float)(a)*fract1))
```

`a`	A floating-point number to be converted to a `fract` number.
macro result	A `fract` number having the closest value to the floating-point number.

ColorToFract

You can use the `ColorToFract` macro to convert a `gxColorValue` number to a `fract` number.

```
#define ColorToFract(a) (((fract)(a)<<14) + ((fract)(a) +2 >>2))
```

`a`	A `gxColorValue` number to be converted to a `fract` number.
macro result	A `fract` number having the same value as the `gxColorValue` number.

FractToColor

You can use the `FractToColor` macro to convert a `fract` number to a `gxColorValue` number.

```
#define FractToColor(a) ((gxColorValue)((a)-((a)>>16)+8191>>14))
```

`a`	A `fract` number to be converted to a `gxColorValue` number.
macro result	The closest `gxColorValue` number to the `fract` number.

Rounding, Truncating, and Square Root Operations

The macros in this section round, truncate, and determine the square root of fixed numbers.

FixedRound

You can use the `FixedRound` macro to round a `Fixed` number to its nearest integer.

```
#define FixedRound(a) ((short) ((Fixed)(a) + fixed1/2 >> 16))
```

`a`	The number to be rounded.
macro result	The closest integer to the `Fixed` number.

FixedTruncate

You can use the `FixedTruncate` macro to obtain an integer that is the greatest integer that is not greater than the given `Fixed` number.

```
#define FixedTruncate(a) ((short)((Fixed)(a) >> 16))
```

`a`	The number that is to be truncated.
macro result	The largest integer that is not greater than the `Fixed` number.

FixedSquareRoot

You can use the `FixedSquareRoot` macro to determine the square root of a fixed number.

```
#define FixedSquareRoot(a) ((Fixed)FractSquareRoot(a) + 64 >>7)
```

`a`	The number for which the square root is to be determined.

macro result The square root of the number.

Mathematical Functions

This section describes the QuickDraw GX functions you can use to perform

- fixed-point operations
- `wide` number operations
- vector operations
- mapping operations
- random number generation
- bit analysis

Fixed-Point Operations

QuickDraw GX provides an assortment of fixed-point mathematical functions that you can use in your application.

FixedMultiply

You can use the `FixedMultiply` function to return the product of two numbers.

```
Fixed FixedMultiply (Fixed multiplicand, Fixed multiplier);
```

`multiplicand`	The number to be multiplied by the multiplier.
`multiplier`	The number by which the multiplicand is to be multiplied.

function result The product of two numbers.

DESCRIPTION

The `FixedMultiply` function multiplies two fixed numbers. The format of the `Fixed` number returned depends on the respective number formats of the multiplicand and multiplier. The operation has a bias of 16 bits; in general, the bias of the resulting number is the sum of the biases of the input numbers, shifted right by 16 bits. If either the multiplicand or the multiplier is `Fixed`, the result of the `FixedMultiply` function will be the same fixed-point format as the other parameter (`long`, `Fixed`, or `fract`).

Table 8-3 shows the bias of the product for different combinations of formats. The dashed line indicates that the resulting bias is not equivalent to long, fixed, or fract. Use the rules of the operation to determine it.

Table 8-3 `FixedMultiply` product bias

	long	**fixed**	**fract**
long	---	long	---
fixed	long	fixed	fract
fract	---	fract	---

SPECIAL CONSIDERATIONS

The `FixedMultiply` function does not pin its result in the case of an overflow; the result returned is modulo 65,536.

FixedDivide

You can use the `FixedDivide` function to return the quotient of a dividend and divisor.

```
Fixed FixedDivide (Fixed dividend, Fixed divisor);
```

`dividend`	The number to be divided.
`divisor`	The number by which the dividend is to be divided.
function result	The quotient of the dividend and the divisor.

DESCRIPTION

The `FixedDivide` function divides the `dividend` parameter by the `divisor` parameter and returns the quotient. The format of the fixed number returned depends on the respective number formats of the `dividend` and `divisor` parameters. The operation has a bias of 16 bits; in general, the bias of the resulting number is the difference between the biases of the input numbers, shifted left by 16 bits. If the `divisor` parameter is fixed, then the result will be the same fixed-point format as the dividend. If both the dividend and divisor are the same fixed-point format, the result will be in `Fixed` format.

Table 8-4 shows the bias for the quotient of two numbers that are of dissimilar formats. The dashed line indicates that the resulting bias is not equivalent to long, fixed, or fract. Use the rules of the operation to determine it.

Table 8-4 `FixedDivide` quotient bias

Denominator	**Numerator**		
	long	fixed	fract
long	fixed	---	---
fixed	long	fixed	fract
fract	---	---	fixed

SPECIAL CONSIDERATIONS

In the case of overflow, `FixedDivide` pins its result to either the `gxPositiveInfinity` or `gxNegativeInfinity` constant.

MultiplyDivide

You can use the `MultiplyDivide` function to multiply two numbers and divide by a third number.

```
long MultiplyDivide (long multiplicand, long multiplier,
                     long divisor);
```

`multiplicand`
The number to be multiplied by the multiplier.

`multiplier`
The number by which the multiplicand is multiplied.

`divisor` The number by which the product is divided.

function result The quotient of the product of two numbers and the divisor.

DESCRIPTION

The `MultiplyDivide` function calculates the quotient of the product of two numbers (parameters `multiplicand` and `multiplier`) and a divisor.

The function uses a 64-bit intermediate result to maintain accuracy and to prevent premature overflow. The parameters do not need to all be the same fixed-point format. The operation has a bias of 0 bits; if the divisor is the same format as the multiplier, the result is the same format as the multiplicand. If the divisor is the same format as the multiplicand, the result is in the same format as the multiplier.

SPECIAL CONSIDERATIONS

In the case of overflow, `MultiplyDivide` pins its result to either the `gxPositiveInfinity` or `gxNegativeInfinity` constant.

Magnitude

You can use the `Magnitude` function to obtain the magnitude of a vector, the length of a line, or the distance between two points.

```
unsigned long Magnitude (long deltaX, long deltaY);
```

`deltaX`	The difference in the x-coordinates of the vector’s end points.
`deltaY`	The difference in the y-coordinates of the vector’s end points.
function result	The magnitude of the vector.

DESCRIPTION

The `Magnitude` function returns $(\texttt{deltaX}^2 + \texttt{deltaY}^2)^{1/2}$, the Euclidean distance between two points whose x-coordinates are separated by `deltaX` and whose y-coordinates are separated by `deltaY`.

The fixed-point format of the result is the same as the fixed-point format for both of the parameters. Make sure that the two parameters use the same format.

FractSineCosine

You can use the `FractSineCosine` function to obtain both the sine and cosine of an angle measured in degrees.

```
fract FractSineCosine (Fixed degrees, fract *cosine);
```

`degrees`	The angle in degrees for which the cosine and sine are required.
`cosine`	A pointer to the location where the cosine of the angle is required.

function result The sine of the angle specified.

DESCRIPTION

Given the `degrees` parameter in degrees, the `FractSineCosine` function returns the sine as the function result and the cosine in the `cosine` parameter. Values for the `degrees` parameter are specified in degrees, not radians. The range of the angle is –32,768 to +32,769.999 degrees.

FractSquareRoot

You can use the `FractSquareRoot` function to calculate the square root of a `fract` number.

```
fract FractSquareRoot (fract source);
```

`source`	The number for which the square root is required.

function result The square root of the `fract` number.

DESCRIPTION

The `FractSquareRoot` function returns the square root of the `fract` number specified by the `source` parameter. The number is interpreted as unsigned and in the range 0 through $4 - (2^{-30})$. This means that bit 31 has a weight of 2, instead of –2. The result is an unsigned number in the range of 0 through 2.

FractCubeRoot

You can use the `FractCubeRoot` function to calculate the cube root of a `fract` number.

```
fract FractCubeRoot (fract source);
```

`source` The `fract` number for which the cube root is required.

function result The cube root of the `fract` number. This number is a signed value.

DESCRIPTION

The `FractCubeRoot` function returns the cube root of a `fract` number.

FractMultiply

You can use the `FractMultiply` function to calculate the product of two numbers.

```
fract FractMultiply (fract multiplicand, fract multiplier);
```

`multiplicand`
The number to be multiplied by the multiplier.

`multiplier`
The number by which the multiplicand is to multiplied.

function result The product of two numbers.

DESCRIPTION

The `FractMultiply` function calculates the product of two numbers, specified in the `multiplicand` and `multiplier` parameters. If the parameters are a and b, the product a × b is returned.

The format of the number returned depends on the respective number formats of the `multiplicand` and `multiplier` parameters. The operation has a bias of 30 bits; in general, the bias of the resulting number is the sum of the biases of the input numbers, shifted right by 30 bits. Thus if either the `multiplicand` or `multiplier` parameter is `fract`, then the result is the same fixed-point format as the other argument.

Table 8-5 shows the bias of the FractMultiply result. The dashed line indicates that the resulting bias is not equivalent to long, fixed, or fract. Use the rules of the operation to determine it

Table 8-5 `FractMultiply` result bias

	long	**fixed**	**fract**
long	---	---	long
fixed	---	---	fixed
fract	long	fixed	fract

SPECIAL CONSIDERATIONS

`FractMultiply` does not pin its result in the case of an overflow; the result returned is modulo 4.

FractDivide

You can use the `FractDivide` function to return the quotient of a dividend and divisor.

```
fract FractDivide (fract dividend, fract divisor);
```

`dividend`	The number to be divided.
`divisor`	The number by which the dividend is to be divided.

function result The quotient of two numbers.

DESCRIPTION

The `FractDivide` function divides the `dividend` parameter by the `divisor` parameter and returns the quotient. If the `dividend` parameter is a and the `divisor` parameter is b, the quotient a / b is returned.

The format of the number returned depends on the respective number formats of the dividend and divisor. The operation has a bias of 30 bits; in general, the bias of the resulting number is the difference between the biases of the input numbers, shifted left by 30 bits. Thus if the divisor is a `fract`, the result is the same format as the dividend. If the `divisor` and the `dividend` parameters are the same format, the result is in `fract` format, as shown inTable 8-6. The dashed line indicates that the resulting bias is not equivalent to long, fixed, or fract. Use the rules of the operation to determine it.

Table 8-6 `FractDivide` result bias

Denominator	Numerator		
	long	fixed	fract
long	fract	---	---
fixed	---	fract	---
fract	long	fixed	fixed

SPECIAL CONSIDERATIONS

In the case of division of a large number by a very small number,the `FractDivide` function pins its result to either the `gxPositiveInfinity` or the `gxNegativeInfinity` constant.

Operations on wide Numbers

QuickDraw GX provides an assortment of 64-bit mathematical functions for your use. You can use `wide` functions to increase the accuracy of calculations.

WideAdd

You can use the `WideAdd` function to add two `wide` numbers.

```
wide *WideAdd(wide *target, const wide *source);
```

`target`	A pointer to the number to be added to. On return, contains the sum of the two numbers.
`source`	A pointer to the number that is to be added to the target number.
function result	A pointer to the result (also a pointer to the target number).

DESCRIPTION

The `WideAdd` function adds the `wide` number in the `source` parameter to the `wide` number in the `target` parameter and returns the target pointer.

WideSubtract

You can use the `WideSubtract` function to subtract one `wide` number from another.

```
wide *WideSubtract(wide *target, const wide *source);
```

`target`	A pointer to the number to be subtracted from. On return, contains the difference between the two numbers.
`source`	A pointer to the number that is to be subtracted from the number at target.
function result	A pointer to the target number.

DESCRIPTION

The `WideSubtract` function subtracts the source number from the target number and returns a pointer to the target number.

WideNegate

You can use the `WideNegate` function to change a `wide` number to its negative.

```
wide *WideNegate(wide *target);
```

`target`	A pointer to the number to be negated. On return, contains the negated number.
function result	A pointer to the target number.

WideShift

You can use the `WideShift` function to shift bits in a `wide` number.

```
wide *WideShift(wide *target, long shift);
```

`target`	A pointer to the number for which the bits are to be shifted. On return, contains the shifted number.
`shift`	The number of bits by which the target is to be shifted to the right.

function result A pointer to the target number.

DESCRIPTION

The shift direction is to the right (a decrease in magnitude) if the `shift` parameter is greater than 0, and to the left if the `shift` parameter is less than 0. The result of a right shift is rounded.

WideMultiply

You can use the `WideMultiply` function to calculate the `wide` product of two `long` numbers.

```
wide *WideMultiply(long multiplicand, long multiplier,
                     wide *target);
```

`multiplicand`	The number to be multiplied by the multiplier.
`multiplier`	The number by which the multiplicand is to be multiplied.
`target`	A pointer to the location where the product of the two numbers is to be stored.

function result A pointer to the target value, which holds the result.

DESCRIPTION

The operation has a bias of 0 bits. The bias of the result is the sum of the biases of the inputs.

WideDivide

You can use the `WideDivide` function to calculate the `long` quotient and `long` remainder for a `wide` dividend and `long` divisor.

```
long WideDivide(const wide *dividend, long divisor,
                long *remainder);
```

`dividend`	A pointer to the `wide` number to be divided.
`divisor`	The number by which the dividend is to be divided.
`remainder`	A pointer to a location to store the remainder of the division.
function result	The quotient of the division.

DESCRIPTION

The `WideDivide` function divides the dividend by the divisor and returns the quotient in the function result and the remainder in the `long` number pointed to by the `remainder` parameter. If the dividend is a and the divisor is b, the quotient a / b is returned with a remainder. The operation has a bias of 0 bits; the bias of the result is the difference between the biases of the dividend and the divisor. The bias of the remainder is the same as the bias of the dividend.

If an overflow occurs, the result is pinned to the closest infinity and the remainder is set to `gxNegativeInfinity` (an impossible remainder).

If the `remainder` parameter is `nil`, no remainder is returned and the `WideDivide` function returns a rounded quotient. Passing `(long *)-1` in the `remainder` parameter is the same as passing `nil` except in the case of an overflow, in which case `gxNegativeInfinity` is returned.

WideWideDivide

You can use the `WideWideDivide` function to calculate a `wide` quotient and `long` remainder for a `wide` dividend and a `long` divisor.

```
wide *WideWideDivide(wide *dividend, long divisor,
      long *remainder);
```

`dividend`	A pointer to the `wide` number to be divided.
`divisor`	The number by which the dividend is to be divided.
`remainder`	A pointer to a location to store the remainder of the division.
function result	A pointer to the quotient (also to the dividend).

DESCRIPTION

The `WideWideDivide` function returns the quotient of the dividend and divisor as its function result and places the remainder in the `remainder` parameter. If the `remainder` parameter is `nil`, `WideWideDivide` returns the rounded quotient. The quotient replaces the dividend. The operation has a bias of 0 bits; the bias of the result is the difference between the biases of the dividend and the divisor. The bias of the remainder is the same as the bias of the dividend.

If the `remainder` parameter is `nil`, no remainder is returned and the `WideDivide` function returns a rounded quotient. Passing `(long *)-1` in the `remainder` parameter is the same as passing `nil`.

Note that this function cannot result in overflow.

WideSquareRoot

You can use the `WideSquareRoot` function to calculate the square root of a `wide` number.

```
unsigned long WideSquareRoot(const wide *source);
```

`source`	A pointer to the number for which the square root is to be calculated.
function result	A number that is the square root of the number in the argument.

DESCRIPTION

The `WideSquareRoot` function returns the square root of the `wide` number pointed to by the `source` parameter. The source value for this function must be an unsigned `wide` value ranging from 0 to $2^{64} - 1$, not -2^{63} to $2^{63} - 1$. If you supply a non-integer value for this function, its bias must be an even number of bits.

WideScale

You can use the `WideScale` function to obtain the bit number of the highest-order nonzero bit in the absolute value of a `wide` number.

```
short WideScale(const wide *w);
```

`w`	A pointer to the number whose scale is desired.
function result	The bit number of the highest order nonzero bit in the absolute value of `w`. The returned value is 63 if the highest-order bit is set, and 0 if the lowest order bit is set. If no bit is set, the return value is –1.

WideCompare

You can use the `WideCompare` function to compare the magnitudes of two 64-bit numbers.

```
short WideCompare(const wide *target, const wide *source);
```

`target`	A pointer to one of the two `wide` numbers to be compared.
`source`	A pointer to the second of the two `wide` numbers to be compared.

function result 1 if the target number is greater, –1 if the source number is greater, and 0 if the two numbers are equal.

Vector Operations

QuickDraw GX provides an assortment of vector mathematics functions for your use.

VectorMultiply

You can use the `VectorMultiply` function to obtain the dot product of two vectors with 64-bit accuracy.

```
wide *VectorMultiply(long count, const long *vector1, long
step1, const long *vector2, long step2, wide *dot);
```

`count`	The size of each vector.
`vector1`	A pointer to one of the two vectors.
`step1`	The index increment for the `vector1` vector.
`vector2`	A pointer to the second of two vectors.
`step2`	The index increment for the `vector2` vector.
`dot`	A pointer to the destination of the result.

function result A pointer to the dot product of the two vectors.

DESCRIPTION

The `VectorMultiply` function calculates the `wide` dot product of the parameters `vector1` and `vector2`. The size of each vector is given by the `count` parameter. The index increment is given by the parameters `step1` and `step2`, respectively. The `dot` parameter points to the destination `wide` number and is returned as the function result.

SEE ALSO

Examples of how to use the `VectorMultiply` function are provided in the section "Performing Vector Operations" beginning on page 8-29.

VectorMultiplyDivide

You can use the `VectorMultiplyDivide` function to calculate the quotient of the dot product of two vectors and a divisor.

```
long *VectorMultiplyDivide(long count, const long *vector1,
                           long step1, const long *vector2,
                           long step2, long divisor);
```

`count`	The size of each vector.
`vector1`	A pointer to one of the two vectors.
`step1`	The index increment for the `vector1` vector.
`vector2`	A pointer to the second of two vectors.
`step2`	The index increment for the `vector2` vector.
`divisor`	The number by which the dot product is to be divided.
function result	The quotient of the dot product of two vectors and a divisor.

DESCRIPTION

The `VectorMultiplyDivide` function calculates the quotient of a dot product of parameters `vector1` and `vector2` and a `divisor` parameter. The size of each vector is given by the `count` parameter. The index increment is given by the parameters `step1` and `step2`, respectively. If the `count` parameter is negative, the terms are alternated. This is equivalent to

```
WideDivide(VectorMultiply(),divisor)
```

Cartesian and Polar Coordinate Point Conversions

QuickDraw GX provides two functions for converting Cartesian to polar coordinates.

PolarToPoint

You can use the `PolarToPoint` function to convert a point in polar coordinates to the identical point in Cartesian coordinates.

```
gxPoint *PolarToPoint(const gxPolar *ra, gxPoint *xy);
```

`ra`	A pointer to the point in polar coordinates.
`xy`	A pointer to the destination of the resulting point in Cartesian coordinates.
function result	A pointer to the converted point (also a pointer to the `xy` parameter).

DESCRIPTION

The `PolarToPoint` function converts the polar coordinate point (*r*, *a*) to the identical Cartesian coordinate point (*x*, *y*). The parameters of the `PolarToPoint` function are the `gxPolar` structure pointer `ra` and a `gxPoint` structure pointer `xy`.

If both pointers point to the same location, the source `gxPolar` structure will be converted to a `gxPoint` structure and will replace the `gxPolar` structure.

SEE ALSO

The `gxPolar` structure is described in the section “Constants and Data Types” beginning on page 8-35. Polar coordinate to Cartesian coordinate conversions are discussed in the section “Cartesian and Polar Coordinate Conversion” beginning on page 8-10. The `PointToPolar` function converts a point in Cartesian coordinates to the identical point in polar coordinates. The `PointToPolar` function is described next.

PointToPolar

The `PointToPolar` function converts a point in Cartesian coordinates to the identical point in polar coordinates.

```
gxPolar *PointToPolar(const gxPoint *xy, gxPolar *ra);
```

`xy`	A pointer to the Cartesian coordinate.
`ra`	A pointer to the destination of the resulting polar coordinate.
function result	The pointer passed in `ra`.

DESCRIPTION

The `PointToPolar` function converts the Cartesian coordinate point (*x*, *y*) to the identical polar coordinate point (*r*, *a*). The parameters of the `PointToPolar` function are a `gxPoint` structure pointer `xy` and a `gxPolar` structure pointer `ra`.

If both pointers point to the same location, the source `gxPoint` structure will be converted to a `gxPolar` structure and will replace the `gxPoint` structure.

SEE ALSO

The `gxPolar` structure is described in the section "Constants and Data Types" beginning on page 8-35. The `PolarToPoint` function converts a point in polar coordinates to the identical point in Cartesian coordinates. The `PolarToPoint` function is described in the previous section.

Random Number Generation

QuickDraw GX provides random number generation functions that can be used in your application.

RandomBits

You can use the `RandomBits` function to return a sequence of pseudorandom numbers.

```
unsigned long RandomBits(long count, long focus);
```

`count`	The number of bits in the number to be generated by the random number generator.
`focus`	The degree of clustering about the mean value.

function result A sequence of pseudorandom numbers.

DESCRIPTION

The `RandomBits` function returns random numbers in the range of 0 to $2^{\texttt{count}} - 1$. A focus of 0 generates numbers that are uniformly distributed.

A positive value for the `focus` parameter generates numbers that are clustered about the mean, analogous to averaging $2^{\texttt{focus}}$ uniform random numbers. A negative focus generates numbers that tend to avoid the mean.

If you define a value *limit* to be 1 << `count`, the result of the `RandomBits` function ranges from 0 to *limit* – 1. Its mean is (*limit* – 1) / 2. The mean is independent of the focus. If the focus is positive, the standard deviation of the numbers generated by the `RandomBits` function is approximately $(0.28868 \times limit) / e^{1.41421 \times \text{focus}}$. As the `focus` parameter gets bigger, two things happen:

- The values cluster about the mean.
- The values approximate a normal distribution (central limit theorem).

If the focus is negative, the `RandomBits` function result is computed as if it were positive; for results less than *limit* / 2, *limit* / 2 is added; for others, *limit* / 2 is subtracted. This causes the distribution to avoid the mean.

To generate a clustering of points around a given value, generate x and y offsets with

```
FractMultiply(radius, RandomBits(31, focus) - fract1);
```

The average distance will be $0.57735 \times \text{radius} / e^{1.41421 \times \text{focus}}$.

A good way to select a value for the focus is to experiment until the desired result is achieved.

SEE ALSO

The `SetRandomSeed` function sets the starting number seed for the random number generator algorithm. The `SetRandomSeed` function is described in the next section. The `GetRandomSeed` function returns the current starting number seed for the random number generator algorithm. The `GetRandomSeed` function is described on page 8-60.

SetRandomSeed

You can use the `SetRandomSeed` function to set the starting number for the random number generator algorithm.

```
void SetRandomSeed(const wide *seed);
```

`seed`	The pointer to the number to be used by the random number algorithm to generate random numbers.

DESCRIPTION

Random number generators are seeded with a value that is used by the algorithm to generate a random number. The seed is then used to generate the next random number.

The `SetRandomSeed` function allows you to select the seed used by the QuickDraw GX random number algorithm. If `SetRandomSeed` is not used, QuickDraw GX will select a default seed of 0. This results in the same sequence of random numbers each time `RandomBits` is called.

In order to obtain a different set of random numbers than those obtained using the default seed value or a previously set seed, use the `SetRandomSeed` function.

SEE ALSO

The `RandomBits` function uses the current seed to generate the next random number. The `RandomBits` function is described on page 8-58. The `GetRandomSeed` function returns the current seed. The `GetRandomSeed` function is described next.

GetRandomSeed

You can use the `GetRandomSeed` function to return the current seed for the random number generating algorithm.

```
wide *GetRandomSeed(wide *seed);
```

`seed`	A pointer to the current random number generator seed.
function result	The pointer passed in the `seed` parameter.

DESCRIPTION

The `GetRandomSeed` function returns the current seed for the random number generator and returns the pointer passed in `seed`.

SEE ALSO

The `RandomBits` function uses the current seed to generate the next random number. The `RandomBits` function is described in the previous section. The `SetRandomSeed` function changes the current seed. The `SetRandomSeed` function is described in the previous section.

Linear and Quadratic Roots

QuickDraw GX provides two functions that solve for the roots of linear and quadratic equations.

LinearRoot

You can use the `LinearRoot` function to obtain the root of a linear equation.

```
long LinearRoot(Fixed first, Fixed last, fract t[]);
```

`first`	The first coefficient.
`last`	The last coefficient.
`t`	An array of fract numbers. On return, it contains the roots of the equation.
function result	The number of roots of the linear equation. This value may be 0 or 1 (or –1 if all values of t are roots).

DESCRIPTION

The `LinearRoot` function computes any t between 0 and 1 in which a(1 – t) + bt = 0. The coefficient a is the parameter `first`. The coefficient b is the parameter `last`. The function returns the number of roots between 0 and 1.

Any root is returned in the `t` array, which only needs to hold one value. If both a and b are zero, the function returns the number –1, indicating that a(1 – t) + bt = 0 for all t.

QuadraticRoot

You can use the `QuadraticRoot` function to calculate the roots of a quadratic equation.

```
long QuadraticRoot(Fixed first, Fixed control, Fixed last, fract
t[]);
```

`first`	The first coefficient.
`control`	The second coefficient.
`last`	The third coefficient.
`t`	An array of fract numbers. On return, it contains the roots of the equation.
function result	The number of roots of the quadratic equation. This value may be 0, 1, or 2 (or –1 if all values of t are roots).

DESCRIPTION

The `QuadraticRoot` function returns roots between 0 and 1 for quadratic equations having the form $a(1 – t)^2 + 2bt(1 – t) + ct^2 = 0$. The coefficient a is the parameter `first`. The coefficient b is the parameter `control`. The coefficient c is the parameter `last`.

All roots are returned in increasing order in the `t` array. The array can have at most two values. If a, b, and c are all zero, then the function returns the number –1, indicating that $a(1 – t)^2 + 2bt(1 – t) + ct^2 = 0$ for all t.

Bit Analysis

QuickDraw GX provides a function that allows you to analyze the bits in a number.

FirstBit

You can use the `FirstBit` function to determine the highest order bit that is set in a number.

```
short FirstBit (unsigned long x);
```

`x`	The number for which the first bit is to be determined.
function result	The bit number of the highest order bit of the number in the argument.

DESCRIPTION

The `FirstBit` function returns the bit number of the highest order bit in a number that is set, or –1 if the number is 0. The highest-order bit is bit 31; the lowest-order bit is bit 0.

DESCRIPTION

The use of the `FirstBit` function is described in the section “Analyzing the Bits in a Number” on page 8-33.

Mapping Functions

QuickDraw GX provides two groups of mapping functions. The first group allows you to manipulate mapping matrices and apply them to other mappings or to points. The second group allows you to modify the transformational properties of a mapping matrix.

Mappings are described in the section “Transformation Operations With Mappings” beginning on page 8-12.

For specific information on mapping matrices as applied to transform objects, view port objects, and view device objects, see the chapters “Transform Objects” and “View-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*.

Manipulating and Applying Mappings

This section describes functions with which you can copy, normalize, reset, and invert a mapping. It also describes functions with which you can apply a mapping to another mapping, and apply a mapping to an array of points.

CopyToMapping

You can use the `CopyToMapping` function to copy a mapping from one location to another location.

```
gxMapping *CopyToMapping(gxMapping *target,
                         const gxMapping *source);
```

`target`	A pointer to the destination mapping. On return, it is a copy of the source mapping.
`source`	A pointer to the mapping to be copied.
function result	A pointer to the copied mapping, which is also the target mapping.

DESCRIPTION

The `CopyToMapping` function copies the mapping pointed to by the `source` parameter into the location pointed to by the `target` parameter. Note that it may be faster in C to simply copy the `gxMapping` structure into another `gxMapping` structure than to call this function.

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

NormalizeMapping

You can use the `NormalizeMapping` function to normalize a mapping.

```
gxMapping *NormalizeMapping(gxMapping *target);
```

`*target`	A pointer to the mapping to be normalized. On return, it is the normalized mapping.
function result	A pointer to the normalized mapping, which is also the target mapping.

DESCRIPTION

The `NormalizeMapping` function normalizes the target mapping. If the mapping's perspective elements (u and v) are 0, each element of the mapping is divided by element w (`target->[2][2]`). If the mapping has a nonzero perspective, each element is shifted to ensure that `fract1`/2 < |u| + |v| +(|w| >> 15) ≤ `fract1`.

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

ResetMapping

You can use the `ResetMapping` function to reset a mapping.

```
gxMapping *ResetMapping(gxMapping *target);
```

`target`	A pointer to the mapping that is to be reset. On return, contains the identity mapping.
function result	A pointer to the reset mapping, which is also the target mapping.

DESCRIPTION

The `ResetMapping` function resets the target mapping to the identity matrix.

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

SEE ALSO

An example of the use of the `ResetMapping` function is provided on page 8-34.

InvertMapping

You can use the `InvertMapping` function to create an inverted copy of a mapping.

```
gxMapping *InvertMapping(gxMapping *target,
                         const gxMapping *source);
```

`target`	A pointer to a mapping structure. On return, contains the inverse of the mapping specified in the `source` parameter.
`source`	A pointer to the mapping to be inverted.

function result A pointer to the inverted mapping, which is also the target mapping.

DESCRIPTION

The `InvertMapping` function creates a copy of the source mapping, inverts it, and returns the inverted mapping in the `target` parameter. If both the `source` and `target` parameters point to the same `gxMapping` structure, that mapping will be replaced by its inverse. If the mapping is not invertible, the function returns nil and the target is not changed.

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

MapMapping

You can use the `MapMapping` function to concatenate two mappings.

```
gxMapping *MapMapping(gxMapping *target, const gxMapping *source);
```

`target`	A pointer to the mapping to be modified. On return, contains the result of the concatenation.
`source`	A pointer to the mapping to be concatenated with the target mapping.

function result A pointer to the resultant mapping, which is also the target mapping.

DESCRIPTION

The `MapMapping` function postmultiplies the target mapping by the source mapping, and returns the result in the `target` parameter.

The result of passing the function result of `MapMapping` tothe `GXMapShape` function is equivalent to passing the result of one call to `GXMapShape` to another call to `GXMapShape`, as shown below (for the shape `s`):

```
GXMapShape(s, target);
GXMapShape(s, source);
```

The same results would be obtained more efficiently and perhaps more accurately by making these calls:

```
MapMapping(target, source);
GXMapShape(s, target);
```

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

SEE ALSO

The `GXMapShape` function is described in the chapter Transform Objects in *Inside Macintosh: QuickDraw GX Objects.*

MapPoints

You can use the `MapPoints` function to apply a mapping to each of the points in an array.

```
void MapPoints(const gxMapping *source, long count,
               gxPoint vector[]);
```

`source`	A pointer to the mapping that is to be applied to the array of points.
`count`	The number of points in the array.
`vector`	The array of points to which the mapping is to be applied. On return, the array contains the transformed points.

ERRORS, WARNINGS, AND NOTICES

Errors

```
mapping_is_nil
parameter_is_nil
number_of_points_exceeds_implementation_limit
```

Warnings

```
map_points_out_of_range
```

SEE ALSO

For an example of a function that applies a mapping to a single point, see Listing 8-2 on page 8-30.

Modifying Mappings

This section describes functions with which you can modify the translational, scaling, rotational, and skewing properties of a mapping.

Similar functions are available that allow you to directly modify the transformational properties of the mapping in the transform object associated with a QuickDraw GX shape. See the chapter "Transform Objects" in *Inside Macintosh: QuickDraw GX Objects* for more information.

MoveMapping

You can use the `MoveMapping` function to change the horizontal and vertical translation factors of a mapping by given amounts.

```
gxMapping *MoveMapping(gxMapping *target, Fixed hOffset,
                        Fixed vOffset);
```

`target`	A pointer to the mapping to be modified. On return, points to the modified mapping.
`hOffset`	The horizontal translation to add to the mapping.
`vOffset`	The vertical translation to add to the mapping.

function result A pointer to the modified mapping, which is also the target mapping.

DESCRIPTION

The `MoveMapping` function postmultiplies the target mapping by a mapping that adds `hOffset` to the x translation and `vOffset` to the y translation of the target mapping.

Passing the result of this function to the `GXMapShape` function is equivalent to calling the `GXMapShape` function and then calling the `GXMoveShape` function.

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

SEE ALSO

The use of the `MoveMapping` function is described in the section "Translation by a Relative Amount" beginning on page 8-17.

The `GXMapShape` and `GXMoveShape` functions are described in the chapter "Transform Objects" in *Inside Macintosh: QuickDraw GX Objects.*

MoveMappingTo

You can use the `MoveMappingTo` function to assign specific values to the horizontal and vertical translation factors of a mapping.

```
gxMapping *MoveMappingTo(gxMapping *target, Fixed hPosition,
                                Fixed vPosition);
```

`target`	A pointer to the mapping that is to be modified. On return, points to the modified mapping.
`hPosition`	The horizontal translation to be assigned to the target mapping.
`vPosition`	The vertical translation to be assigned to the target mapping.
function result	A pointer to the modified mapping, which is also the target mapping.

DESCRIPTION

The `MoveMappingTo` function postmultiplies the target mapping by a mapping that assigns `hPosition` to the x translation and `vPosition` to the y translation of the target mapping. This function sets the translational origin of the mapping; the point (0, 0), when postmultiplied by the mapping that results from this function, will be at location (`hPosition`, `vPosition`).

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

SEE ALSO

The use of the `MoveMappingTo` function is described in the section “Translation to a Specified Point” beginning on page 8-18.

ScaleMapping

You can use the `ScaleMapping` function to change the horizontal and vertical scale factors of a mapping.

```
gxMapping *ScaleMapping(gxMapping *target,
                        Fixed hFactor, Fixed vFactor,
                        Fixed xCenter, Fixed yCenter);
```

`target`	A pointer to the mapping that is to be modified. On return, points to the modified mapping.
`hFactor`	The horizontal scaling factor to apply. A value of 1.0 means no scale change in the x direction.
`vFactor`	The vertical scaling factor to apply. A value of 1.0 means no scale change in the y direction.
`xCenter`	The x-coordinate of the center of scaling.
`yCenter`	The y-coordinate of the center of scaling.

function result A pointer to the modified mapping, which is also the target mapping.

DESCRIPTION

The `ScaleMapping` function postmultiplies the target mapping by a mapping that specifies a horizontal scaling factor of `hFactor` and a vertical scaling factor of `vFactor`, about the point (`xCenter`, `yCenter`). Note that if `hFactor` is 1, `xCenter` irrelevant; likewise, if `vFactor` is 1, `yCenter` is irrelevant.

These scaling factors are in addition to any preexisting scaling factors in the target mapping. The center of scaling is the point that does not move when the scaling is applied.

Passing the result of the `ScaleMapping` function to the `GXMapShape` function is equivalent to calling the `GXMapShape` function and then calling the `GXScaleShape` function. For example, you could make these calls (for the shape `s`):

```
ScaleMapping(target, hFactor, vFactor, xCenter, yCenter);
GXMapShape(s, target);
```

or, you could make these equivalent calls:

```
GXMapShape(s, target);
GXScaleShape(s, hFactor, vFactor, xCenter, yCenter);
```

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

SEE ALSO

The use of the `ScaleMapping` function is described in the section “Scaling” beginning on page 8-20.

The `GXMapShape` and `GXScaleShape` functions are described in the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects.*

RotateMapping

You can use the `RotateMapping` function to change the rotation specified by a mapping.

```
gxMapping *RotateMapping(gxMapping *target, Fixed angle,
                           Fixed xCenter, Fixed yCenter);
```

`target`	A pointer to the mapping to be modified. On return, points to the modified mapping.
`angle`	The amount of rotation (in degrees clockwise) to be added to the mapping.
`xCenter`	The x-coordinate of the center of rotation.
`yCenter`	The y-coordinate of the center of rotation.

function result A pointer to the modified mapping, which is also the target mapping.

DESCRIPTION

The `RotateMapping` function postmultiplies the target mapping by a mapping that specifies a rotation (clockwise if positive) by a specified number of degrees about the point (`xCenter`, `yCenter`).

The rotation is in addition to any preexisting rotation specified by the target mapping.

Passing the result of this function to the `GXMapShape` function is equivalent to calling the `GXMapShape` function and then calling the `GXRotateShape` function. For example, you could make these calls (for the shape `s`):

```
RotateMapping(target, angle, xCenter, yCenter);
GXMapShape(s, target);
```

or, you could make these equivalent calls:

```
GXMapShape(s, target);
GXRotateShape(s, angle, xCenter, yCenter);
```

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

SEE ALSO

The use of the `RotateMapping` function is described in the section "Rotation" beginning on page 8-22.

The `GXMapShape` and `GXRotateShape` functions are described in the chapter "Transform Objects" in *Inside Macintosh: QuickDraw GX Objects.*

SkewMapping

You can use the `SkewMapping` function to change the horizontal and vertical skew specified by a mapping.

```
gxMapping *SkewMapping(gxMapping target,
                       Fixed skewX, Fixed skewY,
                       Fixed xCenter, Fixed yCenter);
```

`target`	A pointer to the mapping that is to be modified. On return, points to the modified mapping.
`skewX`	The scaling factor that determines the amount of skew in the x direction. A value of 0 means no horizontal skew.

`skewY`	The scaling factor that determines the amount of skew in the y direction. A value of 0 means no vertical skew.
`xCenter`	The x-coordinate of the center of skewing.
`yCenter`	The y-coordinate of the center of skewing.
function result	A pointer to the modified mapping, which is also the target mapping.

DESCRIPTION

The `SkewMapping` function postmultiplies the target mapping by a mapping that specifies a horizontal skew factor of `skewX` and a vertical skew factor of `skewY`, about the point (`xCenter`, `yCenter`). Note that if `skewX` is 0, `yCenter` irrelevant; likewise, if `skewY` is 0, `xCenter` is irrelevant.

These skew factors are in addition to any preexisting skew specified in the target mapping. The center of skewing specifies the point at which no translation takes place because of the skewing.

Passing the result of the `SkewMapping` function to the `GXMapShape` function is equivalent to calling the `GXMapShape` function and then calling the `GXSkewShape` function. For example, you could make these calls (for the shape `s`):

```
SkewMapping(target, hFactor, vFactor, xCenter, yCenter);
GXMapShape(s, target);
```

or, you could make these equivalent calls:

```
GXMapShape(s, target);
GXSkewShape(s, skewX, skewY, xCenter, yCenter);
```

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

SEE ALSO

The use of the `SkewMapping` function is described in the section “Skewing” beginning on page 8-24.

The `GXMapShape` and `GXSkewShape` functions are described in the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects.*

Summary of QuickDraw GX Mathematics

Constants and Data Types

Number Formats and Constants

```
typedef long fract;

typedef unsigned short gxColorValue;

struct gxPolar {
   Fixed radius;
   Fixed angle;
};

struct wide {
   long hi;
   unsigned long lo;
};

#define fixed1             ((Fixed) 0x00010000)        /* = 1.0 for Fixed */

#define fract1             ((fract) 0x40000000)        /* = 1.0 for fract */

#define gxColorValue1      ((gxColorValue) 0xFFFF) /* 1.0 for gxColorValue*/

#define gxPositiveInfinity ((Fixed) 0x7FFFFFFF)    /* for Fixed and fract */

#define gxNegativeInfinity ((Fixed) 0x80000000)    /* for Fixed and fract */
```

The Mapping Structure

```
struct gxMapping {
   Fixed map[3][3];
};
```

Number-Conversion Macros

Format Conversions

```
#define FixedToFract(a)      ((fract) (a) << 14)
#define FractToFixed(a)      ((Fixed) (a) + 8192L >> 14)
#define FixedToInt(a)        ((short) ((Fixed) (a) + fixed1/2 >> 16))
#define IntToFixed(a)        ((Fixed)(a) << 16)
#define ff(a)                ((Fixed)(a) << 16)
#define FixedToFloat(a)      ((float)(a) / fixed1)
#define FloatToFixed(a)      ((Fixed)((float) (a) * fixed1))
#define fl(a)                ((Fixed)((float) (a) * fixed1))
#define FractToFloat(a)      ((float)(a)/fract1)
#define FloatToFract(a)      ((fract)((float)(a)*fract1))
#define ColorToFract(a)      (((fract)(a)<<14) + ((fract)(a) +2 >>2))
#define FractToColor(a)      ((gxColorValue)((a)-((a)>>16)+8191>>14))
```

Rounding, Truncating, and Square Root Operations

```
#define FixedRound(a)        ((short) ((Fixed)(a) + fixed1/2 >> 16))
#define FixedTruncate(a)     ((short)((Fixed)(a) >> 16))
#define FixedSquareRoot(a)   ((Fixed)FractSquareRoot(a) + 64 >>7)
```

Mathematical Functions

Fixed-Point Operations

```
Fixed FixedMultiply          (Fixed multiplicand, Fixed multiplier);
Fixed FixedDivide            (Fixed dividend, Fixed divisor);
long MultiplyDivide          (long multiplicand, long multiplier,
                              long divisor);
unsigned long Magnitude      (long deltaX, long deltaY);
fract FractSineCosine        (Fixed degrees, fract *cosine);
fract FractSquareRoot        (fract source);
fract FractCubeRoot          (fract source);
fract FractMultiply          (fract multiplicand, fract multiplier);
fract FractDivide            (fract dividend,fract divisor);
```

Operations on wide Numbers

```
wide *WideAdd                (wide *target, const wide *source);
wide *WideSubtract           (wide *target, const wide *source);
wide *WideNegate             (wide *target);
wide *WideShift              (wide *target, long shift);
wide *WideMultiply           (long multiplicand, long multiplier,
                              wide *target);
long WideDivide              (const wide *dividend, long divisor,
                              long *remainder);
wide *WideWideDivide         (wide *dividend, long divisor, long *remainder);
unsigned long WideSquareRoot
                             (const wide *source);
short WideScale              (const wide *w);
short WideCompare            (const wide *target, const wide *source);
```

Vector Operations

```
wide *VectorMultiply         (long count, const long *vector1, long step1,
                              const long *vector2, long step2, wide *dot);
long *VectorMultiplyDivide   (long count, const long *vector1, long step1,
                              const long *vector2, long step2, long divisor);
```

Cartesian and Polar Coordinate Point Conversions

```
gxPoint *PolarToPoint        (const gxPolar *ra, gxPoint *xy);
gxPolar *PointToPolar        (const gxPoint *xy, gxPolar *ra);
```

Random Number Generation

```
unsigned long RandomBits     (long count, long focus);
void SetRandomSeed           (const wide *seed);
wide *GetRandomSeed          (wide *seed);
```

Linear and Quadratic Roots

```
long LinearRoot              (Fixed first, Fixed last, fract t[]);
long QuadraticRoot           (Fixed first, Fixed control, Fixed last,
                              fract t[]);
```

Bit Analysis

```
short FirstBit               (unsigned long x);
```

Mapping Functions

Manipulating and Applying Mappings

```
gxMapping *CopyToMapping    (gxMapping *target, const gxMapping *source);
gxMapping *NormalizeMapping (gxMapping *target);
gxMapping *ResetMapping     (gxMapping *target);
gxMapping *InvertMapping    (gxMapping *target, const gxMapping *source);
gxMapping *MapMapping       (gxMapping *target, const gxMapping *source);
void MapPoints              (const gxMapping source, long count,
                             gxPoint vector[]);
```

Modifying Mappings

```
gxMapping *MoveMapping      (gxMapping *target,
                             Fixed hOffset, Fixed vOffset);
gxMapping *MoveMappingTo    (gxMapping *target,
                             Fixed hPosition, Fixed vPosition);
gxMapping *ScaleMapping     (gxMapping *target,
                             Fixed hFactor, Fixed vFactor,
                             Fixed xCenter, Fixed yCenter);
gxMapping *RotateMapping    (gxMapping *target, Fixed angle,
                             Fixed xCenter, Fixed yCenter);
gxMapping *SkewMapping      (gxMapping target, Fixed skewX, Fixed skewY,
                             Fixed xCenter, Fixed yCenter);
```

Glossary

all object validation A QuickDraw GX validation level that confirms that all references to all object types are valid, that the properties of the object are valid, and that all internal caches built for all objects are valid. Compare **type validation** and **structure validation**.

application heap A region of memory that is allocated by the Macintosh Memory Manager when an application is launched. This is the memory region reserved for application code and data structures.

attribute mask A means of editing the attributes of a collection object.

attributes A property of many QuickDraw GX objects that is a set of flags that control various aspects of that object's behavior.

bad parameter error A nonfatal QuickDraw GX error indicating that one or more function parameters are incorrect.

bad reference error A QuickDraw GX error indicating that an invalid reference to a view or font device, view group, or view port was made.

bias The number of bits to the right of a binary point in a fixed point number. See also **`gxColor`Value**, **`Fixed`**, **fixed-point number**, **`fract`**, **`long`**, and **`short`.**

bitmap (1) A QuickDraw GX data structure that describes a pixel map on a physical device. A bitmap structure is a property of a view device object. (2) A type of QuickDraw GX shape.

cache See **QuickDraw GX cache.**

cache error A QuickDraw GX error indicating that a memory cache problem occurred.

Cartesian coordinate A coordinate system used for view devices in which the positive x direction is to the right and the positive y direction is down with respect to the origin, at the upper-left corner. A point is defined by specifying the x- and y-coordinates in the format (*x*, *y*). Compare polar coordinate.

child view port For a given view port, a view port immediately below it in the view port hierarchy.

child view port list A property of a view port object that is an array of references to the child view ports of that view port.

clip A QuickDraw GX shape and a property of a transform object, view port object, and view device.

collection An abstract data type that allows you to store information. Unlike an array, a collection allows you to store variable-sized items.

collection index A means of uniquely identifying each item within a collection.

collection item A part of a collection object.

collection object See **collection.**

concatenate An operation consisting of two or more sequential mappings.

data stream A highly structured sequence of bytes that contains all of the information required to store, print, or display QuickDraw GX objects.

dead cache A shape cache that is out of date. The object or environment associated with the cache has been changed since the cache was created.

debugging environment The QuickDraw GX application development environment consisting of the debugging version; errors, warnings, and notices; application-defined message handlers; the drawing errors; validation function; and the MacsBug and GraphicsBug utilities. See also **error**, **warning, notice**, and **message handler.**

debugging version The version of QuickDraw GX that provides validation and an extended set of errors, warnings, and notices. This version is intended for use during application development. See also **non-debugging version**, **error**, **warning**, and **notice.**

default attributes The attributes that determines the initial attribute values assigned to items added to a collection.

default memory size The implementation limit size of the graphic client memory heap that QuickDraw GX will select if the memory size is not specified.

default object A QuickDraw GX object with the properties of a newly created object. Whenever it creates an object, QuickDraw GX assigns it the default properties for that kind of object; an application may then alter those properties through accessor functions.

discontiguous memory One or more non-continuous blocks of memory. For example, a graphics client heaps might be discontiguous.

dispose of To delete a reference to an object. When an application no longer needs an object, it disposes of the object. That action deletes the object from memory if there are no other current references to the object; otherwise, disposing of an object merely decreases its owner count by 1.

drawing error A QuickDraw GX error indicating why your shape did not draw successfully. The `GXGetShapeDrawError` function posts such a single error.

drawing process sequence The sequence in which QuickDraw processes objects: shape, style, ink, transform, view port, and view device.

error A single descriptive phrase that is posted by QuickDraw GX whenever an application is unable to execute. Execution is terminated at the nonexecutable function. Each error message is assigned a unique number in the range –27999 through –27000. Errors are posted in both the debugging and non-debugging versions of QuickDraw GX.

`Fixed` number A 32-bit signed integer with 16 bits to the left and 16 bits to the right of the binary decimal point. A fixed-point number with a bias of 16. `Fixed` numbers range from –32,768 to nearly +32,768. The `fixed` number for 1.0 is 0x0001000.

fixed-point number A signed 16-bit, 32-bit, or 64-bit quantity containing an integer part in the high-order word and a fractional part in the low-order word. Integers are interpreted as real numbers by the use of bias to define where the decimal point is located. Numbers having the `gxColorValue`, `short`, `long`, `fixed`, `fract`, and `wide` number formats are fixed point numbers. See also **bias**, **`long` number**, **`fract` number**, **`short` number**, **`gxColorValue`** and **wide** number.

flatten To convert an object created by your application from its original format to a QuickDraw GX stream format.

font management error A QuickDraw GX error that involves the storage, attributes, or parameter of a font.

font scaler error A QuickDraw GX error that involves the conversion of a glyph outline to a bitmap.

font scaler warning A QuickDraw GX warning that involves the conversion of a glyph outline to a bitmap.

forward To invoke the override of the next handler in the chain for the current message.

`fract` number A 32-bit signed integer with two bits to the left and 30 bits to the right of the binary decimal point. A fixed-point number with a bias of 2. `Fract` numbers range from –2 to +2. The `fract` number for 1.0 is 0x40000000.

geometry A property of a QuickDraw GX shape object**.** A shape's geometry is the specification of the actual size, position, and fill of the shape. For example, for a rectangle shape, the geometry specifies the locations of the rectangle's corners in local coordinates.

global coordinates For QuickDraw GX, the coordinate system used for a view group. For example, a view port's location is described in global coordinates. This coordinate system represents all potential drawing space. The origin, point (0,0), of the global coordinate system is located at the upper-left corner of the main screen. The positive x-axis extends to the right. The positive y-axis extends downward.

GraphicsBug A QuickDraw GX debugging utility that allows detailed analysis of heaps and objects. See also **heaps** and **objects.**

graphics client A region of memory where bookkeeping data is stored for a graphics client heap. This includes the memory starting address, the size and location of all of the heap's memory blocks, and the error, warning, and notice state. See also **graphics client heap.**

graphics client heap A region of memory that contains all of the objects that a QuickDraw GX application creates. A heap that consists of public objects, such as shapes, styles, inks, and transforms, as well as private objects used for heap management. See also **graphics client** and **heap.**

graphics device Any graphics hardware attached to the system.

gxColorValue A 16-bit unsigned integer. A fixed-point number that ranges from 0 to 65,535 to represent the numbers 0 to 1. The integer must be divided by 65,535 to obtain the real number represented. The color value number for 1.0 is 0xFFFF.

handler A recipient and processor of messages. It can be a printing extension, a printer driver, QuickDraw GX printing, or an application. For example, an application can supply a handler for errors, warnings, and messages. See also message chain.

heap An area of memory that is dynamically allocated and deallocated on demand. See also **application heap** and **graphics client heap.**

hit-testing The conversion of a specific geometric location, such as a pixel position in a view port, to logical location (part, control point, or glyph) in the geometry of a shape object. Hit-testing is used to highlight or activate parts of geometric shapes or to highlight or draw a caret within the displayed text of a typographic shape.

identity matrix A mapping matrix that maps a point to the same point. A mapping matrix with the value 1 for the diagonal matrix elements and the value 0 for all other matrix elements.

ignore notice stack A stack that can contain the implementation limit of notice numbers. Notices on the ignore notice stack are not posted by QuickDraw GX.

ignore warning stack A stack that can contain the implementation limit of warning numbers. Warnings on the ignore warning stack are not posted by QuickDraw GX.

implementation limit error A QuickDraw GX error indicating that the implementation limit of a structure has been exceeded. See also implementation limit.

implementation limit An upper or lower bounds of a size, number, or value. This limit is defined by the current version of QuickDraw GX. See also **default memory size**.

instance A single copy of a message handler in memory. See also **instantiate**.

instantiate To create an instance of a message handler separate and unique from all other instances. See also **instance**.

ink A QuickDraw GX object associated with a shape object. An ink object contains information that affects the color of a shape and the transfer mode with which it is drawn.

internal error A nonfatal QuickDraw GX error indicating a damaged file, memory problem, or incorrect implementation.

internal validation An optional validation mode in which object parameter validation occurs whenever an application uses a public function and whenever QuickDraw GX uses an internal function. Compare **public validation.**

interrupt programming A type of programming in which QuickDraw GX allows an application to switch tasks, but only when it is not performing critical functions.

invalid data warning A QuickDraw GX warning indicating that an object contains incorrect data or that extra data was passed.

inverse of a mapping The mathematical inverse of the mapping matrix. A mapping concatenated with its inverse results in the identity matrix.

live cache A QuickDraw GX cache that contains current information. The object associated with the cache has not been changed since the cache was created. See also **dead cache.**

load [an object] To return an unloaded QuickDraw GX object from external storage to memory. QuickDraw GX automatically and transparently loads and unloads objects in the course of managing memory; an application need never know whether an object it accesses is currently loaded or unloaded.

local coordinates For QuickDraw GX, the coordinate space local to each shape. For example, a shape's geometry is described in local coordinates.

lock attribute When set, this attribute prevents an item in a collection from being replaced.

`long` number A 32-bit signed integer. A fixed-point number with a bias of 0. Long numbers range from –2,147,483,648 to +2,147,483,647. The `long` number for 1.0 is 0x00000001.

Macintosh interface functions A set of Macintosh-specific functions. Most other QuickDraw GX functions can exist on any platform.

macro A sequence of predefined directives that the C preprocessor interprets at compile time. When the preprocessor encounters the macro name in the source code, the preprocessor substitutes the macro definition for it. QuickDraw GX provides macros for number format conversions.

MacsBug A Macintosh debugging utility.

map See **mapping.**

mapping A transformation of spatial locations (points) that can be represented by a 3 × 3 perspective matrix. Synonymous with **map** and **mapping matrix.**

mapping matrix See **mapping.**

memory allocation Specification of the starting address of the graphics client in memory.

memory block An area of contiguous memory within a heap or zone**.**

memory size The number of bytes of random access memory allocated to the QuickDraw GX graphics client. The default size is 600 KB**.**

message A notification passed to a message object so that the message object will perform an operation.

message chain One or more handlers that wish to receive and respond to messages. A handler at the top of a message chain always receives a message first. See also **message handler.**

message class The set of messages and methods defined at run time that are understood by message objects.

message handler A component of a message class that can override messages.

Message Manager A low-level software manager that is part of the QuickDraw GX message-passing printing architecture.

message object The recipient and sender of messages.

message override See **override.**

non-debugging environment The QuickDraw GX end-user environment consisting of the non-debugging version, errors and warnings, and application-defined message handlers. See also **error**, **warning**, and **message handler.**

non-debugging version The version of QuickDraw GX that provides a limited set of errors and warnings. This version is intended for use with a debugged application. See also **debugging version**, **error**, and **warning.**

normalize To divide a mapping matrix by the absolute value of matrix element w. A mapping is considered normalized whenever the matrix element w has the value 1.

notice A single descriptive phrase that is posted by the debugging version of QuickDraw GX whenever an unnecessary or redundant function has been performed. Execution continues as if the notice had not been posted. Notices are posted only in the debugging version of QuickDraw GX. A notice number is a unique number in the range –25999 through –25500 assigned to each QuickDraw GX notice message. Each notice number has a unique notice name. See also **notice name.**

notice name A multiple-word phrase that describes the QuickDraw GX notice posted. Each notice name has a unique notice number. See also **notice.**

notice number See **notice.**

object A private QuickDraw GX data structure. An object is defined by properties and is accessed by a reference.

omit byte A means of assigning different data compressions to type constants and object properties that immediately follow this byte.

omit byte mask With the omit byte shift, this is a means of interpreting the meaning of each of the bits in an omit byte.

omit byte shift With the omit byte mask, this is a means of interpreting the meaning of each of the bits in an omit byte.

overflow notice A QuickDraw GX notice indicating that a notice could not be added to the ignore notice stack because the implementation limit had been exceeded. See also implementation limit.

overflow warning A QuickDraw GX warning indicating that a warning could not be added to the ignore warning stack because the implementation limit had been exceeded. See also implementation limit.

override A message handler's implementation of a given message. A message handler's override performs the operation requested by the message received by the message object. A partial override forwards the message. A complete override does not forward the message.

owner A variable, structure, or QuickDraw GX object that references an object. Many objects can be referenced by more than one variable and can thus have multiple owners.

owner count A property of some QuickDraw GX objects that indicates the number of current references to the object.

parameter out of range warning A QuickDraw GX warning indicating that a function parameter is out of the valid range.

parent view port A property of a view port object. A view port's parent is that view port immediately above it in the view port hierarchy**.**

persistence attribute An attribute that causes an item to be included when the Collection Manager flattens a collection. See **flatten.**

polar coordinate A coordinate system in which a point is specified by the length of the radius vector *r* from the origin to the point and the direction of the vector is specified by the polar angle *a*. A point is defined by specifying the coordinates *r* and *a* in the format (*r*, *a*). The polar origin has the coordinates (0, *a*), where *a* is any angle. Compare Cartesian coordinate.

posting The process of generating error, warning, and notice messages by QuickDraw GX. See also **debugging version**, **non-debugging version**, **error**, **warning**, and **notice.**

postmultiplied A term that describes the order in which matrices are multiplied. Matrix [A] is postmultiplied by matrix [B] if matrix [A] is replaced by [A] × [B]. Compare **premultiplied**.

premultiplied A term that describes the order in which matrices are multiplied. Matrix [A] is premultiplied by matrix [B] if matrix [A] is replaced by [B] × [A]. Compare **postmultiplied.**

property An item or set of data in a QuickDraw GX object. A property of an object is analogous to a field (or member) of a data structure; however, a field is accessed through its name, whereas a property is accessed through an accessor function.

public validation The process of checking the validity of the parameters passed by an application. See **validation.**

QuickDraw GX cache Temporary memory that is managed by QuickDraw GX. Each object has a pointer to one or more caches. Each cache is related to only one object. See also **dead cache** and **live cache.**

recoverable error A nonfatal QuickDraw GX error indicating fragmented memory, a problem with the backing store, or a problem with the unflattening process.

reference A long word value, neither a pointer nor a handle, through which an application accesses a QuickDraw GX object. References are created by QuickDraw GX and passed to applications.

reflection The symmetrical movement of a mapping with respect to the **Cartesian coordinate** axes. The movement can be about the x- or y- or both axes.

reserved attributes The attributes of a collection item's 32 attributes that are reserved and cannot be set.

restricted access error A QuickDraw GX error indicating that the object data requested is private and not available.

result out of range warning An application execution warning detected and posted by QuickDraw GX indicating that the function result was out of the valid range.

seed An initialization value used by a random number generator to produce a sequence of values.

shape (1) A graphic or typographic item (such as a geometric shape, bitmap, or a line of text) created and drawn with QuickDraw GX. (2) A set of QuickDraw GX objects that, taken together, describe the type and characteristics of such a graphic or typographic item. A shape consists of a shape object, style object, ink object, and transform object.

shape cache A cache created and maintained by QuickDraw GX for storing the results of intermediate calculations made prior to drawing a shape.

shape fill A property of a shape object. The shape fill specifies whether and how QuickDraw GX fills in the outlines of a shape that is draws.

shape object A QuickDraw GX object that, along with several other objects, describes a QuickDraw GX shape. A shape object specifies the fundamental type and contents of a shape.

shape type A property of a shape object. The shape type specifies the classification (such as point, line, bitmap, or text) of a particular shape.

`short` number A 16-bit signed integer with 16 bits to the left and 0-bits to the right of the binary decimal point. A fixed point number with a bias of 0. The short number for 1.0 is 0x0001.

specific object validation A QuickDraw GX validation level that confirms that all references to a specific object type are valid.

storage warning A QuickDraw GX warning indicating a data stream problem.

stream format The public format available for describing flattened QuickDraw GX objects. Objects in stream format are compressed or flattened. Flattened objects are unflattened when they are converted back to object format. A flattened object may be interpreted by using QuickDraw GX unflattening functions or reconstructed by parsing with an interpreter that uses the stream format.

structure validation A QuickDraw GX validation level that confirms that references to object types are valid and that the properties of the object are valid. Compare **type validation**, and all **object validation**.

tag list A property of some QuickDraw GX objects. The tag list is an array of references to tag objects associated with the object.

tag list position The position of an item in a list of items with the same collection tag.

tag object A QuickDraw GX object whose purpose, structure, and content are entirely controlled by the application creating it. Tag objects exist to allow custom information and behavior to be attached to standard QuickDraw GX objects. Tag objects are classified by tag type; objects reference their tag objects through a tag list.

translation options The use of one or more constants to translate QuickDraw data to QuickDraw GX shapes.

transfer mode A QuickDraw GX data structure, also the property of an ink object, that controls the interaction between the color of a shape and the colors of the background at the location where the shape is drawn.

transform A QuickDraw GX object associated with a shape object. A transform object contains information that affects the visual appearance of a shape when it is drawn.

translator A set of functions that convert QuickDraw data into QuickDraw GX shapes or pictures. The translation approximates the intent of the original QuickDraw images; it does not provide a pixel-by-pixel mapping of the image.

type validation A QuickDraw GX validation level that confirms that references to object types are valid. Compare **structure validation** and all **object validation.**

underflow notice A QuickDraw GX notice indicating that a notice could not be removed from the ignore notice stack because no notice was on the stack.

underflow warning A QuickDraw GX warning indicating that a warning could not be removed from the ignore warning stack because no warning was on the stack.

unexpected result warning A QuickDraw GX warning indicating that a character or font substitution took place or that the geometry of an area or new device is probably incorrect.

unflatten To convert the public, stream-based description of an object or set of objects into the private, native QuickDraw GX object-based format. Compare **flatten**. See also **stream format.**

unload [an object] To move a QuickDraw GX object from memory to temporary external storage. QuickDraw GX automatically and transparently loads and unloads objects in the course of managing memory; an application need never know whether an object it accesses is currently loaded or unloaded.

user attributes The lower 16 bits of an item's attributes; these bits can be defined for purposes suitable to your application.

validation A set of debugging functions that cause one or more actions to occur whenever a QuickDraw GX function is called or whenever the internal memory manager is called. See also **public validation**, **internal validation.**

validation error A QuickDraw GX error detected and posted by the debugging version with validation error checking turned on. The parameters of objects are checked to ensure that the object is valid. See also **validation**.

view device A QuickDraw GX object associated with a view port object. A view device object describes the characteristics of a given physical display device such as a monitor or printer.

view group A QuickDraw GX object that consists of a grouping of view ports and view devices.

view port A QuickDraw GX object associated with a transform object. A view port describes the characteristics of the drawing environment for individual QuickDraw GX shapes.

view port hierarchy An ordered arrangement of view ports that allows for such features as windows within windows, including multiple windows within a single window.

view port list A property of a transform object. This list is an array of references to the view ports that the shapes associated with that transform can be drawn to.

warning A single descriptive phrase that is posted by QuickDraw GX whenever an application executes a function that may likely not provide the result expected. Execution continues internally, as if the warning had not been posted. A warning number is a unique number in the range –26999 through –26000 assigned to each QuickDraw GX warning message. Each warning has a unique warning name.

warning name See **warning.**

warning number See **warning.**

`wide` number A 64-bit signed integer with unspecified bias.

wrong type error A QuickDraw GX error indicating that an invalid type has been assigned to a shape.

Index

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro printer. Final page negatives were output directly from text files on an Optrotech SPrint 220 imagesetter. Line art was created using Adobe Illustrator™ and Adobe Photoshop™. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

WRITERS
Richard Pettijohn, Marq Laube, David Bice, Joanna Bujes

DEVELOPMENTAL EDITORS
Sue Factor, John Hammett

ILLUSTRATORS
Ruth Anderson, Barbara Carey, Sandee Karr, Mai-Ly Pham

PRODUCTION EDITORS
Pat Christenson, Alan Morgenegg

PROJECT MANAGER
Trish Eastman

LEAD WRITER
David Bice

LEAD EDITOR
Laurel Rezeau

ART DIRECTOR / COVER DESIGNER
Barbara Smyth

Special thanks to Pete “Luke” Alexander, Cary Clark, Pable Fernicola, Dave Good, Dave Hersey, Rob Johnson

Acknowledgments to Tom Dowdy, Ken Hittleman, Miki Lee, Chris Yerga