GLG Programming Reference Manual

GLG Toolkit
Version 4.6

Generic Logic, Inc.

Generic Logic, Inc.
6 University Drive 206-125
Amherst, MA 01002
USA
Telephone: (413) 253-7491
FAX: (413) 241-6107
email: support@genlogic.com
web: www.genlogic.com

April 22, 2026
Software Release Version 4.6

GLG Programming Reference Manual

Table of Contents

Preface

GLG Programming Reference Manual

This book provides information about programming with GLG and using different GLG Run-Time environments, including C/C++, Java, C#/.NET, JavaScript and ActiveX Control.

The first section of this manual describes the C API, while the rest of the book describes the C++, Java, C#/.NET, JavaScript and ActiveX Control versions.

Although various programming environments such as C, C++, Java, C#/.NET, JavaScript and ActiveX, use different syntax to invoke the GLG API, the semantics of the API methods is the same in either environment. The C programming section provides the most detailed description of each method in terms of its effect on the underlying GLG objects. It also provides more examples and may be used as a reference for C, C++, Java, C#/.NET and JavaScript programmers.

The book contains the following chapters:

Table of Contents

Integrating GLG Drawings into a Program
A general overview of the process of programming with the GLG Toolkit.

Using the C/C++ version of the Toolkit

Displaying a GLG Drawing
How to load and display a GLG drawing in an application program in different programming environments.

Compiling and Linking GLG Programs
The details of compiling and linking an application program with the GLG libraries.

The GLG Generic API
Describes functions of the GLG Generic API for loading a displaying a GLG drawing is a cross-platform way.

Animating a GLG Drawing with Data Using the Standard API
Describes functions of the GLG Standard API used to animate a drawing with real-time data.

Handling User Input and Other Events
A description of the ways in which a GLG drawing can be used to get input from an application program user.

GLG Intermediate and Extended API
How to query the content of a drawing and modify the drawing from an application program.

Real-Time Chart Functions
A description of functions used with the GLG Real Time Chart object.

GIS Object Functions
A description functions used to handle user interaction with the GLG GIS Object.

GLG Installable Interface Handlers
A description of installable interface handles that may be used to handle complex editor-like user interaction.

Chapter 1

Integrating GLG Drawings into a Program

1

The GLG Graphics Builder is a flexible and powerful tool for the creation and animation of GLG drawings. The GLG Toolkit also provides a variety of ways to incorporate GLG drawings into your applications and supplies several facilities designed to draw and control a GLG drawing from within a user's application program at run time.

The steps for integrating a GLG drawing into a program are straightforward, and most of them are platform-independent:

1. Create a GLG drawing in the GLG Builder using one of the *File, New, Widget* options and add graphical objects, such as predefined components from the Palettes menu or graphical primitives with dynamic actions.

   The drawing is encapsulated in a GLG **widget**, which is a top-level viewport named *$Widget* with its *HasResources* attribute set to YES. The drawing is saved in a file with a *.g* extension. In addition to creating a drawing interactively with the GLG Graphics Builder, it may also be created programmatically at run time using the GLG Extended API Library.

2. Load the saved drawing (*.g* file) into a program and display it in a window. This can be done either using the cross-platform GLG Generic API, or via platform-specific containers:

   C/C++

   - The **GLG Generic API Library** is a platform-independent solution for applications that need source code portability. The *GlgInitialDraw* method of the Generic API performs all platform-specific initialization, creates a window and displays a drawing in it. Cross-platform methods for handling user interaction are also provided. An application developed using the Generic API can be **compiled and executed with no source code changes on both Linux/Unix and Windows**.
   - **On Windows**, the *GLG Custom Control*, the *GlgControlC MFC class* or the *GLG ActiveX Control* can be used to display a drawing in the C/C++ program.
   - **On Linux/Unix**, several flavors of the *GLG Wrapper Widget* are provided for integration in C/C++ programs.
   - **Qt** and **GTK wrapper widgets** are also provided for integrating drawings into these environments on both Windows and Linux/Unix platforms (refer to the *Qt and GTK Integration* section on page 50 for more information).

   Java

   - A *GLG Java Bean* is used to integrate GLG drawings in a Java application, both *Swing* and *JavaFX*. The *GLG Bean* is available as a heavyweight (*GlgJBean*) or lightweight (*GlgJLWBean*) Swing component. The GLG Java Class library provides a 100% pure Java version of the GLG Run Time engine.
   - The Generic API is also available for Java as the *InitialDraw* method of the *GlgObject* class.

- The *GLG Graphics Server* class library can be used with the *JavaEE Framework* to develop server-side AJAX-based applications for web and mobile applications using *Java Servlet* technology.

C#/.NET

- A native *Windows Form User Control* is provided as the *GlgControl* class, and the GLG .NET DLL provides the GLG Run Time engine for the .NET Framework. The GLG .NET DLL can be used in a cross-platform way using **Mono** on Linux. On Windows, C#/.NET applications can also use the GLG ActiveX Control.
- The Generic API is also available for C#/.NET as the *InitialDraw* method of the *GlgObject* class.

VB.NET and VB6

- **VB.NET** applications can use either the C#-based GLG User Control or the GLG ActiveX Control. The GLG ActiveX Control can also be used in **VB6** applications.

JavaScript

- The GLG Generic API is used in JavaScript applications to load and display a GLG drawing on a web page.

3. Animate the drawing with real-time data by setting either resources or tags defined in the drawing. This is done using methods of the **GLG Standard API**.

4. Handle user interaction, such as button clicks, mouse click actions, etc.

The following sections of this chapter provide details about each of these steps, and about issues important to GLG Toolkit programmers.

Creating a Widget Drawing

To use a GLG drawing in a program, it has to be encapsulated in a GLG Widget by placing the drawing in a viewport named *$Widget*. Most drawings are created with the GLG Graphics Builder. Any GLG drawing with a viewport at the top level of its resource hierarchy can become a GLG widget, simply by defining the name of that viewport to be *$Widget*. The viewport's *HasResources* attribute must be set to YES in order for its resources to appear inside the viewport.

A drawing can also be created from a program, using the GLG Extended API. The Extended API is described in detail in the *GLG Intermediate and Extended API* chapter on page 135.

If you save a GLG drawing into a file, there are three different file formats it may use:

- The **binary** format is the fastest format to use for loading drawing files but is not compatible between hardware platforms with different binary data representations. The binary format differs between the C/C++ and Java/C# versions of the Toolkit as well, since Java and C# use their own binary representations of data. The JavaScript API does not support the binary format. The binary format is primarily used for the drawings which are embedded into the C/C++ executables using the code generation option, and should not be used for drawings that will be used on different platforms or with the Java/C#/JavaScript programs.

- The **ASCII** format is slightly slower to load (though more compact) than the binary format and is compatible between different hardware platforms. Is is also compatible across the C/C++ and Java/C#/JavaScript versions of the Toolkit. The ASCII format is the default save format and is also the preferred format for the Java, C#/.NET and JavaScript versions of the Toolkit.
- The **extended** format is the slowest and the least compact but provides the ability to transfer drawing files between different versions of the Toolkit. It may be used to import the drawings saved using a newer release of the Toolkit into an older version (if the drawing uses the new features of the newer release, the conversion for these features or for the whole drawing may fail). The extended format is not supported in Java, C#/.NET and JavaScript.

Displaying a Drawing

Unless the GLG Generic API is used, displaying a drawing is one of the few platform-dependent steps in the process of animating it with input data. The necessary steps for displaying a drawing on Linux/Unix or Windows are both covered in the *Displaying a GLG Drawing* chapter on page 31. You only need to read the sections of that chapter that apply to the windowing environment and the version of the Toolkit you intend to use for your application (C/C++, ActiveX, Java, C#/.NET or JavaScript).

Before a drawing can be displayed by a program, it must be loaded into memory. There are three ways to do this:

- A drawing may be stored in a separate file. This lets the user of an application customize the appearance of the widget by editing the file using the GLG Graphics Builder.This option is also very convenient in the development stage, allowing you to modify the drawing without additional compile/debug cycles.
- A drawing may also be converted into a memory image in a C language format using the Toolkit's *gcodegen* utility. This allows you to compile and link the drawing directly into the program's executable. Embedding a drawing into the executable increases its size and prevents users from modifying the drawing but reduces the number of additional files needed by an application. See the *Code Generation Utility* section on page 349 in the *GLG Programming Tools and Utilities* chapter for more details. Memory images are not supported in the Java, C#/.NET and JavaScript versions of the Toolkit. In Java, the drawing may be placed inside the application's JAR file.
- A program may also generate a drawing from scratch using the GLG Extended API. Although this is possible, it is usually simpler to load a drawing template from a file and reserve these functions for modifying the template drawing and adding new objects to it. The Extended API is described in the *GLG Intermediate and Extended API* chapter on page 135. The Extended API for the Java and C#/.NET versions of the Toolkit is described in the *Using the Java and C# Versions of the Toolkit* chapter. The JavaScript Extended API is almost identical to the Java and C# Extended API and is described in the GLG JavaScript Online Documentation.

Animating a Drawing with Real-Time Data

A program animates a GLG drawing with application data by setting drawing's resources or tags using methods of the GLG Standard API. These functions are described in the *Animating a GLG Drawing with Data Using the Standard API* chapter on page 67.

Resources are hierarchical and can be used to set properties of specific objects in the drawing when the content of the drawing is known in advance. Tags are global and can be used to animate an arbitrary drawing without knowing its content or resource hierarchy. Tags can be assigned by end users in the GLG HMI Configurator or the GLG Builder to specify the tag source of the process database used to animate a particular dynamic parameter. At run time, an application can query a list of tags defined in the drawing and use information in each tag to obtain data from the process database, animating the drawing with the retrieved data.

A *SimpleViewer* example provides a sample implementation of a generic GLG viewer that animates an arbitrary drawing using tags. The example provides source code for various programming environments (C/C++/C#.NET/Java/VB.NET). A *SCADA_Viewer* example provides a sample framework for a more elaborate viewer that also handles drawing navigation, object commands, popup menus and dialogs, as well as alarm display and acknowledgement.

Distributing Remote Server Data Using GLG Data Gateway

The GLG Data Gateway is a **high-performance light-weight** solution for distributing data from a central server to **multiple C/C++, C#, or Java clients**. It ensures reliable, secure communication with automatic fault recovery and built-in data serialization/deserialization for efficient network messaging. With optional CURVE-based encryption, comprehensive configuration, logging, statistics and troubleshooting tools, it is ideal for secure, real-time, cross-platform data exchange.

The *SimpleViewer* and *SCADAViewer* reference frameworks, located in the GLG installation directory, provide source code examples for integrating the GLG Data Gateway into a GLG application. For a standalone implementation, refer to the source code example in the *gateway/standalone_example* directory.

For detailed information and API descriptions, visit the Data Gateway Documentation at the following link:

```
https://www.genlogic.com/doc_html/start.html#DataGatewayDoc
```

Controlling a Drawing from a Program

A GLG drawing is controlled from a program in the same way it is controlled from the GLG Graphics Builder: by setting and querying resources. Resources can be used to change values of any object properties, from a fill color to object visibility, to a number of plots in a chart. Methods for querying or setting resource values are provided in the GLG Standard API.

The GLG Intermediate API provides additional methods that can be used to traverse all objects defined in the drawing to discover the structure of the drawing, constraint object attributes, perform coordinate conversion or implement custom object manipulation, such as dragging objects with the mouse.

Finally, the GLG Extended API provides methods for creating drawings programmatically at run time. It includes methods for creating objects, adding or deleting objects from the drawing, as well as creating and attaching dynamics to objects.

The Intermediate and Extended APIs are described in the *GLG Intermediate and Extended API* chapter on page 135.

Handling User Input

If the drawing is to accept input from a user, some extra steps must be taken to build it. You may need to add input objects (such as buttons or text boxes) to the drawing, or attach actions (such as a mouse click or a mouse over) to objects in the drawing. The *ProcessMouse* attribute of a viewport must be set to the desired set of action types to activate processing of actions attached to objects in the drawing.

At run-time, user interaction is handled in the **input callback**, which is a callback function supplied by the programmer and executed by the GLG Toolkit in response to various possible user actions. A **select callback** can be used for simplified object selection, and a **trace callback** can be used to access low-level events of the native windowing system.

A number of pre-built input objects is provided, accessible in the GLG Graphics Builder via the *Palettes* menu. Custom input objects may also be created as described in the the *Input Objects* chapter of the GLG User's Guide.

Object selection, custom event and command action handling is described in the *Integrated Features of the GLG Drawing* chapter of the GLG User's Guide. The structure of a callback function is described in the *Handling User Input and Other Events* chapter of this manual.

H and V Resources

In the static picture of the GLG drawing architecture that was sketched in the introductory chapters of this guide, all resources are of equal status. All resources are attributes of some object, and they can all be modified or created as the need arises. However, while a drawing is in flux (while it is being drawn, for example), some resources need to be treated differently than others. Because of this need, two different categories of resources exist: those that affect simple values and those that affect the resource hierarchy itself.

The **H** (hierarchy) and **V** (value) drawing resources differ in when they are processed by the widget. The H resources are processed *before* creating the drawing hierarchy, while the V resources are processed *after* it has been established. The H resources can affect the structure of the hierarchy itself, for example by controlling the number of objects in a series, while the V resources are processed after the specified number of objects in the drawing hierarchy have been created.

A series object, for example, replicates its template object to produce a number of instances. The series' factor defines how many objects appear in that series. Referencing the eighth object in a series makes no sense before the instances of objects are created. The value of the series factor is another example of an H resource. Setting the factor value with a V resource would work but would

introduce inefficiency, since the drawing hierarchy would first be created with the number of series instances defined by the old value, then destroyed and the new number of instances would be created. Other example of H resources are the non-global attributes of a series template. The template is replicated at the time of hierarchy setup, and its attributes have to be set before the hierarchy is set up in order for the instances to inherit the template's attribute values. All H resources are processed before the drawing hierarchy is created, which guarantees that all resources depending on the number of objects in the drawing hierarchy are accessible.

V resources are used for setting resources that depend on the structure of the drawing hierarchy or that do not exist or are inaccessible before creating the hierarchy. Consider an example where the GLG Toolkit, in a program called "example," reads a bar graph with 10 data samples from a drawing file. We'd like to change the number of data samples to 100 and to have the color of the 25th data sample appear red when the widget is displayed.

Using H and V Properties of Native Controls

Let's start with setting the value of the "*DataGroup/Factor*" resource of the widget to 100. Because this resource affects the hierarchy of the drawing, we use an H resource. Using the X Windows protocol for setting resource values, we might use a line in the configuration file such as:

```
example*XtNglgHResource0 "DataGroup/Factor d 100"
```

Using the GLG Active X control on Windows, it would be:

HProperty0: *DataGroup/Factor d 100*

Using a V resource to set this value would make the GLG Toolkit create a drawing with 10 data samples, discard it, and create another drawing with 100 data samples. Careful use of H and V resources can avoid this kind of inefficiency.

Now we want to set the color of the 25th data sample to be red. However, the original drawing has only 10 data samples; so the 25th data sample does not exist in the drawing. Because of the 25th data sample does not exist until all the H resources are processed, its color cannot be set using H resources. Trying to do so generates an error message saying that the resource cannot be accessed.

To change the color of a graphical object, you only change the value of a data object in the hierarchy; you do not change the hierarchy. Therefore, a V resource is appropriate for this purpose. These are treated after the specified data sample has been created and is accessible. The line in the configuration file for our example program would then look like this:

```
example*XtNglgVResource0 "DataGroup/DataSample24/FillColor g 1. 0. 0."
```

Note that since the index is zero-based, the number 24 is used to access the 25th data sample.

Using the GLG Active X control on Windows, it would be:

VProperty0: *DataGroup/DataSample24/FillColor g 1. 0. 0.*

To set the resources that do not depend on the number of objects in the drawing hierarchy, you may use either H or V resources. For example, you can set a background color of the widget's drawing using either kind of resource. Note, however, that many resource names depend on the structure of the resource hierarchy to be accurately interpreted.

Handling H and V Resources in the Program Code

When the GLG API is used to set drawing resources, the same considerations apply. To increase the efficiency of your application program and avoid errors, insure that all the H resources that need setting are set first, before the V resources. In our example above, the *Factor* resource should be set first, then the *GlgUpdate* function should be called, and then the V resources set:

```
GlgSetDResource( drawing, "BarGraph/DataGroup/Factor", 100. );
GlgUpdate( drawing );
GlgSetGResource( drawing,
            "BarGraph/DataGroup/DataSample24/FillColor", 1., 0., 0. );
```

Utilities

The GLG Toolkit contains a small number of utility programs for use by an application programmer:

datagen
Generates a variety of data streams used for testing and prototyping. It is primarily used inside the GLG Graphics Builder for animating the drawing with simulated data in the Run mode.

gcodegen
Creates a memory image of a GLG drawing in the C language format. The output of the utility is a file with *.c* extension, containing the image of the drawing in a form of the C source code. The output file can be compiled into the program's executable, making it possible to create a single program executable and eliminate a need to distribute additional drawing files.

gconvert
Converts drawing files from one GLG format to another; can also be used to change resources of a drawing in a batch mode using the *-command* or *-script* command line options.
A GLG drawing file may have one of three different formats: binary, ASCII, or extended. Binary drawings are the fastest to load from the memory image, but ASCII drawings are more portable. The extended format is used to port drawings to different versions of the toolkit.

These tools are described in detail in the *GLG Programming Tools and Utilities* chapter on page 329.

Chapter 2

Using the C/C++ version of the Toolkit

2

2.1 Displaying a GLG Drawing

The first step in operating a GLG drawing from a program is to display that drawing. GLG drawings are environment and platform-independent, as are many of the aspects of controlling and modifying the drawings. The task of displaying the drawings in a display window, however, is peculiar to the windowing system and environment in use.

This chapter describes the task of creating and displaying a GLG drawing in C and C++ programs used in different windowing environments on Linux/Unix and Windows.

This chapter contains four sections:

- The first section describes the task of loading and displaying a GLG drawing using a GTK GLG widget on **Linux**.
- The second section on page 33 describes how to perform these steps using a legacy Motif GLG Wrapper widget on **Linux/Unix**.
- The third section of this chapter on page 39 provides information on loading and displaying a GLG drawing on **Windows**.
- The last section on page 43 describes a **platform-independent GLG Generic API**, which can be used for loading and displaying a GLG drawing in cross-platform way. The source code of the applications created using the Generic API is portable between Linux/Unix and Windows environments.

The Windows version of the GLG Toolkit also includes the **GLG ActiveX Control**, an ActiveX component providing full GLG functionality. The GLG ActiveX control allows GLG drawings to be embedded and used in a variety of user applications and environments that support ActiveX controls. For more information, see the *Using the ActiveX Control* chapter on page 275.

In addition to the C language functions, **GLG C++ bindings** provide **C++ classes** to instantiate a GLG drawing as a class in C++ applications. The Toolkit provides classes for instantiating a GLG drawing in both **GTK, Qt, Motif** and **MFC** environments, as well as **cross-platform classes** that use GLG Generic API and may be used in either environment. See the *GLG C++ Bindings* chapter on page 251 for more information.

The **Qt and GTK integration examples** in the *integration* directory of the GLG installation demonstrate how to deploy GLG drawings in the Qt and GTK applications.

The **Java version** of the toolkit uses either the GLG Java Bean or the GLG Generic API provided by the *GlgObject* class to load and display the drawing.

The **C#/.NET version** uses either the native .NET *GlgControl* or the GLG Generic API. Refer to the *Using the Java and C# Versions of the Toolkit* chapter on page 311 for more information.

The **JavaScript version** of the toolkit uses the GLG Generic API to load and display the drawing on a web page in HTML.

Using GLG GTK Widget on Linux

This section describes how to use the *GtkGlg* widget in the **GTK version** of the GLG C/C++ library on Linux (*libglg_gtk.a*). Refer to the *integration/gtk3_glg_native* directory of the GLG installation for a ready to build example of a GTK project. The *integration/gtkmm3_glg_native* directory contains a GTKMM example.

The GtkGlg widget is a widget wrapper around the GLG library. The widget is used to embed GLG drawings into existing GTK interfaces. After the wrapper widget is instantiated in an application's widget hierarchy, it loads and displays a GLG drawing, providing the GLG API to update the display with new data and access GLG objects in the drawing.

There are a few basic steps involved in displaying a GLG drawing in the GTK application:

- Creating the *GtkGlg* widget. This is done the same way as any other GTK widget, using the *gtk_glg_new* function as shown in the example below.
- Loading the drawing using the *GlgLoadWidgetFromFile* function.
- Attaching callbacks that may be used to handle user interaction with the drawing, such as selecting objects with the mouse or interacting with the input object in the drawing. This is done using the widget's *gtk_glg_add_callback* function. Alternatively, the *GlgAddCallback* function may be used to attach callbacks to the drawing's viewport or any of its child viewports.
- Assigning the drawing to the widget using the *gtk_glg_set_viewport* function.

The following example demonstrates how to load and display a GLG drawing in the *GtkGlg* widget:

```
#include "gtkglg.h"
#include "GlgApi.h"

GtkWidget * glg_widget = gtk_glg_new();
GlgObject viewport = GlgLoadWidgetFromFile( "my_drawing.g" );
gtk_glg_add_callback( glg_widget, GLG_INPUT_CB, InputCallback, NULL );
gtk_glg_set_viewport( glg_widget, viewport );
```

Refer to the *integration/gtk3_glg_native* directory of the GLG installation for a complete example program.

The widget created in the above example can be used anywhere in the GTK widget hierarchy, taking care of displaying the drawing and resizing it as necessary. The widget's *gtk_glg_get_viewport* function may be used to get an object ID of the GLG drawing displayed inside the widget. This object ID is used with the GLG API functions to animate the drawing with data and handle user interaction.

The *gtk_glg_new* widget automatically initializes the Toolkit the first time it is invoked. The *gtk_glg_toolkit_init* method can be invoked to explicitly initialize the Toolkit and process any command-line options.

To change the drawing displayed in the widget, simply assign a new drawing to the widget using its *gtk_glg_set_viewport* function. The widget will take care of destroying the old drawing and displaying a new one. Callbacks (if any) added to the widget will be used to handle a new drawing. Any callbacks added directly to the drawing using *GlgAddCallback* must be added to the drawing before attaching the drawing to the widget using *gtk_glg_set_viewport*.

To remove the drawing displayed in the widget, simply pass NULL to the *gtk_glg_set_viewport* function. The widget automatically dereferences the drawing when the widget is destroyed, which will destroy the drawing if its reference count goes to zero.

The *gtk_glg.h* file also provides the following standard GTK widget macros:

GTK_TYPE_GLG
GTK_GLG
GTK_GLG_CLASS
GTK_IS_GLG
GTK_IS_GLG_CLASS

Refer to the *Linking with the GLG Libraries* chapter on page 46 for information about linking with GLG libraries.

Using Legacy GLG Wrapper Widget on Linux/Unix

This section describes how to use the GLG Wrapper Widget in the **legacy X11 version** of the GLG C/C++ library in the **Linux/Unix** (*libglg.a*). Refer to the *GlgWrapperC (Legacy X11 Version of the Toolkit on Linux/Unix Only)* section on page 273 for details on the C++ bindings that can be used in a C++ programs.

The **GLG Wrapper Widget** is a Motif-based widget wrapper around the GLG library. The widget is used to embed GLG drawings into existing Motif interfaces and inherits its attributes from the *XmManager* Motif class. After the wrapper widget is instantiated in an application's widget hierarchy, it loads and displays a GLG drawing, providing the GLG API to update the display with new data and access GLG objects in the drawing.

There are three basic steps involved in displaying a GLG drawing in the X Windows environment. The first step is creating a GLG Wrapper Widget to contain the drawing.

After the widget is created and the drawing displayed, a program must acquire a **viewport handle** (or **object ID**) to use with the rest of the GLG API. Finally, when the program's execution is completed, it must close the widget and drawing. Briefly, the tools for these steps are as follows:

- To create a GLG Wrapper Widget instance, use *XtCreateWidget* and specify the class variable *GlgWrapperWidgetClass* and either the *XtNglgDrawingFile, XtNglgDrawingObject*, or *XtNglgDrawingImage* resource.
- To destroy a GLG Wrapper Widget, use *XtDestroyWidget* and specify the widget ID of the GLG wrapper.
- To manipulate objects in a viewport of the GLG Wrapper Widget, use *XglgGetWidgetViewport* to create an object ID.

These steps are described in detail in the sections that follow.

Creating the Wrapper Widget

An application program creates a GLG Wrapper Widget the same way it creates any other widget. Use the *XtCreateWidget* function from the X Toolkit, and the *GlgWrapperWidgetClass* from the GLG Toolkit.

Graphical objects to be displayed inside the widget are defined by a supplied GLG drawing. The drawing used for a widget defines the type of the widget and its appearance.

Synopsis

To create a GLG Wrapper Widget with the *XtCreateWidget* function, use the following arrangement of include files in your program:

```
#include <X11/StringDefs.h>
#include <X11/Intrinsic.h>
#include "GlgWrapper.h"
```

Create the widget with a call to the X toolkit:

```
widget = XtCreateWidget( name, glgWrapperWidgetClass,... );
```

Wrapper Widget Resources

The GLG Wrapper Widget is a subclass of the *XmManager* class. Like other widgets, the GLG Wrapper Widget is controlled by querying and setting resources.

Warning: The resources of an X widget like the GLG Wrapper Widget are part of a different set of resources from the resources of a GLG drawing. The structure is similar, but the application is different. X resources refer to the data structure that makes up an entire screen, while the GLG resources only refer to the animated drawing. The two sets do interact; you can use the X resources of the GLG Wrapper Widget to set values for the GLG drawing resources. However, the two must not be confused.

The following sections describe the resources of the GLG Wrapper Widget. The entire list of resources is summarized in the table on page 38.

Loading the Drawing

There are three resources of the GLG Wrapper Widget you can use to define the source of the GLG drawing to be displayed:

XtNglgDrawingFile

A character string specifying the name of the GLG drawing file.

XtNglgDrawingObject

An object ID of an already-loaded viewport object to be used by the GLG Wrapper Widget. The drawing might have been loaded from a file or an image, or generated with the GLG Extended API.

XtNglgDrawingImage

A long integer specifying the memory address of the drawing's memory image. This must be used in conjunction with *XtNglgImageSize*, which must contain an integer defining the size of that memory image. This is acquired from the *gcodegen* utility.

One of these three resources must be defined when a GLG Wrapper Widget is created. If all three are defined, *XtNglgDrawingFile* is used. If the file cannot be read, a warning message is generated, then *XtNglgDrawingImage* is used as a fall back resource. If none of these resources are defined or none of them are readable, an Xt error message is generated.

The drawing specified by the *XtNglgDrawingFile* or *XtNglgDrawingImage* resource must have a viewport named *$Widget* at the top level of the drawing's hierarchy. If this condition is not satisfied, the drawing is not recognized as a valid drawing for the GLG Wrapper Widget. The *HasResources* flag of the viewport should be set to YES. For information about drawing resources, see the *Structure of a GLG Drawing* chapter.

The GLG Wrapper Widget manages the geometry of GLG graphical objects inside it, including any children GLG widgets represented by viewport objects. When the geometry of the widget is changed, it rubberbands all contained objects accordingly.

Setting the Drawing Resources

The GLG Wrapper Widget allows access to all the resources of the GLG drawing it contains in two different ways. After a drawing is displayed, you can use functions from the GLG API to manipulate drawing resources. The manipulation can happen either within the same process that started the GLG Wrapper Widget or from a remote process.

You can also set initial values for drawing resources using the resources of the GLG Wrapper Widget itself. Because the resource hierarchy of a drawing is infinitely variable, it is not feasible to allocate a separate GLG Wrapper Widget resource for every corresponding GLG drawing resource. Instead, the wrapper widget has a fixed number of dynamic resources which are not associated with any particular drawing resource but are simply used as entry points to access the drawing resource hierarchy.

You can also use the *XtSetArgs* function to manipulate the drawing resources through the GLG Wrapper Widget resources. However, this is a clumsy mechanism for an application program, and is not recommended. Using the GLG API gives an application program much finer control over when the drawing is updated and does not incur Xt resource setting overhead.

The GLG Wrapper Widget contains two sets of twenty resources, named *XtNglgHResource0* to *XtNglgHResource19* and from *XtNglgVResource0* to *XtNglgVResource19*. (The difference between H and V resources is discussed in the *H and V Resources* section of the *Integrating GLG Drawings into a Program* chapter.) These resources contain character strings which in turn contain the names and values of GLG drawing resources. The strings have the following syntax:

```
<resource_name>  <resource_type>  <values>
```

<resource_name>

The name of the GLG drawing resource, including the complete path (omitting the "*$Widget*"). For example, "*XLabelGroup/XLabel4/String*" accesses the text string of the fifth

label on the X axis in a graph widget.

<resource_type>

The type of the resource and can be one of the following:

d Indicates the resource value is represented by a single floating point number (a scalar), like a line width, a font number or a value of a data sample

s The resource value is a string, like a text string of a label text object

g The resource value is a set of three floating point numbers, like a geometrical point with X, Y and Z coordinates or a color, in which case the three components represent the color's Red, Green, and Blue values.

<values>

A value or a set of values for the resource. The values given depend on the resource type.

The following strings are examples of specifying resources:

```
"DataGroup/DataSample3/Value d 2.5"
"DataGroup/DataSample5/Value d 4"
"XLabelGroup/XLabel4/String s April"
"DataGroup/DataSample6/FillColor g 0.5 0 0.9"
```

The GLG Wrapper Widget does not need forty different dynamic resources of this type. In one sense, it would be adequate to have just one, and call it repeatedly. However, it is often useful to be able to keep the initial values of drawing resource in the X resource database, or in the X configuration files. Having many of these dynamic resources allows an application program to set several different drawing resources from the X resource database.

In the event that there are not enough GLG Wrapper Widget dynamic resources for your purpose, you can use the *XtNglgHInitCB* and *XtNglgVInitCB* callbacks to set the initial resources using functions of the GLG API.

Callback Resources

The *XtNglgHInitCB* callback is called after the viewport object containing the widget's drawing has been created, but before the drawing hierarchy is created, allowing you to set H resources from the callback function itself. You can use the GLG API functions to do this, although you cannot modify the drawing itself with the Extended API from within one of these callback functions.

Setting H resources in this callback before the hierarchy is created eliminates the possible inefficiencies that arise when setting an H resource overrides an already existing part of the object hierarchy. The *call_data* parameter of the callback is not used.

The *XtNglgVInitCB* callback is called after the widget's drawing hierarchy has been created, but before its graphical objects are drawn, allowing you to set V resources that depend on the number of objects in the hierarchy. These resources may be used to set the attributes of the created instances of objects for the initial appearance.

The *XtNglgSelectCB* callback is invoked after each mouse click event and provides a list of objects selected with the mouse.

The *XtNglgInputCB* callback is invoked after each user interaction with objects in the drawing and provides additional information which may be used to handle input events.

Trace callbacks (*XtNglgTraceCB* and *XtNglgTrace2CB*) can be used in addition to the input and selection callbacks to handle low-level events. If attached, these callbacks are invoked for any native event received by any of the drawing's viewports. The trace callback is invoked just before processing the event by the select and input callbacks, and the trace2 callback is invoked after the event has been processed.

The *XtNglgHierarchyCB* callback is invoked when *SubWindow* or *SubDrawing* objects in the widget's drawing load their templates. The callback is invoked twice: before the hierarchy setup of the template and after the hierarchy setup but before the template is drawn.

For more information about the input, select, trace and hierarchy callbacks, refer to the *Callback Events* chapter on page 115 of this manual.

The GLG Wrapper Widget also provides Motif-style *XtNglgMotifSelectCB* and *XtNglgMotifInputCB* callbacks, which receive the *GlgInputCBStruct* and *GlgSelectCBStruct* callback structures as parameters. These structures are defined in the *GlgWrapper.h* include file and provide a Motif-style callback interface. The *GlgInputCBStruct* structure contains *message_obj* parameter, and the *GlgSelectCBStruct* structure contains the *selection_list* parameter, providing the same functionality as the other styles of these callbacks. Refer to the *Callback Events* chapter on page 115 of this manual for details.

Sequence of Events

When a GLG Wrapper Widget is realized with a GLG drawing in it, the following sequence of events occurs:

1. The GLG drawing (supplied by the *XtNglgDrawingFile* or *XtNglgDrawingImage* resource) is loaded.

2. All current values of *XtNglgHResource<N>* resources are applied to the drawing.

3. The *XtNglgHInitCB* callback is invoked (if supplied).

4. The drawing hierarchy is set up.

5. All current values of *XtNglgVResource<N>* resources are applied to the drawing.

6. The *XtNglgVInitCB* callbacks is invoked (if supplied).

Setting the *XtNglgDrawingFile* and *XtNglgDrawingImage* resources after the widget has been realized loads the new drawing and discards all changes made to the old drawing. After reloading a new drawing file or image, the widget executes the same initialization sequence described above. Note that applying the current values of the GLG Wrapper Widget resources may generate an error message if some of the drawing resources specified are not applicable for the new drawing. To avoid the error message, set the offending wrapper widget resource to NULL.

Summary of GLG Wrapper Widget Resources

When creating a GlgWrapper widget instance, the following resources are retrieved from the argument list or from the resource database:

GLG Wrapper Widget Resource Summary

Name	Type	Default	Description
XtNglgDrawingFile	String	NULL	Name of the file containing a GLG drawing.
XtNglgDrawingImage	XtPointer	NULL	Address of the memory image of a GLG drawing.
XtNglgImageSize	long	0	Size of the memory image.
XtNglgHResource0 XtNglgHResource1 ... XtNglgHResource19	String String String	NULL NULL NULL	Entry points for accessing **H** resources of graphical objects in the drawing.
XtNglgVResource0 XtNglgVResource1 ... XtNglgVResource19	String String String	NULL NULL NULL	Entry points for accessing **V** resources of graphical objects in the drawing.
XtNglgHInitCB	XtCallback list	NULL	Callback list for setting initial values of **H** resources using convenience functions.
XtNglgVInitCB	XtCallback list	NULL	Callback list for setting initial values of **V** resources using convenience functions.
XtNglgSelectCB	XtCallback list	NULL	Callback list for selecting objects in the widget.
XtNglgMotifSelectCB	XtCallback list	NULL	Callback list for selecting objects in the widget, Motif-style callback structure format.
XtNglgInputCB	XtCallback list	NULL	Callback list for the input activity in the widget.
XtNglgMotifInputCB	XtCallback list	NULL	Callback list for the input activity in the widget, Motif-style callback structure format.
XtNglgTraceCB	XtCallback list	NULL	Callback list for trace callbacks.
XtNglgTrace2CB	XtCallback list	NULL	Callback list for trace2 callbacks.
XtNglgHierarchyCB	XtCallback list	NULL	Callback list for hierarchy callbacks.

Obtaining a Viewport Handle

To use any of the functions from the GLG API, you must get an object ID for the main viewport of the drawing. This is the one called *$Widget*. (See page 35.) To obtain this object ID from the widget, use the *XglgGetWidgetViewport* function.

```
GlgObject XglgGetWidgetViewport( widget )
   Widget widget;
```

Parameters

widget

The Xt widget ID that specifies the GLG Wrapper Widget.

The GLG Wrapper Widget contains a viewport. The viewport manages all aspects of the drawing widget's behavior and appearance. When accessing resources or performing any other operations on a GLG drawing, the object ID returned by *XglgGetWidgetViewport* is passed as a "handle" identifying the viewport. The viewport is created only after the GLG Wrapper Widget has been realized, so this function returns NULL if called before then.

Closing the Wrapper Widget

To close the GLG Wrapper Widget gracefully, use the *XtDestroyWidget* function from the X Toolkit. It accepts only one argument, the widget ID returned by the *XtCreateWidget* function.

```
XtDestroyWidget( widget )
   Widget widget;
```

GLG Custom Control for Windows

This section describes how to use the C/C++ version of the GLG library on Windows using the **Windows API**. Refer to the *GLG C++ Bindings* chapter on page 251 for details of the GLG **C++** and **MFC classes**. Refer to the *Using the ActiveX Control* chapter on page 275 for a description of using the **GLG ActiveX control**.

The GLG Custom Control is a Windows wrapper designed for embedding GLG drawings in the programs using a **Windows API**. It allows embedding GLG drawings into the Windows interface hierarchy as a custom window control (a window with a custom window type). The GLG Custom Control manages the geometry of GLG graphical objects in it, including any child GLG widgets represented by nested viewport objects. When the geometry of the Control window changes, it proportionally resizes all the GLG objects inside it.

The first step in displaying a GLG drawing in the Windows environment is creating a window data structure to contain that drawing. This is the **GLG Custom Control**. After the window is created and the drawing displayed, a program must acquire a **viewport handle** (or **object ID**) which is used for updating the drawing with data via the GLG API. Finally, when the program's execution is completed, it must close the window and drawing. Briefly, the tools for these steps are as follows:

- To create a GLG Custom Control instance, load the GLG drawing and use *CreateWindow*. Other functions from the GLG API are used to set up the drawing hierarchy and display the drawing.
- To destroy a GLG Custom Control, use *DestroyWindow* and specify the Window ID of the control.
- To manipulate objects in a viewport of the GLG Wrapper Widget, use *GlgGetWindowViewport* to obtain an object ID of the viewport.

These steps are described in detail in the sections that follow.

Creating the GLG Custom Control

There are four steps involved in displaying a GLG drawing on Windows:

1. Read a GLG drawing into the program. This can be done from a file, with *GlgLoadWidgetFromFile*, or from memory, with *GlgLoadWidgetFromImage*.

2. Use *GlgSetDefaultViewport* to identify the viewport to be used as the drawing widget.

3. Create a Custom Control using the drawing. This is done with the *CreateWindow* function of the Windows API. The Windows class of the GLG Custom Control is *GlgControl*.

4. Use *GlgInitialDraw* to draw the initial drawing. You can delay this step until after you use *GlgGetWindowViewport* to create a widget ID for the GLG drawing. At that point, you can use either *GlgInitialDraw* or a combination of *GlgSetupHierarchy* and *GlgUpdate*.

The following example shows how to create a GLG Custom Control:

```
#include "GlgApi.h"

HWND GlgControl;
GlgObject drawing;

drawing = GlgLoadWidgetFromFile( "BarGraph.g" );
GlgSetDefaultViewport( drawing );
GlgControl = CreateWindow( "GlgControl", "GlgControlTest",  WS_VISIBLE
                         | WS_CHILD | WS_BORDER, 50, 50, 100, 150,
                         parent, NULL, hInstance, NULL );
(LPVOID) GlgInitialDraw( drawing );
```

Note that only one GLG Custom Control can be attached to each instance of the loaded drawing. To display several copies of a drawing, you must load or copy several instances of it into memory, creating each control with the *CreateWindow* function.

Setting Initial Resources

A GLG drawing represents a hierarchy of graphical objects. This hierarchy is created when the drawing is used in a GLG Custom Control. If your application needs to change a resource that affects the structure of a drawing's resource hierarchy, the most efficient way is to do it *before* the drawing hierarchy is created (that is, before using the drawing to create a GLG Custom Control).

For example, if you want to change the number of bars in a bar graph's drawing, you should do it before using the drawing for creating a GLG Custom Control:

```
HWND GlgControl;
GlgObject BarGraphDrawing;

BarGraphDrawing = GlgLoadWidgetFromFile( "BarGraph.g" );

/* Setting the number of bars in the bar graph to 30. */
GlgSetDResource( BarGraphDrawing, "DataGroup/Factor", 30. );
```

```
GlgControl = CreateWindow( "GlgControl", "BarGraph Custom Control",
                WS_VISIBLE | WS_CHILD | WS_BORDER, 50, 50, 100, 150,
                parent, NULL, hInstance, NULL );
GlgInitialDraw( BarGraphDrawing );
```

Resources that must be set before creating the hierarchy include factors of series objects, templates of series objects, Global attribute flags and some others. These are called H resources, and are described in the *H and V Resources* section of the *Integrating GLG Drawings into a Program* chapter.

Some resources, like attributes of an individual instance of the series object's template, may be accessed only after the drawing hierarchy is created. These are the V resources. For example, setting an initial value of the first bar in the bar graph must be done *after* the bar instances are created:

```
double GetData();
HWND GlgControl;
GlgObject BarGraphDrawing;

BarGraphDrawing = GlgLoadWidgetFromFile( "BarGraph.g" );

GlgControl = CreateWindow( "GlgControl", "BarGraph Custom Control",
            WS_VISIBLE | WS_CHILD | WS_BORDER, 50, 50, 100, 150,
            parent, NULL, hInstance, NULL );
GlgSetupHierarchy( BarGraphDrawing );

/* Setting the value of the first data sample for the initial
            appearance. */
GlgSetDResource( BarGraphDrawing, "DataGroup/EntryPoint", GetData() );

GlgUpdate( BarGraphDrawing ); /* Draw control's graphics */
```

The rest of the resources may be changed at any time, although remember that if you want to change some resources for the initial appearance of the control, do it before the call to the *GlgUpdate* or *GlgInitialDraw* functions.

Obtaining a Viewport Handle

To use any of the functions from the GLG API, you must get an object ID for the main viewport of the drawing. This is the one called *$Widget*. (See page 35.) To obtain this object ID from the widget, use the *GlgGetWindowViewport* function.

```
GlgObject GlgGetWindowViewport( glg_control_window )
   HWND glg_control_window;
```

The *glg_control_window* argument is the Window ID returned by the *CreateWindow* function. If a window other than a GLG Custom Control is passed as a parameter, the result is undefined. The **viewport handle** is the return value of this function.

A viewport is a GLG Toolkit object which manages all aspects of the GLG Custom Control's behavior and appearance. When accessing resources or performing any other operations on the Control, the viewport handle is passed as a parameter to the functions of the GLG API Library.

Closing the Custom Control

At the end of a program's execution, you should use the *DestroyWindow* Windows function to close the GLG Custom Control in an orderly fashion. By default, this is done for you when you exit the program.

Messages

The GLG Custom Control provides a window procedure to work with the Windows environment.

On non-TrueColor systems, the GLG Custom Control creates its own color palette and handles palette related messages. However, Windows sends palette messages to the top level window only, therefore the application program must be responsible for propagating these messages to the GLG Custom Control. This is also true for some focus-related messages, as shown in the example below.

An example of how messages are propagated to the GLG Custom Control is shown in the following example:

```
extern HWND GlgControl;    /* A children window. */
LRESULT CALLBACK MyWindowProc( hWnd, iMessage, wParam, lParam )
   HWND hWnd;
   UINT iMessage;
   WPARAM wparam;
   LPARAM lparam;
{
   switch( iMessage )
   {
      /* Propagate palette messages to the Control. */
      case WM_PALETTECHANGED:
      case WM_QUERYNEWPALETTE:
         if( GlgControl )
            return
             GlgControlWindowProc( GlgControl, iMessage, wParam,
               lParam );

      /* Let the Control know about focus change, etc. */
      case WM_SETFOCUS:
      case WM_KILLFOCUS:
      case WM_CANCELMODE:
         if( GlgControl )
           GlgControlWindowProc( GlgControl, iMessage, wParam, lParam
                 );
         break;

      ... /* Some other application specific cases. */
   }

   return DefWindowProc( hWnd, iMessage, wParam, lParam );
}
```

The *GlgControlWindowProc* function has the standard arrangement of windows procedure parameters, whose definitions can be found in Windows programming manuals. It is included in the Windows version of the GLG API library.

Loading and Displaying a GLG Drawing using Generic API

The GLG Generic API can be used to load the drawing into a program and display it in a cross-platform way, freeing the programmer from a need to use platform-specific native integration code. An application written using the simple GLG Generic API can be compiled and run on either Windows or Linux/Unix without any changes to the source code. GLG demos and most of the coding examples use the GLG Generic API and may be compiled and run on both Windows and Linux/Unix.

Linux Note: On Linux, two versions of the GLG library are provided:

- the GTK version *libglg_gtk.a* uses GTK/Cairo
- the legacy X11 version *libglg.a* uses X11/Motif and has a different look and feel.

  An application that uses the generic API can be built with either version of the library. To use the GTK version, the GLG_GTK macro has to be defined when compiling the program. This macro is already defined in the GTK version of the sample makefiles in the *src* directory of the GLG installation.

The following example loads and displays a GLG drawing using the GLG Generic API:

```
#include "GlgApi.h"
#include "GlgMain.h"

int GlgMain( argc, argv, InitialAppContext )
   int argc;
   char * argv[];
   GlgAppContext InitialAppContext;
{
   GlgObject viewport;

   GlgInit( False, InitialAppContext, argc, argv );

   viewport = GlgLoadWidgetFromFile( "bar1.g" );
   GlgInitialDraw( viewport );

   return GlgMain( InitialAppContext );
}
```

The *GlgMain.h* include file defines a cross-platform program entry point, *GlgMain*.

The *GlgInitialDraw* function creates a top-level window and displays a GLG drawing in it in a cross-platform way. This code may be compiled and run on both Linux/Unix and Windows platforms. Refer to the GLG programming examples in the *examples* directory for a complete example of an application that loads, displays and updates GLG drawing with data.

Alternatively, a call to *GlgInitialDraw* can be split into two equivalent calls, *GlgSetupHierarchy* and *GlgUpdate*, to perform any required initialization before displaying the drawing:

```
GlgSetDResource( viewport, "Chart/NumPlots", 3.);
GlgSetupHierarchy( viewport );  /* Creates 3 plots in the chart. */
```

```
/* Set any required plot properties. */
GlgSetDResource( viewport, "Chart/Plots/Plot#2/LineWidth", 3. );
GlgUpdate( viewport );  /* Display the drawing. */
```

Refer to the *The GLG Generic API* section on page 52 for the list of Generic API methods, and to the *GLG C++ Bindings* chapter on page 251 for information on using the **C++ version** of the GLG Generic API.

Coding Examples of Displaying and Animating a GLG Drawing

Subdirectories of the *examples_c_cpp* directory of the GLG installation contain various examples of loading, displaying and animating a GLG drawing. Some examples, such as an animation example in the *examples_c_cpp/animation* directory, demonstrate how to use both the Generic API and the native windowing system controls:

- The version with the filename ending with a letter "G" uses the GLG Generic API which can be used on both Linux/Unix and Windows.
- The version ending with "X" shows an example of using the legacy X11 GLG Wrapper Widget under X Windows.
- The version ending with "W" uses the GLG Custom Control on Windows.
- The files with a ".c" extension demonstrate the C version, and files with a ".cpp" extension present the C++ version of the same example.

Compiling and running Example Programs

Visual Studio projects are provided in each directory for building demos and most of the examples on Windows. On Linux/Unix, makefiles for building demos and examples are provided in the *src* subdirectory of the GLG installation directory. Refer to the *Building GLG Demos and Examples* section below for more information.

2.2 Compiling and Linking GLG Programs

Building GLG Demos and Examples

Visual Studio Projects for Windows

On Windows, Visual Studio projects are provided for all GLG demos and most of the example programs. The project files are located in each demo or example directory. To build, simply open a project file in the Visual Studio, select the desired platform (Win32 or x64) and build the solution.

The sample projects use the GLG library in a form of a DLL. Refer to the *Using Static GLG Libraries* section on page 47 for information about using GLG static libraries.

When using the Toolkit version that target older Visual Studio 2010, the Microsoft Visual C++ 2010 Redistributable Package needs to be installed. The GLG installation provides an option to install it, and the *vcredist_x86_10.0_sp1.exe* file is also provided in the *lib* directory of the GLG installation.

Makefiles for Linux/Unix

On Linux, two sets of makefiles for building demos and examples are provided in the *src* subdirectory of the GLG installation directory:

- *makefile_gtk* and *makefile2_gtk* for the GTK version of the GLG C/C++ Library
- *makefile_x11* and *makefile2_x11* for the legacy X11 version of the library.

On non-linux Unix systems, only the legacy X11 version is provided.

Each set contains two different makefiles:

- *makefile_gtk* (and *makefile_x11* for the legacy version) grabs all source files in the current directory and links them in a single executable. It can be used for building examples in directories that contain only one version of the example, such as the **SCADA Viewer example** in the *SCADA_Viewer/C* directory.

  Note: If this makefile is used in a directory which contains multiple versions of the same demo, such as examples in the *examples_c_cpp/Dashboard* directory, it will generate multiple definition errors. Use *makefile2_gtk* for directories with multiple versions of the example.

- *makefile2_gtk* (and *makefile_x11* for the legacy version) may be used to compile and link individual source files in a directory that contains several versions of the same example, such as examples in the *examples_c_cpp/Dashboard* directory. The *TARGET* and *ADD_FILES* variables in the *makefile2* file should be edited to define source files that have to be included in the build.

To build a project on Linux/Unix, copy one of the above mentioned makefiles into the project's directory, edit it to define a path to the GLG installation directory and other platform options listed at the beginning of the makefile (see the comments inside the makefile for specific instructions). To build **C++ projects**, also change the *CC* variable in the makefile from *gcc* to *g++*. Then change to the project's directory and use *make* to build the project using the *-f* option to specify the makefile:

```
cd <project_dir>
make -f makefile_gtk
```

Alternatively, the copied makefile can be renamed as *makefile* to use *make* without the *-f* option.

Using Constant Strings

By default, C++ bindings use constant strings. This may be changed by defining the CONST_CHAR_PTR macro with a value of 0 before including the *GlgClass.h* file.

The C API uses non-constant strings by default. This can be changed by defining the GLG_C_CONST_CHAR_PTR macro with the value of 1 before including the *GlgApi.h* file.

Linking with the GLG Libraries

The details of the process of linking an application program with the GLG Toolkit library depends on the operating system and windowing environment where the application will run.

Linux/Unix

The *libglg* library supplied with the GLG Toolkit contains all functions described in this guide, including the GLG Standard Library and GLG Generic API Library. There are several versions of the library for different programming environments:

- *libglg_gtk.a*
  The GTK version of the library. The library is provided on Linux and provides integration with the native Linux GTK environment.
- *libglg.a*
  The legacy X11/Motif version of the library. On Linux, it was built using OpenMotif 2.3 to support XFT fonts and UTF-8 locales. On other Unix platforms, it was built with the Motif version provided by a vendor.
- *libglg_x11.a*
  The X11 version of the library for use with the Qt and legacy X11 GTK integrations. This version uses an X11 window as a drawing surface instead of a Motif widget.

For applications that use map server functionality, the *libglg_map* map server library and the *libtiff*[1] library are also required.

All libraries are provided in the *lib* subdirectory of the GLG Toolkit installation. The GLG libraries must be included in the link list before the system libraries. Sample makefiles provided in the *src* subdirectory of the GLG installation contain additional platform-dependent libraries that may be selected by uncommenting them depending on the platform (refer to the *Makefiles for Linux/Unix* section on page 45 for more information).

In the legacy X11 version of the GLG library, if a static Motif library is used instead of the default shared library, the following libraries must also be included: *libXmu, libXft, libXext* and *libXp*. On Solaris and AIX, some extra system libraries (*libnsl* and *libsocket* for Solaris, and *libiconv* for AIX) must also be included, as shown in the sample makefiles.

The linking process for the legacy X11 version also requires *libz*[2]*, libjpeg*[3]*, libpng*[4] and *libfreetype*[5] libraries and related system libraries. On Linux, these libraries are provided by the OS, but a static version of the libraries is also provided in the *lib/syslib* directory of the GLG Toolkit installation for reference. On other Unix systems, these libraries are included in the *lib* directory.

The following line shows the GLG-related fragment of the link command for the GLG program using the Standard API and the Map Server on Linux:

```
... -lglg_gtk -lglg_map ...
```

---

1. Copyright © 1988-1997, Sam Leffler. Copyright © 1991-1997 Silicon Graphics, Inc.
2. Copyright © 1995-2002 Jean-loup Gailly and Mark Adler.
3. Copyright © 1991-1998, Thomas G. Lane, The Independent JPEG Group.
4. Copyright © 2004, 2006-2014 Glenn Randers-Pehrson.
5. Copyright © 1996-2000 by David Turner, Robert Wilhelm, and Werner Lemberg, FreeType Team.

If an application does not use the Map Server and GIS functionality, the *libglg_map* library may be replaced with the *libglg_map_stub* stub. The following line shows a fragment of the link command for the GLG program using the Standard API without the Map Server and GIS functionality:

```
... -lglg_gtk -lglg_map_stub ...
```

The GLG Extended Library is called *libglg_ext* and is used as an add-on to the GLG Standard API library. The following two lines show examples of linking with and without the Map Server and GIS functionality:

```
... -lglg_ext -lglg_gtk -lglg_map ...

... -lglg_ext -lglg_gtk -lglg_map_stub ...
```

The GLG Extended Library can be omitted if an application does not use any of the Extended API functions.

The GLG Intermediate Library *libglg_int* may be used instead of the GLG Extended Library *libglg_ext.*

On platforms that support both the 32 and 64 bit executables, the 64 bit versions of libraries and utilities are provided in the *lib_64* directory.

Windows

Using GLG DLLs

The DLL version of the GLG Standard Library is called *Glg.dll*. To use it, link your application with the *Glg.lib* library.

The GLG Intermediate Library DLL is called *GlgIn.dll* and includes the GLG Standard API. This means you don't have to link with the *Glg.lib* if using *GlgIn.dll*. Link only with the *GlgIn.lib* library to use the Intermediate API DLL.

The GLG Extended Library DLL is called *GlgEx.dll* and includes the GLG Standard and Extended APIs. This means you don't have to link with the *Glg.lib* if using *GlgEx.dll*. Link only with the *GlgEx.lib* library to use the Extended API DLL.

Using Static GLG Libraries

On Windows, an application can be linked with either a static or dynamic GLG library. Two versions of static libraries are provided: one compiled with the *multi-threaded DLL (/MD)* option, and another compiled with the *multi-threaded (/MT)* option. The libraries compiled with the */MT* option use the *MT* suffix to differentiate them from the libraries compiled with */MD*, for example: *GlgLibMT.lib* vs. *GlgLib.lib*. A version of the library matching the compilation options of the application code should be used for linking with the GLG static libraries.

The static versions of the GLG Standard API Library are called *GlgLib.lib* and *GlgLibMT.lib.*

The static versions of the GLG Intermediate Library are called *GlgInLib.lib* and *GlgInLibMT.lib*. They also require linking with the corresponding version of the GLG Standard Library: *GlgLib.lib* or *GlgLibMT.lib*.

The static versions of the GLG Extended Library are called *GlgExLib.lib* and *GlgExLibMT.lib*. They also require linking with the corresponding version of the GLG Standard Library: *GlgLib.lib* or *GlgLibMT.lib*.

The project files supplied with the Toolkit use dynamic linking with the GLG DLLs by default. In order to use the static version of the GLG libraries, define GLG_STATIC in your code:

```
#define GLG_STATIC
```

This must appear before the *GlgApi.h* file, or it may be specified as a preprocessor definition in the project's settings.

To use the static GLG library, an application code must also include a call to the *GlgInit* method, passing an explicit application instance as the application context parameter. (If the dynamic library is used, this is done automatically by the DLL's *DllMain* entry point.)

Using OpenGL in Application Executables

The same application executable may be run in either the OpenGL rendering mode or the native windowing system (GDI) mode. Refer to the *Command-line Options* section on page 295 of the *GLG User's Guide and Builder Reference Manual* for information on the **command-line options that specify rendering mode to be used at run-time**.

Linux Note: On Linux, the GTK version of the Toolkit uses Cairo renderer that supports anti-aliasing and alpha-blending, and doesn't need OpenGL for rendering 2D graphics. The OpenGL renderer is disabled by default in the GTK version but can still be used if hidden surface removal is needed to render 3D objects. Refer the description of the *ForceOpenGL* viewport attribute on page 101 of the *GLG User's Guide and Builder Reference Manual* for more information.

Refer to the *OpenGL or GDI (Native Windowing System) Renderer* section on page 32 of the *GLG User's Guide and Builder Reference Manual* for **information on the OpenGL and GDI drivers**.

If the graphics card supports OpenGL version 3.00 and above, a shader-based Core profile may be used for rendering. Refer to the *OpenGL Versions, Compatibility and Core Profiles* section on page 34 of the *GLG User's Guide and Builder Reference Manual* for more information on **OpenGL versions**.

Refer to the *OpenGL on Linux Laptops with NVidia Optimus / Bumblebee* section on page 37 of the *GLG User's Guide and Builder Reference Manual* for a detailed description of the **libraries used by both hardware and software OpenGL renderers**.

Refer to the *OpenGL Setup and Diagnostics* section on page 36 of the *GLG User's Guide and Builder Reference Manual* for **setup and diagnostic options of the OpenGL driver**.

OpenGL Libraries

The **hardware renderer on Linux/Unix** systems uses the *libGL* and *libGLU* libraries provided by the graphics card vendor. These libraries are dynamically loaded by the Toolkit and do not need to be linked in the application executable. If an application uses OpenGL calls itself, it can link the OpenGL libraries, and the Toolkit will use the linked-in libraries.

On Linux, the *libGL* library from the *libgl1-mesa-glx* package provides support for hardware acceleration. The *libGL* library from the *libgl1-mesa-swx11* package supports only the software renderer.

For the **low-priority software renderer on Linux/Unix**, GLG uses the *libOSMesa* library, as well as the *libGLU* library which provides some utility functions.

Linux Note: On Linux, the **software OpenGL renderer is used by default only by the legacy X11 version** of the Toolkit. The **GTK version** uses Cairo that supports anti-aliasing and alpha-blending, and, by default, does not use OpenGL for rendering 2D graphics. If hardware-accelerated OpenGL is needed to perform hidden surface removal for rendering 3D objects or to increase performance of real time charts with large number of data samples, it may be enabled using the *ForceOpenGL* viewport attribute.

On Linux/Unix, the legacy X11 versions of the GLG editors use the *libOSMesa* library provided by the GLG installation in the *glg/lib* directory for the software renderer. If the *libGLU* library is not provided by the hardware renderer, the *libtess_util* library from the *glg/lib* directory is also used instead of the *libGLU* library.

The GLG editors and demos are configured to automatically use the *libOSMesa* library from the GLG installation. If a GLG application wants to use *libOSMesa* library from the *glg/lib* directory for the software renderer, it should include *glg/lib* in LDD_LOAD_PATH or copy the libraries to a place where they will be found by the linker.

An application can also set the *GLG_OPENGL_MESA_LIB_PATH* environment variable or the *GlgOpenGLMesaLibPath* global configuration resource to point to the *libOSMesa* library to be used. Alternatively, *-glg-mesa-lib-path* command-line option may be used to point to the *libOSMesa* library. If the *GLG_DIR* environment variable is set, the library will also be searched in the $*GLG_DIR/lib* directory.

If the hardware OpenGL is not used, the software OpenGL also needs the *libtess_util* library. The library will be searched in the directory where the *libOSMesa* library is located. To specify a different location, use either the *GLG_OPENGL_TESS_LIB_PATH* environment variable, the *GlgOpenGLTessLibPath* global configuration resource, or the *-glg-tess-lib-path* command-line option.

On **Windows**, OpenGL uses Opengl32.dll and Glu32.dll libraries. These libraries are always present, but they support the hardware-accelerated renderer only if drivers for a graphics card with the OpenGL support are installed. Otherwise, only the software renderer is enabled. If hardware acceleration is enabled by the graphics card, the OpenGL libraries on Windows support both hardware and software-based rendering, with no additional libraries required.

On both Linux/Unix and Windows, all OpenGL libraries are dynamically loaded at run time if they are available, so that the executable may be used even if the libraries are absent.

Qt and GTK Integration

The Toolkit provides the Qt, GTK and GTKMM integration files in the *integration* directory which contains a separate subdirectory for each of the environments. Each subdirectory contains an integration example with a source code showing how to use a native environment-specific GLG widget for the corresponding environment: *QGlgWidget*, *glg_gtk,* and *GtkmmGlgWidget*. All files required to build the project are also provided.

On Linux, two versions of the GTK and GTKMM integrations are provided:

- The native integration uses the GTK version of the GLG library (*libglg_gtk.a*) that seamlessly integrates into the GTK environments and supports native GTK input objects inside the GLG drawing.
- The legacy X11 integration is also provided for backward compatibility. It uses the X11 version of the GLG library (*libglg_x11.a*) and does not support native GTK input objects inside the GLG drawing.

The Qt integration on Linux also uses *libglg_x11.a* library and supports native Qt input objects inside the GLG drawing.

On Windows, the standard GLG library is used with the Qt integration.

Error Processing and Debugging

Default Error Handler

C/C++ Error Handler

All GLG library errors are reported using the GLG default error handler. In the Linux/Unix environment, it prints an error message on the console, and on Windows, displays a message box with an error message. On both platforms the default handler also emits an audio beep and logs the error into the *glg_error.log* file.

Log File Location

The log file will be created in the directory specified by the following environment variables:

GLG_LOG_DIR_<*Major*>_<*Minor*>
GLG_LOG_DIR
GLG_DIR_<*Major*>_<*Minor*>
GLG_DIR

The environmental variables are searched in the order they are listed above. The <*Major*> and <*Minor*> are replaced by the major and minor version numbers, for example GLG_DIR_4_6.

If environment variables are not defined, the system attempts to create *glg_error.log* in the following order of priority:

- **GLG Graphics Builder / HMI Configurator:** The GLG installation directory.
- **Application Executables:** The executable's directory.
- **Fallback:** The current working directory (if the executable path is indeterminable).

Java, C# and JavaScript Error Handler

In the Java version, the errors are printed on the standard output stream or in the Java Console, and the errors are not logged in a file.

The C#/.NET version displays an error message box, beeps and logs the errors into the g*lg_error.log* file, the same as the C/C++ version.

In the JavaScript version, the errors are printed on the browser console and are not logged in a file.

Map Server Error Handler

The Map Server errors are logged in the *glm_error.log* file, which will be created in the directory specified by the GLM_LOG_DIR_<*Major*>_<*Minor*> and GLM_LOG_DIR environment variables. If they are not set, either GLG_LOG_DIR or GLG_DIR versions of environment variables listed above are used to determine the directory in which the Map Server log file will be created. If neither environment variable is defined, the current directory is used.

Installing a Custom Error Handler

To intercept GLG errors or to change the default error reporting behavior, you may install a custom error handler using the *GlgSetErrorHandler* function. (This is part of the standard GLG API; see the the *Animating a GLG Drawing with Data Using the Standard API* chapter.). Severe internal errors (like invalid object handles) will still be logged into the *glg_error.log* file, but errors involved with the use of the GLG Toolkit, like not being able to set or get a resource, will not.

In the GTK version of the GLG C/C++ library on Linux, the GTK errors and warnings are reported via the GTK error handler. Refer to the GTK documentation for information on the GTK error handling and debugger break points.

In the legacy X11 version of the GLG library on Linux/Unix, the *XtError* function is used to report widget-related errors (such as an error setting the wrapper widget's resource). You can use *XSetErrorHandler* function to install a custom error handler for Xt errors. The *XSetErrorHandler* function can be used to install a custom error handler for X Windows errors.

Debugging C/C++ Errors

To break on a GLG error in a debugger, set a breakpoint in the *GlgDefaultErrorHandler* C function, or in your custom error handler if one is installed as described above. When using the GLG DLL on Windows, explicitly install a custom error handler via *GlgSetErrorHandler* and set the breakpoint there.

2.3 The GLG Generic API

This section describes the **GLG Generic API for C**. Refer to the *GLG C++ Bindings* chapter on page 251 for information on using the **C++ version** of the GLG Generic API.

If it is important for your application to be cross-platform compatible (or if you don't want to learn another windowing API), you may use the GLG Generic API instead of the GLG Custom Control on WIndows or the GLG Wrapper Widget for Linux/Unix. It contains a generic, platform independent API for creating and manipulating a GLG Widget, as well as a small set of functions for cross-platform development. An application written using the GLG Generic API may be ported between Windows and the Linux/Unix environment by simply recompiling the code. The supplied demos and other coding examples of GLG Generic API may be compiled and run in both Windows and Linux/Unix environments.

Linux Note: On Linux, an application that uses GLG Generic API may be built with either GTK or legacy X11 version of the GLG C/C++ library.

The GLG Generic API is a subset of the GLG Standard API that provides functions for initializing the Toolkit's run time environment, loading a drawing and displaying it in a program using the capabilities of the native windowing system on Windows or Linux/Unix. After loading and displaying the drawing, it is animated using the rest of the functions of the GLG Standard API, which are also cross-platform. A coding example of using the Generic API is provided in the *Loading and Displaying a GLG Drawing using Generic API* section on page 43.

Though the functions provided are all platform-independent, the GLG Generic API library is not intended as a complete cross-platform development library. Instead, it supplies a small set of functions which are sufficient for a platform-independent use of GLG drawings and the GLG Toolkit. It is up to the application developer to write the rest of the application in a platform-independent way.

Function Summary

The GLG Generic API includes the following functions:

- **GlgInitLocale** sets locale.
- **GlgInit** initializes the Toolkit.
- **GlgTerminate** terminates the Toolkit and frees any allocated memory.
- **GlgLoadWidgetFromFile** loads a GLG Widget drawing from a file.
- **GlgLoadWidgetFromImage** loads a GLG Widget drawing from a generated memory image.
- **GlgLoadWidgetFromObject** extracts a GLG widget drawing from a GLG object.
- **GlgSetupHierarchy** sets up a GLG Widget's drawing hierarchy.
- **GlgResetHierarchy** resets a GLG Widget's drawing hierarchy in preparation for being dereferenced with *GlgDropObject*.
- **GlgInitialDraw** draws a GLG Widget for the first time.

- **GlgMainLoop** polls for events.
- **GlgDialogSetupHierarchy** sets up drawing hierarchy of a dialog drawing.
- **GlgConfigureWindow** sets the size and/or position of a viewport's window.
- **GlgDialogInitialDraw** draws a GLG drawing as a dialog for the first time.
- **GlgAddCallback** adds input, selection and other callbacks to a GLG drawing.
- **GlgAddWorkProc** adds a work procedure.
- **GlgRemoveWorkProc** removes an active work procedure.
- **GlgAddTimeOut** adds an interval timer.
- **GlgRemoveTimeOut** removes an active interval timer.
- **GlgSleep** suspends program's execution.
- **GlgBell** produces an audio bell.
- **GlgRand** generates a random number.

These functions are described in detail in the following pages.

Generic Program Entry Point

To define a cross-platform application program's entry point both Linux/Unix and Windows environments, include the *GlgMain.h* header file at the beginning of the program and define the *GlgMain* function in the following way:

```
#include "GlgApi.h"
#include "GlgMain.h"

int GlgMain( argc, argv, InitialAppContext )
   int argc;
   char * argv[];
   GlgAppContext InitialAppContext;
{
   GlgInit( False, InitialAppContext, argc, argv );

   ... /* Place code here. */
}
```

The defined *GlgMain* function will be called when the program is started. The *argc* and *argv* parameters are analogous to the corresponding parameters of the standard *main()* function. The *InitialAppContext* parameter must be passed to the *GlgInit* function as shown above.

GLG Generic API Function Descriptions

In order to use any of the functions in the GLG Generic API, you must include the following header file in your program:

```
#include "GlgApi.h"
```

GlgAddCallback

Adds a callback function to a viewport.

```
void GlgAddCallback( viewport, callback_type, callback, client_data )
   GlgObject viewport;
   GlgCallbackType callback_type;
   GlgCallbackProc callback;
   GlgAnyType client_data;
```

Parameters

viewport

Specifies the viewport the callback will be added to. For the GLG_INPUT_CB callback type, it may be a light viewport.

callback_type

Specifies the type of a callback to be added, such as GLG_SELECT_CB, GLG_INPUT_CB, GLG_TRACE_CB, GLG_TRACE2_CB or GLG_HIERARCHY_CB. There are also special callbacks used with the GLG Wrapper Widget; see page 36.

callback

Specifies a callback function to be called.

client_data

Specifies client data to be passed to the callback function when it is called.

This function adds the specified callback to a viewport. A callback is a user-supplied function that is called by the GLG Toolkit upon some action:

- Selection callback is invoked when a user selects an object inside the viewport.
- Input callback is invoked when an input widgets, such as a slider or a toggle, receives some user input. It is also invoked when an object in the viewport is selected.
- Trace callback is invoked for every native windowing system event received by the viewport or its child viewports and may be used to handle these events.
- Hierarchy callback may be used to initialize loaded subdrawings.

See the *Callback Events* section of the *Handling User Input and Other Events* chapter for a description of callback functions.

Only one callback of each callback type may be added to one widget or an individual viewport. Any subsequent invocations of this function will overwrite the previous value of the callbacks. To remove a callback, call *GlgAddCallback* with NULL as a value of the callback parameter.

Note: Callbacks must be added before the drawing's hierarchy is set up.

Several input and selection callbacks may be added to different viewports on different levels of the drawing hierarchy. However, only the first encountered callback on the lowest level of the hierarchy will be called. For example, if an input callback is attached to a viewport and its child viewport, only the child viewport’s callback will be invoked when input event occurs in the child viewport.

If an event occurs in a child viewport that doesn't have a callback attached, the callback of the first encountered parent viewport that has the callback attached will be invoked.

The viewport's *ProcessMouse* attribute controls processing of the mouse selection events and tooltips inside the viewport and its child viewports.

GlgAddTimeOut

Adds an interval timer in a platform-independent way.

```
GlgLong GlgAddTimeOut( app_context, interval, timer_callback,
            client_data )
   GlgAppContext app_context;
   GlgLong interval;
   GlgTimerProc timer_callback;
   GlgAnyType client_data;
```

Parameters

app_context

Specifies the application context returned by the *GlgInit* function.

interval

Specifies the time interval in milliseconds.

timer_callback

Specifies a procedure to be called when the time expires.

client_data

Specifies the client data to be passed to the specified procedure when it is called.

This function adds a procedure which is called when the specified time interval expires. It returns the **timer ID** which can be used with *GlgRemoveTimeOut*. The interval timer is called only once and is removed immediately after it has been called. The *GlgRemoveTimeOut* function may be used to remove an active timeout. If you want the timer procedure to be called continuously, the timer procedure itself should call *GlgAddTimeOut*.

The following is a prototype for a timer procedure function:

```
typedef void (*GlgTimerProc)( client_data, timer_id )
   GlgAnyType client_data;
   GlgLong * timer_id;
```

The *timer_id* parameter is the same as the one returned by the *GlgAddTimeOut* function. The *client_data* argument is the same as the one supplied in the *GlgAddTimeOut* function.

GlgAddWorkProc

Adds a work procedure in a platform-independent way.

```
GlgLong GlgAddWorkProc( app_context, work_proc, client_data )
   GlgAppContext app_context;
   GlgWorkProc work_proc;
   GlgAnyType client_data;
```

Parameters

app_context

Specifies the application context returned by the *GlgInit* function.

work_proc

Specifies a work procedure function to be called repeatedly.

client_data

Specifies client data to be passed to the work procedure when it is called.

This function adds a work procedure. A work procedure is a function to be called repeatedly while the application is waiting for an event. It returns a **work procedure ID** that can be used by *GlgRemoveWorkProc*.

The following code is the prototype for a work procedure function:

```
typedef GlgBoolean (*GlgWorkProc)( client_data )
   GlgAnyType client_data;
```

The *client_data* argument provides user-defined data to the work procedure. It is defined by the call to *GlgAddWorkProc*.

If a work procedure returns TRUE, it is removed and will not be called again. If it returns FALSE, it will be called continuously until the application calls the *GlgRemoveWorkProc* function. You can register several work procedures and they will be performed one at a time. Work procedures must return quickly to avoid response time delays.

GlgBell

Produces a bell sound in a platform-independent way.

```
void GlgBell( viewport )
   GlgObject viewport;
```

Parameters

viewport

Specifies a viewport or a light viewport. This viewport object must be displayed; it specifies the display at which to produce the bell. On Windows, this parameter is ignored and may be NULL.

GlgConfigureWindow

Sets size and/or position of a viewport's window.

```
void GlgConfigureWindow( viewport, x, y, width, height, mask,
                         parent_viewport )
   GlgObject viewport;
   GlgLong x, y;
   GlgLong width, height;
   GlgConfigureMask mask;
   GlgObject parent_viewport
```

Parameters

viewport

Specifies the viewport to be configured.

x, y

Specify the x and y position in screen coordinates.

width, height

Specify the width and height in screen coordinates.

mask

Specifies bitwise flags that control the action to perform: set size, set position or both:

`GLG_POSITION_MASK`
: Requests to set window position using the *x* and *y* parameters and ignore the position defined by the viewport's control points. For top-level viewports, the *x* and *y* parameters specify position in screen coordinates relative to the origin of the screen. For child viewports and embedded dialog viewports, position parameters specify position relative to the origin of their parent viewport's window.

`GLG_SIZE_MASK`
: Requests to set window size using the *width* and *height* parameters and ignore the width and height defined by the viewport's control points.

`GLG_POSITION_AND_SIZE_MASK`
: Requests to set both the size and position, is equivalent to:

```
GLG_POSITION_MASK | GLG_SIZE_MASK
```

`GLG_RESET_POSITION_MASK`
: Resets the position request to restore the position defined by the viewport's control points.

`GLG_RESET_SIZE_MASK`
: Resets the size request to restore the width and height defined by the viewport's control points.

The set and reset flags are mutually exclusive and can't be used together for the same size or position attribute.

parent_viewport

Specifies an optional parent viewport for dialog viewports displayed via the *GlgDialogSetupHierarchy* or *GlgDialogInitialDraw* functions, or NULL otherwise.

Once the viewport's size or position is set using *GlgConfigureWindow*, the viewport's control points are no longer used for controlling the viewport's size and/or position. If only the viewport's position is set and not its size, the size will be still controlled by the viewport's control points, and vice versa if only the size is set and not the position. The *mask* parameter of the function contains options to restore the use of the control points for determining the viewport's size or position after it was set with *GlgConfigureWindow.*

GlgDialogInitialDraw

Display a GLG drawing as a dialog for the first time after it has been created or loaded:

```
void GlgDialogInitialDraw( viewport, parent_viewport )
   GlgObject viewport;
   GlgObject parent_viewport;
```

Parameters

viewport

Specifies the viewport containing the dialog’s drawing.

parent_viewport

Specifies the viewport of the main drawing the dialog will be a child of.

This is similar to the *GlgInitialDraw* function described on page 61, with the only difference that the drawing will be displayed as a child dialog of the main drawing specified by the *parent_viewport* parameter.

The *GlgConfigureWindow* function described on page 57 may be used to set the dialog's size and position.

GlgDialogSetupHierarchy

Provides an explicit request to set up the object hierarchy of a drawing displayed in a dialog window.

```
void GlgDialogSetupHierarchy( viewport, parent_viewport )
   GlgObject viewport;
   GlgObject parent_viewport;
```

Parameters

viewport

Specifies the viewport containing the dialog’s drawing.

parent_viewport

Specifies the viewport of the main drawing the dialog will be a child of.

This is similar to the *GlgSetupHierarchy* method described on page 64, with the only difference that the drawing will be set up as a child dialog of the main drawing specified by the *parent_viewport* parameter.

This method needs to be invoked only on the initial draw of the dialog. After the dialog has been set up, a generic *GlgSetupHierarchy* method may be used for all objects, including the dialog.

The *GlgConfigureWindow* function described on page 57 may be used to set the dialog's size and position.

GlgGetURLCB

Fetches data from a URL and saves it in a temporary file.

```
char * GlgGetURLCB( request, reserved )
   char * request;
   void * reserved;
```

Parameters

request

The URL string to fetch the data from, or NULL to delete the previously fetched temporary file.

reserved

This parameter is reserved for future implementations and must be NULL.

By default, this function is used for fetching URLs by the GLG editors on all platforms, as well as by the GLG C API runtime on Windows. *GlgSetGetURLCallback* described on page 65 may be used to activate the default callback for the GLG C API runtime on Linux:

```
GlgSetGetURLCallback( GlgGetURLCB );
```

To use it on Linux, the *GLG_WGET_PATH* environment variable should be set and contain the path to the *wget* command that will used for fetching URLs.

When *GlgGetURLCB* is invoked to fetch the next URL, it deletes a temporary file containing the previously fetched URL.

GlgInit

Initializes the GLG Toolkit.

```
GlgAppContext GlgInit( tk_initialized, app_context, argc, argv )
   GlgBoolean tk_initialized;
   GlgAppContext app_context;
   int argc;
   char ** argv;
```

Parameters

tk_initialized

Specifies whether or not X Toolkit was already initialized by the program that uses the legacy X11 version of the GLG library. It allows using the GLG Toolkit in an application that creates its own application context. If this parameter is False, the X Toolkit will be initialized by the function. Otherwise, it is assumed that the program has already initialized it.

This parameter is used only in the legacy X11 version of the GLG library and must be *False* for the GTK version of the GLG Library on Linux and on Windows.

app_context

With the **legacy X11 version** of the GLG library, this argument specifies the Xt application context if it has already been created by the program. If NULL is passed, an application context will be created by this function. Otherwise, the provided application context will be used.

With the **GTK version** of the GLG library, the argument is not used and must be NULL.

On Windows, the argument specifies an application instance handle, which is supplied by a parameter of the GlgMain or WinMain program entry points. In an MFC application, a call to *AfxGetInstanceHandle* function may be used to obtain application instance handle. If the GLG library is used in the form of a DLL, the application instance handle is obtained automatically and the value of NULL may be used for the argument. If a static GLG library is used, an application instance handle must be supplied explicitly.

argc

Specifies the number of command line parameters. On Windows, it will be provided by the *GlgMain* cross-platform entry point, otherwise zero may be used.

argv

Specifies the command line parameter list. On Windows, it will be provided by the *GlgMain* cross-platform entry point, otherwise zero may be used.

If the function creates an application context, it returns the created context, otherwise it returns the application context that was passed to it. The return value is later used with some other functions of the GLG Generic API.

On Linux/Unix, this function is automatically invoked if the GLG library is deployed using one of the native GLG widget wrappers.

On Windows, the dynamically linked library (DLL) version of the GLG library automatically calls this function as well. If the static version of the GLG library is used, the function has to be explicitly called by the application. In either case, it's always a good idea to explicitly call this function at the beginning of the application code, in which case it will also process command-line options recognized by the Toolkit and passed via the *argc* and *argv* parameters, such as:

`-verbose`
: generates extended diagnostic output. A verbose mode can also be set by setting the *GLG_VERBOSE* environment variable to *True*. The output is saved in the *glg_error.log* file. On Linux/Unix, the output is also displayed in the terminal.

`-glg-debug-opengl`
: generates extended diagnostic output for the OpenGL driver. The output can also be activated by setting the *GLG_DEBUG_OPENGL* environment variable to *True*. The output is saved in the *glg_error.log* file. On Linux/Unix, the output is also displayed in the terminal.

`-glg-disable-error-dialogs`

Disables error and warning message dialogs on Windows. Alternatively, the dialogs can also be disabled by setting the *GLG_DISABLE_ERROR_DIALOGS* environment variable to *True*. The messages will still be logged in the *glg_error.log* file.

`-glg-disable-opengl`
disables OpenGL driver in favor of the native windowing driver.

`-glg-enable-opengl`
enables OpenGL driver if present. The OpenGL driver will be used only for the viewports which have their *OpenGLHint* attribute set to *ON*.

The OpenGL driver may also be enabled or disabled by setting either the *GlgOpenGLMode* global configuration resource or the *GLG_OPENGL_MODE* environment variable to the following values:

0 - disable the OpenGL driver
1 - enable the OpenGL driver
-1 - don't change the default setting.

GlgInitLocale

Sets the program locale.

```
GlgBoolean GlgInitLocale( locale )
   char * locale;
```

Parameters

locale

Specifies program locale. The value of NULL may be passed to use the current system locale.

The function invokes *setlocale(LC_ALL, locale)* and performs any other platform-dependent locale initialization activity. On Linux/Unix platforms, this function must be used before the *GlgInit* function.

GlgInitialDraw

Draws a GLG viewport for the first time after it has been created or loaded.

```
void GlgInitialDraw( viewport )
   GlgObject viewport;
```

Parameters

viewport

Specifies a viewport containing a GLG drawing.

This function is used to render a viewport drawing after it was loaded or created. If the GLG Wrapper Widget is used on Linux/Unix, or the GLG Control is used on Windows, this function is called automatically, and need not be explicitly called.

GlgLoadWidgetFromFile

Loads a GLG widget from a file or URL.

```
GlgObject GlgLoadWidgetFromFile( filename )
   char * filename;
```

Parameters

filename

Specifies the name of the widget's drawing file or URL. For details of using URLs on Linux, refer to the description of the *GlgGetURLCB* function on page 59.

This function loads a drawing from a file or URL and searches the drawing for a viewport object named "$Widget". If the viewport is found, it references and returns a handle to it, otherwise it produces an error message and returns NULL.

After the viewport has been used, it may be dereferenced using the *GlgDropObject* function to avoid memory leaks. Refer to the description of the *GlgReferenceObject* function on page 136 for details on the reference counting.

GlgLoadWidgetFromImage

Loads a GLG widget from a memory image.

```
GlgObject GlgLoadWidgetFromImage( image_address, image_size )
   void * image_address;
   GlgLong image_size;
```

Parameters

image_address

Specifies the address of the widget's drawing image generated by the GLG Code Generation Utility *gcodegen*.

image_size

Specifies the image size. This is also generated by *gcodegen*.

This function loads a drawing from the drawing image and searches the drawing for a viewport named "$Widget". If the viewport is found, it references and returns it, otherwise it produces an error message and returns NULL. After the viewport has been used, it may be dereferenced using the *GlgDropObject* function.

GlgLoadWidgetFromObject

Loads a widget from a GLG object.

```
GlgObject GlgLoadWidgetFromObject( object )
   GlgObject object;
```

Parameters

object

Specifies a GLG object to be used as a widget's drawing.

This function searches the object to be used as a drawing for a viewport named "$Widget". If the viewport is found, it references and returns it, otherwise it produces an error message and returns NULL. This function could be used to load a widget created by using the GLG Extended API, or to load a sub-section of an already loaded drawing. After the viewport has been used, it may be dereferenced using the *GlgDropObject* function.

GlgMainLoop

Implements an event polling loop in a platform independent way.

```
GlgLong GlgMainLoop( app_context )
   GlgAppContext app_context;
```

Parameters

app_context

Specifies the application context returned by the *GlgInit* function.

On Windows, the return value of this function may be used as the return value of the *GlgMain* function.

GlgRand

Produces a random number in a platform-independent way.

```
double GlgRand( low, high )
   double low;
   double high;
```

Parameters

low, high

Specify the low and high range of the random numbers generated.

This function produces a random number in the specified range by using the platform's *rand* function.

GlgRemoveTimeOut

Removes a timer procedure.

```
void GlgRemoveTimeOut( timer_id )
   GlgLong timer_id;
```

Parameters

timer_id

Specifies the ID of the timer procedure to be removed. The ID was returned by *GlgAddTimeOut*.

This function removes an active timer. If the timer has already been removed and is not active, the results are undefined.

GlgRemoveWorkProc

Removes a work procedure.

```
void GlgRemoveWorkProc( workproc_id )
   GlgLong workproc_id;
```

Parameters

workproc_id

Specifies an id of the work procedure to be removed. The id was returned by *GlgAddWorkProc*.

This function removes an active work procedure. If the work procedure has already been removed and is not active, the results are undefined.

GlgResetHierarchy

Resets the object hierarchy.

```
void GlgResetHierarchy( object )
   GlgObject object;
```

Parameters

viewport

Specifies an object to be reset.

Resets the object hierarchy of a top-level viewport or drawing. If the object was displayed using the Generic API, this function must be called before destroying a top-level object with *GlgDropObject*. This function should not be called for viewports used in the GLG integrated containers, such as the GLG Wrapper Widget or GLG Custom Control.

Warning: Do not confuse this function with the *GlgReset* function of the standard API. *GlgReset* is used to redraw a displayed drawing and to regenerate the object hierarchy.

GlgSetupHierarchy

Provides an explicit request to set up the object hierarchy.

```
void GlgSetupHierarchy( object )
   GlgObject object;
```

Parameters

object

Specifies an object to be set up.

When invoked on a drawing which has been loaded but not yet displayed, the method sets up the drawing's object hierarchy to prepare the drawing for rendering. The drawing should contain either a top-level viewport object, or a group containing several top-level viewports.

After the initial draw (when the object hierarchy has already been set up), the method can be used to set up any type of object after its resources were changed. Unlike the *GlgUpdate* method, *GlgSetupHierarchy* sets up the object without repainting it.

GlgSetGetURLCallback

Installs a custom callback used to fetch data from URLs.

```
void GlgSetGetURLCallback( callback )
   GlgGetURLCallback callback;
```

Parameters

`callback`

Specifies a custom callback to use for fetching URLs.

The function returns the previously installed callback, if any, otherwise NULL.

The function prototype for the new handler is as follows:

```
char * get_url_callback( request, reserved )
   char * request;
   void * reserved;
```

The *request* parameter contains the URL string to fetch data from, or NULL if the callback is invoked to delete the last fetched temporary file. The *reserved* parameter is reserved for future implementations. The callback should return the temporary file name containing the fetched data or NULL if fetching failed. It should also return NULL if the callback is invoked to delete the previously fetched temporary file.

A custom callback implementation on Linux should delete a temporary file containing the previously fetched URL when it is invoked to fetch the next URL. When *GlgTerminate* is invoked, the callback will be invoked with NULL request to clean up the last used temporary file.

URLs may be used to load drawing files from remote servers, as well as to use map server installed on a remote web server.

A default *GlgGetURLCB* callback described on page 59 is provided as part of the GLG API library and is activated by default for the GLG editors on all platforms, as well as for the GLG C API runtime on Windows. *GlgSetGetURLCallback* may be used activate the default callback for the GLG C API runtime on Linux:

```
GlgSetGetURLCallback( GlgGetURLCB );
```

To use the default callback on Linux, the *GLG_WGET_PATH* environment variable should be set and contain the path to the *wget* command that will be used for fetching URLs.

GlgSleep

Suspends application execution for a specified interval in a platform independent way.

```
void GlgSleep( sleep_interval )
   GlgLong sleep_interval;
```

Parameters

sleep_interval

Specifies a sleep interval in milliseconds.

GlgTerminate

Terminates the application's use of the GLG Toolkit.

```
void GlgTerminate( void )
```

This function destroys all created GLG structures and frees allocated memory, flushing the internal memory pools. No GLG functions may be called after a call to *GlgTerminate*.

2.4 Animating a GLG Drawing with Data Using the Standard API

The GLG Application Program Interface (API) provides a way for a user's C program to animate a GLG drawing. The API consists of a set of functions for querying and changing the resources of a GLG drawing. As a convenience to the programmer, it also provides some functions for frequently needed functions, such as specialized string manipulation, etc.

Note that the drawing must be displayed before it can be animated. For information on displaying a GLG drawing from a program, see the *Displaying a GLG Drawing* chapter on page 31. Though the details of displaying a drawing are platform-dependent, the functions described in this chapter are not. The syntax of the API described in this chapter is the same on Linux/Unix and Windows.

There are three flavors of the GLG API:

- The **Standard API** enables an application to display a GLG drawing, animate it via the use of either resources or data tags, as well as handle user interaction.
- The **GLG Intermediate API** is a subset of the Extended API that provides all of the Extended API methods for manipulating the drawing, except for the methods for creating, adding or deleting objects from the drawing at run time. It is typically used for introspection (examining the content of a drawing), advanced input handling, as well as for obtaining object IDs for direct data updates, which increases update performance by shortcutting the resource name-based resource search.
- The **GLG Extended API** adds functionality that allows an application program to edit or create the drawing at run time.

The rest of this chapter describes functions of the GLG Standard API.

Overview

Once a GLG drawing is created and displayed, it is a simple matter to animate it. Animating a GLG drawing is done simply by manipulating its resources. This means that the only actions a program need take is to query or to set some resource value, and occasionally to redisplay the drawing with the new resource values.

The simplicity of operation means that the GLG API is quite a small one. It consists of the following functions:

- **GlgSetDResource**, **GlgSetDResourceIf** and **GlgSetDTag** set a scalar resource or tag. For information about the different attribute data types (geometrical, scalar, and string), see the *The Attribute Object* section in the *Structure of a GLG Drawing* chapter.
- **GlgSetGResource**, **GlgSetGResourceIf** and **GlgSetGTag** set a geometrical resource or tag.
- **GlgSetSResource**, **GlgSetSResourceIf** and **GlgSetSTag** set a string resource or tag.
- **GlgSetSResourceFromD**, **GlgSetSResourceFromDIf** and **GlgSetSTagFromD** set a string resource or tag from a scalar according to a chosen format.
- **GlgSetResourceFromObject** Sets a resource using another object.
- **GlgGetDResource** and **GlgGetDTag** return a scalar resource or tag.

- **GlgGetGResource** and **GlgGetGTag** return a geometrical resource or tag.
- **GlgGetSResource** and **GlgGetSTag** return a string resource or tag.
- **GlgHasResourceObject** checks if a named resource exists.
- **GlgHasTagName** and **GlgHasTagSource** check if a tag with a specified tag name or tag source exists.
- **GlgCreateTagList** returns a list of data tags defined in the drawing.
- **GlgCreateAlarmList** returns a list of alarms defined in the drawing.
- **GlgSetZoom** provides programmatic access to the integrated zooming and panning features.
- **GlgSetZoomMode** sets or resets the GIS or Chart zoom mode.
- **GlgUpdate** redisplays the drawing using the current resource values. All resource changes are stored until this function is called or until an Expose event (or paint event on Windows) is received.
- **GlgExportPostScript** saves a PostScript image of the widget's current state into a file.
- **GlgNativePrint** performs native printing (Windows and GTK version on Linux only).
- **GlgSaveImage** and **GlgSaveImageCustom** save an image of the drawing in the JPEG or PNG format.
- **GlgGenerateImage** and **GlgGenerateImageCustom** create an image of the drawing and return it as a pixmap on Linux/Unix, or a bitmap on Windows.
- **GlgReset** reinitializes a widget drawing.
- **GlgSendMessage** sends a message to an object to request some action to be performed.
- **GlgSetAlarmHandler** defines a function that will process alarm messages.
- **GlgSetTooltipFormatter** sets a custom tooltip formatter.

The GLG Standard API also includes the following methods described the *GLG Intermediate and Extended API* chapter:

- **GlgReferenceObject** increments the reference count of an object.
- **GlgDropObject** decrements the reference count of an object.
- **GlgGetElement** returns the object at the specified index in a container.
- **GlgGetSize** queries the size of a container object

The **GlgSetResourceObject** function of the Intermediate API may also be used with the Standard API for setting global configuration resources.

The **GlgAddCallback** function is used to add callbacks for handling user input in the form of input, selection and other events. While this function is a part of the GLG Standard API, it is described in detail in the *Callback Events* chapter on page 115.

GlgSetDefaultViewport is provided in the Windows version of the API and is used for setting a default drawing to be used by the GLG Wrapper Widget or GLG Custom Control.

The Standard API also includes functions specific to the **Real-Time Chart** and **GIS** objects. These functions are described in the *Real-Time Chart Functions* chapter on page 212 and the *GIS Object Functions* chapter on page 228.

The GLG Standard API also includes the following convenience and utility functions that simplify tasks common to GLG C programs.

- **GlgGetObjectName**, **GlgGetObjectType,** and **GlgGetDataType** provide convenient shortcuts for querying corresponding properties of an object.
- **GlgGetRefCount** returns an object's reference count.
- **GlgStrClone** creates a copy of a string.
- **GlgCreateIndexedName** creates an enumerated resource name.
- **GlgConcatResNames** creates composite resource names from components.
- **GlgConcatStrings** creates a new string from two input stings.
- **GlgAlloc** allocates memory using the Toolkit's memory allocator.
- **GlgFree** frees the memory allocated by *GlgAlloc* and GLG string utility functions.
- **GlgSetErrorHandler** replaces the GLG Toolkit error handler.
- **GlgGetSelectionButton** is used inside a callback procedure to query the mouse button that caused the callback to be invoked.
- **GlgGetModifierState** provides a cross-platform way to query the state of the *Control* and *Shift* keys, as well as the type of the mouse click: a single or double click.
- **GlgSetFocus** provides a cross-platform way to set the keyboard focus.
- **GlgExportStrings** exports all text strings defined in the drawing into a string translation file.
- **GlgImportStrings** imports translated text strings from a string translation file.
- **GlgExportTags** exports tag names and tag sources of all tags defined in a drawing into a file.
- **GlgImportTags** replaces tag names and tag sources in a drawing with the information imported from a tag translation file.
- **GlgChangeObject** sends a change message to the object without actually changing the object's properties, may be used to force the object to redraw.
- **GlgGetNativeComponent** returns an ID of a native windowing system component used to render the viewport.
- **GlgSetBrowserObject** defines the object whose resources will be displayed by the resource, tag and alarm browser widgets.
- **GlgSetBrowserSelection** sets a new value of a browser's selection and filter text boxes for the resource, tag, alarm and data browser widgets after the browser has been set up.
- **GlgSetEditMode** disables viewport's input objects for editing.
- **GlgSetLParameter** and **GlgGetLParameter** are used to set or query values of global parameters that control memory allocation at run time.
- **GlgPreAlloc** is used to preallocate memory under control of a mission-critical real-time appli-

cation.

- **GlgSetCursorType** sets the cursor displayed when the mouse enters the drawing or one of its child viewports.
- **GlgSetCursor** (Windows only) sets a custom cursor for a drawing or one of its child viewports.
- **GlgFindFile** may be used to find a file in the GLG search path.
- **GlgGetDir** returns the directory portion of the given file or directory path.
- **GlgGetRelativePath** creates a filepath relative to the specified path.
- **GlgCreateRelativePath** is an extended version of *GlgGetRelativePath* that provides additional options.
- **GlgGetExePath** retrieves the full path of the application's executable.
- **GlgError** displays error and information messages.
- **GlgXPrintDefaultError** prints information about an X error on a console or logs it into a file (legacy X11 version on Linux/Unix only).
- **GlgLoadExeIcon** loads small and big icons from the executable and associates them with the top level windows of a viewport (Windows only).
- **GlgControlWindowProc** passes messages to the Glg Control when low-level Windows API is used instead of the GLG Generic API to display a drawing (Windows only).
- **GlgGetMajorVersion** and **GlgGetMinorVersion** retrieve the library's version numbers.
- **GlgPrintObjectCounts** and **GlgPrintObjectCountChanges** may be used to diagnose memory and object leaks.

Note: Several resources in a GLG drawing may affect the actual object hierarchy of the drawing. The *Factor* attribute of a series object, for example, controls the number of objects that appear to be members of that series. When manipulating the resources of a drawing with the GLG API, it is useful to set the resources that affect the object hierarchy first, then call *GlgUpdate*, and then set the resources that only affect the value of objects in that hierarchy. Alternatively, *GlgSetupHierarchy* can be used instead of *GlgUpdate* to perform hierarchy setup without triggering paint events. For more information about this distinction between resources, see the *H and V Resources* section of the *Integrating GLG Drawings into a Program* chapter.

Using Resource and Tag Wildcards

Functions that set resource or tags values support * and ? wildcards for pattern matching. The asterisk (*) matches any sequence of characters, and the question mark (?) matches any single character.

To enable wildcard matching, prefix the resource or tag name with *"$!"*. For example, *"$!Clear**"* will target all tags with names beginning with *"Clear"*.

When using wildcards in resource paths, the wildcards prefix must be added to each level of resource hierarchy where pattern matching is required. For example, the following string targets all nested elements with names beginning with *"Element"* inside parent objects with names beginning with *"Group"*:

```
$!Group*/$!Element*/LineWidth
```

Note: Wildcards can only be used for matching named object resources; they cannot be used for matching default attribute names (e.g. *LineWidth*, *FillColor*).

Resource Access Optimization

If NULL is passed as the *resource_name* parameter of functions for setting and querying resources, the data object supplied by the *object* parameter will be used directly, avoiding an overhead of a resource search. This can be used to optimize performance when the same resource is accessed repeatedly. The *GlgGetResourceObject* function of the GLG Intermediate API may be used to obtain an object ID of a data object once and then use it repeatedly.

For example:

```
// Store a data object for repeated use.
GlgObject data_obj =
   GlgGetResourceObject( viewport, "WaterTank/Level" );
...
// Use the stored data object to update the drawing with data.
if( data_obj )
   GlgSetDResource( data_obj, NULL, new_water_level_value );
```

The *GlgAddDataSampleNode* function of the GLG Intermediate API can also be used instead of setting values of chart entry points to prefill a chart with a large number of data samples.

Function Descriptions

To use any of the functions described in this chapter, the application program must include the function definitions from the GLG API header file. Use the following line to include these definitions:

```
#include "GlgApi.h"
```

Some resources have enumerated values. For example, the *ScrollType* resource of a graph may have values GLG_SCROLLED or GLG_WRAPPED. The *GlgApi.h* file has defined constants to use for setting these resource values.

Note that virtually all of the GLG API library functions have the same first argument: a *GlgObject* parameter called *object*. This argument is a **handle** pointing to an object in the drawing. Usually, it indicates the viewport object that is the widget in the drawing. This handle is returned by the platform-specific display functions described in the *Using the C/C++ version of the Toolkit* chapter. The *object* parameter may also provide an object ID of a graphical object inside the viewport.

The following describes the interfaces of the GLG Standard Library functions.

GlgAlloc

Allocates memory using the Toolkit's memory allocator and returns a pointer to the allocated memory.

```
void * GlgAlloc( size )
   GlgLong size;
```

Parameters

size

Specifies the size of a memory block in bytes.

The memory must be freed with *GlgAlloc* when it is no longer used.

GlgChangeObject

Sends a change message to the object without actually changing the object's properties.

```
void GlgChangeObject( object, resource_name )
   GlgObject object;
   char * resource_name;
```

Parameters

object

Specifies the object to send the message to.

resource_name

If NULL, the message is sent to the object specified by the *object* parameter. Otherwise, the message is sent to the object's child specified by the resource name. The resource path is relative to the *object* parameter.

This function may be used to send a change message to an object's attribute. Sending the message to an object attribute is equivalent to setting the attribute to the same value as its current value using one of the *GlgSetDResource*, *GlgSetSResource* or *GlgSetGResource* functions.

For example, sending the message to the *Factor* attribute of a volatile series object forces the series to recreate its instances without actually changing the value of the *Factor*. If the change message is sent to the *ImageFile* attribute of an image object, the image will reload the image file (the image's *EnableCache* attribute should be set to NO to disable cache for images that need to be reloaded).

Sending a change message to a drawable object forces the object to redraw itself. If the message is sent to an object like a polygon or a text object, the area of the object's bounding box gets invalidated and redrawn. If an object is a viewport, it redraws its content.

GlgConcatResNames

Creates a composite resource name from two components.

```
char * GlgConcatResNames( resource_name1, resource_name2 )
   char * resource_name1, * resource_name2;
```

Parameters

resource_name1

Specifies the first path component.

resource_name2

Specifies the second path component.

This function creates and returns a resource name formed by concatenating the two components with the / character between them. The function checks if a trailing / separator is already present in the first string, and adds it only if the separator is not present.

Either argument may be NULL, in which case the returned string will precisely equal the input value of the other argument. The string containing the returned resource name should be freed later with the *GlgFree* function.

For example, the following code fragment:

```
char * resource_name;

resource_name = GlgConcatResNames( "DataGroup", "DataSample2" );
printf( "resource_name = %s\n", resource_name );
GlgFree( resource_name );
```

produces the following output:

```
resource_name = DataGroup/DataSample2
```

Multiple calls to the *GlgConcatResNames* function may be used to create composite names from more than two components:

```
char
   * temp_name,
   * resource_name;

temp_name = GlgConcatResNames( "DataGroup", "DataSample2" );
resource_name = GlgConcatResNames( temp_name, "Value" );
GlgFree( temp_name )
printf( "resource_name = %s\n", resource_name );
GlgFree( resource_name );
```

produces the following output:

```
resource_name = DataGroup/DataSample2/Value
```

GlgConcatStrings

Concatenates two character strings.

```
char * GlgConcatStrings( string1, string2 )
   char * string1, string2;
```

Parameters

string1, string2
The strings to be concatenated.

This function creates a new string from two input strings. Either input string may be NULL, in which case the returned string is a copy of the non-NULL input string. If both strings are NULL, then NULL is returned. The output of this function must be freed with the *GlgFree* function. That function also understands null data, so the following code may be used without needing to check for NULL values:

```
string3 = GlgConcatStrings( string1, string2 );
   ...
   GlgFree( string3 );
```

GlgControlWindowProc (Windows only*)*

Passes messages to the GLG Control when a low-level Windows API is used to display a drawing instead of the GLG Generic API on Windows:

```
GlgLong CALLBACK GlgControlWindowProc( HWND hwnd, UINT msg,
            WPARAM wParam, LPARAM lParam )
```

Parameters

hwnd
The window ID of the GLG Control.

msg
The message to be passed.

wParam, lParam
The message parameters.

This method is used by the window procedure of the control's parent window to pass messages to the control. It is used only if low-level Windows API is used to display a GLG drawing on Windows. The GLG Generic API provides a platform-independent way to display GLG drawings on any platform without resorting to using low-level platform-specific API. The *examples_c_cpp\animation\GlgAnimationW.c* file in the GLG installation directory on Windows provides an example of using this function.

GlgCreateIndexedName

Creates a string with a name for an enumerated resource.

```
char * GlgCreateIndexedName( template_name, resource_index )
   char * template_name;
   GlgLong resource_index;
```

Parameters

template_name

A character string containing the template name to be expanded. This name uses the **%** character to define the expansion position.

resource_index

Specifies the number to use for expanding the **%** character in the template name.

This function creates and returns a resource name by replacing the first (leftmost) occurrence of the **%** expansion character within the template name with the number corresponding to the *resource_index* parameter. If the template name does not contain the expansion character, the number is added to the end of the name. The returned expanded name should later be freed with the *GlgFree* function.

Example

The following code fragment:

```
char * name;
int i;
for( i=0; i<3; ++i )
{
   name = GlgCreateIndexedName( "XLabelGroup/XLabel%/String", i );
   printf( "Name: %s\n", name );
   GlgFree( name );
}
```

produces the following output:

```
Name: XLabelGroup/XLabel0/String
Name: XLabelGroup/XLabel1/String
Name: XLabelGroup/XLabel2/String
```

The *GlgCreateIndexedName* function may be used repeatedly if a template name has more than one expansion character. For example, the following code fragment:

```
char * name1, * name2;
name1 = GlgCreateIndexedName( "Chart%/Plots/Plot#%/Value", 4 );
name2 = GlgCreateIndexedName( name1, 3);
GlgFree( name1 );
printf( "Name: %s\n", name2 );
GlgFree( name2 );
```

produces the following output:

```
Name: Chart4/Plots/Plot#3/Value
```

GlgCreateTagList

Creates and returns a list of data tags defined in the drawing.

```
GlgObject GlgCreateTagList( object, unique_tag_sources )
   GlgObject object;
   GlgBoolean unique_tags;
```

Parameters

object

An object whose tags are queried. The top level viewport may be used to query the list of tags of the whole drawing.

unique_tag_sources

If set to *GlgTrue*, only the first encountered tag of the highest priority will be added to the list for tags with the same tag source. Input tags are prioritized over output tags, and enabled tags over disabled tags. If the parameter value is set to *GlgFalse*, all instances with the same tag source will be reported.

This function creates and returns a group containing all named tag objects, or NULL if no tags were found. Each element of the group is an attribute object that has a tag attached. The *GlgGetElement* and *GlgGetSize* functions may be used within the Standard API to handle the returned group object. The returned group object must be dereferenced using the *GlgDropObject* function when it is no longer needed.

If *object* is not a viewport, the returned list of tags will include only the tags contained inside the object. For a viewport, the tag list will contain all tags defined in the viewport's drawing.

Example

The following code prints information about all tags defined in the drawing:

```
char * tag_name, * tag_source;
int i, size;
GlgObject tag_list, tag_object;

tag_list = GlgCreateTagList( viewport, GlgFalse );
if( tag_list )
{
   size = GlgGetSize( tag_list );
   for( i=0; i<size; ++i )
   {
      tag_object = GlgGetElement( group, i );
      GlgGetSResource( tag_object, "TagName", &tag_name );
      GlgGetSResource( tag_object, "TagSource", &tag_source );
      printf( "TagName: %s, TagSource: %s\n", tag_name, tag_source );
   }
   GlgGropObject( tag_list );
}
```

GlgCreateAlarmList

Creates and returns a list of alarms defined in the drawing.

```
GlgObject GlgCreateAlarmList( object )
   GlgObject object;
```

Parameters

object

An object whose alarms are queried. The top level viewport may be used to query a list of alarms of the whole drawing.

This function creates and returns a group containing all alarm objects, or NULL if no alarm objects were found. Each element of the group is a data object that has an alarm attached. The *GlgGetElement* and *GlgGetSize* functions may be used within the Standard API to handle the returned group object. The returned group object must be dereferenced using *GlgDropObject* when it is no longer needed.

If *object* is not a viewport, the returned alarm list will include only alarms contained inside the object. For a viewport, the alarm list will contain all alarms defined in the viewport's drawing.

Example

The following code prints information about all alarms defined in the drawing:

```
char * alarm_label;
int i, size;
GlgObject alarm_list, data_object;

alarm_list = GlgCreateAlarmList( viewport );
if( alarm_list )
{
   size = GlgGetSize( alarm_list );
   for( i=0; i<size; ++i )
   {
      /* The data object the alarm is attached to. */
      data_object = GlgGetElement( group, i );

      /* AlarmLabel may be unnamed, extract it as the XformAttr1
         resource of the alarm object. */
      GlgGetSResource( data_object,
                  "Alarm/XformAttr1", &alarm_label );
      printf( "AlarmLabel: %s\n", alarm_label );
   }
   GlgGropObject( alarm_list );
}
```

GlgError

Displays error and information messages using the currently installed error handler.

```
void GlgError( error_type, message )
   GlgErrorType error_type;
   char * message;
```

Parameters

error_type

Specifies the type of the message, may have the values listed below. The list includes a description of the action performed by the default error handler for each message type:

`GLG_INTERNAL_ERROR`

reserved for the Toolkit's internal use; generates an error message, emits an audio beep and logs the error into the *glg_error.log* file

`GLG_USER_ERROR`

generates an error message and emits an audio beep; on Windows, also logs the error into the *glg_error.log* file

`GLG_WARNING`

generates a warning message and emits an audio beep; on Windows, also logs the warning into the *glg_error.log* file

`GLG_INFO`

on Windows, logs the message into the *glg_error.log* file; on Linux/Unix, prints the message on the terminal.

The location of the *glg_error.log* file is determined by the GLG_LOG_DIR and GLG_DIR environment variables as described in the *Error Processing and Debugging* section on page 50. In the Linux/Unix environment, all messages are printed in the program standard error output regardless of their message types.

message

The message to be displayed.

GlgExportPostScript

Generates a PostScript output of the current state of the viewport's graphics.

```
GlgBoolean GlgExportPostScript( object, file, xpos, ypos, width,
                                height, portrait, stretch, center )
   GlgObject object;
   char * file;
   double xpos, ypos, width, height;
   GlgBoolean portrait, stretch, center;
```

Parameters

object

Specifies a viewport. If *object* is a light viewport, the output will be generated for its parent viewport.

file

Specifies a filename in which to save the generated PostScript output.

xpos, ypos, width, height

Specify the rectangle on the page that will contain the Postscript output: *xpos* and *ypos* define the coordinates of the upper left corner of the rectangle, *width* and *height* define its dimensions. All coordinates are specified in the world coordinate system, which maps a rectangle defined by the points (0;0) and (2000;2000) to a page.

portrait

Specifies the orientation of the PostScript output on the page. If the value of this parameter is *GlgTrue*, a portrait orientation is used. Otherwise, a landscape orientation is used.

stretch

Specifies whether or not PostScript output should be stretched when mapping it to the specified rectangle. If the value of this parameter is *GlgTrue*, the PostScript output will be stretched to fit the rectangle exactly. This may distort printed objects. If the value is *GlgFalse*,

the ratio of height to width of the widget will be preserved in the PostScript output. This may leave some parts of the specified rectangle blank.

center

Specifies whether or not PostScript output should be centered inside the specified output rectangle when *stretch=GlgFalse* results in unused space on the sides of the rectangle. If the value of this parameter is *GlgTrue*, the output will be centered inside the output area, otherwise it will be anchored in the upper left corner of the rectangle.

The drawing's hierarchy must be set up in order to generate PostScript output. Use the *GlgUpdate* function before calling *GlgExportPostScript* to make sure the drawing is up to date. The function returns *GlgTrue* if PostScript output is successfully written into the file.

GlgExportStrings

Writes all text strings defined in the drawing into a string translation file:

```
GlgLong GlgExportStrings( object, filename, separator1, separator2 )
   GlgObject object;
   char * filename;
   GlgLong separator1;
   GlgLong separator2;
```

Parameters

object

Specifies a viewport or drawing object whose text strings to export.

filename

Specifies the string translation file to write.

separator1, separator2

Defines characters to be used as string separators in the generated file.

The function exports strings of the S (string) resource objects, such as the *String* attribute of a text object. It does not export strings that are not values of S resource objects, such as object names, tag names and tag sources, which ensures that the program logic is not affected when the translated file is used by *GlgImportStrings* to modify text strings in the drawing.

The function returns the number of exported strings or -1 in case of an error. Refer to the *Localization Support* chapter on page 255 of the *GLG User's Guide and Builder Reference Manual* for information about the string translation file format.

GlgExportTags

Writes tag names and tag sources of all tags defined in a drawing into a file:

```
GlgLong GlgExportTags( object, filename, separator1, separator2 )
   GlgObject object;
   char * filename;
   GlgLong separator1;
   GlgLong separator2;
```

Parameters

object

Specifies a viewport or drawing object whose tags to export.

filename

Specifies the tag file to write.

separator1, separator2

Defines characters to be used as string separators in the generated file.

Each line of the generated file contains current values of a tag's *TagName* and *TagSource*, as well as their new values. The file can be edited to modify new values, and then imported into the drawing using *GlgImportTags*.

This makes it possible to use the drawing as a template, and then modify it on the fly to use a different set of tags. This technique may be used to create a single drawing, and then use it at run time to display different sections of a plant or different rooms of a data center. Tag remapping may also be done programmatically as shown in the GLG demos.

It can also be used to localize *TagNames* by translating them to different languages.

The function uses the same file format as the *GlgExportStrings* function. Refer to the *Localization Support* chapter on page 255 of the *GLG User's Guide and Builder Reference Manual* for information about the file format.

The function returns the number of exported tags or -1 in case of an error.

GlgFindFile

Searches for a file, first in the current directory, then in the directory provided by the *path* parameter and then in the GLG search path.

If the file was found, the function returns its file path, otherwise it returns a copy of the *filename* parameter. The returned file path can be passed to *fopen* to open the file.

```
char * GlgFindFile( filename, path, check_glg_path )
   char * filename;
   char * path;
   GlgBoolean check_glg_path;
```

Parameters

filename

Specifies a relative filename to search for.

path

If not NULL, specifies an additional search path to be searched before the GLG search path.

check_glg_path

If set to *GlgTrue*, the GLG search path will be included in the search.

The method returns an allocated string that has to be freed with *GlgFree*.

GlgFree

Frees memory used to store character strings or allocated using *GlgAlloc*.

```
void GlgFree( pointer )
   char * pointer;
```

Parameters

pointer

Specifies a memory pointer to be freed.

This function is used to free memory allocated by *GlgAlloc* and GLG string utilities. This is not to be used to free the memory occupied by any GLG object (*GlgObject*). Use *GlgDropObject* and *GlgReferenceObject* to manage these objects. The function checks for NULL address pointers, and will not fail or generate an error message if a NULL pointer is passed.

GlgGetDataType

Returns the data type (GLG_D, GLG_S or GLG_G) of a data or attribute object.

```
GlgDataType GlgGetDataType( data_object )
   GlgObject data_object;
```

Parameters

data_object

Specifies a data or attribute object to query.

GlgGetDir

Returns the directory portion of the specified path.

```
char * GlgGetDir( path )
   char * path;
```

Parameters

path

Specifies a file or directory path.

The function returns a copy of the *path* string with the filename component removed from the path. The filename is the last component after the last file separator (a forward slash on Linux/Unix, either a forward or backward slash on Windows). A trailing file separator is not removed from the returned string.

If NULL is passed as the *path* parameter, NULL will be returned. An allocated string representing a current directory (*"./"* or *".\"* on Windows) will be returned if an empty string is passed as *path*.

A copy of the path string will be returned without changing it (except for an added trailing file separator, if needed) if:

- the *path* ends with the file separator, in which case it is considered a directory path
- the *path* points to a real directory on the file system
- the *path* represents the current directory (*".."*) or the parent directory (*".."*), with or without a trailing file separator.

The method returns an allocated string that has to be freed with *GlgFree*.

GlgGetExePath

Returns the path of the executable.

```
char * GlgGetExePath()
```

This function provides a cross-platform way to retrieve the executable's path across both Windows and Linux/Unix systems.

The function returns an allocated string that has to be freed with *GlgFree*.

Note: For Linux/Unix platforms, passing *argc* and *argv* to *GlgInit* (or to the *GlgSessionC* constructor in C++) is required to identify the path of the executable.

GlgGetRelativePath

Creates a filepath relative to the specified base path.

```
char * GlgGetRelativePath( base_path, filename )
   char * base_path;
   char * filename;
```

Parameters

base_path

Specifies a base path. If NULL, the path of the program's executable will be used.

Note: For Linux/Unix platforms, passing *argc* and *argv* to *GlgInit* (or to the *GlgSessionC* constructor in C++) is required to identify the path of the executable when *base_path* is NULL.

filename

Specifies a filename or a relative filepath. If an absolute path is specified, it will be returned as is.

This function returns an allocated string that has to be freed with *GlgFree*.

If *filename* specifies an absolute path, it will be returned as is. Otherwise, *GlgGetRelativePath* extracts the directory portion of *base_path* using *GlgGetDir*, appends *filename* and returns the resulting string. If *base_path* is a directory path that does not point to an existing directory, it must end with *'/'*, or either *'/'* or *'\'* on Windows.

GlgCreateRelativePath

An extended version of *GlgGetRelativePath* that checks if the file exists and provides additional options for handling non-existent files.

```
char * GlgCreateRelativePath( path, filename,
                              return_null_if_not_found, check_if_exists )
   char * path;
   char * filename;
   GlgBoolean return_null_if_not_found;
   GlgBoolean check_if_exists;
```

Parameters

path

Specifies a base path. If NULL, the path of the program's executable will be used.

Note: For Linux/Unix platforms, passing *argc* and *argv* to *GlgInit* (or to the *GlgSessionC* constructor in C++) is required to identify the path of the executable when *base_path* is NULL.

filename

Specifies a filename or a relative filepath. If an absolute path is specified, it will be used as is.

return_null_if_not_found

Controls the return value in case when *check_if_exists=GlgTrue* and the file or directory pointed to by the created relative path does not exist. If set to *GlgTrue*, NULL is returned, otherwise a copy of the *filename* string is returned.

check_if_exists

Specifies if the function should check if the file or directory pointed to by the created relative path exists.

The method returns an allocated string that has to be freed with *GlgFree*.

If *filename* specifies an absolute path, it will be used as is. Otherwise, *GlgCreateRelativePath* extracts the directory portion of *base_path* using *GlgGetDir* and appends *filename*. If *base_path* is a directory path that does not point to an existing directory, it must end with *'/'*, or either *'/'* or *'\'* on Windows.

If *check_if_exists=GlgTrue*, the function checks if the created file path points to an existing file and handles it depending on the setting of the *return_null_if_not_found* parameter. If *check_if_exists=GlgFalse*, the created file path is returned without any checking.

GlgGetDResource
GlgGetDTag

Returns the current value of a scalar resource or tag.

```
GlgBoolean GlgGetDResource( object, resource_name, d_value_ptr )
   GlgObject object;
   char * resource_name;
   double * d_value_ptr;

GlgBoolean GlgGetDTag( object, tag_source, d_value_ptr )
   GlgObject object;
   char * tag_source;
   double * d_value_ptr;
```

Parameters

object

Specifies a GLG object.

resource_name, tag_source

Specifies the name of the double resource or tag to query. See the *Resource Access Optimization* section for additional information.

d_value_ptr

Specifies a pointer with which to return the resource or tag value.

If a scalar resource or tag with the input name exists, this function copies its current value into the address specified by *d_value_ptr* and returns *GlgTrue*, otherwise it generates an error message and returns *GlgFalse*.

GlgGetGResource
GlgGetGTag

Returns the set of three values making up a geometrical resource or tag:

```
GlgBoolean GlgGetGResource( object, resource_name, g_value1_ptr,
            g_value2_ptr, g_value3_ptr )
   GlgObject object;
   char * resource_name;
   double * g_value1_ptr, * g_value2_ptr, * gvalue3_ptr;

GlgBoolean GlgGetGTag( object, tag_source, g_value1_ptr, g_value2_ptr,
            g_value3_ptr )
   GlgObject object;
   char * tag_source;
   double * g_value1_ptr, * g_value2_ptr, * gvalue3_ptr;
```

Parameters

object

Specifies a GLG object.

resource_name, tag_source

Specifies the name of the geometrical resource or tag to query. See the *Resource Access Optimization* section for additional information.

g_value0_ptr, g_value1_ptr, g_value2_ptr

Specify pointers to the locations to which the XYZ or RGB values of the resource or tag are returned.

If a geometrical resource or tag with the input name exists, this function copies its current three values to the addresses specified with the *g_value* pointers and returns *GlgTrue*. Otherwise it generates an error message and returns *GlgFalse*.

For a geometrical point, *g_value0_ptr*, *g_value1_ptr* and *g_value2_ptr* are set to the X, Y and Z coordinates of the point, respectively. For a color resource or tag, they will be set to the R, G and B values of the color, respectively.

GlgGetLParameter

Queries a value of a global parameter.

```
GlgBoolean GlgGetLParameter( char * name, GlgLong * value )
   char * name;
   GlgLong value;
```

Parameters

name

Specifies parameter name.

value

A pointer used to return a parameter's value.

Refer to the *Appendix D: Global Parameters* section on page 416 for information on available global parameters. The function returns *GlgTrue* on success.

GlgGetMajorVersion
GlgGetMinorVersion

Returns library's major and minor version numbers.

```
GlgLong GlgGetMajorVersion( void )

GlgLong GlgGetMinorVersion( void )
```

GlgGetModifierState

Returns the state of the requested modifier.

```
GlgBoolean GlgGetModifierState( modifier )
   GlgModifierType modifier;
```

Parameters

modifier

Specifies a modifier to query: GLG_SHIFT_MOD, GLG_CONTROL_MOD or GLG_DOUBLE_CLICK_MOD.

GlgGetNativeComponent

Returns an ID of a native windowing system component used to render the viewport:

```
GlgAnyType GlgGetNativeComponent( object, resource_name, type )
   GlgObject object;
   char * resource_name;
   GlgComponentQueryType type;
```

Parameters

object

Specifies a viewport or a viewport's parent.

resource_name

Specifies a resource name for accessing the child viewport inside its parent. Use NULL when the *object* parameter specifies the viewport.

type

Specifies the type of a component to retrieve:

`GLG_WIDGET_QUERY`

Returns an ID of the native component used to render the viewport:

- a window handle on Windows
- when using the GTK version of the GLG library on Linux, including native GTK integration, returns a GTK container widget that holds the content of the viewport
- a Motif widget ID when using the legacy X11 version of the GLG library on Linux/Unix
- an X11 window ID when using the *libglg_x11.a library* for the Qt integration on Linux/Unix. This window ID will be NULL if a viewport uses a native Qt widget, such as a text box or a list (see the *Native Widgets* section on page 288 of the *GLG User's Guide and Builder Reference Manual*).

`GLG_SHELL_QUERY`

Returns an ID of the viewport's top-level parent:

- a window handle of the top-level window on Windows
- a top level GTK window when using the GTK version of the GLG library on Linux
- a top level Motif shell ID the viewport belongs to when using the legacy X11 version of the GLG library on Linux/Unix
- in the Qt and X11 GTK integration that uses the *libglg_x11.a* library on Linux/Unix, it returns an X11 window ID of the top parent's Qt widget, or an X11 window ID of the first encountered parent viewport with *ShellType* set to either GLG_DIALOG_SHELL or GLG_APPLICATION_SHELL.

GLG_CHILD_WIDGET_QUERY

In the GTK version of the GLG library on Linux, returns the widget ID of a native input widget, such as a text or combo box, or NULL if the viewport doesn't use native widgets.

In the legacy X11 version of the GLG library on Linux/Unix, returns a widget ID of a child Motif widget inside a scroll pane for a native widget that uses a scroll pane, such as a text edit or list widget. For these widgets, GLG_WIDGET_QUERY returns the widget ID of the scroll pane, and GLG_CHILD_WIDGET_QUERY returns the list or text widget itself.

GLG_V_SCROLLBAR_QUERY

GLG_H_SCROLLBAR_QUERY

In the legacy X11 version of the GLG library on Linux/Unix, returns a Motif widget ID of a requested scrollbar widget for native widgets that use a scroll pane.

GLG_DISPLAY_QUERY

In the X11 versions of the GLG library, returns the X11 display pointer for the viewport's window.

GLG_WINDOW_QUERY

- In the X11 versions of the GLG library, returns an X11 window ID of the viewport's window.
  In Qt integration that uses the *libglg_x11.a* library on Linux/Unix, this window ID will be NULL for a viewport that uses a native Qt widget, such as a text box or a list. For these viewports, GLG_NB_WIDGET_QUERY may be used to get the Qt widget used by the viewport.
- on Windows, returns a viewport's window handle (same as GLG_WIDGET_QUERY).

GLG_NB_WIDGET_QUERY

In the Qt integration that uses the *libglg_x11.a* library on Linux/Unix, returns a Qt widget used by the viewport, or NULL if the viewport does not use a native Qt widget.

GLG_TOP_WIDGET_QUERY

In the Qt/GTK integration that uses the *libglg_x11.a* on Linux/Unix, returns Qt/GTK parent widget of the top viewport, such as *QGlgWidget* or *gtk_glg*.

The function returns NULL if the requested component doesn't exist in the current runtime environment or hasn't yet been created.

GlgGetObjectName

Returns an object's name. It returns a pointer to the internal string that should not be modified or freed.

```
char * GlgGetObjectName( object )
   GlgObject object;
```

Parameters

object

Specifies an object to query.

GlgGetObjectType

Returns an object's type (GLG_PLOYGON, GLG_TEXT, etc.).

```
GlgObjectType GlgGetObjectType( object )
   GlgObject object;
```

Parameters

object

Specifies an object to query.

GlgGetRefCount

Returns an object's reference count.

```
GlgLong GlgGetObjectType( object )
   GlgObject object;
```

Parameters

object

Specifies an object to query.

GlgGetSelectionButton

Queries the mouse button that initiated the callback.

```
GlgLong GlgGetSelectionButton( void )
```

This function returns 1, 2, or 3 to indicate which button caused the callback to be invoked. It may be used only inside callback functions that are activated on a mouse click, otherwise the return value is undefined.

GlgGetSResource
GlgGetSTag

Returns the value of a string resource or tag.

```
GlgBoolean GlgGetSResource( object, resource_name, s_value_ptr )
   GlgObject object;
   char * resource_name;
   char ** s_value_ptr;

GlgBoolean GlgGetSTag( object, tag_source, s_value_ptr )
   GlgObject object;
   char * tag_source;
   char ** s_value_ptr;
```

Parameters

object

Specifies a GLG object.

resource_name, tag_source

Specifies the name of the string resource or tag to query. See the *Resource Access Optimization* section for additional information.

s_value_ptr

A pointer with which to return the resource or tag value.

If a string resource or tag with the given name exists, this function copies the current resource or tag string into an internal buffer, sets the *s_value_ptr* to point to the buffer and returns *GlgTrue*. Otherwise it returns *GlgFalse*.

Warning: The returned pointer points to GLG internal data structures and should not be modified. The pointer is valid only immediately after a call to the *GlgGetSResource* function. To store the returned string, create a copy of it using the *GlgStrClone* function.

GlgHasResourceObject

Checks if a named resource exists:

```
GlgBoolean GlgHasResourceObject( object, resource_name )
   GlgObject object;
   char * resource_name;
```

Parameters

object

Specifies a GLG object whose resource to query.

resource_name

Specifies the name of the resource object to query.

If a resource object with the input name exists, this function returns *GlgTrue*, otherwise it returns *GlgFalse*.

GlgHasTagName
GlgHasTagSource

Check if a data tag with a specified tag name or tag source exists:

```
GlgBoolean GlgHasTagName( object, tag_name )
   GlgObject object;
   char * tag_name;

GlgBoolean GlgHasTagSource( object, tag_source )
   GlgObject object;
   char * tag_source;
```

Parameters

object

Specifies a GLG object whose tag to query.

tag_name, tag_source

Specifies the tag name or tag source to query, supporting wildcards: ? (matches any single character) and * (matches any sequence of characters).

If a data tag with the given tag name or tag source exists, this function returns *GlgTrue*, otherwise it returns *GlgFalse*.

GlgHasTag

Check if a data or export tag with a specified tag name exists:

```
GlgBoolean GlgHasTag( object, tag_name, tag_type_mask )
   GlgObject object;
   char * tag_name;
   GlgTagType tag_type_mask;
```

Parameters

object

Specifies a GLG object whose tag to query.

tag_name

Specifies the tag name to query, supporting wildcards: ? (matches any single character) and * (matches any sequence of characters).

tag_type_mask

Defines the type of tags to query:
- GLG_DATA_TAG
- GLG_EXPORT_TAG
- GLG_EXPORT_DYN_TAG.

For export tags, tag type constants may be combined with bitwise OR to query multiple subtypes of export tags.

If a tag with the given tag name exists, this function returns *GlgTrue*, otherwise it returns *GlgFalse*.

GlgImportStrings

Replaces text strings in the drawing with the strings loaded from a string translation file:

```
GlgLong GlgImportStrings( object, filename, verbose )
   GlgObject object;
   char * filename;
   GlgBoolean verbose;
```

Parameters

object

Specifies the viewport or drawing object whose text strings to replace.

filename

Specifies the string translation file to load.

verbose

If true, generates an error for each string that was not modified because it was missing a matching entry in the string translation file.

The function returns the number of imported strings or -1 in case of an error. Refer to the *Localization Support* chapter on page 255 of the *GLG User's Guide and Builder Reference Manual* for information about the string translation file format.

GlgImportTags

Replaces tag names and tag sources of tags in a drawing with information loaded from a tag remapping file:

```
GlgLong GlgImportTags( object, filename, verbose)
   GlgObject object;
   char * filename;
   GlgBoolean verbose;
```

Parameters

object

Specifies the viewport or drawing object whose tags to change.

filename

Specifies the tags remapping file to load.

verbose

If true, generates an error for each tag that was not modified because it was missing a matching entry in the tag remapping file.

Each line of the tag remapping file contains the current values of a tag's *TagName* and *TagSource*, as well as their new values. When the file is imported into the drawing, the tags get changed to use new values from the imported file.

This makes it possible to use the drawing as a template, and modify it on the fly to use a different set of tags. This technique may be used to create a single drawing, and then use it at run time to display different sections of a plant or different rooms of a data center. Tag remapping may also be done programmatically as shown in the GLG demos.

It can also be used to localize *TagNames* by translating them to different languages.

The function uses the same file format as the *GlgImportStrings* function. Refer to the *Localization Support* chapter on page 255 of the *GLG User's Guide and Builder Reference Manual* for information about the file format.

The function returns the number of imported tags or -1 in case of an error.

GlgLoadExeIcon (Windows only)

Loads small and big icons from the executable and associates them with the top level window of a viewport. The method should be invoked after the drawing hierarchy has been set up.

```
void GlgLoadExeIcon( viewport, icon_id )
   GlgObject viewport;
   GlgLong icon_id;
```

Parameters

viewport

Specifies the drawing viewport.

icon_id

Specifies an icon ID in the resource file.

GlgNativePrint

Invokes native printing of the corresponding platform. This function is available only on Windows and in the GTK version of the GLG library on Linux.

Windows version

```
GlgBoolean GlgNativePrint( viewport, print_dc, x,y, width, height,
                           stretch, center )
   GlgObject viewport;
   HDC print_dc;
   double xpos, ypos, width, height;
   GlgBoolean stretch, center;
```

GTK version on Linux

```
GlgBoolean GlgNativePrint( viewport, print_context, x,y, width,
                           height, stretch, center )
   GlgObject viewport;
   GtkPrintContext * print_context;
   double xpos, ypos, width, height;
   GlgBoolean stretch, center;
```

Parameters

viewport
Specifies the viewport whose contents are to be printed. If *viewport* is a light viewport, the output will be generated for its parent viewport.

print_dc (Windows)
The printer device context, acquired from a call to the *PrintDlg* function.

print_context (GTK version on Linux)
The printing context, acquired via the *gtk_print_operation_new* / *gtk_print_operation_run* GTK printing interface.

xpos, ypos, width, height, stretch, center
These arguments are used in the same way as the corresponding arguments to the *GlgExportPostScript* function, described on page 78.

Windows

Given the printer device context, this function prints the contents of a viewport and all its children viewports into that device context. It is the responsibility of the application to set the abort procedure, provide a cancel printing dialog, disable the application window during printing, and call the *StartDoc*, *StartPage*, *EndDoc*, and *EndPage* functions before and after calling this function. These procedures and the use of these functions is described in the Windows API programming reference

There is no orientation parameter with the Windows native printing. The page orientation is determined by the user through the print dialog.

The function can be used to add GLG printing output to a page which already had some output on it. In this case, that other output may set the printer device context parameters to values incompatible with the GLG output. Before calling this function, therefore, the device transformations must be reset to the state they were in right after calling the *PrintDlg* function.

GTK Version on Linux

Given the printer context, this function prints the content of a viewport and all its children viewports into that printing context.

GlgOnDrawMetafile

Provides MFC metafile output support (Windows only). Is used by the MFC container's OnDrawMetafile method to produce metafile for the GLG child window.

```
GlgBoolean GlgOnDrawMetafile( viewport, print_dc )
   GlgObject viewport;
   HDC print_dc;
```

Parameters

viewport

Specifies a viewport object. If *viewport* is a light viewport, the output will be generated for its parent viewport.

print_dc

Specifies the printing device context for metafile output.

Returns *GlgTrue* if metafile generation succeeds, otherwise returns *GlgFalse*.

GlgOnPrint

Provides MFC printing support (Windows only). Is used by the MFC container's OnPrint method to print GLG child window.

```
GlgBoolean GlgOnPrint( viewport, print_dc )
   GlgObject viewport;
   HDC print_dc;
```

Parameters

viewport

Specifies a viewport. If *viewport* is a light viewport, the output will be generated for its parent viewport.

print_dc

Specifies the printing device context.

Returns *GlgTrue* if printing succeeds, otherwise returns *GlgFalse*.

GlgPreAlloc

Preallocates memory under control of mission critical real-time applications:

```
GlgBoolean GlgPreAlloc( size, type )
   GlgLong size;
   GlgPreAllocType type;
```

Parameters

size

Specifies the size of the memory to allocate.

type

Specifies the allocation type. The following lists available types and the meaning of the corresponding *size* parameter:

- GLG_PREALLOC_MEMORY - allocates a specified number of spare memory blocks.
- GLG_PREALLOC_POLYGON_POINT_BUFFERS - allocates a buffer of the requested size to be used for rendering large polygons. It can be set to the maximum expected number of points in one polygon.
- GLG_PREALLOC_GRAPH_POINT_BUFFERS - allocates a buffer of the requested size used for rendering plots of real-time charts. It can be set to the maximum expected number of points in one plot.
- GLG_PREALLOC_TESS_BUFFERS - allocates a buffer of the requested size for tessellation

vertices. The buffer is used by the OpenGL driver for tessellating filled polygons; it is not used by the GDI driver.

Mission-critical applications can use this function to preallocate memory on start-up and then allocate more memory as needed outside of the application's critical sections. The *GlgGetLParameter* function can be used to query the amount of spare memory in the preallocated memory blocks.

If the buffers are not preallocated, they are allocated and automatically adjusted as needed.

The function returns *GlgTrue* on success.

GlgReset

Reinitializes the drawing by resetting the drawing hierarchy, then setting it up again and rendering the drawing.

```
GlgBoolean GlgReset( object )
   GlgObject object;
```

Parameters

object

Specifies a viewport object.

Calling *GlgReset* restores all volatile resource changes to the values the resources had when the viewport was first drawn. It also discards currently displayed chart data, unless the chart's *Persistent* attribute is set to *GlgTrue*. The function returns *GlgTrue* if the widget is successfully reinitialized; otherwise it returns *GlgFalse*.

The function should be invoked only for the top-level viewports, not the children viewports.

GlgSaveImage
GlgSaveImageCustom

Generates a JPEG or PNG image of the current state of the viewport's graphics and saves it into a file:

```
GlgBoolean GlgSaveImage( object, resource_name, filename, format )
   GlgObject object;
   char * resource_name;
   char * filename;
   GlgImageFormat format;

GlgBoolean GlgSaveImageCustom( object, resource_name, filename,
                               format, x, y, width, height, gap )
   GlgObject object;
   char * resource_name;
   char * filename;
   GlgImageFormat format;
   GlgLong x, y, width, height, gap;
```

Parameters

object

Specifies a viewport object.

resource_name

Specifies the resource name of a child viewport whose image to save, or NULL to save the image of the viewport specified by the *object* parameter. The resource name is relative to the viewport specified by the *object* parameter.

filename

Specifies a filename in which to save the generated image.

format

Specifies an image format, must be GLG_JPEG or GLG_PNG.

x, y, width, height

Specifies the area of the drawing for generating an image with *GlgSaveImageCustom*, which can be bigger or smaller than the visible area of the viewport. The x and y parameters define the offset in screen pixels relative to the origin of the viewport's window, which has screen coordinates of (0,0). The width and height parameters define the width and height of the generated image in screen pixels. These parameters may be obtained by querying the bounding box of either the whole drawing or of the area of the drawing for which the image needs to be generated.

If the width and height parameters are 0, the image of the whole drawing is generated using the drawing's bounding rectangle as the image generation extent. If the viewport is zoomed in, this area may be significantly bigger than the visible area of the viewport, and the image size may be quite large. The viewport's zoom factor defines the scaling factor for objects in the drawing when the image is saved.

gap

Specifies the padding space between the extent of the drawing and the border of the generated image in case when zero width and height parameters are specified for *GlgSaveImageCustom*.

For a light viewport, the output will be generated for its parent viewport.

The *GlgSaveImage* function saves an image of the visible part of the viewport, clipping out the parts of the drawing that extends outside of it. The *GlgSaveImageCustom* saves the whole drawing without such clipping (if width and height parameters are 0), or saves just the specified area of the drawing.

The drawing's hierarchy must be set up in order to generate an image. The function returns *GlgTrue* if the image was successfully generated.

GlgGenerateImage
GlgGenerateImageCustom

Creates an image of the current state of the viewport's graphics and returns it as a pixmap on Linux/Unix, or a bitmap on Windows. The *PIXMAP* is defined as *Pixmap* on Linux/Unix, and *HBITMAP* on Windows.

```
PIXMAP GlgGenerateImage( object, resource_name, image )
   GlgObject object;
   char * resource_name;
   PIXMAP image;

PIXMAP GlgGenerateImageCustom( object, resource_name, pixmap,
                               x, y, width, height, gap )
   GlgObject object;
   char * resource_name;
   PIXMAP image;
   GlgLong x, y, width, height, gap;
```

Parameters

object

Specifies a viewport object.

resource_name

Specifies the resource name of a child viewport whose image to generate, or NULL to generate the image of the viewport specified by the *object* parameter. The resource name is relative to the viewport specified by the *object* parameter.

image

If not NULL, specifies the pixmap or the bitmap to be used. This pixmap or bitmap will be filled and returned instead of creating and returning a new pixmap or bitmap.

x, y, width, height, gap

Specify parameters of a custom image. Refer to the *GlgSaveImageCustom* function described above for detailed information.

GlgSendMessage

Sends a message to an object to request some action to be performed.

```
GlgAnyType GlgSendMessage( object, resource_name, message,
                           param1, param2, param3, param4 )
   GlgObject object;
   char * resource_name;
   char * message;
   GlgAnyType param1, param2, param3, param4;
```

Refer to the *Input Objects* chapter on page 263 of the *GLG User's Guide and Builder Reference Manual* for a list of messages supported by each type of the available input handlers.

Parameters

object

Specifies an object to send the message to. In most cases, it is a viewport with a GLG input handler attached, such as a button or a slider.

resource_name

If the parameter is set to NULL, the message is sent to the object specified by the *object* parameter. Otherwise, the message is sent to the *object's* child specified by the resource name. The resource path is relative to the *object* parameter. For example, to send the message to a viewport's handler, use the viewport as the *object* parameter and "*Handler*" as the value of the *resource_name* parameter.

message

A string defining the type of the message.

param1, param2, param3, param4

Message parameters whose type depends on the message type.

The function serves as an escape mechanism for actions that can not be easily accomplished by setting or querying a single resource, such us adding items to a list widget or querying a state of a multiple-selection list. For messages that execute some query, the function returns the query result, otherwise it just executes the requested action.

Refer to the *Input Objects* chapter on page 263 of the *GLG User's Guide and Builder Reference Manual* for the description of the function's parameters and returned value for each of the GLG input handlers.

GlgSetAlarmHandler

Installs a global alarm handler function that will be invoked to process alarm messages.

```
void GlgSetAlarmHandler( alarm_handler )
   GlgAlarmHandler alarm_handler;
```

Parameters

alarm_handler

Specifies a function that will be invoked to process alarm messages generated by the application's drawings.

An alarm handler has the following function prototype:

```
typedef void (*GlgAlarmHandler)( data_object, alarm_object,
            alarm_label, action, subaction, reserved )
   GlgObject data_object;
   GlgObject alarm_object;
   char * alarm_label;
   char * action;
   char * subaction;
   GlgAnyType reserved;
```

The *data_object* parameter is the resource whose value has changed. The *alarm_object* parameter is the alarm attached to the *data_object* to monitor its value; it is the alarm that generated the alarm message. The *alarm_label* parameter supplies the *AlarmLabel* used to identify *alarm_object*. The *action* and *subaction* parameters provide details about the condition that caused the alarm; refer to the *Alarm Messages* section on page 218 of the *GLG User's Guide and Builder Reference Manual* for more information.

An alarm handler should not change object hierarchy by actions such as adding or deleting objects from the drawing or changing a factor of a series object.

GlgSetBrowserObject

Sets the object for browsing with the GLG resource, tag or alarm browser widgets.

```
void GlgSetBrowserObject( browser, object )
   GlgObject browser;
   GlgObject object;
```

Parameters

browser

Specifies a viewport of the browser widget.

object

Specifies an object to browse.

The browser will display the object's resources, tags or alarms.

GlgSetBrowserSelection

Sets a new value of a browser's selection and filter text boxes for resource, tag, alarm and data browsers after the browser object has been set up. Setting a new selection will display a new list of matching entries.

```
void GlgSetBrowserSelection( object, resource_name, selection,
            filter )
   GlgObject object;
   char * resource_name;
   char * selection;
   char * filter;
```

Parameters

object

Specifies a viewport of the browser widget or its parent.

resource_name

Specifies a resource path to access the browser from the parent supplied by the *object* parameter, or NULL if the *object* parameter supplies the browser's viewport.

selection

Specifies a new selection string.

filter

Specifies a new filter string.

The browser will display a new list of matching items. Before the browser's drawing hierarchy has been set up, the selection and filter may be set via corresponding resources. To change selection or filter after hierarchy setup, this function should be used.

GlgSetCursorType

Sets a cursor for a drawing or one of its child viewports.

```
GlgBoolean GlgSetCursorType( viewport, resource_name, cursor_type )
   GlgObject viewport;
   char * resource_name;
   GlgCursorType cursor_type;
```

Parameters

viewport

Specifies a viewport.

resource_name

Specifies the resource name of a child viewport to set the cursor of, or NULL to set the cursor of the viewport specified by the *viewport* parameter. The resource name is relative to the viewport specified by the *viewport* parameter.

cursor_type

Specifies one of the system cursor types to be used: GLG_DEFAULT_CURSOR, GLG_CROSSHAIR_CURSOR, GLG_WAIT_CURSOR, GLG_HAND_CURSOR, or GLG_MOVE_CURSOR.

This function uses a subset of cursor types that can be used in a cross-platform way.

The specified cursor will be used when the mouse moves inside the viewport the cursor is assigned to. In the GTK version of the GLG library on Linux, the cursor will be changed for the top window that contains the viewport.

GlgSetCursor

Sets a custom cursor for a drawing or one of its child viewports (Windows only).

```
GlgBoolean GlgSetCursor( viewport, resource_name, cursor )
   GlgObject viewport;
   char * resource_name;
   GlgLong cursor;
```

Parameters

viewport

Specifies a viewport.

resource_name

Specifies the resource name of a child viewport to set the cursor of, or NULL to set the cursor of the viewport specified by the *viewport* parameter. The resource name is relative to the viewport specified by the *viewport* parameter.

cursor

Specifies one of the predefined Windows cursor types.

The custom cursor will be used when the mouse moves inside the viewport the custom cursor is assigned to.

While *GlgSetCursorType* uses only predefined cross-platform subset of cursor types, *GlgSetCursor* can use all cursor types defined on Windows.

GlgSetDefaultViewport

Sets a drawing to be used by the GLG Custom Control on Windows.

```
void GlgSetDefaultViewport( viewport )
   GlgObject viewport;
```

Parameters

viewport

Specifies a GLG viewport object to be used as a widget. The viewport object may be loaded using *GlgLoadWidgetFromFile*, *GlgLoadWidgetFromImage*, or *GlgLoadWidgetFromObject* functions. It may also be created programmatically using functions from the GLG Extended API.

One instance of a viewport may be used to create only one GLG Custom Control since every control needs its own copy of the viewport. To create several custom controls with the same drawing, load a new instance of the drawing for each control or use *GlgCopyObject* to create instances of the drawing, then select them using *GlgSetDefaultViewport* before creating each control or widget. This function references the viewport object.

GlgSetDResource
GlgSetDResourceIf
GlgSetDTag

Set a new value for a resource or tag of type D. For information about the data types used by GLG objects, see the *Structure of a GLG Drawing* chapter.

```
GlgBoolean GlgSetDResource( object, resource_name, d_value )
   GlgObject object;
   char * resource_name;
   double d_value;
```

```
GlgBoolean GlgSetDResourceIf( object, resource_name, d_value,
            if_changed )
   GlgObject object;
   char * resource_name;
   double d_value;
   GlgBoolean if_changed;

GlgBoolean GlgSetDTag( object, tag_source, d_value, if_changed )
   GlgObject object;
   char * tag_source;
   double d_value;
   GlgBoolean if_changed;
```

Parameters

object

Specifies a GLG object.

resource_name, tag_source

Specifies the name of the scalar resource or tag to be set. For tags, all tags with the specified *tag_source* will be set to the supplied value. May use wildcards, see the *Using Resource and Tag Wildcards* section. See the *Resource Access Optimization* section for additional information.

d_value

Specifies a new value for the resource or tag.

if_changed

If set to *GlgFalse*, the graphical updates will be performed even if the new value is the same as the old one (equivalent to *GlgSetDResource* function). Setting the parameter to *GlgTrue* optimizes performance of applications that set resource or tag values regardless whether the values have changed. The parameter is ignored when updating chart entry points to allow plotting straight lines when the plotted value does not change over time.

If a scalar resource or tag with the given name exists, this function sets it to a new value and returns *GlgTrue*, otherwise it prints an error message and returns *GlgFalse*.

GlgSetEditMode

Sets the viewport's edit mode which disables input objects in the viewport while the drawing is being edited.

```
GlgBoolean GlgSetEditMode( viewport, resource_name, edit_mode )
   GlgObject viewport;
   char * resource_name;
   GlgBoolean edit_mode;
```

Parameters

viewport

Specifies a viewport object (either a viewport or a light viewport).

resource_name

Specifies the resource name of a child viewport to set the editing mode of, or NULL to set the editing mode of the viewport specified by the *viewport* parameter. The resource name is relative to the viewport specified by the *viewport* parameter.

edit_mode

GlgTrue to set the editing mode, or *GlgFalse* to reset.

If the viewport's edit mode is turned on, the input objects in the viewport will not react to the mouse and keyboard events. The rest of the input objects outside of the viewport will still be active. The edit mode is usually set for viewports that serve as a drawing area; the edit mode ensures that the input objects do not react to mouse clicks when they are dragged with the mouse to a new position. This is similar to the behavior of the Builder's *Drawing Area*.

The edit mode is global and may be set only for one viewport used as a drawing area. Setting the edit mode of a new viewport resets the edit mode of the previous viewport. The viewport's edit mode is inherited by all its child viewports. The function returns *GlgTrue* for success, otherwise it generates an error message and returns *GlgFalse*.

GlgSetErrorHandler

Replaces the GLG Toolkit error handler and returns the previous error handler.

```
GlgErrorHandler GlgSetErrorHandler( new_handler )
   GlgErrorHandler new_handler;
```

Parameters

new_handler

Specifies a new error handler, supplied by the user, to be called on an error condition. This does not include internal errors of the GLG Toolkit, if any are detected: they are still reported using the default handler.

The function prototype for the new handler is as follows:

```
void new_handler( error_message, error_type )
   char * error_message;
   GlgErrorType error_type;
```

The *error_type* parameter may have the following values:

GLG_INTERNAL_ERROR
GLG_USER_ERROR
GLG_WARNING
GLG_INFO

Refer to the description of the *GlgError* function on page 77 for more information on the error types.

For more information about error handlers, see the *Error Processing and Debugging* section on page 50.

GlgSetFocus

Sets the keyboard focus to an object's viewport. If the object is a viewport, sets the focus to this viewport, otherwise sets the focus to the parent viewport of the object.

```
void GlgSetFocus( object, resource_name )
   GlgObject object;
   char * resource_name;
```

Parameters

object

Specifies an object.

resource_name

Specifies the resource name of a child object to use, or NULL to use the object specified by the *object* parameter. The resource name is relative to the object specified by the *object* parameter.

GlgSetGResource
GlgSetGResourceIf
GlgSetGTag

Set the values of a geometrical resource or tag.

```
GlgBoolean GlgSetGResource( object, resource_name, g_value1, g_value2,
            g_value3 )
   GlgObject object;
   char * resource_name;
   double g_value1, g_value2, g_value3;

GlgBoolean GlgSetGResourceIf( object, resource_name, g_value1,
            g_value2, g_value3, if_changed )
   GlgObject object;
   char * resource_name;
   double g_value1, g_value2, g_value3;
   GlgBoolean if_changed;

GlgBoolean GlgSetGTag( object, tag_sourcee, g_value1, g_value2,
            g_value3, if_changed )
   GlgObject object;
   char * tag_source;
   double g_value1, g_value2, g_value3;
   GlgBoolean if_changed;
```

Parameters

object

Specifies a GLG object.

resource_name, tag_source

Specifies the name of the geometrical resource or tag to be set. For tags, all tags with the specified *tag_source* will be set to the supplied values. May use wildcards, see the *Using Resource and Tag Wildcards* section. See the *Resource Access Optimization* section for

additional information.

g_value0, g_value1, g_value2

Specify the new XYZ or RGB values for the resource or tag.

if_changed

If set to *GlgFalse*, the graphical updates will be performed even if the new value is the same as the old one (equivalent to *GlgSetDResource* function). Setting the parameter to *GlgTrue* optimizes performance of applications that set resource or tag values regardless whether the values have changed. The parameter is ignored when updating chart entry points to allow plotting straight lines when the plotted value does not change over time.

If a geometrical resource or tag with the given name exists, this function sets it to the input values and returns *GlgTrue*. Otherwise, the function prints an error message and returns *GlgFalse*.

GlgSetLParameter

Sets a value of a global parameter.

```
GlgBoolean GlgSetLParameter( char * name, GlgLong value )
   char * name;
   GlgLong value;
```

Parameters

name

Specifies parameter name.

value

Specifies parameter value.

Refer to the *Appendix D: Global Parameters* section on page 416 for information on available global parameters. The function returns *GlgTrue* on success.

GlgSetResourceFromObject

Sets a data or matrix resource object's value using another object.

```
GlgBoolean GlgSetResourceFromObject( object, resource_name,
            data_or_matrix_object )
   GlgObject object;
   char * resource_name;
   GlgObject data_or_matrix_object;
```

Sets the value of the resource specified with the object and resource name to a value defined by the *data_or_matrix_object*. The object type (and data type for the data object) of the resource specified by the resource name should match the type of the *data_or_matrix_object*. May use wildcards in the *resource_name* string, see the *Using Resource and Tag Wildcards* section. See the *Resource Access Optimization* section for additional information.

GlgSetSResource
GlgSetSResourceIf
GlgSetSTag

Replaces the string in a string resource or tag.

```
GlgBoolean GlgSetSResource( object, resource_name, s_value )
   GlgObject object;
   char * resource_name;
   char * s_value;

GlgBoolean GlgSetSResourceIf( object, resource_name, s_value,
            if_changed )
   GlgObject object;
   char * resource_name;
   char * s_value;
   GlgBoolean if_changed;

GlgBoolean GlgSetSTag( object, tag_source, s_value, if_changed )
   GlgObject object;
   char * tag_source;
   char * s_value;
   GlgBoolean if_changed;
```

Parameters

object

Specifies a GLG object.

resource_name, tag_source

Specifies the name of the string resource or tag to be set. For tags, all tags with the specified *tag_source* will be set to the supplied value. May use wildcards, see the *Using Resource and Tag Wildcards* section. See the *Resource Access Optimization* section for additional information.

s_value

Specifies the new string for the resource or tag.

if_changed

If set to *GlgFalse*, the graphical updates will be performed even if the new value is the same as the old one (equivalent to *GlgSetDResource* function). Setting the parameter to *GlgTrue* optimizes performance of applications that set resource or tag values regardless whether the values have changed. The parameter is ignored when updating chart entry points to allow plotting straight lines when the plotted value does not change over time.

If a string resource or tag with the given name exists, this function creates a copy of the input string, and sets that copy as the new value of the resource or tag. The function returns *GlgTrue* if the resource or tag has been successfully changed, otherwise it generates an error message and returns *GlgFalse*. The memory associated with the old string is automatically freed.

GlgSetSResourceFromD
GlgSetSResourceFromDIf
GlgSetSTagFromD

Replace the string in a string resource or tag with a value specified by an input number and a format string.

```
GlgBoolean GlgSetSResourceFromD( object, resource_name, format,
            d_value )
   GlgObject object;
   char * resource_name;
   char * format;
   double d_value;

GlgBoolean GlgSetSResourceFromDIf( object, resource_name, format,
            d_value, if_changed )
   GlgObject object;
   char * resource_name;
   char * format;
   double d_value;
   GlgBoolean if_changed;

GlgBoolean GlgSetSTagFromDIf( object, tag_source, format, d_value,
            if_changed )
   GlgObject object;
   char * tag_source;
   char * format;
   double d_value;
   GlgBoolean if_changed;
```

Parameters

object

Specifies a GLG object.

resource_name, tag_source

Specifies the name of the string resource or tag to be set. May use wildcards, see the *Using Resource and Tag Wildcards* section. See the *Resource Access Optimization* section for additional information.

format

Specifies the format to use when setting the resource or tag string. This is a printf-style format string.

d_value

Specifies the numerical value to be converted to a string according to the format.

if_changed

If set to *GlgFalse*, the graphical updates will be performed even if the new value is the same as the old one (equivalent to *GlgSetDResource* function). Setting the parameter to *GlgTrue* optimizes performance of applications that set resource or tag values regardless whether the values have changed. The parameter is ignored when updating chart entry points to allow plotting straight lines when the plotted value does not change over time.

If a string resource or tag with the input name exists, this function sets it to a new string containing the *d_value* parameter in the format specified with the *format* argument. The function returns *GlgTrue* for success, otherwise it generates an error message and returns *GlgFalse*.

GlgSetTooltipFormatter

Supplies a custom tooltip formatter, returns the previously set formatter, if any.

```
GlgTooltipFormatter GlgSetTooltipFormatter( formatter )
   GlgTooltipFormatter formatter;
```

Parameters

formatter

Specifies a custom tooltip formatter.

A tooltip formatter is a function which is invoked every time a new tooltip string needs to be generated. It can create and return a tooltip string, or return NULL to use a default tooltip string. An empty string may be returned to prevent the tooltip from showing up.

When the tooltip string is no longer needed, it will be freed by the Toolkit using the *GlgFree* call. Therefore, it must be allocated using *GlgAlloc* or one of the GLG string utility functions, such as *GlgStrClone* or *GlgConcatStrings*.

If the *GlgTooltipFormatterOnErase* global configuration resource is set to 1, the tooltip formatter function will also be invoked with NULL parameters when the tooltip is erased.

A custom tooltip formatter has the following function prototype:

```
typedef char * (*GlgTooltipFormatter)
             ( GlgObject viewport, GlgObject object,
               GlgObject tooltip_string,
               GlgLong root_x, GlgLong root_y )
```

A custom tooltip formatter is invoked with the following parameters:

viewport

The parent viewport of the object.

object

The object with the tooltip action.

tooltip_string

The data object of S (string) type containing the tooltip string.

root_x

The X coordinate of the cursor relative to the root window.

root_y

The Y coordinate of the cursor relative to the root window.

GlgSetZoom

Provides a programmatic interface to integrated zooming and panning.

```
GlgBoolean GlgSetZoom( object, resource_name, type, value )
   GlgObject object;
   char * resource_name;
   GlgLong type;
   double value;
```

Parameters

object

Specifies a viewport or a light viewport.

resource_name

Specifies the resource name of a child viewport to zoom or pan, or NULL to zoom or pan the viewport specified by the *object* parameter. The resource name is relative to the viewport specified by the *object* parameter.

type

Specifies the type of the zoom or pan operation to perform. The value of this parameter matches the corresponding zooming and panning accelerators and may have the following values:

'i' - zoom in relatively to the center of the drawing by the factor specified by the *value* parameter (the value of 2 is used as a default if *value*=0).
In the Chart Zoom Mode, it zooms in only in the X/Time direction.

'I' - same as *'i'* but zooms in relatively to the current mouse position.
In the Chart Zoom Mode, it zooms out only in the Y direction.

'o' - zoom out relatively to the center of the drawing by the factor specified by the *value* parameter (the value of 2 is used as a default if *value*=0).
In the Chart Zoom Mode, it zooms out only in the X/Time direction.

'O' - same as *'o'* but zooms out relatively to the current mouse position.
In the Chart Zoom Mode, it zooms out only in the Y direction.

't' - start generic ZoomTo mode (left-click and drag the mouse to finish, *value* is ignored).
In the Chart Zoom Mode, if the first point of the ZoomTo box is located within the X or Y axis area, zooming will be performed only in the direction of the selected axis. For example, if the user defines the ZoomTo box in the X axis area, the chart will be zoomed only in the X direction.

'_' - start ZoomToX mode, which zooms only in the X direction and preserves the Y scale (left-click and drag the mouse to finish, *value* is ignored). It is especially useful in the Chart Zoom Mode.

'|' - start ZoomToY mode, which zooms only in the Y direction preserves the X scale (left-click and drag the mouse to finish, *value* is ignored). It is especially useful in the Chart Zoom Mode.

'@' - start ZoomToXY mode (left-click and drag the mouse to finish, *value* is ignored)

'T' - start custom ZoomTo mode (left-click and drag the mouse to ZoomTo, *value* is ignored). A custom zoom mode lets the user define the ZoomTo area without performing the zoom operation. An application can use the selected ZoomTo rectangle as the input to

implement custom zooming or object selection logic. An application can handle Zoom events in input callback to perform a custom zoom operation. Refer to the *Appendix B: Message Object Resources* section of the *GLG Programming Reference Manual* for details of the *Zoom* event.

'e' - abort ZoomTo mode, *value* is ignored

'*s'* - start generic dragging mode (left-click and drag the drawing with the mouse to pan or scroll).
In the Chart Zoom Mode, if the user clicks and drags the mouse within the X or Y axis area, scrolling will be performed in the direction matching the direction of the selected axis.

'^' - start vertical dragging mode (left-click and drag the drawing with the mouse to pan or scroll in the vertical direction). It is especially useful in the Chart Zoom Mode.

'*>'* - start horizontal dragging mode (left-click and drag the drawing with the mouse to pan or scroll in the horizontal direction). It is especially useful in the Chart Zoom Mode.

'*&'* - start XY dragging mode (left-click and drag the drawing with the mouse to pan or scroll).

'f' - fit the drawing to the visible area of the viewport, *value* is ignored (Drawing Zoom Mode only). The X/Y ratio is maintained depending on the setting of the viewport's *Stretch* attribute.

'F' - fit the area of the drawing defined by an object named *GlgFitArea* to the visible area of the viewport, *value* is ignored (Drawing Zoom Mode only). The X/Y ratio is maintained depending on the setting of the viewport's *Stretch* attribute.

'n' - reset zoom, *value* is ignored.
In the GIS zoom mode with map in the ORTHOGRAPHIC projection, resets zooming but keeps the GIS center unchanged. In the RECTANGULAR projection, resets zooming and paning but keeps the rotation angle unchanged.
In the Chart Zoom Mode, resets the Y range to fit all chart plots in the visible area of the chart in Y direction.

'N' - reset zoom, *value* is ignored.
In the Chart Zoom Mode, resets the Y range and changes the chart's X span to show all accumulated data samples in the visible area of the chart.

'u' - pan up by the fraction of the widow height specified by the *value* parameter (the value of 0.5 is used as a default if *value*=0).
In the Chart Zoom Mode, scroll the chart up by a fraction of the chart's data area height.

'd' - pan down by the fraction of the widow height specified by the *value* parameter (the value of 0.5 is used as a default if *value*=0).
In the Chart Zoom Mode, scroll the chart down by a fraction of the chart's data area height.

'l' - pan left by the fraction of the widow width specified by the *value* parameter (the value of 0.5 is used as a default if *value*=0).
In the Chart Zoom Mode, scroll the chart left by a fraction of the chart's data area width.

'r' - pan right by the fraction of the widow width specified by the *value* parameter (the value of 0.5 is used as a default if *value*=0).
In the Chart Zoom Mode, scroll the chart right by a fraction of the chart's data area width.

'x' - pan horizontally by the distance in screen coordinates specified by the *value* parameter, the sign of the value defines the pan direction

'y' - pan vertically by the distance in screen coordinates specified by the *value* parameter, the sign of the value defines the pan direction

'X' - pan horizontally by the distance in world coordinates specified by the *value* parameter, the sign of the value defines the pan direction.
In the GIS Zoom Mode, pans by the latitude degrees.
In the Chart Zoom Mode, scrolls by the units of the X axis.

'Y' - pan vertically by the distance in world coordinates specified by the *value* parameter, the sign of the value defines the pan direction.
In the GIS Zoom Mode, pans by the longitude degrees.
In the Chart Zoom Mode, scrolls by the units of the first Y axis.

'U' - anchor on the upper edge, *value* is ignored (Drawing Zoom Mode only)

'D' - anchor on the lower edge, *value* is ignored (Drawing Zoom Mode only)

'R' - anchor on the right edge, *value* is ignored (Drawing Zoom Mode only)

'L' - anchor on the lower edge, *value* is ignored (Drawing Zoom Mode only)

'A' - rotate the drawing clockwise around X axis by the angle specified by the value parameter (Drawing Zoom Mode only)

'a' - rotate the drawing counterclockwise around X axis by the angle specified by the value parameter (Drawing Zoom Mode only)

'B' - rotate the drawing clockwise around Y axis by the angle specified by the value parameter (Drawing Zoom Mode only)

'b' - rotate the drawing counterclockwise around Y axis by the angle specified by the value parameter (Drawing Zoom Mode only)

'C' - rotate the drawing clockwise around Z axis by the angle specified by the value parameter (Drawing Zoom Mode or GIS Zoom Mode with the rectangular projection only)

'c' - rotate the drawing counterclockwise around Z axis by the angle specified by the value parameter (Drawing Zoom Mode or GIS Zoom Mode with the rectangular projection only)

value

Specifies a value for zoom and pan operations as described above.

The function performs a specified zoom or pan operation regardless of the setting of the viewport's *Pan* and *ZoomEnabled* attributes. In the Drawing Zoom Mode, if the function attempts to set a very large zoom factor which would result in the overflow of the integer coordinate values used by the native windowing system, the zoom operation is not performed and the function returns *GlgFalse*.

The left mouse button is the default button for performing the ZoomTo operation, as well as for panning and scrolling the drawing by dragging it with the mouse. These defaults can be changed by setting the *GlgZoomToButton* and *GlgPanDragButton* global configuration resources. If the default is changed, the left-click used in the description of the affected operations changes to the click with the mouse button assigned to the respective ZoomTo or Pan operation.

GlgSetZoomMode

Sets or resets a viewport's zoom mode by supplying a GIS or Chart object to be zoomed. In the Drawing Zoom Mode (default), the drawing displayed in the viewport is zoomed and panned. If the GIS Zoom Mode is set, any zooming and panning operation performed by the *GlgSetZoom* method will zoom and pan the map displayed in the viewport's GIS Object, instead of being applied to the viewport's drawing. In the Chart Zoom Mode, the content of the chart is zoomed and panned.

```
GlgBoolean GlgSetZoom( object, resource_name,
                       zoom_object, zoom_object_name )
   GlgObject object;
   char * resource_name;
   GlgObject zoom_object;
   char * zoom_object_name;
```

Parameters

object

Specifies a viewport object.

resource_name

Specifies the resource name of a child viewport whose zoom mode to set, or NULL to set the Zoom Mode of the viewport specified by the *object* parameter. The resource name is relative to the viewport specified by the *object* parameter.

zoom_object

Specifies a GIS or Chart object (or its parent if *zoom_object_name* is not NULL) for the GIS or Chart zoom mode, or NULL for the drawing zoom mode.

zoom_object_name

If the parameter is NULL, the GIS or Chart object supplied by the *zoom_object* parameter is selected as the GIS or Chart object to be zoomed. If the parameter is not NULL, it specifies the resource name of the GIS or Chart object to be zoomed. The resource name is relative to the *zoom_object* parameter, or to the viewport specified by the *object* and *resource_name* parameters if the *zoom_object* parameter is NULL.

zoom_mode

The zoom mode to set:

- GLG_DRAWING_ZOOM_MODE to zoom the drawing
- GLG_GIS_ZOOM_MODE to zoom a GIS Object's map
- GLG_CHART_ZOOM_MODE to zoom a Chart object.

GlgStrClone

Creates a copy of a character string.

```
char * GlgStrClone( string )
   char * string;
```

Parameters

string

The string to be copied. This can be a NULL pointer, in which case the return value is also NULL.

This function creates and returns a copy of the input string. The copy should later be freed with the *GlgFree* function.

GlgUpdate

Updates a drawing to show new resource values.

```
GlgBoolean GlgUpdate( object )
   GlgObject object;
```

Parameters

object

Specifies a viewport to update. If *object* is a light viewport, update of its parent viewport will be performed.

If *object* is a light viewport, an update of its parent viewport will be performed. All resource changes are held until the *GlgUpdate* function is called or until the viewport's window receives an expose event. (That is, it must be redrawn because another window that was obscuring a part of the viewport's window has been moved. This causes the viewport to redraw itself, using its new data.)

The function returns *GlgTrue* if the drawing has been successfully updated; otherwise it returns *GlgFalse*.

Note: Some drawing resources affect the object hierarchy of the drawing. For example, the *Factor* attribute of a series object controls the number of template instances that will be created by the series. If a number of instances is increased by setting the series' *Factor*, the new instances will be created and their resources can be accessed only after the *GlgUpdate* or *GlgSetupHierarchy* function is invoked. If the series object is not persistent, any change (either increase or decrease) of its *Factor* attribute recreates its instances, and either the *GlgUpdate* or *GlgSetupHierarchy* function should be called before accessing resources of the series' instances.

GlgPrintObjectCounts

Prints a total object count for each of the GLG object types.

```
void GlgPrintObjectCounts( void )
```

This function may be used to diagnose memory and object leaks. On Windows, object counts output is stored in the *glg_error.log* file. On Linux/Unix, the output is also printed to *stdout*.

GlgPrintObjectCountChanges

Prints object count changes since the last call to this function.

```
GlgPrintObjectCountChanges( void )
```

This function may be used to diagnose memory and object leaks. On Windows, object counts output is stored in the *glg_error.log* file. On Linux/Unix, the output is also printed to *stdout*.

GlgXPrintDefaultError

Prints information about an X error on a console or logs it into a file. The function can be used for printing error information in custom X error handlers and is available only in the legacy X11 version of the Toolkit on Linux/Unix.

```
GlgBoolean GlgXPrintDefaultError( display, event, file )
   Display *display;
   XErrorEvent *event;
   FILE * file;
```

Parameters

display

Specifies the connection to the X server.

event

X error event.

file

File to log the output to. *stdout* may be used to print the output on the console.

The function returns *GlgTrue* if it detects OpenGL GLX errors, otherwise it returns *GlgFalse*.

2.5 Handling User Input and Other Events

Callback Events

A **callback** is a function or method that is invoked under certain conditions. For example, a callback is called each time a user selects something in a GLG drawing with the mouse. The code is supplied by the application developer.

A callback can be invoked for one of a few possible reasons:

- The user has used the mouse to select some graphical object in the drawing area of the widget, generating object selection and custom selection events.
- The user has moved the mouse over some object in the drawing, which may generate selection events as well.
- An input widget, such as button or slider, has received some input from the user. This usually arrives in the form of mouse motion or button clicking, although the text widget accepts typed input as well.
- The window has received some window event, such as a window closing request, resize event, etc.
- A *SubWindow* object has loaded the drawing that will be displayed in it, or a *SubDrawing* object has loaded its template.
- The GLG Wrapper Widget also invokes callback functions at startup, when resources are set. For information about the wrapper widget and its callback lists, see the *Callback Resources* section of the *Using Legacy GLG Wrapper Widget on Linux/Unix* chapter on page 33.
- An alarm handler is a special global callback that handles alarm events. Refer to the *GlgSetAlarmHandler* section on page 98 for more information.

In addition to these callbacks, the GLG Toolkit provides **trace** callbacks that are invoked whenever a GLG viewport receives **any** event. The trace callbacks are used as an escape mechanism to provide low-level access to native windowing events. There are two trace callbacks: a **trace** callback invoked before dispatching the event for processing, and a **trace2** callback invoked after the event has been processed by the Toolkit.

Attaching Callbacks to a Viewport Object

A callbacks is attached to a viewport with the *GlgAddCallback* function, which takes the callback type and the callback function as parameters:

```
void GlgAddCallback( viewport, type, callback, client data )
   GlgObject viewport;
   GlgCallbackType type;
   GlgCallbackProc callback;
   GlgAnyType client_data;
```

The callback type may be one of the values defined in the *GlgApi.h* file:

GLG_INPUT_CB
GLG_SELECT_CB
GLG_TRACE_CB
GLG_TRACE2_CB
GLG_HIERARCHY_CB.

All callbacks use the same function prototype:

```
typedef void (*GlgCallbackProc)( viewport, client_data, call_data )
   GlgObject viewport;
   GlgAnyType client_data;
   GlgAnyType call_data;
```

Parameters

viewport

A viewport handle corresponding to the viewport to which the callback was attached. For the GLG_INPUT_CB callback, it may be a light viewport.

client_data

Application defined data specified by the *GlgAddCallback* function.

call_data

Callback-specific data supplying information about the event which caused the callback.

A callback must be attached to a viewport before the viewport's object hierarchy is set up (after the viewport is loaded but before it is drawn). Child viewports may have their own callbacks attached, different from the callbacks of their parent viewport. Only one callback of each type can be attached to one viewport object. To remove a callback, use NULL as a callback parameter.

Adding Callbacks to a GtkGlg widget on Linux

The GTK version of the GLG C/C++ library uses the *GtkGlg* widget to display a GLG drawing in a GTK application. The widget's *gtk_glg_add_callback* function is used to attach callbacks to the widget as shown in the following example:

```
GtkWidget * glg_widget = gtk_glg_new();
GlgObject viewport = GlgLoadWidgetFromFile( "my_drawing.g" );
gtk_glg_add_callback( glg_widget, GLG_INPUT_CB, InputCallback, NULL );
gtk_glg_set_viewport( glg_widget, viewport );
```

The callbacks must be attached before the drawing is displayed. Alternatively, the *GlgAddCallback* function may be used to attach a callback to the drawing's viewport or any of its child viewports.

Refer to the *integration/gtk3_glg_native* directory of the GLG installation for a complete example program.

Adding Callbacks to a Legacy GLG Wrapper Widget

In the Motif/Xt environment, a GLG callback can also be attached to the top level viewport of the GLG Wrapper Widget's drawing as an Xt callback using the *XtAddCallback* function. The callback types include

XtNglgSelectCB
XtNglgMotifSelectCB

XtNglgInputCB
XtNglgMotifInputCB

XtNglgTraceCB
XtNglgTrace2CB

XtNglgHierarchyCB

XtNglgHInitCB
XtNglgVInitCB

Notice that for the select and input callbacks two callback types are provided: one following the GLG generic cross-platform format and one following the Motif calling convention. The callbacks must be added before the widget is realized and its drawing hierarchy is set up.

There are two widget initialization callbacks: *HInit* and *VInit*, which allow an application to initialize drawing's resources before it is displayed in the widget. The *HInit* callback is invoked after the widget's drawing is loaded but **before** its hierarchy is setup. It may be used to initialize values of **H** resources, such as a number of instances in a series object or a number of points in a graph. The *VInit* callback is invoked **after** the hierarchy setup but before the drawing is displayed. It may be used for initializing **V** resources, such as initial data values for a graph.

The Xt callbacks use the standard *XtCallbackProc* prototype and pass the callback information to the application using the callback's *call_data* parameter. The following description of specific callback types includes the description of Motif-style callbacks.

Selection Callback

The **selection callback** provides a simplified name-based interface to handle object selection. The input callback provides a more elaborate alternative for handling selection events using either object IDs, or custom events and selection commands which can be attached to objects in the Graphics Builder.

A selection callback is called when the user selects objects in the widget with a mouse click. If no objects are selected, the *call_data* parameter will be NULL. If some objects are selected by the click, the *call_data* parameter will contain a list of names of all selected objects. This list is a NULL-terminated list of pointers to strings, each of which represents the complete path name of

one object selected by the mouse click. If more than one object is selected, the list will contain several strings. The objects drawn on top will be listed first. The pointer to the list as well as pointers to the individual strings point to internal data structures which should not be modified.

The *GlgGetSelectionButton* function may be used to find out which mouse button was pressed to activate the selection callback.

The path name obtained in the selection callback may be used to query or access resources of the selected object. For example, if the fifth bar of a bar graph was selected, one of the path names will be *DataGroup/DataSample4*. To query the value of this data sample, concatenate this path name with "*Value*" using the *GlgConcatResNames* function and use the resulting *DataGroup/DataSample4/Value* path name in a call to *GlgGetDResource*.

The following example shows how to highlight the selected data sample of a graph with a different color in the selection callback:

```
void Select( viewport, client_data, call_data )
   GlgObject viewport;
   GlgAnyType client_data;
   GlgAnyType call_data;
{
   int i;
   char
      ** name_array,
      * name,
      * resource_name;

   if( !name_array )
      return;

   for( i=0; name = name_array[ i ]; ++i )
      if( strstr( name, "DataGroup/DataSample" ) )
      {
         /* Concatenate the name of the data sample with "FillColor".
            */
         resource_name = GlgConcatResNames( name, "FillColor" );
         GlgSetGResource( viewport, resource_name,
                          1., 0., 0. );    /* Red.*/
         GlgFree( resource_name );
      }
}
```

Legacy Motif Widget Selection Callback

The legacy GLG Motif Wrapper Widget provides the Motif-style select callback (*XtNglgMotifSelectionCB*). The *call_data* argument supplied to Motif version of the selection callback function is a structure defined as follows:

```
typedef struct _GlgSelectCBStruct
{
   GlgCallbackType reason;
   XEvent * event;           /* Event that triggered the callback */
   GlgObject viewport;       /* Viewport that received the mouse click
                                that triggered the selection
                                callback */
   Widget widget;            /* Wrapper's widget ID */
   long num_selected;        /* The length of the selection list */
   char ** selection_list;   /* Same as in the Xt selection callback */

} GlgSelectCBStruct;
```

Note that the structure contains the same list of selected objects that is sent to a callback function when using the standard GLG selection callback described above.

Input Callback

The Toolkit translates the low-level events of the native windowing system, such as mouse moves, and mouse clicks, into high-level GLG events, such as object selection events, custom events and command actions associated with GLG objects, as well as input events such as slider moves and button presses. An **input callback** is the primary mechanism for handling any GLG events occurring in an application. It is invoked for several types of input activities:

- When some input activity is detected by an input widget, like moving a slider or a knob with the mouse. The widget has an input handler attached to the widget's viewport to process incoming events, and the type of the widget is defined by the type of the handler.
- When objects in the drawing are selected with the mouse over or mouse click events.
- When custom events or command actions associated with objects in the drawing are triggered in response to mouse move or click events.
- When window events occur, such as closing a top level GLG window.

To activate processing of selection and custom selection events on the mouse click and mouse move, the viewport's *ProcessMouse* attribute must be set to GLG_MOUSE_CLICK or GLG_MOUSE_MOVE_AND_CLICK, respectively. The rest of the event are always processed and do not require any additional settings.

If a system event in a viewport is translated into some GLG event, the input callback of the viewport object is invoked. If the viewport doesn't have an input callback attached, the input callback of its closest parent with input callback is invoked. The *call_data* parameter of the callback contains a *message* object used to pass all information about the event to the callback. This information is present in the message object in the form of named resources of the message object and may be obtained from it by using methods for querying resources (*GlgGetSResource*, *GlgGetDResource*

and *GlgGetGResource*). The message object should be passed as the first parameter, and a resource name defining the resource to query should be passed as the second parameter. For example, the following code extracts the value of the message object's *Origin* resource:

```
char * origin;
GlgGetSResource( message_object, "Origin", &origin );
```

If the input callback was invoked in response to an activation of an action or a command attached to an object, the message will also contain the *ActionObject* resource that may be used to access all information associated with the action or command.

The details of the message object for each event type are described in the *Message Object* section on page 124.

Legacy Motif Widget Input Callback

The legacy GLG Motif Wrapper Widget provides the Motif-style input callback (*XtNglgMotifInputCB*). The *call_data* argument supplied to Motif version of the input callback is a structure defined as follows:

```
typedef struct _GlgInputCBStruct
{
   GlgCallbackType reason;
   GlgObject viewport;     /* Viewport that received the event that
                              triggered the input callback. */
   Widget widget;          /* Wrapper's widget ID */
   GlgObject message_obj;  /* Same as in the Xt input callback */

} GlgInputCBStruct;
```

Note that the structure contains the same message object that is sent to a callback function when using the standard GLG input callback; see the *Message Object* section, below.

Trace Callbacks

The **trace callbacks** are used as an escape mechanism to provide low-level access to native windowing events. They are invoked whenever a viewport receives a native windowing event and passes the event to the application code for processing. If a viewport object has a trace callback attached, the trace callback will be invoked for all events received by the viewport and all of its child viewports. There are two trace callbacks: the **trace** callback is invoked before the event is dispatched for processing, and the **trace2** callback is invoked after the event has been processed.

As with the other callbacks, it is attached to a viewport with the *GlgAddCallback* function(or through the resources of the GLG Wrapper Widget.

The information necessary to completely identify an event may be gathered by parsing window data returned to the callback function, or by invoking functions in the GLG Standard or Extended API.

The *call_data* passed to these callback functions is a structure containing platform-specific data described below.

Trace Data for the GTK Version on Linux

In the GTK version of the GLG library on Linux, the following *call_data* structure is used:

```
typedef _GlgTraceCBStruct
{
   GlgCallbackType reason;   /* GLG_TRACE_CB or GLG_TRACE2_CB */
   GlgObject viewport;       /* Viewport that received the event. */
   GlgObject light_viewport; /* Light viewport of the event or NULL. */
   GXEvent * event;          /* GLG-style event that triggered the
                              callback. */
   GtkWidget * widget;       /* Widget ID of the event viewport. */
   GdkEvent * event1;        /* GDK event, may be NULL for some
                              GXEvents. */
} GlgTraceCBStruct;
```

The callback structure contains information about the viewport which received the event (indicated by the *viewport* field), as well as the event information. It also contains a pointer to the GXEvent structure, which may be used to provide details of the event. The *GXEvent* structure mimics the X11 *XEvent* structure, which allows the same code to be used with both the GTK and the legacy X11 version of the GLG library, as shown in the GLG C/C++ demos and examples.

The *GTK_GLG* macro have to be defined to compile the code with the GTK version of the library. The macro is already defined in the GTK version of the makefiles provided in the *src* directory of the GLG installation.

For the mouse and key events, the *light_viewport* field contains the light viewport under the mouse (if any), or NULL otherwise. The *widget* field provides information about native GTK widget used to render the viewport. The *reason* field is set to the type of the callback: GLG_TRACE_CB or GLG_TRACE2_CB. The *event1* is a pointer to a native GDK event and may be NULL for some *GXEvents*.

The following shows event types and names of the corresponding events used by the *GXEvent* structure:

```
typedef enum _GXEventType
{
   KeyPress          = 2,
   KeyRelease        = 3,
   ButtonPress       = 4,
   ButtonRelease     = 5,
   MotionNotify      = 6,
   EnterNotify       = 7,
   LeaveNotify       = 8,
   Expose            = 12,
   VisibilityNotify  = 15,
   UnmapNotify       = 18,
   MapNotify         = 19,
   ConfigureNotify   = 22,
   LASTEvent         = 35
} GXEventType;
```

```
typedef union _GXEvent
{
   GXEventType type;
   GXAnyEvent xany;
   GXKeyEvent xkey;
   GXButtonEvent xbutton;
   GXMotionEvent xmotion;
   GXCrossingEvent xcrossing;
   GXExposeEvent xexpose;
   GXVisibilityEvent xvisibility;
   GXUnmapEvent xunmap;
   GXMapEvent xmap;
   GXConfigureEvent xconfigure;
} GXEvent;
```

Refer to the *GlgApi.h* file for information on the fields contained in each of the event structures of the *GXEvent*.

Trace Data for the Legacy X11 Version on Linux/Unix

In the legacy X11 version of the GLG library on Linux, the following *call_data* structure is used:

```
typedef _GlgTraceCBStruct
{
   GlgCallbackType reason;   /* GLG_TRACE_CB or GLG_TRACE2_CB */
   GlgObject viewport;       /* Viewport that received the event. */
   GlgObject light_viewport; /* Light viewport of the event or NULL. */
   GXEvent * event;           /* Event that triggered the callback. */
   Widget widget;            /* Widget ID of the event viewport. */
   Window window;            /* Window ID of the event viewport. */
   Display * display;        /* Display of the event viewport. */
   XEvent * event1;          /* Reserved */
} GlgTraceCBStruct;
```

For the mouse and key events, the *light_viewport* field contains the light viewport under the mouse (if any), or NULL otherwise. The *widget*, *window* and *display* fields provide information about the corresponding parameters of the viewport's native components. The *reason* field is set to the type of the callback: GLG_TRACE_CB or GLG_TRACE2_CB. The *event1* pointer is reserved for future use.

Windows Trace Data Structure

On Windows, the following trace callback structure is used:

```
typedef _GlgTraceCallbackStruct
{
   GlgCallbackType reason;   /* GLG_TRACE_CB or GLG_TRACE2_CB */
   GlgObject viewport;       /* Viewport that received the event. */
   GlgObject light_viewport; /* Light viewport of the event or NULL. */
   MSG * event;              /* Event that triggered the callback. */
   HWND widget;              /* Window of the event viewport. */
   HWND window;              /* Window of the event viewport. */
   void * display;           /* NULL */
   void * event1;            /* Reserved */
} GlgTraceCallbackStruct;
```

The callback structure contains information about the viewport which received the event (indicated by the *viewport* field), as well as the event information.

The *event* field contains the native Windows event that provides more detail about the event. The *widget* and *window* fields contain the handle of the viewport's native window. The *reason* field is set to the type of the callback: GLG_TRACE_CB or GLG_TRACE2_CB. The *display* field is set to NULL and the *event1* pointer is reserved for future use.

For the mouse and key events, the *light_viewport* field contains the light viewport under the mouse (if any), or NULL otherwise.

Light Viewport Events

In addition to native windowing system events, the trace callbacks may receive a few events generated by the GLG object engine for light viewports: GLG_LV_ENTER_NOTIFY, GLG_LV_LEAVE_NOTIFY and GLG_LV_RESIZE. On Windows, the GLG engine also generates viewport enter notify and leave notify events which are not generated by the native windowing system: GLG_MSW_ENTER_NOTIFY and GLG_MSW_LEAVE_NOTIFY.

Hierarchy Callback

The **hierarchy callback** is used to get access to the drawing to be displayed in the *SubWindow* object before the drawing is displayed. *SubWindows* may be used to switch drawings displayed in them, and an application may want to install a separate input callback for each loaded drawing to handle user interaction instead of having one giant input callback that handles input events from all drawings. An input callback has to be added to the viewport of a drawing displayed in the subwindow before the drawing is set up and drawn. The hierarchy callback is invoked with the ID of the subwindow's drawing after it is loaded but before its hierarchy is setup, making it possible to add input callbacks as well as initialize the drawing with data before it is painted on the screen.

Similarly, the hierarchy callback is also invoked for the *SubDrawing* objects when they load their template drawings; the callback provides an object ID of the loaded object that will be displayed in the subdrawing.

The hierarchy callback is attached to a viewport with the *GlgAddCallback* function. A hierarchy callback may be added to the top level viewport or any of its children viewports. If any of a subwindow's (or subdrawing's) parent viewports has a hierarchy callback added, the callback of the closest parent viewport will be invoked each time the subwindow loads a new drawing or the subdrawing loads its template. The hierarchy callback is invoked multiple times for each template drawing: when the drawing is loaded but before it is set up, when the drawing is set up but before it is drawn, as well as before and after the drawing is reset.

The *call_data* passed to the hierarchy callback is a structure that provides an object ID of the *SubWindow* object that triggered the callback, as well as an object ID of the viewport of the loaded drawing:

```
typedef _GlgHierarchyCallbackStruct
{
   GlgCallbackType reason; /* GLG_HIERARCHY_CB */

   /* When the callback is invoked. */
   GlgHierarchyCallbackType condition;

   /* Subwindow or subdrawing object that loaded its template. */
   GlgObject object;

   /* Template instance of the subwindow or subdrawing. */
   GlgObject subobject;

} GlgHierarchyCallbackStruct;
```

The *reason* field of the callback structure is always set to GLG_HIERARCHY_CB. The *condition* field indicates when the callback is invoked:

GLG_BEFORE_SETUP_CB
before an object's hierarchy setup
GLG_AFTER_SETUP_CB
after an object's hierarchy setup
GLG_FINISHED_SETUP_CB
after finishing an object's setup: all transformed values are valid
GLG_BEFORE_RESET_CB
before an object's hierarchy reset
GLG_AFTER_RESET_CB
after an object's hierarchy reset.

The *object* field provides an object ID of the *SubWindow* or *SubDrawing* object that triggered the callback. The *subobject* field contains an ID of the drawing that will be displayed in the subwindow, or ID of the object that will be shown in the subdrawing (the *Instance* attribute).

The source code of the **SCADA Viewer example** provides an example of using the callback.

Message Object

A **message object** is a GLG object like any other. It is a group object, containing data in the form of named attributes just like other GLG objects. It is used to pass data to the input callbacks that may be associated with a viewport.

Any message object has the following named attributes:

Resource Name	Data Type	Description
Format	String	Defines the format of the message object and the resources present in it. It is defined by the event type and, for input messages, the type of the input handler which detected the input activity and sent the message. It may have values such as *Button*, *Slider*, *Knob*, *ObjectSelection*, *CustomEvent*.

Resource Name	Data Type	Description
Origin	String	Contains the name of the viewport that initiated the message.
FullOrigin	String	Contains the full path name of the viewport that initiated an input or window message, or the full path name of the object that initiated a custom event or a command message. This resource will be set to an empty string for other types of messages.
Action	String	Describes the input action occurred. It may have values such as *ValueChanged*, *Activated*, *CustomEvent*, *MouseClick* and so on, depending on the event type.
SubAction	String	Describes the action in more details. For example, for the slider's *ValueChanged* action it describes what caused the value change and may have values such as *Click* or *Motion*.
Object	*GlgObject*	The top-level object that triggered the *Input* callback, a custom event or a command action. This resource is present in all messages except the *ObjectSelection* message.
OrigObject	*GlgObject*	The low-level object that triggered the *Input* callback, a custom event or a command action. This resource is present in all messages except the *ObjectSelection* message.

A message object can have other resources as well, depending on the type of the event and, for input object events, the type of the input handler that generated the event. For events triggered by an Action attached to an object, such as Command or Custom Event, the message contains the *ActionObject* resource.

For events generated by input objects, such as a slider or a button, the message object includes handler-related resources of the input widget. For example, the message object coming from a GLG slider object may also contain such resources as *ValueX*, *ValueY*, *Granularity*, *Increment*, *DisableMotion* and others. These resources may be queried from either the message object or the viewport object that generated the message.

Refer to the *Appendix B: Message Object Resources* on page 377 for a complete list of all message types and their resources. Refer to the *Input Handlers* chapter of the GLG User's Guide for a complete list of resources for each input handler type.

Examples of Using Callbacks in Application Code

The following examples demonstrate how to use GLG callbacks. Refer to the source code of the GLG demos for additional examples.

Adding callbacks

The following code adds both selection and input callbacks to the *drawing* viewport object:

```
GlgAddCallback( drawing, GLG_SELECT_CB, SelectCB, callback_data );
GlgAddCallback( drawing, GLG_INPUT_CB, InputCB, callback_data );
```

Processing Selected Objects using Selection Callback

A selection callback has several parameters, one of which is a list of names of all selected objects, including complete path identifying the place of objects in the drawing hierarchy. The following example shows a selection callback which prints names of all selected objects:

```
void SelectCB( drawing, client_data, call_data )
   GlgObject drawing;
   GlgAnyType client_data;
   GlgAnyType call_data;
{
   char ** name_ array;
   char * name;
   int i;

   name_array = (char **) call_data;

   if( !name_array )
      PrintOnConsole( "Nothing was selected." );
   else
      for( i=0; name = name_array[ i ]; ++i )
         PrintOnConsole( name );
}
```

The selection callback may be used with a minimal GLG configuration, such as the Basic Edition of the Graphics Builder and the GLG Standard API.

Processing Selected Objects Using Input Callback

While the select callback provides a simplified name-based interface for handling object selection, the input callback provides a more elaborate mechanism based on object IDs. Several object selection techniques are available:

- A targeted object selection via the use of a custom event or command action attached to an object at design time. When an object is selected with the mouse, the input callback is invoked with a message that contains detailed information about the action that triggered the callback.
- An *ObjectSelection* message that provides a list of IDs of all selected objects on the lowest level of the object hierarchy. It does not require anything to be done at design time but uses the Intermediate API in the application code to retrieve IDs of selected objects.

The object selection event is generated when the mouse is moved over an object in the drawing, or an object is selected with a mouse click. The custom selection events and commands are generated when the selected object has a custom event or command action. The *ProcessMouse* attribute of the viewport object has to include a combination of the *Move* and *Click* masks to enable object selection

and custom selection events on the corresponding mouse events. Refer to the *Integrated Features of the GLG Drawing* chapter on page 237 of the *GLG User's Guide and Builder Reference Manual* for more information on object selection and custom selection events.

The following example shows how to process selection using both types of object selection messages.

```
void InputCB( viewport, client_data, call_data )
   GlgObject viewport;
   GlgAnyType client_data;
   GlgAnyType call_data;
{
   GlgObject
      message_obj,
      selection_array,
      selected_object,
      command,
      action_object,
      action_data;
   char
      * format,/* Message format */
      * action,/* Message action */
      * selecte_object_name,  /* Name of the selected object. */
      * event_label,  /* Event label of a custom event or command. */
      * command_type; /* Type of a command to be executed on object
                         selection. */
   double button_index; /* Button that caused object selection. */
   int i, size;

   message_obj = (GlgObject) call_data;
   GlgGetSResource( message_obj, "Format", &format );
   GlgGetSResource( message_obj, "Action", &action );

   if( strcmp( format. "Command" ) == 0 )
   {
      /* This code handles command actions attached to anobject. */

      /* Retrive command information using the Standard API. */
      GlgGetSResource( message_obj,
            "ActionObject/Command/CommandType",&command_type );
      GlgGetSResource( message_obj, "EventLabel", &event_label );
      GlgGetSResource( message_obj, "Object/Name",
                       &selected_object_name );

      /* Retrieve the object IDs of the command and the selected
         object using the Intermediate API. */
      selected_object = GlgGetResourceObject( message_obj, "Object" );
      command =
         GlgGetResourceObject( message_obj, "ActionObject/Command" );

      ExecuteCommand( selected_object, command_type, command );
   }
   else if( strcmp( format, "CustomEvent" ) == 0 )
   {
```

```
      /* This code handles custom events attached toan object.
         This code handles both custom event actions and the
         old-style custom events. */

      GlgGetSResource( message_obj, "EventLabel", &event_label );

      /* Don't process events with empty EventLabel. An event with an
         empty EventLabel is generated for MouseOver events when the
         mouse moves away from the object, and for MouseClick events
         when the mouse button is released.
      */
      if( strcmp( event_label, "" ) == 0 )
         return;

      /* Retrieve command and selected object (Intermediate API).
         */
      selected_object = GlgGetResourceObject( message_obj, "Object" );

      /* Retrieve the selected object's name using Standard API. */
      GlgGetSResource( message_obj, "Object/Name",
                       &selected_object_name );

      /* For the old-style MouseClick actions, query the mouse button
         that generated this custom selection event (0 for custom
         MouseOver events). */
      GlgGetDResource( message_obj, "ButtonIndex", &button_index );

      /* Prints MouseClick or MouseOver. */
      PrintOnConsole2( "Custom event action: ", action );
      PrintOnConsole2( "Selected object: ", selected_object_name );

      /* Retrieve ActionObject resources of the message object
         (Intermediate API). ActionObject will be NULL for custom
         events added prior to GLG v.3.5. */
      action_obj =
         GlgGetResourceObject( message_obj, "ActionObject" );

      /* Retrieve the action's ActionData. If present, itsproperties
         may be used for processing the event. */
      if( action_obj )
         action_data =
            GlgGetResourceObject( action_obj, "ActionData" );

   }
   /* Handle default object selection message. */
   else if( strcmp( format, "ObjectSelection" ) == 0 )
   {
      selection_array =
         GlgGetResourceObject( message_obj, "SelectionArray" );
      if( !selection_array )
      {
         PrintToConsole( "No objects selected by ", action );
         return;
      }
```

```
            /* For MouseClick actions, query the mouse button that caused the
               the selection (0 for MouseMove events). */
            GlgGetDResource( message_obj, "ButtonIndex", &button_index );

            size = GlgGetSize( selection_array );
            for( i=0; i < size; ++i )
            {
               selected_object = GlgGetElement( selection_array, i );
               GlgGetSResource( selected_object, "Name",
                                &selected_object_name );

               /* Print MouseMove or MouseClick. */
               PrintToConsole2( "Selected by: ", action );
               if( !selected_object_name || !*selected_object_name )
                  PrintToConsole( "Unnamed object is selected." );
               else
                  PrintToConsole2( "Selected object: ",
                                   selected_object_name );
            }
         }
      }
```

The Enterprise Edition of the Graphics Builder is required to add Command actions or custom event actions to objects at design time. The GLG HMI Configurator can be also used to add Command actions to objects. The GLG Intermediate API may be used for more efficient and flexible processing of the Command and Custom Event actions in the application code. In the above example, the *GlgGetResourceObject* Intermediate API call is used to retrieve object ID of the selected object as well as the command object.

Handling the *ObjectSelection* message requires the use of the GLG Intermediate API to retrieve IDs of the selected objects.

Processing Input Object Events

There are two methods for processing user input from input objects, such as a button, a slider or a text input field, in an application code:

- using low-level input object events based on input object names
- using targeted input commands via the use of Input Command actions attached to input objects at design time. The *InputAction* parameter of the action object specifies the condition that triggers the input callback. When an input object receives user input, the input callback is invoked with a message that contains detailed information about the event.

The following example demonstrates a very simple input callback which terminates the application when a *Quit* button is pressed. The example demonstrates how to handle the default input object events, as well as an attached command actions.

```
void InputCB( drawing, client_data, call_data )
   GlgObject drawing;
   GlgAnyType client_data;
   GlgAnyType call_data;
```

```
{
   GlgObject message_object;
   char
      * origin, /* Name of the input object */
      * format, /* Message type */
      * action, /* Reason */
      * command_type; /* Command to be executed */

   message_object = (GlgObject) call_data;

   GlgGetSResource( message_object, "Format", &format );

   /* Handle default input object events. */
   if( strcmp( format, "Button" ) == 0 )
   {
      GlgGetSResource( message_object, "Action", &action );
      /* Query the name of the button that generated the message. */
      GlgGetSResource( message_object, "Origin", &origin );

      if( strcmp( action, "Activate" ) == 0 &&
          strcmp( origin, "Quit" ) == 0 )
         exit( 0 );
   }
   /* Handle command actions attached to a button. */
   else if( strcmp( format, "Command" ) == 0 )
   {
      GlgGetSResource( message_object,
            "ActionObject/Command/CommandType", &command_type );
      if( strcmp( command_type, "Quit" ) == 0 )
         exit( 0 );
   }
}
```

The Enterprise Edition of the Graphics Builder is required to add Input Command actions to input objects at design time. The GLG HMI Configurator can be also used to add Input Command actions. The GLG Intermediate API may be used for more efficient and flexible processing of the Input Command actions in the application code.

Refining Input Object Selection

Default Input Object Events

The following code fragment shows an example of an input callback that uses default input object events. The code determines which of the two sliders in the drawing (the temperature slider or pressure slider) received user input, and calls the appropriate function to process the new value specified by the user.

```
void InputCallback( viewport, client_data, call_data )
   GlgObject viewport;
   GlgAnyType client_data;
   GlgAnyType call_data;
{
   GlgObject message_obj;
   char
      * format,
      * action,
      * full_origin;
   double value;

   message_obj = (GlgObject)call_data;

   /* Extract callback data from the message object. */
   GlgGetSResource( message_obj, "Format", &format );
   GlgGetSResource( message_obj, "Action", &action );
   GlgGetSResource( message_obj, "FullOrigin", &full_origin );

   /* Ignore if message came not from a slider or not a ValueChanged. */
   if( strcmp( format, "Slider" ) != 0 || strcmp( action,
            "ValueChanged" ) != 0 )
      return;

   /* Obtain the new value of the slider. */
   GlgGetDResource( message_obj, "Value", &value );

   /* Call an appropriate function depending on the slider name. */

   if( strcmp( full_origin, "TemperatureBlock/Slider" ) == 0 )
      ProcessTemperatureInput( widget, value );
   else if( strcmp( full_origin, "PressureBlock/Slider" ) == 0 )
      ProcessPressureInput( widget, value );
   else
      Error( "Unknown name." );
}
```

Alternatively, Input Command actions may be attached to the sliders to execute the attached command as shown below.

Input Commands

The following input callback code demonstrates handling Input Command actions attached to the temperature and pressure sliders in the drawing. The code is completely generic and may be used with any drawing, since it uses the command data to execute the command without any reliance on the names of input objects in the drawing. The GLG Intermediate API is used to retrieve IDs of the input object and the command object.

```
void InputCallback( viewport, client_data, call_data )
   GlgObject viewport;
   GlgAnyType client_data;
   GlgAnyType call_data;
{
   GlgObject
      message_obj,
      command_obj,
      input_obj;
   char
      * format,
      * command_type,
      * value_resource,
      * tag;
   double value;

   message_obj = (GlgObject)call_data;

   /* Extract callback data from the message object. */
   GlgGetSResource( message_obj, "Format", &format );
   if( strcmp( format, "Command" ) != 0 )
      return;

   GlgGetSResource( message_obj, "ActionObject/Command/CommandType",
                    &command_type );

   if( strcmp( command_type, "WriteValueFromWidget" ) == 0 )
   {
      command_obj =
         GlgGetResourceObject( message_obj, "ActionObject/Command" );

      /* Input object that triggered the message. */
      input_obj = GlgGetResourceObject( message_obj, "Object" );

      /* The resource path to the input object'svalue.*/
      GlgGetSResource( command, "ValueResource", &value_resource );

      /* New value of input object. */
      GlgGetDResource( input_obj, value_resource, &value );

      /* The process controller tag to write the value to. */
      GlgGetSResource( command, "OutputTagHolder/TagSource", &tag );

      /* Write the new value. */
      SendValueToController( tag, value );
   }
```

```
}
```

Trace Callback examples

For examples of using the trace callback, refer to the source code of the **GLG Diagram demo**.

Hierarchy Callback examples

For examples of using the hierarchy callback, refer to the source code of the **SCADA Viewer example**.

2.6 GLG Intermediate and Extended API

The GLG Standard API enables an application to display a GLG drawing and animate it by querying and setting resources. The GLG Extended API adds functionality that allows an application program to edit or create the drawing at run time. The GLG Intermediate API is a subset of the Extended API that provides all of the Extended API methods for manipulating the drawing, except for the methods for creating, adding or deleting objects from the drawing at run time.

The **GLG Intermediate API** may be used to modify GLG drawings created in the GLG Graphics Builder and to implement elaborate custom logic to handle user interaction. The Intermediate API may also be used to obtain object IDs of resources in the drawing to access them directly, eliminating the resource path parsing required by the Standard API methods and increasing update performance of large drawings. The Intermediate API provides extensive functionality for changing an object's geometry and layout, creating and removing constraints, as well as advanced introspection capabilities for traversing the drawing and determining its content at run time.

The **GLG Extended API** includes all method of the Standard and Intermediate APIs and adds methods for creating, copying and adding objects to the drawing at run time. The Extended API may be used for adding objects to the drawing at run time or creating a drawing on the fly based on a configuration file. It may also be used for adding dynamics, tags and custom properties to objects at run time.

Both the Intermediate and Extended APIs are available in all deployment options supported by the Toolkit, including C/C++, Java, C#/.NET, JavaScript and (on Windows) ActiveX Control.

Refer to the *DEMOS* and *examples* directories for examples of using the Intermediate and Extended API methods in various programming environments.

Overview

Because of the abstract approach of the GLG architecture, the Extended API is small and simple, with very few functions to worry about, while still providing all the tools you need to create highly elaborate drawings. The power of the library comes from the symmetry of the architecture, where everything in a drawing is represented as a data object. This lets you use the same functions in many different combinations to obtain different effects, using the resource notation to control their actions.

There are two libraries that provide the Extended API functionality: **GLG Intermediate Library** and **GLG Extended Library**. The Intermediate Library provides all of the Extended API methods, except for the methods for creating and deleting objects at run time. The Extended Library contains all Extended API functions, including the methods for creating objects at run time.

In order to minimize the size of the API, all GLG functions are **generic** in the sense that some of their arguments are of the *GlgAnyType* type. The actual type of these parameters is determined dynamically at run time by their context (as defined by the values of the other parameters).

The GLG Intermediate API includes the following functions:

- **GlgLoadObject** loads a GLG object from a file.
- **GlgLoadObjectFromImage** loads a GLG object from a memory image created by the GLG Code Generation Utility.
- **GlgSaveObject** saves a GLG object into a file.
- **GlgReferenceObject** references an object.
- **GlgDropObject** dereferences an object.
- **GlgReorderElement** changes the object's position inside a container.
- **GlgContainsObject** finds if the object belongs to a container.
- **GlgGetNamedObject** finds a named object in a container.
- **GlgGetElement** returns the object at the specified index in a container.
- **GlgGetIndex** returns the index of the first occurrence of the object in a container.
- **GlgGetStringIndex** returns the index of the first occurrence of the string in a string container.
- **GlgFindObject** finds an object in a container.
- **GlgGetSize** queries the size of a container object.
- **GlgSetStart** and **GlgSetEnd** initialize a container object for the forward or backward traversing.
- **GlgIterate** and **GlgIterateBack** traverse a container object in the forward or backward direction.
- **GlgIsAt** may be used to check the current position of the iteration pointer.
- **GlgInverse** inverses the order of elements inside of a container.
- **GlgGetResourceObject** gets the object ID of a resource object.
- **GlgCreateResourceList** returns a list of an object's resources.
- **GlgCreateResourcePath** returns a relative resource path that can be used to access the object as a resource of the parent object.
- **GlgQueryTags** returns a list of all tags of the requested tag type defined in the object.
- **GlgHasTag** checks if a data or export tag with a specified tag name exists.
- **GlgGetTagObject** returns a tag object with a specified tag name, data tag source, export tag category or a list of tags matching the specified tag name, tag source or category pattern.
- **GlgGetAction** returns the first found action with the matching parameters attached to the object.
- **GlgGetAlarmObject** returns an alarm object with a specified alarm label, or a list of alarms matching the specified alarm label pattern.
- **GlgConstrainObject** adds constraints.
- **GlgUnconstrainObject** removes constraints.

- **GlgSuspendObject** suspends the object for editing.
- **GlgReleaseObject** releases the object after editing.
- **GlgGetParent** returns an object's parent object, if one exists.
- **GlgGetBoxPtr** returns an object's bounding box.
- **GlgCreateSelectionNames** returns the names of all the objects intersecting a given rectangle.
- **GlgCreateSelectionMessage** finds an object selected by a given selection criteria in a rectangular area and returns a selection message for the first found matching action attached to the object.
- **GlgCreateSelection** returns an array of objects intersecting a given rectangle.
- **GlgGetDrawingMatrix** returns the matrix used to draw the object.
- **GlgCreateInversedMatrix** inverts a transformation matrix object.
- **GlgTransformPoint** transforms a single point.
- **GlgCreatePointArray** creates and returns an array containing all control points of an object.
- **GlgMoveObjectBy, GlgMoveObject, GlgScaleObject, GlgRotateObject, GlgPositionObject, GlgFitObject** and **GlgTransformObject** transform an object in various ways.
- **GlgLayoutObjects** performs various alignment and layout operations.
- **GlgGetMatrixData** and **GlgSetMatrixData** query and set matrix data values.
- **GlgScreenToWorld** and **GlgWorldToScreen** convert coordinates between the screen and world coordinate systems.
- **GlgTranslatePointOrigin** converts screen coordinates of a point in one viewport to the screen coordinates relative to another viewport.
- **GlgRootToScreenCoord** converts screen coordinates relative to the root window to the screen coordinates in the specified viewport.
- **GlgGetParentViewport** returns a parent viewport or a parent light viewport of a GLG object.
- **GlgGetObjectViewport** is similar to GlgGetParentViewport but can also return the object itself, if it is requested and the object is a viewport.
- **GlgConvertViewportType** converts a viewport to a light viewport, or a light viewport to a viewport.
- **GlgFindObjects** finds an object's parents or children that match the supplied criteria and returns matching objects. Alternatively, the function can process the matching objects in-place without returning them.
- **GlgTraverseObjects** and **GlgTraverseObjectsExt** traverse all object's children, invoking the supplied custom function for each traversed object.
- **GlgTraceObject** highlights all objects in the drawing that depend on a tag or resource object.
- **GlgSetCustomSetupHandler** installs a custom setup handler that will be invoked to perform custom processing of objects tagged using the *CustomSetup* flag.

- **GlgSetObjectData** and **GlgGetObjectData** are used to associate arbitrary custom data with a graphical object.
- **GlgIsDrawable** tests if the object is drawable.
- **GlgDeleteTags** deletes all object's data tags or all object's public properties.
- **GlgSerialize** is used to implement custom save methods.
- **GlgSerializeFull** saves a GLG object in a raw memory block.

The Intermediate API also includes functions specific to the **Real-Time Chart** and **GIS** objects. These functions are described in the *Real-Time Chart Functions* chapter on page 212 and the *GIS Object Functions* chapter on page 228.

The GLG Extended API includes the following functions:

- **GlgCreateObject** creates a GLG object of any type.
- **GlgCopyObject** creates a copy of an object.
- **GlgCloneObject** creates a specialized copy (clone) of an object.
- **GlgAddObjectToTop, GlgAddObjectToBottom, GlgAddObjectAt** and **GlgAddObject** add an object to some container object. This can be used to add an object to a group or viewport as well as to add a point to a polygon.
- **GlgDeleteTopObject, GlgDeleteBottomObject, GlgDeleteObjectAt, GlgDeleteThisObject** and **GlgDeleteObject** delete an object from a container object.
- **GlgAddAttachmentPoint** adds an attachment point to a reference object.
- **GlgFlush** sets container size to the requested size.
- **GlgSetElement** replaces the object at the specified index in a container.
- **GlgSetXform** attaches a transformation to an object.
- **GlgSetResourceObject** replaces the resource with a new object.

Handling GLG Objects

As with the standard GLG API library, the programmer's task is made simpler by the dynamic typing of GLG objects. Most of the functions in the Extended API operate on several different kinds of objects. The action each one takes often depends on the type of its input object.

The Reference Count

Each GLG object occupies space in the computer memory. Some objects are larger than others, but all of them take up some space. In order to make certain that only objects used by a program are kept in memory, each object is equipped with a **reference count**. This is simply an integer with which an object keeps track of the number of operations that care about its existence. The GLG Extended API provides functions to increase and decrease the reference count: *GlgReferenceObject* and *GlgDropObject*. When an object's reference count reaches zero, it is deleted, and its memory reclaimed. The *GlgGetRefCount* function can be used to query an object's reference count.

When an object is created, its reference count is 1. Adding the object to a group increases its reference count, making sure that the object is kept around while it is needed. If you want the object to be deleted when the group is deleted, you can decrease the reference count once that object is safely contained in the group. This restores the reference count to 1. This is the standard sequence for creating GLG objects: create object, place in group or viewport, drop reference count. If the group is deleted or exploded, its members may be deleted, unless they are referenced beforehand.

An object is "responsible" for managing the reference count of its subsidiary objects. For example, a polygon object is responsible for managing the reference counts of its attribute objects. Similarly, when an object is inserted into a container, the container increments that object's reference count. When the object is removed from the group, its reference count is decremented. (Note that this can cause an object to be inadvertently deleted if the group was the only current reference to the object. If you want to remove an object from a group and put it somewhere else, you must increment its reference count before removing it.)

If these guidelines are met, any object with a reference count of zero may be safely deleted, and its memory reclaimed. This will prevent memory leaks, keeping all the system's memory available to the application.

Using Mouse Coordinates and Pixel Mapping

The Toolkit's rendering engine uses the OpenGL-style pixel mapping for consistent rendering across all windowing systems and programming environments. The integer pixel values are mapped to the center of the pixel, which means that the x and y coordinates of the center of the upper left pixel of a window are (0.5;0.5) instead of (0;0). When using the mouse position to obtain screen coordinates passed to the GLG functions, add GLG_COORD_MAPPING_ADJ (defined as 0.5 in *GlgApi.h*) to the x and y screen coordinates of the mouse for better precision.

Include Files

To use any of the functions described in this chapter, the application program must include the function definitions from the GLG API header file. Use the following line to include those definitions:

```
#include "GlgApi.h"
```

Intermediate API Function Descriptions

GlgConstrainObject

Constrains an attribute or a point of an object to an attribute or a point of another object.

```
GlgBoolean GlgConstrainObject( from_attribute, to_attribute )
   GlgObject from_attribute;
   GlgObject to_attribute;
```

Parameters

from_attribute

Specifies the attribute or point object to be constrained. To obtain the object ID of the *from_attribute*, the *GlgGetResourceObject* function must be used with a default resource name rather than a user-defined resource name as the last part of resource path. For example, "*my_object/FillColor*" must be used to get the object ID of the *FillColor* attribute of *my_object*, and not "*my_object1/my_color*".

For objects that have a fixed number of control points (arc, text, etc.), use the default attribute name (*"PointN"*) to access an object's *N*th control point. For objects with a variable number of points (e.g. polygons), use the *GlgGetElement* or *GlgIterate* function to get its points and use them as the *from_attribute*.

to_attribute

Specifies the attribute or the point to constrain to. This object may be queried by using either the default or a user-defined resource name.

This function constrains the attribute or the point specified by the *from_attribute* parameter to the attribute or the point specified by the *to_attribute* parameter. If two attributes are constrained, changing the value of either attribute changes the values of both. For more information about constraining, see the *Constraints* section in the *Structure of a GLG Drawing* chapter.

If the object whose attribute (*from_attribute)* is being constrained has already been drawn, an error message will be generated. You should either constrain object attributes before drawing the object or use the *GlgSuspendObject* function to suspend drawn objects before constraining their attributes.

You cannot constrain any objects beside attribute objects (a control point is an attribute object as well). The *from_attribute* and *to_attribute* attribute objects should be of the same data subtype. That is, you cannot constrain a color attribute object (G type) to the line width attribute object (D type) because they have different data types.

If any of the conditions above are not satisfied, the function fails and returns *GlgFalse*. If constraining is successful, the function returns *GlgTrue*.

Note that constraining an attribute object causes all the attributes of the attribute object to be identical to the attributes of the object to which it is constrained, including its name, transformation and value. If a transformation is added to an attribute object, it is automatically added to all the attribute objects that are constrained to this attribute object in order to maintain the constraints. Similarly, if the object name is changed, it is automatically changed for all the objects that are constrained to the object. These changes are permanent and remain in effect even after the constraints are removed.

GlgContainsObject

Checks if an object belongs to a container.

```
GlgBoolean GlgContainsObject( container, object )
   GlgObject container;
   GlgObject object;
```

Parameters

container

Specifies the container object.

object

Specifies the object to search for.

This function returns *GlgTrue* if the object belongs to the container and *GlgFalse* otherwise.

GlgConvertViewportType

Converts a viewport to a light viewport, or a light viewport to a viewport.

Returns a newly created object on success or NULL on error.

```
GlgObject GlgConvertViewportType( object )
   GlgObject object;
```

Parameters

object

A viewport or a light viewport to be converted.

If a converted object is added to the drawing, the original object must be deleted, since both objects reuse the graphical objects inside the viewport and cannot be both displayed at the same time.

GlgCreateInversedMatrix

Inverts a matrix object.

```
GlgObject GlgCreateInversedMatrix( matrix )
   GlgObject matrix;
```

Parameters

matrix

Specifies the matrix to be inverted.

Creates and returns the inverse of the input matrix. The returned matrix must be dereferenced using *GlgDropObject* when not needed any more.

This function may be used to invert the drawing transformation obtained with the *GlgGetDrawingMatrix* function. While the drawing transformation converts world to screen coordinates, the inverse of it may be used to convert the screen coordinates of objects and object bounding boxes back to world coordinates.

GlgCreatePointArray

Creates and returns an array containing all control points of an object.

```
GlgObject GlgCreatePointArray( object, type )
   GlgObject object;
   GlgControlPointType type;
```

Parameters

object

Specifies the object.

type

Specifies the type of points to be returned: control points, or control and attachment points.

The function returns an array of G data objects used as control points and must be dereferenced using *GlgDropObject* when finished.

GlgCreateResourceList

Returns a list of an object's resources.

```
GlgObject GlgCreateResourceList( object, list_named_res,
            list_def_attr, list_aliases )
   GlgObject object;
   GlgBoolean list_named_res, list_def_attr, list_aliases;
```

Parameters

object

The input object.

list_named_res

A boolean value indicating whether the returned list should include the input object's named resources.

list_def_attr

A boolean value indicating whether the returned list should include the input object's default attributes. The resource objects referred to by the default resource names are not actually included in the list. Instead a dummy scalar data object (type D) is included whose name is the same as the default attribute.

list_aliases

Indicates whether the returned list should include aliases. The resource objects referred to by the aliases are not included in the list. Instead a dummy scalar data object (type D) is included whose name is the same as the alias.

This function creates and returns the array of an object's resources. The list_named_res, list_def_attr and list_aliases parameters let you choose whether the array should include named resources, default attributes, aliases or any combination of the above.

The returned array has one entry for each resource. The entries are not sorted and are listed in the order of the occurrence inside their category. The named resources (if any) are listed first, then default resources and finally the aliases. The returned array must be dereferenced when finished.

For named resources, the entries are the object IDs of the resource objects themselves. For default resources and aliases, the entries are dummy scalar data objects named after the default attributes or aliases. For both types of entries, the name of the entry object is the name of the resource listed. The following code fragment shows how to print the list of resources:

```
GlgObject object, list, list_element;
GlgLong size, i;
char * name;

list = GlgCreateResourceList( object, GlgTrue, GlgTrue, GlgFalse );
if( list )
{
   size = GlgGetSize( list );
   for( i=0; i < size; ++i )
   {
      list_element = GlgGetElement( list, i);
      GlgGetSResource( list_element, "Name", name );
      printf( "Resource Name: %s\n", name );
   }
   GlgDropObject( list );
}
```

The *GlgCreateResourceList* function returns the resource list on only one level of the resource hierarchy. To see the resource lists of the returned resources, invoke the *GlgCreateResourceList* recursively using one of the following techniques:

- Use *GlgCreateResourceList* on the resources in the returned list. This will only work for the named resources since the default resources and aliases are represented by dummy objects.
- Get the name of a resource in the returned resource list and use the *GlgGetResourceObject* function to obtain its object ID, then call *GlgCreateResourceList* with that object ID. This method is slower but will work for both named resources, default attributes and aliases.

GlgCreateResourcePath

Returns a relative resource path that can be used to access the object as a resource of the parent object.

```
char * GlgCreateResourcePath( object, parent )
   GlgObject object;
   GlgObject parent;
```

Parameters

object

The object whose resource path is being queried. The object must be named.

parent

The parent object relative to which the resource path is evaluated. It must have its

HasResources flag set.

The function returns the resource path, or null if the resource path can't be determined. The returned path must be freed after use with *GlgFree* to avoid memory leaks.

GlgCreateSelectionMessage

Searches all objects inside the given rectangle for an action with the specified selection trigger type attached to an object and returns a selection message for the first found action attached to the object:

```
GlgObject GlgCreateSelectionMessage( top_vp, rectangle, selected_vp,
            selection_type, button )
   GlgObject top_vp;
   GlgRectangle * rectangle;
   GlgObject selected_vp;
   GlgSelectionEventType selection_type;
   GlgLong button;
```

Parameters

top_vp

The top viewport of the selection query (must be an ancestor of *selected_vp* or the same as *selected_vp*). It may be either a viewport or a light viewport.

rectangle

The bounding rectangle in screen coordinates of the *selected_vp* viewport. Any graphical shape whose rendering intersects this rectangle, including shapes in the *top_vp* viewport, is included in the search.

selected_vp

The viewport or the light viewport relatively to which the bounding rectangle is defined. When used with the mouse events, this is the viewport in which the mouse event occurred.

selection_type

This is an enumerated value specifying the selection type. The GLG_MOVE_SELECTION value selects custom mouse move events attached to an object, GLG_CLICK_SELECTION selects mouse click events and GLG_TOOLTIP_SELECTION selects the tooltip events.

button

The mouse button for the GLG_CLICK_SELECTION selection type.

This function may be used in a trace callback to determine whether a mouse event was meant to trigger an action attached to a graphical object in a drawing. The function returns a message equivalent to the message received in the input callback. If no matching actions were found, *NULL* is returned.

GlgCreateSelectionNames

Returns a list of names of objects intersecting a given rectangle.

```
GlgObject GlgCreateSelectionNames( top_vp, rectangle, selected_vp )
   GlgObject top_vp;
   GlgRectangle * rectangle;
   GlgObject selected_vp;
```

Parameters

top_vp

The top viewport or light viewport of the selection query (must be an ancestor of *selected_vp* or the same as *selected_vp*). It may be either a viewport or a light viewport.

rectangle

The bounding rectangle in screen coordinates of the *selected_vp* viewport. All graphical shapes whose rendering intersects this rectangle, including shapes in the *top_vp* viewport, are included in the selection list.

selected_vp

The viewport or the light viewport relatively to which the bounding rectangle is defined. When used with the mouse events, this is the viewport in which the mouse event occurred.

This function provides a low-level API to determine whether a mouse event was meant to select graphical objects in a drawing. The *Action* object provides a higher level interface for custom object selection processing.

The function returns an array of strings containing the names of the objects that overlap with the given rectangle completely or partially. Both the selected objects on the bottom of the hierarchy and all their named parents will be included. The objects in the array are listed in the reversed order compared to their drawing order, so that the objects that are at the bottom of the drawing list and are drawn on top of other objects will be listed first in the array.

The name string is a complete resource path name (relative to the *top_vp*) which can be used get the object ID of the object:

```
GlgObject selected_object = GlgGetResourceObject( top_vp, path_name );
```

This function may be called from a trace callback function to implement the custom handling of mouse events. The function returns NULL if no objects intersect the input rectangle, or if none of the intersected objects have names.

A more powerful *GlgCreateSelection* function returns object IDs of the selected objects and may be used to handle unnamed objects.

The array returned by this function must be dereferenced with the *GlgDropObject* function when the program is finished using it.

GlgCreateSelection

Returns a list of the objects intersecting a given rectangle.

```
GlgObject GlgCreateSelection( top_vp, rectangle, selected_vp )
   GlgObject top_vp;
   GlgRectangle * rectangle;
   GlgObject selected_vp;
```

Parameters

top_vp

The top viewport or light viewport of the selection query (must be an ancestor of *selected_vp*). It may be either a viewport or a light viewport.

rectangle

The bounding rectangle in screen coordinates of the *selected_vp* viewport. Any graphical shape whose rendering intersects this rectangle is included in the selection list.

selected_vp

The viewport or the light viewport relatively to which the bounding rectangle is defined. When used with the mouse events, this is the viewport in which the mouse event occurred.

This function is similar to the *GlgCreateSelectionNames* function, but it returns an array of objects instead of an array of object names. The objects in the array are listed in the reversed order compared to their drawing order, so that the objects that are at the bottom of the drawing list and are drawn on top of other objects will be listed first in the array.

While *GlgCreateSelectionNames* reports both the selected objects on the bottom of the hierarchy and all their parents, this function reports only objects at the bottom and doesn't include their parents. For example, if a polygon in a group is selected, only the polygon is reported and not the group. To get information about the parents of selected objects, use the *GlgGetParent* function.

The function returns NULL if no objects intersect the input rectangle. The returned array must be dereferenced with the *GlgDropObject* function when the program is finished using it.

GlgDropObject

Decrements an object's reference count.

```
void GlgDropObject( object )
   GlgObject object;
```

Parameters

object

Specifies the object to be dereferenced.

This function decreases the object's reference count by 1. If the reference count goes to 0, the object is destroyed. Destroying an object dereferences all its subsidiary objects and may destroy them as well if their reference counts become zero.

The *GlgDropObject* function should be used to dereference the object after creating, copying, loading or referencing the object, as was shown in the first coding example for the *GlgReferenceObject* function.

GlgDeleteTags

Deletes all object's data tags or all object's public properties. Returns *GlgTrue* on success.

```
GlgBoolean GlgDeleteTags( object, tag_type_mask )
   GlgObject object;
   GlgTagType tag_type_mask;
```

Parameters

object

Specifies the object to delete the tags from.

tag_type_mask

Specifies a bitwise mask that defines the type of tags to be deleted: data tags (GLG_DATA_TAG), public properties (GLG_EXPORT_TAG) or both.

GlgEnableAttachmentPoints

Controls the use of attachment points to determine object extents when the objects are aligned. Returns the previous state of the setting.

```
GlgBoolean GlgEnableAttachmentPoints( state )
   GlgBoolean state;
```

Parameters

state

Specifies if attachment points should be used in addition to the control points to determine object extents when the objects are aligned using control points.

GlgFindObject

A generic find function: finds an object in a container. Use *GlgContainsObject*, *GlgGetNamedObject*, *GlgGetElement* or *GlgGetIndex* convenience functions for specialized tasks.

```
GlgObject GlgFindObject( container, object, search_type, reserved )
   GlgObject container;
   void * object;
   GlgSearchType search_type;
   void * reserved;
```

Parameters

container

Specifies the container object.

object

Specifies the data to search for. Depending on the search type, it may be an object ID, object name, or object's index in the container's object list.

search_type

Specifies the search type and may have the following values:

GLG_FIND_OBJECT
Finds an object by the object ID (pass the object ID as the value of the *object* parameter).

GLG_FIND_BY_NAME
Finds an object by its name (pass the name as the value of the object parameter).

GLG_FIND_BY_INDEX
Finds an object by its position in the container object list (pass the zero-based position index as the value of the *object* parameter).

reserved

A reserved parameter for future extensions; must be NULL.

This function searches for an object based on the specified name, index, or object ID. If the object is found, the function modifies the container's internal position pointer to point to the found object and returns the object; otherwise it leaves the position pointer intact and returns NULL. The container's position pointer may be affected by other *GlgFindObject*, *GlgAddObject* and *GlgDeleteObject* calls. Any function using the position pointer (*GLG_CURRENT* access type) must be called immediately after the *GlgFindObject* call before any other call which may affect the pointer.

GlgFindObjects

Finds either one or all parents or children of the specified object that match the supplied criteria:

```
GlgBoolean GlgFindObjects( object, data )
   GlgObject object;
   GlgFindObjectsData * data;
```

Parameters

object

Specifies the object whose parents or children to search.

data

The structure that specifies search criteria and returns the search results:

```
typedef struct _GlgFindObjectsData
{
   GlgObjectMatchType match_type;
   GlgBoolean find_parents;
   GlgBoolean include_top_object;
   GlgBoolean find_first_match;
   GlgBoolean search_inside;
   GlgBoolean search_drawable_only;
   GlgBoolean search_templates;
   GlgBoolean dont_add_matches;
   GlgBoolean (*custom_match)( GlgObject object, void * custom_data );
   void * custom_data;
   GlgObjectType object_type;
   char * object_name;
   char * resource_name;
   GlgObject object_id;
   GlgObject found_object;
   GlgBoolean found_multiple;
} GlgFindObjectsData;
```

The structure has to be cleared by writing zeros before assigning values to its fields. This can be done using the *GlgInitStruct* macro defined in the *GlgApi.h* file as follows:

```
GlgFindObjectsData find_data;
GlgInitStruct( &find_data, sizeof( find_data ) );
```

The structure's fields control the way the search is performed:

match_type

Specifies a bitwise mask of the object matching criteria:

- GLG_OBJECT_TYPE_MATCH finds objects with an object type matching the type specified by the *object_type* field.
- GLG_OBJECT_NAME_MATCH finds objects with a name matching the name specified by the *object_name* field.
- GLG_RESOURCE_MATCH finds objects that have the resource specified by the *resource_name* field.
- GLG_OBJECT_ID_MATCH finds an object with the ID specified by the *object_id* field. This can be used to find out if the object supplied as the *object* parameter is a child or a parent of the object provided by *object_id*.
- GLG_CUSTOM_MATCH finds objects that match a custom matching criterion supplied by a custom function in the *custom_match* field.

Multiple criteria can be ORed together. All of the criteria in the mask must be true in order for an object to match.

find_parents

If set to *GlgTrue*, ancestors (parents, grandparents and so on) of the object are searched. Otherwise, the object's descendants (children, grandchildren and so on) are searched.

include_top_object

If set to *GlgTrue*, the top object passed as the *object* parameter will be included in the search results if it matches the search criteria, otherwise it will not be included.

find_first_match

If set to *GlgTrue*, the search is stopped after the first match and the first matching object is returned, otherwise all matching objects are returned.

search_inside

If set to *GlgFalse*, the object's children or parents are not searched if the object itself matches. Otherwise, the search proceeds to process the object's children or parents even if the object matches.

search_drawable_only

If set to *GlgTrue* when searching descendants, only drawable objects are searched. For example, when this flag is set, a polygon object's attributes such as *FillColor* and *EdgeColor*, will not be searched, which can be used to speed up the search. Refer to the description of the *GlgIsDrawable* method on page 162 for more information.

search_templates

If set to *GlgFalse* when searching descendants, the templates of series, subdrawings and subwindows will be excluded from the search, and only the rendered instances inside these objects will be included. This is a default setting, since the template objects are not drawn and therefore do not need to be processed.

This flag is ignored when the search is performed before hierarchy setup, when the instances have not yet been loaded, and only the templates are available.

dont_add_matches

If set to *GlgFalse*, *GlgFindObjects* returns matching objects in the *found_object* field of the *GlgFindObjectsData* structure. If multiple objects are found, the returned *found_object* will contain a GLG group that is created to hold the matching objects. This group is owned by the caller and has to be dereferenced with *GlgDropObject* when finished.

If the *custom_match* function processes objects in-place, as shown in the **SCADA Viewer example**, the returned group containing matching object is not used. In this case, the *dont_add_matches* field may be set to *GlgTrue* to prevent storing matching objects at all. This setting can be used only with the GLG_CUSTOM_MATCH search criterion. If *dont_add_matches* is set to *GlgTrue*, the returned *found_object* field will always be NULL.

custom_match

Provides a custom matching function that is invoked for each searched object. The function can perform custom matching and should return *GlgTrue* to indicate a match. The function will also receive custom data supplied by the *custom_data* field.

custom_data

Custom data for the *custom_match* function.

object_type

The type for the GLG_OBJECT_TYPE_MATCH search. Only objects of this type will be matched.

object_name

The name for the GLG_OBJECT_NAME_MATCH search. Only objects with this name will be matched.

resource_name

The resource name for the GLG_RESOURCE_MATCH search. Only objects that have this resource will be matched. The resource has to be an object.

object_id

The name for the GLG_OBJECT_ID_MATCH search. Only the object with this object ID will be matched. This type of search can be used to find out if an object is either a child or a parent of another object.

found_object

Returns a single found object or a group containing multiple found objects, according to the state of the *found_multiple* parameter. If a group containing multiple objects is returned, the group should be dereferenced using *GlgDropObject* to avoid memory leaks. It is set to NULL if no matching objects were found.

found_multiple

Will be set to *GlgTrue* or *GlgFalse* to indicate the type of the returned result. If it is set to *GlgTrue*, the result of the search stored in the *found_object* field contains a group of several matching objects. Otherwise (*GlgFalse*), *found_object* contains a single matching object.

The function returns *GlgTrue* if at least one matching object was found.

GlgFitObject

Fits an object to the specified rectangular area:

```
GlgBoolean GlgFitObject( object, coord_type, box, keep_ratio )
   GlgObject object;
   GlgCoordType coord_type;
   GlgCube * box;
   GlgBoolean keep_ratio;
```

Parameters

object

Specifies the object to fit.

coord_type

Specifies the coordinate system to interpret the fit area in. If GLG_SCREEN_COORD is used, the area is defined in screen pixels. If GLG_OBJECT_COORD is used, the area is defined in the GLG world coordinates.

box

The area to fit the object to. If the Z extent of the box is 0, 2D fitting is performed and the object is not scaled in the Z dimension. If the Z extent is not 0, 3D fitting is performed.

keep_ratio

If the parameter is set to *GlgTrue*, the function preserves the object's X/Y ratio by using the smallest of the required X and Y scale factors for both directions. If *keep_ratio* is set to *GlgFalse*, different scale factors may be used for X and Y scaling to fit the object to the box more precisely.

This function transforms the object to fit it to the area, calculating new control point values. The object's hierarchy must be setup to use this function.

The following types defined in the *GlgApi.h* file are used:

```
typedef struct _GlgCube
{
   GlgPoint p1; /* Lower left */
   GlgPoint p2; /* Upper right */
} GlgCube;
```

GlgGetAction

Returns the first found action with the matching parameters attached to the object.

```
GlgObject GlgGetAction( object, action_type, trigger_type, button,
            armed_state, double_click_state, action, subaction,
            enabled_only )
   GlgObject object;
   GlgActionType action_type,
   GlgTriggerType trigger_type,
   GlgLong button,
   GlgArmedStateType armed_state,
   GlgDoubleClickStateType double_click_state,
   char * action,
   char * subaction,
   GlgBoolean enabled_only
```

Parameters

object

Specifies the object whose alarms to query.

action_type

An action type. If zero (0), actions of any type will be considered.

trigger_type

A trigger type. If zero (0), actions with any trigger type will be considered.

button

A mouse button. If zero (0), actions that are activated by any mouse button will be considered.

armed_state

The setting of the action's ProcessArmed attribute.

double_click_state

The setting of an action's *DoubleClick* attribute.

action

The input action string for action objects with GLG_INPUT_TRIGGER trigger. If NULL, action objects with any input action string will be considered.

subaction

Specifies input subaction string for action objects with GLG_INPUT_TRIGGER trigger. If NULL, action objects with any input subaction string will be considered.

enabled_only

If True, only actions enabled by the settings of their *Enabled* attribute will be considered.

GlgGetAlarmObject

Returns a tag object with a specified alarm label, or a list of alarms matching the specified alarm label pattern.

```
GlgObject GlgGetAlarmObject( object, search_string, single_alarm,
            reserved )
   GlgObject object;
   char * search_string;
   GlgBoolean single_alarm;
   GlgLong reserved;
```

Parameters

object

Specifies the object whose alarms to query.

search_string

Specifies a string or a pattern for the search (may include *?* and * wildcards).

single_alarm

Defines a single alarm (*GlgTrue*) or multiple alarm (*GlgFalse*) mode.

reserved

Reserved for future use, must be 0.

In a single alarm mode, the function returns the data object that has an attached alarm with an alarm label matching the search criteria.

In the multiple alarm mode, the function returns a group containing all data objects that have alarms with matching alarm labels attached to the objects. The returned group must be dereferenced using *GlgDropObject* when it is no longer needed.

GlgGetBoxPtr

Returns an object's bounding box.

```
GlgCube * GlgGetBoxPtr( object )
   GlgObject object;
```

Parameters

object

Specifies the object whose bounds are to be returned.

Returns a pointer to the object's 3-D bounding box. The box boundaries are in absolute screen coordinates (pixels) and are valid only after the object has been drawn. (That is, after the hierarchy has been set up.) The returned pointer points to internal structures which are valid only while the object exists and should not be modified. The object box may be converted to world coordinates by using the *GlgGetDrawingMatrix*, *GlgCreateInversedMatrix* and *GlgTransformPoint* functions.

The following types defined in the *GlgApi.h* file are used:

```
typedef struct _GlgPoint
{
   double x, y, z;
} GlgPoint;

typedef struct _GlgCube
{
   GlgPoint p1; /* Lower left */
   GlgPoint p2; /* Upper right */
} GlgCube;
```

GlgGetDrawingMatrix

Returns the transformation matrix used to draw an object.

```
GlgObject GlgGetDrawingMatrix( object )
   GlgObject object;
```

Parameters

object

Specifies the object to query.

This function returns a matrix object that combines matrices of all transformations used to draw the object. Transformation matrices are only used for geometric transformations, so this function may be called only for graphical objects, like polygons, arcs and text objects. This function must be called after the object has been drawn. (That is, after its hierarchy has been set up.) The function returns the internal matrix object which is valid only immediately after the function call. The matrix should not be modified or destroyed. To prevent the matrix from being destroyed, you may reference it with *GlgReferenceObject* (and dereference later), or, even better, create a copy of it.

Querying the drawing matrix of a viewport returns the drawing matrix used to position the viewport's control points, and not the matrix used to render the objects inside it. To get the drawing matrix the viewport uses to draw objects inside it, query the drawing list of the viewport (the *Array* resource of the viewport) and use it as a parameter of the *GlgGetDrawingMatrix* function.

GlgGetElement

Returns the object at the specified position in a container.

```
GlgObject GlgGetElement( container, index )
   GlgObject container;
   GlgLong index;
```

Parameters

container

Specifies the container object.

index

Specifies the object position inside the container.

This function returns NULL if the specified index is invalid. While the function provides the Intermediate API functionality, it is also available in the Standard API when used with the group object returned by *GlgCreateTagList*.

GlgGetIndex

Returns the index of the first occurrence of an object in a container.

```
GlgLong GlgGetIndex( container, object)
   GlgObject container;
   GlgObject object;
```

Parameters

container

Specifies the container object.

object

Specifies the object to search for.

This function returns -1 if the object wasn't found in the container.

GlgGetMatrixData

Queries matrix coefficients:

```
void GlgGetMatrixData( matrix, matrix_data )
   GlgObject matrix;
   GlgMatrixData * matrix_data;
```

Parameters

matrix

Specifies the matrix object to query.

matrix_data

The structure to hold the matrix's coefficients.

The *GlgMatrixData* data type is defined in the *GlgApi.h* file:

```
typedef struct _GlgMatrixData
{
   long type;
   double matrix[4][4];
} GlgMatrixData;
```

GlgGetNamedObject

Returns an object ID of the object in a container with the specified name.

```
GlgObject GlgGetNamedObject( container, name )
   GlgObject container;
   char * name;
```

Parameters

container

Specifies the container object.

name

Specifies the object's name.

This function returns named objects of a container with no regards to the resource hierarchy, and is different from the *GlgGetResourceObject* function (which returns named resources on the current level of the resource hierarchy but not necessarily the elements of the container).

This function returns NULL if the object with the specified name wasn't found in the container.

GlgGetObjectData

Retrieves and returns custom data attached to the object via *GlgSetObjectData*:

```
void * GlgGetGetObjectData( object )
   GlgObject object;
```

Parameters

object

An object to retrieve the data from.

GlgGetObjectViewport

Returns a parent viewport of a drawable GLG object.

```
GlgObject GlgGetObjectViewport( object, heavy_weight, self)
   GlgObject object;
   GlgBoolean heavy_weight;
   GlgBoolean self;
```

Parameters

object

A drawable GLG object.

heavy_weight

Controls the type of a parent viewport to be returned. If set to *GlgFalse*, the method returns the closest parent viewport regardless of its type: either a heavyweight or light viewport. If set to *GlgTrue*, it returns the closest heavyweight parent viewport. For example, if the *object* is inside a light viewport, it will return the heavyweight viewport which is a parent of the light viewport.

self

If set to *GlgTrue*, the function returns the *object* itself if the *object* is a viewport or a light viewport and *heavy_weigth* is *GlgFalse*, or if the *object* is a viewport and *heavy_weight* is *GlgTrue*.

The object's hierarchy must be set up to use this function.

GlgGetParent

Returns an object's parent object, if one exists, or an array of all parents for constrained data objects.

```
GlgObject or ( object, num_parents )
   GlgObject object;
   GlgLong * num_parents;
```

Parameters

object

Specifies the object whose parents are to be returned.

num_parents

A pointer used to return the number of *object*'s parents. If it is NULL, an error message will be generated if *object* has more than one parent.

This function returns an ID for an object's parent object. For constrained data objects, there may be more than one parent object. In this case, the function returns an array of parent object IDs and sets *num_parents*, which may be used to check whether a data object is constrained. NULL is returned if *object* has no parents or in case of an error.

The returned object ID is a pointer to an internal data structure and it should not be dereferenced with *GlgDropObject*. To keep it from being destroyed when the parent object is destroyed, you can reference it with *GlgReferenceObject* (and dereference later). If the return value is an array of parents and you want to keep it around, the safer method to keep it is to create a copy. This prevents the array's elements from being modified by the GLG internals.

This function must be called after the hierarchy is created.

GlgGetParentViewport

Returns a parent viewport of a drawable GLG object.

```
GlgObject GlgGetParentViewport( object, heavy_weight )
   GlgObject object;
   GlgBoolean heavy_weight;
```

Parameters

object

A drawable GLG object.

heavy_weight

Controls the type of a parent viewport to be returned. If set to *GlgFalse*, the method returns the closest parent viewport regardless of its type: either a heavyweight or light viewport. If set to *GlgTrue*, it returns the closest heavyweight parent viewport. For example, if the *object* is inside a light viewport, it will return the heavyweight viewport which is a parent of the light viewport.

If the *object* is a top level viewport that has no parent, NULL is returned.

The object's hierarchy must be setup to use this function, otherwise an error message is generated and NULL is returned.

GlgGetResourceObject

Returns the ID of a resource object.

```
GlgObject GlgGetResourceObject( parent, resource_name )
   GlgObject parent;
   char * resource_name;
```

Parameters

parent

Specifies the object whose resources are queried.

resource_name

A string that specifies a resource path to access the resource object inside the *parent*.

This function finds and returns a named resource of the *parent* object or an object ID of its attribute. It cannot be used to access attributes that are not objects, such as an object's *Name*.

If the resource is not found, the function returns NULL. No error message is generated in this case, so this function can be used to query an object's existence.

The object ID returned by the function is not referenced and is valid only immediately after the function call. Reference the object if the resource ID has to be stored for later use to make sure the object is not destroyed.

GlgGetSize

Queries the size of a container object.

```
GlgLong GlgGetSize( container )
   GlgObject container;
```

Parameters

container

Specifies the container object.

This function returns the size of a container object. For a viewport or group object, the size is the number of objects in the viewport or group. For a polygon, the size is the number of vertices. While the function provides the Intermediate API functionality, it is also available in the Standard API when used with a group object returned by *GlgCreateTagList*.

GlgGetStringIndex

Returns the index of the first occurrence of the string in a string container.

```
GlgLong GlgGetStringIndex( container, string)
   GlgObject container;
   char * string;
```

Parameters

container

Specifies the container object.

string

Specifies the string to search for.

This function returns -1 if the string wasn't found in the container.

GlgQueryTags

Returns a list of all tags of the requested tag type defined in the object.

```
GlgObject GlgQueryTags( object, tag_type_mask)
   GlgObject object;
   GlgTagType tag_type_mask;
```

Parameters

object

Specifies the object whose tags to query.

tag_type_mask

Defines the type of tags to query: data tags or export tags. For export tags, tag type constants may be *OR*ed to query multiple subtypes of export tags.

The function returns a list of all attribute objects that have tags of the specified type attached. The returned group object must be dereferenced using *GlgDropObject* when it is no longer needed.

Each element of the returned group is an attribute object that has a tag attached. The *GlgGetElement* and *GlgGetSize* functions may be used to handle the returned group object.

GlgHasTag

Checks if a data or export tag with a specified tag name exists. Returns True if a tag with the given *TagName* exists, otherwise returns False without generating an error messag.

```
GlgBoolean GlgHasTag( object, tag_name, single_tag, tag_type_mask)
   GlgObject object;
   char * tag_name;
   GlgTagType tag_type_mask;
```

Parameters

object

Specifies the object whose tags to query.

tag_name

Specifies the *TagName* to query, supporting wildcards: ? (matches any single character) and * (matches any sequence of characters).

tag_type_mask

Defines the type of tags to query: data tags or export tags. For export tags, tag type constants may be *OR*ed to query multiple subtypes of export tags.

GlgGetTagObject

Returns a tag object with a specified tag name, data tag source, export tag category or a list of tags matching the specified tag name, tag source or category pattern.

```
GlgObject GlgGetTagObject( object, search_string, by_name,
            unique_tags, single_tag, tag_type_mask)
   GlgObject object;
   char * search_string;
   GlgBoolean by_name;
   GlgBoolean unique_tags;
   GlgBoolean single_tag;
   GlgTagType tag_type_mask;
```

Parameters

object

Specifies the object whose tags to query.

search_string

Specifies a string or a pattern for the search (may include *?* and * wildcards).

by_name

If set to *GlgTrue*, the search is performed by matching the tag names, otherwise the search is done by matching the tag sources for data tags or matching the tag categories for export tags.

unique_tags

If set to *GlgTrue*, only the first encountered tag of the highest priority will be listed when several data tags with the same tag name or tag source exist in the drawing. Input tags are prioritized over output tags, and enabled tags over disabled tags. If set to *GlgFalse*, all tags matching the search criteria will be included in the returned list. The parameter is ignored in the single tag mode and when searching for export tags.

single_tag

Defines the single tag (*GlgTrue*) or multiple tag (*GlgFalse*) mode. In the single tag mode, the first found tag object of the highest priority matching the search criteria will be returned. Input tags are prioritized over output tags, and enabled tags over disabled tags.

tag_type_mask

Defines the type of tags to query: data tags or export tags. For export tags, tag type constants may be *OR*ed to query multiple subtypes of export tags.

In a single tag mode, the function returns the first attribute object that has an attached tag matching the search criteria.

In the multiple tag mode, a group containing all attribute objects with matching tags will be returned. The *unique_tags* controls if only one tag is included in case there are multiple data tags with the same tag name or tag source. It is ignored in the single tag mode and when searching for export tags. The returned group must be dereferenced using *GlgDropObject* when it is no longer needed.

GlgInverse

Inverses the order of elements inside a container object.

```
void GlgInverse( container )
   GlgObject container;
```

Parameters

container

Specifies the container object.

GlgIsAt

Checks the position of the iteration pointer (the current element of the container).

```
GlgBoolean GlgIsAt( container, position )
   GlgObject object;
   GlgPositionType position;
```

Parameters

container

Specifies the container object.

position

Specifies the position that is being checked using one of the following constants:

- for GLG_START, True is returned if the iteration pointer is positioned before the first element of the container
- for GLG_END, True is returned if the iteration pointer is positioned past the last element of the container
- for GLG_FIRST, True is returned if the iteration pointer is positioned at the first element of the container
- for GLG_LAST, True is returned if the iteration pointer is positioned at the last element of the container.

GlgIsDrawable

Returns *GlgTrue* if the object is drawable.

```
GlgBoolean GlgIsDrawable( object )
   GlgObject object;
```

Parameters

object

Specifies the object.

Drawable objects can be placed directly in a drawing area, have control points and a bounding box, have the *Visibility* attribute and can contain custom properties, actions and aliases.

For a chart object, the chart itself is drawable, but its plots are not.

GlgIterate
GlgIterateBack

Traverses the container object forward or backward.

```
GlgObject GlgIterate( container )
   GlgObject container;

GlgObject GlgIterateBack( container )
   GlgObject container;
```

Parameters

container

Specifies the container object.

For the forward traversing, *GlgIterate* advances the iteration pointer and returns the next element of *container*. For backward traversing, *GlgIterateBack* decrements the iteration pointer and returns the previous element.

The *GlgSetStart* function must be used to initialize the container for iterating before *GlgIterate* is invoked the first time, and *GlgSetEnd* must be used to initialize the container for backward iteration with *GlgIterateBack*. The add, delete and search operations affect the iteration state and should not be performed on the container while it is being iterated.

To perform a safe iteration that is not affected by the add and delete operations, a copy of the container can be created using the *GlgCloneObject* function with the GLG_ SHALLOW_CLONE clone type. The shallow copy will return a container with a list of objects IDs, and it can be safely iterated while objects are added or deleted from the original container.

Alternatively, *GlgGetElement* can be used to traverse elements of the container using an element index, which is not affected by the search operations on the container.

The following coding example shows how to iterate all objects in a container in the forward and backward direction using *GlgIterate* and *GlgIterateBack*:

```
int i, size;
GlgObject object;

size = GlgGetSize( container );
if( size != 0 )
{
   GlgSetStart( container );     /* Initialize direct traversing. */
   for( i=0; i<size; ++i )
   {
      object = GlgIterate( container );
      ... /* Code to process the object */
   }

   GlgSetEnd( container );    /* Initialize for backward traversing. */
   for( i=0; i<size; ++i )
   {
      object = GlgIterateBack( container );
      ... /* Code to process the object */
   }
}
```

GlgLayoutObjects

Performs operations ranging from setting the object's width and height to aligning and layout of groups of objects.

```
GlgBoolean GlgLayoutObjects( object, sel_elem, type, distance,
                             use_box, process_subobjects )
   GlgObject object;
   GlgObject sel_elem;
   GlgLayoutType type;
   double distance;
   GlgBoolean use_box;
   GlgBoolean process_subobjects;
```

Parameters

object

Specifies the object or the group of objects to perform the requested operations upon.

sel_elem

Specifies the anchor object for alignment operations. If *null* value is specified, the first encountered object in the specified alignment direction is used.

type

Specifies the type of the layout action to perform. May have the following values:

ALIGN_LEFT

Aligns the left edge of elements within the group with the left edge of the anchor element.

ALIGN_RIGHT

Aligns the right edge of elements within the group with the right edge of the anchor element.

ALIGN_HCENTER

Aligns the center of elements within the group with the center of the anchor element horizontally.

ALIGN_TOP

Aligns the top edge of elements within the group with the top edge of the anchor element.

ALIGN_BOTTOM

Aligns the bottom edge of elements within the group with the bottom edge of the anchor element.

ALIGN_VCENTER

Aligns the center of elements within the group with the center of the anchor element vertically.

SET_EQUAL_VSIZE

Sets the height of elements within the group to the height of the anchor.

SET_EQUAL_HSIZE

Sets the width of elements within the group to the width of the anchor.

SET_EQUAL_SIZE

Sets the width and height of elements within the group to the width and height of the anchor.

SET_EQUAL_VDISTANCE

Equally distributes the group's elements in vertical direction as measured by the distance between their centers.

SET_EQUAL_HDISTANCE

Equally distributes the group's elements in horizontal direction as measured by the distance between their centers.

SET_EQUAL_VSPACE

Equally distributes space gaps between group's elements in vertical direction.

SET_EQUAL_HSPACE

Equally distributes space gaps between group's elements in horizontal direction.

SET_VSIZE

Sets the height of the *object* or of all objects in the *object* group to the value specified by the *distance* parameter:

- if *process_subobjects=False*, sets the *object*'s height
- if *process_subobjects=True*, sets the height of all elements within the group.

SET_HSIZE

Sets the width of the *object* or of all objects in the *object* group to the value specified by the *distance* parameter:

- if *process_subobjects=False*, sets the *object*'s width
- if *process_subobjects=True*, sets the width of all elements within the group.

SET_VDISTANCE

Set the vertical distance between centers of the group's elements to a value specified by the distance parameter.

SET_HDISTANCE

Set the horizontal distance between centers of the group's elements to a value specified by the *distance* parameter.

SET_VSPACE

Sets space gaps between group's elements in vertical direction to a value specified by the *distance* parameter.

SET_HSPACE

Sets the space gaps between the group's elements in horizontal direction to a value specified by the *distance* parameter.

distance

The distance in screen coordinates for positioning objects, it is interpreted depending on the layout action.

use_box

Controls how the objects extent is determined:

- *GlgTrue* to use the *object*'s real bounding box to align objects
- *GlgFalse* to use the *object*'s control points to determine the *object*'s extent, which could be different from the real bounding box for objects such as a *Text* object with a single control point.

process_subobjects

Controls how to apply the layout action if the object is a group. If set to *GlgTrue*, the action is applied to all objects contained in the group, otherwise the action (such as SET_HSIZE) is applied to the group itself.

The function returns *GlgFalse* if errors were encountered during the requested layout action.

GlgLoadObject

Loads an object from a GLG drawing file or URL.

```
GlgObject GlgLoadObject( filename )
   char * filename;
```

Parameters

filename

Specifies the name of the file or URL to load the object from. For details of using URLs on Linux, refer to the description of the *GlgGetURLCB* function on page 59.

This function loads an object from the specified file or URL and returns the object ID. If the file is not accessible or is not in the GLG save format, an error message is generated and NULL is returned.

If the object is loaded successfully, the reference count of the returned object is set to 1. The loaded object has to be explicitly dereferenced after it has been used to avoid memory leaks. Refer to the description of the *GlgReferenceObject* function on page 136 for details on the reference counting.

GlgLoadObjectFromImage

Loads an object from a memory image created by the GLG Code Generation Utility.

```
GlgObject GlgLoadObjectFromImage( image_address, image_size )
   void * image_address;
   GlgLong image_size;
```

Parameters

image_address

Specifies the address of the drawing image generated by the GLG Code Generation Utility.

image_size

Specifies the image size.

This function loads an object from the specified drawing's memory image and returns the object ID. If the memory is corrupted, an error message is generated and NULL is returned. If the object is loaded successfully, the reference count of the returned object is set to 1. The loaded object must be explicitly dereferenced after it has been used to avoid memory leaks. Refer to the description of the *GlgReferenceObject* function for details on the reference counting. For information about the Code Generation Utility, see the *GLG Programming Tools and Utilities* chapter.

GlgMoveObject

Moves an object by a move vector:

```
GlgBoolean GlgMoveObject( object, coord_type, start_point, end_point )
   GlgObject object;
   GlgCoordType coord_type;
   GlgPoint * start_point;
   GlgPoint * end_point;
```

Parameters

object

Specifies the object to be moved.

coord_type

Specifies the coordinate system for interpreting the move vector. If GLG_SCREEN_COORD is used, the move vector is in screen pixels. For example, this may be used to move the object by the number of screen pixels as defined by the mouse move. If GLG_OBJECT_COORD is used, the move vector is in the GLG world coordinates.

start_point

The start point of the move vector. If the parameter is NULL, (0,0,0) is used as the start point.

end_point

The end point of the move vector.

This function moves an object's control points by the distance defined by the move vector, calculating new control point values. The object's hierarchy must be setup to use this function.

The *GlgPoint* data type is defined in the *GlgApi.h* file:

```
typedef struct _GlgPoint
{
   double x, y, z;
} GlgPoint;
```

GlgMoveObjectBy

Moves an object by the specified distance:

```
GlgBoolean GlgMoveObjectBy( object, coord_type, x, y, z )
   GlgObject object;
   GlgCoordType coord_type;
   double x, y, z;
```

Parameters

object

Specifies the object to move.

coord_type

Specifies the coordinate system for interpreting the move distance. If GLG_SCREEN_COORD is used, the move distance is in screen pixels. For example, this may be used to move the object by the number of screen pixels as defined by the mouse move. If GLG_OBJECT_COORD is used, the move distance is specified in the GLG world coordinates.

x, y, z

The X, Y and Z move distances.

This function moves an object's control points by the X, Y and Z distances, calculating new control point values. The object's hierarchy must be setup to use this function.

GlgPositionObject

Positions an object at the specified location:

```
GlgBoolean GlgPositionObject( object, coord_type, anchoring, x, y, z )
   GlgObject object;
   GlgCoordType coord_type;
   GlgLong anchoring;
   double x, y, z;
```

Parameters

object

Specifies the object to position.

coord_type

Specifies the coordinate system for interpreting the position. If GLG_SCREEN_COORD is used, the position is defined in screen pixels. If GLG_OBJECT_COORD is used, the position is defined in the GLG world coordinates.

anchoring

Specifies what part of the object's bounding box will be anchored at the specified position. It's formed as a bitwise "or" of the horizontal anchoring constant (*GLG_HLEFT*, *GLG_HCENTER* or *GLG_HRIGHT*) with the vertical anchoring constants (*GLG_VTOP*, *GLG_VCENTER* or *GLG_VBOTTOM*). For example, using *(GLG_VTOP | GLG_HLEFT)* causes the upper left corner of the object's bounding box to be positioned at the specified location.

x, y, z

The X, Y and Z coordinates of the desired object position.

This function positions the object, calculating new control point values. The object's hierarchy must be setup to use this function.

GlgReferenceObject

Increments an object's reference count.

```
GlgObject GlgReferenceObject( object )
   GlgObject object;
```

Parameters

object

Specifies the object to be referenced.

This function increases the reference count of an object by 1 and returns the object's ID. The *GlgDropObject* function may be used to dereference the object when it is not needed any more.

When an object is created, its reference count is set to 1. Any object you create programmatically has to be explicitly dereferenced using the *GlgDropObject* function after the object has been used, otherwise memory leaks will occur. It is a good practice to dereference objects immediately after they have been added to a group or used as a part of other objects; this guarantees that you will not forget to dereference the object later. The following example illustrates this:

```
{
   GlgObject polygon;

   /* Create a polygon object with default values of the attributes.
         */
   polygon = GlgCreateObject( GLG_POLYGON, NULL, NULL, NULL, NULL,
         NULL );

   /* Add the polygon to a group. */
   GlgAddObjectToBottom( group, polygon );

   /* Adding polygon to a group references it; we may dereference
         it now to let the group manage it.
    */
   GlgDropObject( polygon );
}
```

Dereferencing an object does not necessarily destroy the object. The object may still be referenced by groups it was added to or by other objects that use it. An object may also be referenced programmatically to make sure it is kept around and is not destroyed automatically by the Toolkit. The object is actually destroyed only when the last reference to it is dropped.

You can use the *GlgReferenceObject* function to keep objects around, and to make sure that an object ID is still valid. Simply use the function to increment the reference count of the object, and then decrement the count with the *GlgDropObject* function when the object is no longer needed.

The *GlgReferenceObject* function may also be used to keep an object while it is being deleted from one group and added to another group. Deleting the object from a group decrements its reference count and may destroy the object if it is not referenced by anything else. Referencing the object using *GlgReferenceObject* prevents the object from being destroyed. The *GlgDropObject* function is used after the operation is completed to return the object's reference count to its initial state.

The following code fragment illustrates this:

```
GlgReferenceObject( object );   /* Keeping the object around. */

/* Deleting the object from group1 automatically dereferences the
   object and may destroy it if it's not referenced.
 */
if( GlgContainsObject( group1, object ) )
   GlgDeleteThisObject( group1, object );

/* Adding the object to the group2 automatically references it, so
   that it is safe to dereference it now.
 */
GlgAddObjectToBottom( group2, object );

GlgDropObject( object );      /* We may drop it now. */
```

In general, it is good practice to increment the reference count of an object before performing any operation on it and then dereference the object when the operation is complete. This provides a guarantee that the object will not be inadvertently destroyed during the operation.

GlgReleaseObject

Releases an object that was suspended for editing.

```
void GlgReleaseObject( object, suspend_info )
   GlgObject object;
   GlgObject suspend_info;
```

Parameters

object

Specifies the object to be released after it was suspended for editing.

suspend_info

Suspension information returned by the previous call to the *GlgSuspendObject* function.

This function releases previously suspended object and is intended to be used only in conjunction the *GlgSuspendObject* function.

GlgReorderElement

Changes the object's position inside a container.

```
void GlgReorderElement( container, old_index, new_index )
   GlgObject object;
   GlgLong old_index;
   GlgLong new_index;
```

Parameters

container

Specifies the container object.

old_index

Specifies the index of an object to be reordered.

new_index

Specifies a new position for the object.

This function moves the object at the *old_index* to the *new_index* position. It generates an error message if indices are out of range.

GTK Note: GTK doesn't have a notion of windows' Z order. While the GLG driver tries to maintain the window order as much as possible, windows' Z order may not be correct when a mix of the *DrawingArea* viewports and native widget viewports (such as a text entry, a combo box, etc.) is displayed in the GTK version of the Toolkit on Linux. This may be the reason why viewports reordered with *GlgReorderElement* do not appear in the requested Z order.

GlgRootToScreenCoord

Converts screen coordinates relative to the root window to the screen coordinates in the specified viewport.

```
GlgBoolean GlgRootToScreenCoord( viewport, point )
   GlgObject viewport;
   GlgPoint2 * point;
```

Parameters

viewport

Specifies the viewport whose screen coordinate system to convert to.

point

The point to be converted. The result is placed back into the structure pointed by this pointer.

The viewport's hierarchy must be set up to use this function.

The *GlgPoint2* data type is defined in the *GlgApi.h* file:

```
typedef struct _GlgPoint2
{
   double x, y;
} GlgPoint2;
```

GlgRotateObject

Rotates an object:

```
GlgBoolean GlgRotateObject( object, coord_type, center, x, y, z )
   GlgObject object;
   GlgCoordType coord_type;
   GlgPoint * center;
   double x, y, z;
```

Parameters

object

Specifies the object to rotate.

coord_type

Specifies the coordinate system for interpreting the rotation center. If GLG_SCREEN_COORD is used, the center is defined in screen pixels. If GLG_OBJECT_COORD is used, the center is defined in the GLG world coordinates.

center

The center of rotation. If the parameter is NULL, the object is rotated relative to the center of its bounding box.

x, y, z

The X, Y and Z rotation angles.

This function rotates an object's control points by the specified rotation angles and relative to the specified center, calculating new control point values. If more than one rotation angle is specified, X rotation is applied the first, then Y and Z. The object's hierarchy must be setup to use this function.

The *GlgPoint* data type is defined in the *GlgApi.h* file:

```
typedef struct _GlgPoint
{
   double x, y, z;
} GlgPoint;
```

GlgSaveObject

Saves an object into a file.

```
GlgBoolean GlgSaveObject( object, filename )
   GlgObject object;
   char * filename;
```

Parameters

object

Specifies the object to be saved; it may be any GLG object.

filename

Specifies the name of the file to save the object into.

This function saves the specified object and all its subsidiary objects into a file. If the file is not accessible for writing, the function returns *GlgFalse*, otherwise it returns *GlgTrue*. A GLG object of any type may be saved in this fashion and later loaded using the *GlgLoadObject* function. You can use the *GlgSaveFormat* configuration resource described the *Appendices* chapter to modify the save type.

For applications that load a drawing, modify it and save if back into a file, the drawing should be loaded using the *GlgLoadObject* function instead of *GlgLoadWidget*. *GlgLoadWidget* extracts the *$Widget* viewport from the loaded drawing, discarding the rest of the drawing, while *GlgLoadObject* returns an object that represents the whole drawing. Using *GlgSaveObject* to save a viewport loaded with *GlgLoadWidget* will result in a loss of information contained in the discarded part of the drawing, such as the type of the coordinate system used to display the viewport, which will make it difficult to load and edit the drawing in the Builder. The following example demonstrates the proper technique:

```
GlgObject drawing = GlgLoadObject( "drawing.g" );
GlgObject viewport = GlgLoadWidgetFromObject( drawing ); /* Extracts $Widget viewport */
... /* Code that modifies the drawing. */
GlgSaveObject( drawing, "new_drawing.g" );
```

GlgScaleObject

Scales an object:

```
GlgBoolean GlgScaleObject( object, coord_type, center, x, y, z )
   GlgObject object;
   GlgCoordType coord_type;
   GlgPoint * center;
   double x, y, z;
```

Parameters

object

Specifies the object to scale.

coord_type

Specifies the coordinate system for interpreting the scale *center*. If GLG_SCREEN_COORD is used, the center is defined in screen pixels. If GLG_OBJECT_COORD is used, the center is defined in the GLG world coordinates.

center

The center of scaling. If the parameter is NULL, the object is scaled relative to the center of its bounding box.

x, y, z

The X, Y and Z scaling factors. Use the same value for all three factors to scale the *object* uniformly in all dimensions.

This function scales an object's control points by the specified scale factors and relative to the specified center, calculating new control point values. The object's hierarchy must be setup to use this function.

The *GlgPoint* data type is defined in the *GlgApi.h* file:

```
typedef struct _GlgPoint
{
   double x, y, z;
} GlgPoint;
```

GlgScreenToWorld

Converts a point's coordinates from the screen to the GLG world coordinate system.

```
GlgBoolean GlgScreenToWorld( object, inside_vp, in_point, out_point )
   GlgObject object;
   GlgBoolen inside_vp;
   GlgPoint * in_point;
   GlgPoint * out_point;
```

Parameters

object

Specifies the object whose world coordinate system to convert to. The world coordinate system of an object includes the effect of all drawing transformations attached to the object and all its parents.

inside_vp

If *object* is a viewport or a light viewport, *inside_vp* specifies which world coordinate system to use. If *inside_vp=GlgTrue,* the method will use the world coordinate system used to draw objects inside this viewport *object*. If *inside_vp=GlgFalse*, the method will use the world coordinate system used to draw this viewport *object* inside its parent viewport.

The value of this parameter is ignored for objects other than a viewport or light viewport.

in_point

The point to be converted.

out_point

The point structure to hold the converted value.

The object's hierarchy must be setup to use this function.

The *GlgPoint* data type is defined in the *GlgApi.h* file:

```
typedef struct _GlgPoint
{
   double x, y, z;
} GlgPoint;
```

When converting the cursor position to world coordinates, add GLG_COORD_MAPPING_ADJ to the cursor's x and y screen coordinates for precise mapping.

GlgSerialize

Saves a GLG object into a data stream by invoking a custom serialization callback, which can be used to implement custom save methods.

```
GlgBoolean GlgSerialize( object, func, user_data )
   GlgObject object;
   GlgBoolean (*func)( void * data, GlgLong size, void * user_data );
   void * user_data;
```

Parameters

object

Specifies the object to be saved; it may be any GLG object.

func

Specifies a custom serialization callback function that will be invoked to save the object. The function will be invoked repeatedly with the *data* and *size* parameters supplying the pointer to the data that needs to be saved and the size of the data in bytes. The *user_data* parameter will supply *user_data* passed to the *GlgSerialize* function.

At the end of the serialization process, the function will be invoked with *data=NULL* and *size=-1* to flush any remaining data. The function must return *GlgTrue* to continue, or *GlgFalse* to abort serialization, in which case *GlgSerialize* will return *GlgFalse* as well.

user_data

Specifies the user data to be passed to the callback function.

GlgSerializeFull

Saves a GLG object into a raw memory block.

```
void * GlgSerializeFull( object, GlgLong * size )
```

Parameters

object

Specifies the object to be saved; it may be any GLG object.

size

A pointer to a variable that will be set to the size of the returned memory block.

The function returns a pointer to the memory block containing the saved object image, or NULL on error. The size of the memory block is returned via the *size* parameter.

GlgSetCustomSetupHandler

Installs a custom setup handler that will be invoked to perform custom processing of objects tagged using the *CustomSetup* flag in the GLG drawing:

```
void GlgSetCustomSetupHandler( handler )
   GlgCustomSetupHandler handler;
```

Parameters

handler

Specifies the custom setup handler function to be invoked.

A custom setup handler is a function which is invoked at different times during the object lifetime of objects tagged using the *CustomSetup* flag. The handler is invoked for each tagged object with the object passed as a parameter. The handler can perform custom processing based on the information stored in the object.

A custom setup handler has the following function prototype:

```
typedef void (*GlgCustomSetupHandler)( GlgObject object,
                                       GlgCustomSetupType type )
```

A custom setup handler is invoked with the following parameters:

object

A tagged object to be processed.

type

The phase when the handler is invoked, which may be one of the following values:

GLG_BEFORE_HI_SETUP

Before an object's hierarchy setup.

GLG_AFTER_HI_SETUP

Before an object's hierarchy setup.

GLG_FINISHED_SETUP

After finishing an object's setup, when all transformed values are valid.

GLG_BEFORE_RESET_SETUP

Before an object's reset.

GLG_AFTER_RESET_SETUP

After an object's reset.

The execution order of the custom setup callback relative to the hierarchy callback of the containing object depends on the invocation type:

- For GLG_BEFORE_HI_SETUP and GLG_BEFORE_RESET_SETUP, the custom setup callback is triggered **after** the corresponding hierarchy callbacks of the containing object (including the object itself, if it has a hierarchy callback attached).
- For GLG_AFTER_HI_SETUP, GLG_FINISHED_SETUP, and GLG_AFTER_RESET_SETUP, the callback is triggered **before** those corresponding hierarchy callbacks.

The custom setup may be used to automate object data tags assignment based on the object ID, or to attach *Input* and *Trace* callbacks to objects representing custom widgets that handle user interaction. The **SCADA Viewer example** in the GLG installation directory provides a source code of an application framework that demonstrates the use of custom setup handler to process special custom widgets.

GlgSetMatrixData

Sets matrix coefficients:

```
void GlgSetMatrixData( matrix, matrix_data )
   GlgObject matrix;
   GlgMatrixData * matrix_data;
```

Parameters

matrix

Specifies the matrix object.

matrix_data

The new matrix's coefficients.

The *GlgMatrixData* data type is defined in the *GlgApi.h* file:

```
typedef struct _GlgMatrixData
{
   long type;
   double matrix[4][4];
} GlgMatrixData;
```

GlgSetObjectData

Attaches arbitrary data to a graphical object:

```
void GlgSetSetObjectData( object, data )
   GlgObject object;
   void * data;
```

Parameters

object

An object the data will be attached to.

data

The data that will be attached.

GlgSetStart
GlgSetEnd

Initializes the container object for the forward or backward traversing.

```
void GlgSetStart( container )
   GlgObject container;
```

```
void GlgSetEnd( container )
   GlgObject container;
```

Parameters

container

Specifies the container object.

GlgSetStart sets the current position pointer before the first element of the container, preparing the container object for forward traversing with *GlgIterate*.

GlgSetEnd sets the current position pointer past the last element of the container, preparing it for backward traversing with *GlgIterateBack*.

GlgSuspendObject

Suspends an object for editing.

```
GlgObject GlgSuspendObject( object )
   GlgObject object;
```

Parameters

object

Specifies an object to be suspended.

Some operations, such as attaching a transformation to an object, constraining the object's attribute and some others, may be carried out only before the object has been drawn. If the object has been drawn and you still need to perform these operations, the *GlgSuspendObject* function must be used to suspend the object for editing. If the *GlgSuspendObject* function is not used, any attempt to perform these modifications will fail, producing an error message.

The *GlgSuspendObject* function may be called only for the graphical objects, such as a viewport, group, polygon, or others.

The *GlgReleaseObject* function must be called after the editing operations have been completed. If an object is suspended, it is not allowed to call an update function before the object has been released.

The *GlgSuspendObject* function returns an object which keeps suspension information. This information should be passed to the *GlgReleaseObject* function when the object is released.

GlgTransformObject

Transforms all control points of an object with an arbitrary transformation:

```
GlgBoolean GlgTransformObject( object, xform, coord_type, parent )
   GlgObject object;
   GlgObject xform;
   GlgCoordType coord_type;
   GlgObject parent;
```

Parameters

object

Specifies the object to transform: a drawable object or a G data object representing a control point.

xform

Specifies the transformation to be applied to the object's points.

coord_type

Specifies the coordinate system to interpret the transformation in:

- If GLG_SCREEN_COORD is used, the parameters of the transformation are considered to be in screen pixels.

  For example, this may be used to move the *object* by the number of screen pixels as defined by the mouse move. The start and end coordinates of the mouse move can be supplied using the GLG_TRANSLATE_XF transformation as the *xform* parameter.
- If GLG_OBJECT_COORD is used, the transformation parameters are interpreted as GLG world coordinates of the *object*, which includes all transformations applied to the *object*.
- If GLG_PARENT_COORD is used, the transformation parameters are interpreted as GLG world coordinates of the *parent* object, which includes all transformations applied to the *parent*, but excludes transformations attached to the *object*.

parent

The object's parent. It is required if *coord_type=GLG_PARENT_COORD*, or if the *object* is a G data object representing a control point. NULL may be passed in all other cases.

This function applies the given transformation to an object's control points, calculating new control point values. The object's hierarchy must be setup to use this function.

GlgTransformPoint

Transforms a single point.

```
void GlgTransformPoint( matrix, in_point, out_point )
   GlgObject matrix;
   GlgPoint * in_point;
   GlgPoint * out_point;
```

Parameters

matrix

Specifies the matrix to be applied to the input point.

in_point

The input point to be transformed.

out_point

The transformed point.

This function applies the given transformation matrix to the *in_point* parameter, and returns the result in *out_point*.

The *GlgPoint* data type is defined in the *GlgApi.h* file:

```
typedef struct _GlgPoint
{
   double x, y, z;
} GlgPoint;
```

GlgTranslatePointOrigin

Converts screen coordinates of a point in one viewport to the screen coordinates of another viewport.

```
GlgBoolean GlgTranslatePointOrigin( from_viewport, to_viewport,
                                    point )
   GlgObject from_viewport;
   GlgObject to_viewport;
   GlgPoint * point;
```

Parameters

from_viewport

Specifies the viewport in which the point's screen coordinates are defined.

to_viewport

Specifies the viewport whose screen coordinate system to convert to.

in_point

The point to be converted. The result is placed back into the structure pointed by this pointer.

The objects' hierarchy must be setup to use this function.

The *GlgPoint* data type is defined in the *GlgApi.h* file:

```
typedef struct _GlgPoint
{
   double x, y, z;
} GlgPoint;
```

In the GTK version on Linux, the conversion will fail if the viewports belong to different top level windows, in which case the function will return *GlgFalse*.

GlgTraceObject

Highlights all objects in the drawing that depend on the tag or resource object that is being traced.

```
GlgBoolean GlgTraceObject( object, state, is_widget, top_parent,
                           func, data )
   GlgObject object;
   GlgBoolean state;
   GlgBoolean is_widget;
   GlgBoolean (*func)( GlgObject parent, void * data );
   void * data;
```

Parameters

object

Specifies the object to trace. The object must be set up.

state

May be set to True to highlight the objects in the drawing that depend on the traced object or set to False to unhighlight them.

is_widget

If set to True, the parent objects that have resource named *IsWidget* will be highlighted or unhighlighted depending on the *state* parameter.

top_parent

If non-null, the parent objects that are direct children of *top_parent* will be highlighted or unhighlighted.

func

An optional function that provides a custom condition. The function will be invoked for each parent of a traced object to check if a custom condition is satisfied. If the function returns True, the parent will be highlighted or unhighlighted.

data

Specifies a pointer to the custom data passed to the *func* function.

GlgTraceObject searches all parents of the traced object and highlights or unhighlights them (as directed by the *state* parameter) based on the specified conditions. The parent search is performed until any of the specified conditions (*is_widget*, *top_parent* or a custom criteria evaluated by the supplied function) is satisfied. If no conditions are specified, all immediate drawable parents of the traced object will be highlighted or unhighlighted.

The function returns True if any parents were highlighted or unhighlighted.

The dependent parent objects are highlighted with an outline. The attributes of the outline are defined by the *GlgHighlightPolygon* global configuration resource.

Internally, object highlighting is done setting an object's *HighlightFlag* attribute to True. An application can also set and reset this attribute of drawable objects via the *GlgSetDResource* function to highlight or unhighlight individual objects of interest.

GlgTraverseObjects

Traverses the object hierarchy of the object and invokes the supplied function on the object itself and all its subobjects, including object attributes.

```
void GlgTraverseObjects( object, func, data )
   GlgObject object;
   GlgBoolean (*func)( GlgObject object, void * data );
   void * data;
```

Parameters

object

Specifies the object to traverse.

func

Specifies the custom function to be invoked. The function's return result is used to modify the traversal: if the function returns *GlgFalse* for a subobject, the traversal of objects inside the subobject will not be performed.

data

Specifies a pointer to the custom data passed to the function.

GlgTraverseObjectsExt

Traverses the object hierarchy of the object and invokes the supplied functions on the object itself and all its subobjects, including object attributes. One function is invoked when entering an object and another when exiting it.

```
void GlgTraverseObjects( object, func, data )
   GlgObject object;
   GlgBoolean (*enter_func)( GlgObject object, void * data );
   GlgBoolean (*exit_func)( GlgObject object, void * data );
   void * data;
```

Parameters

object

Specifies the object to traverse.

enter_func

Specifies the custom function to be invoked when entering a new object. The function's return value is used to modify the traversal: if the function returns *GlgFalse* for a subobject, the traversal of objects inside the subobject will not be performed.

exit_func

Specifies the custom function to be invoked when exiting an object. The function should return *GlgTrue*. If NULL is supplied as the parameter value, the exit function is not used and *GlgTraverseObjectExt* behaves the same way as the *GlgTraverseObjects* function.

data

Specifies a pointer to the custom data passed to the function.

GlgUnconstrainObject

Unconstrains an object's attribute or an object's point.

```
GlgBoolean GlgUnconstrainObject( attribute )
   GlgObject attribute;
```

Parameters

attribute

Specifies the attribute or point object to be unconstrained.

This function removes any constraints applied to an attribute object. Any changes made to the object while it was constrained to another object are permanent and will be in effect even after the constraints are removed. The removal of the constraints only means that changes to this object are no longer copied to the other objects to which it was once constrained.

If any of the objects which are using this attribute object (or any objects with attributes constrained to it) have already been drawn, an error message will be produced. Use the *GlgSuspendObject* function to suspend drawn objects before removing constraints from their attributes.

GlgWorldToScreen

Converts a point's coordinates from the GLG world coordinate system to the screen coordinate system.

```
GlgBoolean GlgWorldToScreen( object, inside_vp, in_point, out_point )
   GlgObject object;
   GlgBoolen inside_vp;
   GlgPoint * in_point;
   GlgPoint * out_point;
```

Parameters

object

Specifies the object whose world coordinate system to convert from. The world coordinate system of an object includes the effect of all drawing transformations attached to the object and all its parents.

inside_vp

If *object* is a viewport or a light viewport, *inside_vp* specifies which world coordinate system to use. If *inside_vp=GlgTrue,* the method will use the world coordinate system used to draw objects inside this viewport *object*. If *inside_vp=GlgFalse*, the method will use the world coordinate system used to draw this viewport *object* inside its parent viewport.

The value of this parameter is ignored for objects other than a viewport or light viewport.

in_point

The point to be converted.

out_point

The point structure to hold the converted value.

The object's hierarchy must be setup to use this function.

The *GlgPoint* data type is defined in the *GlgApi.h* file:

```
typedef struct _GlgPoint
{
   double x, y, z;
} GlgPoint;
```

Get and Set Resource Function Extension

The *GlgGet*Resource* and *GlgSet*Resource* functions may be used to set or query the value of an attribute object directly, without searching for the resource name. To do this, pass the attribute object ID as the *object* parameter and use NULL as the *resource_name* parameter. It will set the attribute object directly, speeding up the operation by eliminating the search for the resource name.

The following example shows how to use this feature in a function that moves a polygon:

```
GlgBoolean MovePolygon( polygon, by, direction )
   GlgObject polygon;
   double by;
   GlgLong direction;
{
   long i, size;
   GlgObject polygon;
   double x, y, z;

   size = GlgGetSize( polygon );
   GlgSetStart( polygon );     /* Initialize traversing. */
   for( i=0; i<size; ++i )
   {
      /* Get the next point. */
      point = GlgIterate( polygon );

      /* Query the point's coordinates. */
      GlgGetGResource( point, NULL, &x, &y, &z );

      /* Move the point in the desired direction. */
      switch( direction )
      {
         case 'x':
            GlgSetGResource( point, NULL, x + by, y, z );
            break;
         case 'y':
            GlgSetGResource( point, NULL, x, y + by, z );
            break;
         case 'z':
            GlgSetGResource( point, NULL, x, y, z + by );
            break;
         default:
            fprintf( stderr, "Invalid direction.\n" );
            return False;
      }
   }
   return True;
}
```

The above source code is used for example purpose only. The same functionality may be accomplished by the *GlgMoveObjectBy* function.

Extended API Function Descriptions

GlgCreateObject

Creates a GLG object of any type.

```
GlgObject GlgCreateObject( type, name, data1, data2, data3, data4 )
   GlgObjectType type;
   char * name;
   GlgAnyType data1;
   GlgAnyType data2;
   GlgAnyType data3;
   GlgAnyType data4;
```

Parameters

type

Specifies the type of GLG object to be created.

name

An optional argument specifying a name to be assigned to the object. This name can be used later for accessing the object and its attributes without keeping the object ID returned by *GlgCreateObject*. Use NULL as the value of this parameter to create an object without a name. The object's name is an editable attribute of the object, so the name can be assigned or changed later using the resource access functions.
Using the object ID is the fastest method for accessing the object, but it requires keeping the object ID and referencing the object to make sure it is not destroyed while the ID is being used. Using the object name to access the object via the resource mechanism is slightly slower, but it is fast enough for most applications and it ensures the proper use of the object ID.

data1, data2, data3, data4

Parameters which depend on the type of the object to be created. These are described below.

This function creates and returns an object of the specified type. The reference count of the created object is set to 1. Any created object has to be explicitly dereferenced when it is not needed any more or has been referenced by some other object (for example, adding an object to a group references the object). Refer to the description of the *GlgReferenceObject* function for the details on reference counting and object usage.

Most of the entities in the GLG Toolkit, such as graphical objects, container objects, transformations and even attributes, are objects. The **GlgCreateObject** function is used to create any of these objects and is probably the most complex function of the API.

Using GlgCreateObject

The *type* parameter specifies the kind of object to be created. Its possible values and the usage of the accompanying data parameters are outlined in the table below. More detailed descriptions of their use follow the table.

The following table summarizes the usage of the data parameters:

Object Type	data1	data2	data3	data4
Any object type not listed below	NULL	NULL	NULL	NULL
GLG_POLYGON GLG_SPLINE	number of vertices GlgLong optional polygon default: 2 spline default: 3	NULL	NULL	NULL
GLG_PARALLELOGRAM	subtype (GLG_PA_PA for parallelogram GLG_RECT_PA for rectangle) GlgParallType mandatory	NULL	NULL	NULL
GLG_TEXT	string char * optional default: empty string	text subtype GlgTextType optional default: GLG_FIXED_TEXT	NULL	NULL
GLG_IMAGE	image filename char* optional default: NULL	image subtype GlgImageType optional default: GLG_FIXED_IMAGE	NULL	NULL
GLG_GROUP GLG_ARRAY	element type GlgContainerType optional default: GlgObject	initial size GlgLong optional default: 0	NULL	NULL
GLG_LIST	element type GlgContainerType optional default: GlgObject	NULL	NULL	NULL
GLG_VECTOR	element type GlgContainerType optional default: GlgObject	size GlgLong mandatory	NULL	NULL

Object Type	data1	data2	data3	data4
GLG_REFERENCE	template GlgObject mandatory for all reference types (use NULL for file references)	referencing type GlgRefType optional default: GLG_SUBDRAWING_REF	NULL	NULL
GLG_SERIES	template GlgObject mandatory	path transformation of any type GlgObject optional translate xform	NULL	NULL
GLG_SQUARE_SERIES	template GlgObject mandatory	NULL	NULL	NULL
GLG_CONNECTOR	connector subtype GlgObjectType mandatory	number of points for recta-linear path or 0 GlgLong optional default: 2	NULL	NULL
GLG_FRAME	frame subtype GlgFrameType mandatory	frame factor GlgLong mandatory	NULL	NULL
GLG_GIS	dataset file char * optional default: NULL	NULL	NULL	NULL
GLG_DATA	data type (D, S or G) GlgDataType mandatory	string value (S data only) char * optional default: NULL	NULL	NULL
GLG_ATTRIBUTE	data type (D, S or G) GlgDataType mandatory	attribute role GlgXformRole mandatory	NULL	NULL
GLG_TAG	tag name char * optional default: NULL	tag source char * optional default: NULL	tag comment char * optional default: NULL	NULL

Object Type	data1	data2	data3	data4
GLG_XFORM	xform role GlgXformRole mandatory	xform subtype GlgXformType mandatory	xform object GlgObject optional default: NULL	xform object GlgObject optional default: NULL
GLG_ALIAS	alias name char * optional default: NULL	resource path char * optional default: NULL	NULL	NULL
GLG_HISTORY	resource name mask char * optional default: NULL	NULL	entry point data type (D, S or G) GlgDataType optional default: GLG_D	NULL
GLG_FONTTABLE	number of font types GlgLong optional Toolkit default	number of font sizes GlgLong optional Toolkit default	NULL	NULL
GLG_LINE_ATTR	line attributes subtype GlgLineAttrType optional GLG_LINE_-NO_FILL	NULL	NULL	NULL

Usage Summary for *GlgCreateObject*

When an object is created, it is created with default values for its attributes. After the object has been created, the attributes may be assigned values different from the default ones using the resource access functions in the Standard API.

Graphical Objects

Graphical objects are displayable objects used to represent graphical shapes in the drawing.The usage of the *GlgCreateObject* data parameters for the graphical objects is outlined in the following list:

GLG_ARC

No data parameters are used. Use NULL for each parameter value.

GLG_CONNECTOR

data1

Specifies a *mandatory* connector subtype (*GlgObjectType*) which may have the following values:

GLG_POLYGON

Recta-linear connectors

GLG_ARC
Arc connectors

data2
An optional number of control points for the recta-linear connectors (use NULL for arc connectors). The default value for the recta-linear connectors is 2, resulting in one path segment connecting the two control points. By using a value greater than two, more than one path segment is produced (i.e. using a value of 4 yields 3 path segments connecting 4 control points).

data3, **data4**
Not used. Use NULL for each parameter's value.

GLG_FRAME

data1
Specifies a *mandatory* frame subtype (*GlgFrameType*), which can have the following values:

GLG_1D
GLG_2D
GLG_3D
GLG_FREE

Refer to the *GLG Objects* chapter for the details on the frame subtypes. The point frame is the same as a free frame with only one point.

data2
Specifies a *mandatory* frame factor. The frame factor value defines the number of frame intervals, which is one smaller than the number of control points. The frame factor is a creation-time parameter and can't be changed after the frame has been created.

data3, **data4**
Not used. Use NULL as the parameter's value.

GLG_IMAGE

data1
A filename (*char**) of an image file in the GIF, JPEG, PNG or BMP (Windows only) format.

data2
An optional image object subtype (*GlgImageType*) which may have the following values:

GLG_FIXED_IMAGE
GLG_SCALED_IMAGE

Refer to the *GLG Objects* chapter for details on the image subtypes. The default value is **GLG_FIXED_IMAGE**.

data3, **data4**
Not used. Use NULL for each parameter's value.

GLG_MARKER

No data parameters are used. Use NULL for each parameter value.

GLG_PARALLELOGRAM

No data parameters are used. Use NULL for each parameter value.

GLG_POLYGON

data1

Specifies an optional number of polygon points (*GlgLong*). If NULL is used as the parameter's value, a default value of two (2) is used and a polygon with two points is created.

data2, data3, **data4**

Not used. Use NULL for each parameter's value.

GLG_POLYLINE

No data parameters are used. Use NULL for each parameter value.

GLG_POLYSURFACE

No data parameters are used. Use NULL for each parameter value.

GLG_TEXT

data1

An optional string (*char**) to be assigned to the text object. The text object will create a copy of the string. Using NULL causes the default value (the empty string) to be used.

data2

The optional text object subtype (*GlgTextType*) which may have the following values:

GLG_FIXED_TEXT
GLG_FIT_TO_BOX_TEXT
GLG_WRAPPED_TEXT
GLG_TRUNCATED_TEXT
GLG_WRAPPED_TRUNCATED_TEXT
GLG_SPACED_TEXT

Refer to the *GLG Objects* chapter for the details on the text subtypes. The default value is **GLG_FIXED_TEXT**.

data3, **data4**

Not used. Use NULL for each parameter's value.

Example: Creating a Polygon

The following example shows a fragment of code which creates a polygon with three points, sets values of some attributes, and adds the polygon to a viewport:

```
{
GlgObject polygon;

/* Create a polygon with 3 points named "Polygon1". */
polygon = GlgCreateObject( GLG_POLYGON, "Polygon1",
            (GlgAnyType)3, 0, 0, 0 );
/* Set some of the polygon's attributes, use the default
   attribute values for the rest of the attributes.
*/
GlgSetDResource( polygon, "OpenType", (double) GLG_CLOSED );
GlgSetDResource( polygon, "FillType", (double) GLG_EDGE );
GlgSetGResource( polygon, "FillColor", /* Green */ 0., 1., 0. );

/* Add the polygon to a viewport object for displaying. */
GlgAddObject( viewport, polygon, GLG_BOTTOM, 0 );

/* The group have referenced the added polygon.
   Dereference the polygon to let the group manage it.
   This is required since any object is created with the
   reference count set to 1.
*/
GlgDropObject( polygon );
}
```

Since the GLG polygon is a container of points, its points may be accessed via the *GlgGetElement* or *GlgIterate* function. Each point is a GLG data object of G type, and its coordinates may be set using the GlgSetGResource function, using the point as the *object* parameter and NULL as the *resource_name* parameter.

Refer to the demos and examples directories for examples of complete programs using the GLG Extended API.

Containers and Composite Objects

A container object may hold elements of different types as defined by the creation parameters described below and may be used to keep, save, load, and copy collections of user-defined values. A container may be used to group graphical objects in the drawing, in which case the container is a graphical object which may be added to the drawing. There are two types of containers, different in their internal representation: **GLG_ARRAY** (dynamic array) and **GLG_LIST** (linked list). There is also a **GLG_GROUP** type, which may be used to create a container object when the internal representation of the container is not important and whose default value is **GLG_ARRAY**.

Other GLG objects may inherit container functionality. For example, a viewport object serves as a container of graphical objects, and a polygon is a container of points.

GLG_GROUP

GLG_ARRAY

GLG_LIST

data1

Specifies the type of the container's elements and may have the following values:

GLG_OBJECT

for containers to hold GLG objects of any type or hierarchies of GLG objects. This may also be used to hold double floating values, strings or geometrical points in form of the GLG data or attribute objects. Objects are referenced when added to the container, and dereferenced when they are deleted or their container gets destroyed.

GLG_NON_DRAWABLE_OBJECT

ADVANCED: same as GLG_OBJECT but is used when the container object will not be drawn. This may be used for creating lightweight containers which are never added to the drawing to be displayed, as well as for creating custom properties that should not be setup. For example, if a group of custom properties contains drawable objects, it should not be setup to avoid the drawable objects being drawn when the custom properties are setup.

GLG_STRING

for containers to hold C character strings. The strings added to the container will be clones of the input strings. Note that the *GlgFindObject* and *GlgIterate* functions return pointers to the internal strings which must not be altered or freed. Deleting a string from a container will automatically free it. Empty strings and NULL pointers are allowed in the container.

GLG_LONG

for containers to hold integer values or addresses. The container's elements will be of the longest integer type and are guaranteed to hold memory pointers. The memory pointed to by a pointer is not copied or freed automatically: this must be handled by the application which uses the container.

data2

Specifies a maximum number of objects in the group (*GlgLong*). This parameter is optional and is used only for **GLG_GROUP** and **GLG_ARRAY** containers. If you know the maximum number of objects in the group in advance, you may use this parameter to slightly optimize the group's performance. If the number is not known in advance, use NULL as the parameter's value. It will not cause an error to exceed the specified number of objects in the group at run time.

data3, **data4**

Not used. Use NULL as the parameter's value.

GLG_VIEWPORT.

No data parameters are used. Use NULL for each parameter's value.

GLG_SERIES

GLG_SQUARE_SERIES

data1

Specifies a *mandatory* template object (*GlgObject*). This object is replicated by the series according to the series' factors. The series objects are created with the default value of the *Factor* attribute equal to one. The same template object should not be used for more than one series.

data2, **data3**, **data4**
Not used. Use NULL as the parameter's value.

GLG_REFERENCE

data1
A *mandatory* template object (*GlgObject*) for a container or object reference. Use NULL for file reference objects.

data2
An optional reference subtype (*GlgRefType*) which may have the following values:

GLG_REFERENCE_REF
Template and file reference objects.

GLG_CONTAINER_REF
Container objects.

Refer to the *GLG Objects* chapter for details on the reference subtypes. The default value is **GLG_REFERENCE_REF**.

data3, **data4**
Not used. Use NULL for each parameter's value.

Non-Graphical Objects

Non-graphical objects are used to keep data values as well as control a wide variety of the visible features of graphical objects. Refer to the *GLG Objects* chapter of the *GLG User's Guide and Builder Reference Manual* for more information. The following list outlines the usage of the *GlgCreateObject* data parameters for non-graphical objects:

GLG_DATA.

data1
Specifies a *mandatory* data type (*GlgDataType*) which may have the following values:

GLG_D
D data (double)

GLG_S
S data (string)

GLG_G
G data (triplet of double values to define XYZ or RGB)

data2
Specifies an optional string value for the *S* data type. Use NULL for *D* and *G* data types.

data3, **data4**
Not used. Use NULL as the parameter's value.

GLG_ATTRIBUTE.

data1
Specifies a *mandatory* data type (*GlgDataType*) which may have the following values:

GLG_D
D attribute (double)

GLG_S
S attribute (string)

GLG_G
G attribute (triplet of double values to define XYZ or RGB)

data2

Specifies a *mandatory* attribute role (*GlgXformRole*) which may have the following values:

GLG_GEOM_XR
GLG_COLOR_XR
GLG_GDATA_XR
GLG_DDATA_XR
GLG_SDATA_XR

The attribute's role determines the type of transformations that will affect the object. Refer to the *GLG Objects* chapter for more information.

data3, **data4**

Not used. Use NULL as the parameter's value.

GLG_ALIAS

data1

Specifies the alias name (*char**). This is an alternative (logical) name for accessing the resource pointed to by the alias. The default value is NULL.

data2

Specifies the resource path for the alias (*char**). The default value is NULL.

data3, **data4**

Not used. Use NULL as the parameter's value.

GLG_HISTORY

data1

Specifies the resource name mask (*char**) used to access resources to be scrolled. The default value is NULL.

data2

Not used. Use NULL as the parameter's value.

data3

Specifies the data type of the entry point. Possible values are GLG_D, GLG_S and GLG_G. This data type must match the data type of the resources pointed to by the resource name mask. The default value is GLG_D.

data4

Not used. Use NULL as the parameter's value.

Transformation Objects

This section details procedures for creating the transformation objects. An introduction to these objects can be found in the *GLG Objects* chapter.

GLG_XFORM

data1

Specifies the role the transformation plays in the object hierarchy (*GlgXformRole*). It is a mandatory parameter and may have the following values:

GLG_GEOM_XR - for a geometrical transformation
GLG_COLOR_XR - for a color transformation
GLG_DDATA_XR - for a scalar transformation.
GLG_SDATA_XR - for a string transformation.

The **data2** parameter is a mandatory parameter, and specifies the type of the transformation (*GlgXformType*) and may have the following values. Its possible values vary with the value of the **data1** parameter.

The parameters used to control a transformation are set using the resource mechanism after the transformation object is created. So, for example, to create a move transformation that moves a geometric object 50 units in the X direction takes two steps. First, you create a MoveX transformation object (**GLG_TRANSLATE_X_XF**), and only then do you set its attributes to move the desired amount. For geometric transformations, the most commonly manipulated resource is divided into two attributes, and the transformation depends on the product of the two. For a MoveX transformation, the two attributes are the X distance and the *Factor* attributes. This allows the application to scale to the data it is to receive.

The following transformations may be attached to an attribute of any type (D, S or G), and use the following values of the **data2** parameter:

GLG_LIST_XF
: The transformed value is selected from a list of values based on the transformation's index attribute, which is 0-based. When a list transformation is created, the **data3** parameter can be used to pass an array of values using a group containing GLG_ATTRIBUTE objects. The type of attribute object in the group should match the type of attribute the transformation is attached to (D, S or G).

GLG_THRESHOLD_XF
: The transformed value is selected from a list of values based on the transformation's input value, which is compared with a list of thresholds. When a list transformation is created, the **data3** parameter can be used to pass an array of values using a group containing GLG_ATTRIBUTE objects. The type of attribute object in the group should match the type of attribute the transformation is attached to (D, S or G). The **data4** parameter can be used to pass an array of thresholds, which is a group containing GLG_ATTRIBUTE objects of scalar (D) type.

GLG_SMAP_XF

GLG_DMAP_XF
: The transformed value is selected by mapping an input key to an output value by matching the input key with the list of keys. The type of the keys is determined by the type of the transformation: D (double) or S (string). The type of the output values is determined by the type of the attribute the transformation is attached to. When a transformation is created, the **data3** and **data4** parameters can be used to pass a list of values and a list of keys. The lists are passed using a group containing GLG_ATTRIBUTE objects of an appropriate type.

GLG_JAVA_SCRIPT_XF
: Creates a *JavaScript* transformation that sets the output value to the result of a Java Script expression based on a variable number of input parameters.

GLG_TRANSFER_XF
: This transformation is used to transfer a value from one attribute object to another. It can be used to implement a "one-way" constraint, where changes in one object affect another but not vice versa. It has only two attributes. The first attribute is meant to be constrained to the source attribute, while the second may be used to attach a transformation to the constrained value.

GLG_IDENTITY_XF
This transformation is used to transfer a value from one attribute object to another. It is used in the internal design of the GLG Control Widgets.

GLG_CHANGE_ALARM_XF
Creates a *Change Alarm* transformation that generates an alarm when its input value changes.

For a transformation attached to a scalar (D) value, the **data2** values can be:

GLG_RANGE_CONVERSION_XF
Creates a *Range Conversion* transformation used to map an input range to an output range of scalars. For example, if the input range is (0,1) and the output range is (100,200), then an input value of 0.5 will produce a transformed value of 150.

GLG_RANGE_CHECK_XF
Creates a *Range Check* transformation that changes the output value when its input value goes out of range.

GLG_RANGE_ALARM_XF

GLG_RANGE2_ALARM_XF
Creates a *Range Alarm* or *Range2 Alarm* transformation that generates an alarm when its input value goes out of range.

GLG_TIMER_XF
Creates a *Timer* transformation which periodically changes a value of an object's attribute to implement blinking or other types of animation.

GLG_DIVIDE_XF
Creates a *Divide* transformation used to divide an attribute value by the *Divisor* attribute.

GLG_LINEAR3_XF
Creates a *Linear* transformation that calculates the output value as a linear combination of the transformation's parameters.

GLG_BOOLEAN_XF
Creates a *Boolean* transformation that sets the output value to a boolean combination of the transformation's parameters.

GLG_COMPARE_XF
Creates a *Compare* transformation that sets the output value to 0 or 1 depending on the result of comparing two transformation parameters.

GLG_BITMASK_XF
Creates a *Bitmask* transformation that sets the output value to a value formed by interpreting the states of four input parameters as bits of a 4-bit integer.

GLG_D_FROM_G_XF
Creates a *D From G* transformation that extracts an X, Y or Z coordinate (or an R, G or B color value) from a G data object containing a triplet of XYZ or RGB values.

GLG_SCREEN_FACTOR_XF
Creates a *Screen Factor* transformation that proportionally increases or decreases a scalar attribute depending on zooming and resizing of a parent viewport.

For a transformation attached to a string (S) value, the **data2** values can have the following values:

GLG_S_FORMAT_XF
Creates a *Format S* transformation. The transformed value of the string attribute is equal to an input string value formatted with a format string.

GLG_D_FORMAT_XF
Creates a *Format D* transformation. The transformed value of the string attribute is equal to an input scalar value formatted with a format string.

GLG_TIME_FORMAT_XF
Creates a *Time Format* transformation to format POSIX time values and display them as strings.

GLG_STRING_CONCAT_XF
Creates a *String Concatenation* transformation that concatenates several strings.

Transformations for color attributes may also have the following value:

GLG_COLOR_SCALE_XF
Creates a *Bitmask* transformation that converts a base color by either increasing or decreasing its intensity, storing the result as the color attribute's transformed value.

Transformations for color attributes and attributes of G type other than control points may also have the following value:

GLG_G_FROM_D_XF
Creates a *G From D* transformation that forms a single data object of G type (such as an XYZ point or RGB color) from three D (double) data objects that define individual X, Y, Z or R, G, B components.

Transformations for geometrical or color attributes can have the following values:

GLG_TRANSLATE_X_XF
GLG_TRANSLATE_Y_XF
GLG_TRANSLATE_Z_XF
GLG_TRANSLATE_XYZ_XF
Create a parametric translation transformation along the specified axis or axes.

GLG_TRANSLATE_XF
Creates a parametric translation transformation defined by a vector with a start and end point.

GLG_SCALE_X_XF
GLG_SCALE_Y_XF
GLG_SCALE_Z_XF
Create a parametric scale transformation; scaling affects only the specified coordinate.

GLG_SCALE_XYZ_XF
Creates a parametric scale transformation.

GLG_ROTATE_X_XF
GLG_ROTATE_Y_XF
GLG_ROTATE_Z_XF
Create a parametric rotate transformation around the axis defined by the transformation type.

GLG_SHEAR_X_XF
GLG_SHEAR_Y_XF
GLG_SHEAR_Z_XF
Create a parametric shear transformation; the shear amount increases along the axis defined by the transformation type.

Geometrical transformations attached to control points can have the following values:

GLG_WORLD_OFFSET_XF
Creates a *World Offset* transformation.

GLG_PIXEL_OFFSET_XF
Creates a *Pixel Offset* transformation.

GLG_FIXED_OFFSET_XF
Creates a *Fixed Offset* transformation.

Geometrical transformations attached to objects and their control points can have the following values:

GLG_MATRIX_XF
Creates a matrix transformation. A matrix transformation is more compact than the parametric, although its action cannot be easily controlled by accessing resources. To create a matrix transformation, first create the parametric transformation which defines the desired transformations and then use this transformation as the value of the **data3** parameter. The **data4** parameter is not used; use NULL for its value. After the matrix transformation has been created, the parametric transformation is not needed any more and may be dropped using the *GlgDropObject* function. To create a complex matrix transformation, a concatenation of several transformations may be used as the **data3** parameter.

Once a matrix transformation is created, it can be changed with the *SetResourceFromObject* function.

GLG_CONCATENATE_XF
Creates a concatenate transformation; it may be a concatenation of two matrix, parametric, or concatenate transformations. In this case **data3** and **data4** should contain the two transformation objects to be linked. Since a concatenate transformation references its components, they may be dereferenced using the *GlgDropObject* function after the concatenate transformation has been created.

The order of the transformation objects in the argument list corresponds to the order in which the transformations are applied when the resulting concatenate transformation is used to draw any object. The object is transformed with the first transformation, then with the second. Either object may itself be a concatenate transformation object, containing two or more other transformations.

A transformation may be used to create several concatenate transformations. In this case the corresponding transformations of the created concatenate transformations are constrained to have the same parameter values. Use the *GlgCopyObject* function to create copies of the transformation before passing them to the *GlgCreateObject* function to avoid constraints.

GLG_PATH_XF
Creates a parametric path transformation.When a path transformation is created, the **data3** parameter can be used to pass an array of path points. This array may be extracted from any polygon with its *Array* resource. Use a copy of the array if you don’t want the path of the transformation to be constrained to the path polygon. If the **data3** parameter is NULL, a default path transformation with two points is created.

Example: Concatenate Transformation

The following fragment of code shows an example of creating a concatenate transformation. It creates a rotate and translate transformations and then combines them in a concatenate transformation:

```
{
   GlgObject rotate_xform, move_xform;

   /* Create a rotate transformation. */
   rotate_xform =
      GlgCreateObject( GLG_XFORM, "Rotate", GLG_GEOM_XR,
            GLG_ROTATE_Z_XF, NULL, NULL );

   /* Change the center of rotation from a default value (0;0;0) to the
            value (500; 0; 0).
   */
   GlgSetGResource( rotate_xform, "XformAttr1", 500., 0., 0. );

   /* Set the rotation angle to 90 degrees. */
   GlgSetDResource( rotate_xform, "XformAttr2", 90. );

   /* Create a translate transformation. */
   move_xform =
      GlgCreateObject( GLG_XFORM, "Move", GLG_GEOM_XR,
            GLG_TRANSLATE_X_XF, NULL, NULL );

   /* Set the move amount to 500 in world coordinates. */
   GlgSetDResource( move_xform, "XformAttr2", 500. );
   /* Create a concatenate transformation. */
   concat_xform =
      GlgCreateObject( GLG_XFORM, "Concat", GLG_GEOM_XR,
            GLG_CONCATENATE_XF, rotate_xform, move_xform );

   /* Dereference transformations; we do not need them any more. */
   GlgDropObject( rotate_xform );
   GlgDropObject( move_xform );
}
```

Example: Matrix Transformation

The following example shows how to create a matrix transformations using the concatenate transformation from the previous example. It creates two different matrix transformations from the same concatenate transformation using different transformation parameters:

```
{
   GlgObject matrix_xform1, matrix_xform2;

   /* Set transformation parameters for the first matrix. */
   GlgSetDResource( concat, "Rotate/XformAttr2", 60. ); /* Angle */
   GlgSetDResource( concat, "Move/XformAttr1", 500. );  /* Amount */

   /* Create the first matrix transformation. */
   matrix_xform1 =
      GlgCreateObject( GLG_XFORM, "Matrix", GLG_GEOM_XR,
            GLG_MATRIX_XF, concat_xform, NULL );

   /* Set transformation parameters for the second matrix. */
   GlgSetDResource( concat, "Rotate/XformAttr2", 30. );   /* Angle */
   GlgSetDResource( concat, "Move/XformAttr1", 1500. ); /* Amount */

   /* Create the second matrix transformation. */
   matrix_xform2 =
      GlgCreateObject( GLG_XFORM, "Matrix", GLG_GEOM_XR,
            GLG_MATRIX_XF, concat_xform, NULL );

   /* Dereference the concatenation transformation if not needed any
            more.
   */
   GlgDropObject( concat_xform );
}
```

Refer to the *GLG Objects* chapter for a description of all the transformation attributes.

GlgAddObjectToTop
GlgAddObjectToBottom
GlgAddObjectAt

Add an object to a container at the specified position: top, bottom or position defined by the index.

```
GlgBoolean GlgAddObjectToTop( container, object )
   GlgObject container;
   GlgObject object;

GlgBoolean GlgAddObjectToBottom( container, object )
   GlgObject container;
   GlgObject object;

GlgBoolean GlgAddObjectAt( container, object, index )
   GlgObject container;
   GlgObject object;
   GlgLong index;
```

Parameters

container

Specifies the container object.

object

Specifies the object to be added to the container object.

index

Specifies the position in a container.

These functions add an object to a container object at the specified position and return *GlgTrue* if the object was successfully added or *GlgFalse* otherwise. The container object may have different types: it may be a group, a viewport or a polygon.

If the container object is a group or viewport, you may add any graphical object to it. Any attempt to add a non-graphical object will fail and an error message will be generated. The objects at the end (the bottom) of the list are drawn last and will appear in front of other objects.

If a polygon is passed as a container object parameter to this function, you may add points to the polygon. Any attempt to add an object different from a geometrical data object will fail, generating an error message.

Note: For viewports with integrated scrolling enabled by the viewport's *Pan* attribute, child viewport objects added to the bottom of the viewport object list will appear in front of the integrated scrollbars until the viewport is reset. Other graphical objects added to the bottom of the container will also appear on top of the integrated scrollbars that use light viewports. Resetting the viewport reorders the scrollbars to be the last in the object list. To achieve proper rendering without resetting the viewport, use *GlgAddObjectAt* to add the child viewport at a proper position before the integrated scrollbars. If a single X or Y scrollbar is used, the position index will be *list_size - 1*, where list_size is the number of objects in the viewport. If both X and Y scrollbars are active, *list_size - 3* position should be used.

GlgAddObject

A generic function for adding an object to a container.

```
GlgBoolean GlgAddObject( container, object, access_type, pos_modifier
           )
   GlgObject container;
   GlgObject object;
   GlgAccessType access_type;
   GlgPositionModifier pos_modifier;
```

Parameters

container

Specifies the container object.

object

Specifies the object to be added to the container object.

access_type

Specifies the position at which the object is inserted. This parameter may have the following values:

GLG_TOP
Adds the object at the beginning of the object list.

GLG_BOTTOM
Adds the object at the end of the object list.

GLG_CURRENT
Adds the object at the position defined by the last call to *GlgFindObject*.

pos_modifier

Used only with the GLG_CURRENT access type and may have the following values:

GLG_BEFORE
Adds object before the object defined by the last call to *GlgFindObject*.

GLG_AFTER
Adds input object after the object defined by the last call to *GlgFindObject*.

If the access type is not GLG_CURRENT, use 0 as the value of this parameter.

This function adds an object to a container object and returns *GlgTrue* if the object was successfully added or *GlgFalse* otherwise. The container object may have different types: it may be a group, a viewport or a polygon.

If the container object is a group or viewport, you may add any graphical object to it. Any attempt to add a non-graphical object will fail and an error message will be generated. The objects at the end (the bottom) of the list are drawn last and will appear in front of other objects.

If a polygon is passed as a container object parameter to this function, you may add points to the polygon. Any attempt to add an object different from a geometrical data object will fail, generating an error message.

A container object keeps an internal current position pointer, which is set by a successful call to the *GlgFindObject* function. For a call to *GlgAddObject* immediately following the *GlgFindObject* call, the *pos_modifier* parameter defines where to add the new object: right before or right after the object returned by *GlgFindObject*. The position modifier also defines how the insert position defined by the current position pointer is adjusted after the insertion. For GLG_BEFORE, the current position is left unchanged, for GLG_AFTER it is incremented by 1. This allows you to call *GlgAddObject* repeatedly after an initial call to *GlgFindObject*.

For example, the following sequence of calls:

```
GlgFindObject( container, object_B, GLG_FIND_OBJECT, 0 );
GlgAddObject( container, object_1, GLG_CURRENT, GLG_BEFORE );
GlgAddObject( container, object_2, GLG_CURRENT, GLG_BEFORE );
GlgAddObject( container, object_3, GLG_CURRENT, GLG_BEFORE );
```

for a group with objects:

```
(object_A, object_B, object_C)
```

results in a group with the following object order:

```
(object_A, object_1, object_2, object_3, object_B, object_C).
```

The sequence of calls:

```
GlgFindObject( container, object_B, GLG_FIND_OBJECT, 0 );
GlgAddObject( container, object_1, GLG_CURRENT, GLG_AFTER );
GlgAddObject( container, object_2, GLG_CURRENT, GLG_AFTER );
GlgAddObject( container, object_3, GLG_CURRENT, GLG_AFTER );
```

for the same group results in:

```
(object_A, object_B, object_1, object_2, object_3, object_C).
```

Keep in mind that any other calls to *GlgFindObject*, *GlgAddObject*, and *GlgDeleteObject* may affect the current position pointer.

GlgAddAttachmentPoint

Adds an attachment point to a reference object and returns the added point.

```
GlgObject GlgAddAttachmentPoint( object, dx, dy, dz )
   GlgObject object;
   double dx, dy, dz;
```

Parameters

object

Specifies a reference object (other than a subwindow) to add an attachment point to.

dx, dy, dz

Specify world coordinate offsets for initial position of the attachment point relatively to the reference's single control point. When the reference object is resized, the offsets will be automatically adjusted to maintain the point's position relatively to the object.

GlgCopyObject

Creates a copy of an object.

```
GlgObject GlgCopyObject( object )
   GlgObject object;
```

Parameters

object

Specifies the object to be copied.

This function creates and returns a copy of an object. The reference count of the created copy is set to 1. The created copy must be explicitly dereferenced after it has been used to avoid memory leaks. Refer to the description of the *GlgReferenceObject* function for details on reference counting.

GlgCopyObject is equivalent to using GlgCloneObject with the FULL_CLONE cloning type. Refer to the *GlgCloneObject* section below for more details on cloning types.

The *GlgCopyObject* function may be used to create multiple instances of an object after the object has been created and its attributes have been set to the desired values. Using *GlgCopyObject* can also save repeated calls to the resource-setting functions.

GlgCloneObject

Creates a specialized copy (clone) of an object.

```
GlgObject GlgCloneObject( object, clone_type )
   GlgObject object;
   GlgCloneType clone_type;
```

Parameters

object

Specifies the object to be cloned.

clone_type

This is an enumerated value, specifying the type of clone desired. The possible value are GLG_FULL_CLONE, GLG_STRONG_CLONE, GLG_WEAK_CLONE, GLG_CONSTRAINED_CLONE and GLG_SHALLOW_CLONE.

This function creates a copy of an object, according to the rules of cloning. There are four different kinds of clones available for a given object. The difference between them depends on their treatment of the original object's attributes according to the *Global* flag. There is also a special shallow clone type which can be applied only to the GLG group objects.

Full Clone

A full clone is a copy of the original object. No attributes of the copy are constrained to the attributes of the original.

Constrained Clone

A constrained clone is a copy of the original object all of whose attributes are constrained to the corresponding attributes of the original.

Exception: The Constrained Clone does not constrain the attributes that have their *Global* flag set to NONE, such as the constrained points of the parallelogram, frame and connector objects, the *Buffer* attribute of the Transfer transformation and some other special attributes.

Strong Clone

A strong clone is a copy of the original object. If any of the object's attributes are global or semi-global (have their *Global* flags set to GLG_GLOBAL or GLG_SEMI_GLOBAL), they will be constrained to the corresponding attribute of the original.

Weak Clone

A weak clone interprets SEMI-GLOBAL to mean the same as LOCAL. It is similar to a strong clone, but only the global attributes are constrained.

Shallow Clone

A shallow clone is a special clone type that can be applied only to the group objects. When a shallow copy of a group is created, the copy will contain the same object IDs as the original

group, as opposed to the other clone types which clone elements of the group, creating new objects with new object IDs. The shallow clone may be used by a program to create a list of objects in the original group, for example for safe traversing of the list while deleting the objects from the original group.

GlgDeleteTopObject
GlgDeleteBottomObject
GlgDeleteObjectAt

Delete an object from a specified position in a container: top, bottom or a position defined by an index.

```
GlgBoolean GlgDeleteTopObject( container )
   GlgObject container;

GlgBoolean GlgDeleteBottomObject( container )
   GlgObject container;

GlgBoolean GlgDeleteObjectAt( container, index )
   GlgObject container;
   Glglong index;
```

Parameters

container

Specifies a container object.

index

Specifies an index of the object to be deleted from the container.

These functions delete an object from the container object and returns *GlgTrue* if the object was successfully deleted or *GlgFalse* otherwise.

GlgDeleteObject

A generic function for deleting an object from a container or replacing it.

```
GlgBoolean GlgDeleteObject( container, object, access_type,
                            pos_modifier )
   GlgObject object;
   GlgObject container;
   GlgAccessType access_type;
   GlgPositionModifier pos_modifier;
```

Parameters

container

Specifies a container object.

object

An object to replace the deleted element with, or NULL to delete an element without replacing.

access_type

Specifies the position of the object to be deleted and may have the following values:

GLG_TOP
Deletes the object at the beginning of the object list.

GLG_BOTTOM
Deletes the object at the bottom of the object list.

GLG_CURRENT
Deletes the object at the position defined by the last call to *GlgFindObject*.

pos_modifier

Used only with the GLG_CURRENT access type. It defines how to adjust the current position after the delete operation. This parameter allows using the function repeatedly to delete several objects before or after the object defined by the previous call to the *GlgFindObject* function. It may have the following values:

GLG_BEFORE
Moves the current position pointer to the previous object in the list.

GLG_AFTER
Moves the current position pointer to the next object in the list.

If the access type is not GLG_CURRENT, use 0 as the value of the parameter.

This function deletes an object from the container object and returns *GlgTrue* if the object was successfully deleted or *GlgFalse* otherwise.

The object to be deleted is defined by the *access_type* parameter and may be the first (GLG_TOP) or the last object (GLG_BOTTOM) in the container's object list. It may also be an object defined by the previous successful call to *GlgFindObject* (GLG_CURRENT).

For example, the following sequence of calls:

```
GlgFindObject( container, object_D, GLG_FIND_OBJECT, 0 );
GlgDeleteObject( container, NULL, GLG_CURRENT, GLG_AFTER );
GlgDeleteObject( container, NULL, GLG_CURRENT, GLG_AFTER );
GlgDeleteObject( container, NULL, GLG_CURRENT, GLG_AFTER );
```

for a group with objects:

```
object_A, object_B, object_C, object_D, object_E, object_F, object_G
```

results in deleting objects *object_D*, *object_E*, and *object_F*.

The sequence of calls:

```
GlgFindObject( container, object_B, GLG_FIND_OBJECT, 0 );
GlgDeleteObject( container, NULL, GLG_CURRENT, GLG_BEFORE );
GlgDeleteObject( container, NULL, GLG_CURRENT, GLG_BEFORE );
GlgDeleteObject( container, NULL, GLG_CURRENT, GLG_BEFORE );
```

for the same group deletes objects *object_D*, *object_C*, and *object_B*, in that order.

Keep in mind that any other calls to *GlgFindObject*, *GlgAddObject*, and *GlgDeleteObject* may affect the current position pointer. The current position pointer must be used immediately after the *GlgFindObject* call.

GlgDeleteThisObject

Finds and deletes the object from a container.

```
GlgBoolean GlgDeleteThisObject( container, object )
   GlgObject container;
   GlgObject object;
```

Parameters

container

Specifies a container object.

object

Specifies an object to be deleted from the container.

This function finds and deletes an object from the container object. It returns *GlgTrue* if the object was successfully deleted or *GlgFalse* if object was not found in the container.

GlgFlush

Set the container object's size:

```
void GlgFlush( container, size)
   GlgObject container;
   GlgLong size;
```

Parameters

container

Specifies the container object.

size

Specifies the new size.

This function sets the container size to the requested size. If the new size is less than the current container size, extra elements are removed. If the new size is greater than the current size, the function adds elements by making a copy of the last element in the container.

The container may be a group, a viewport, a polygon (as a container of points), a spline or a connector. The function can only be invoked if the container is not set up. If the container is set up (drawn), the function must be surrounded by the *GlgSuspendObject* and *GlgReleaseObject* calls.

GlgSetAttachmentMoveMode

Changes the attachment point move mode and returns the previous mode state.

```
GlgBoolean GlgSetAttachmentMoveMode( state )
   GlgBoolean state;
```

Parameters

state

Specifies the state of the attachment point move mode. By default, the attachment point move mode is off, and moving the attachment point will move the reference object it is attached to. If the mode is on, moving the attachment point will change its position relatively to the reference object.

GlgSetElement

Replaces the object at the specified position in a container.

```
GlgBoolean GlgSetElement( container, index, new_object )
   GlgObject container;
   GlgLong index;
   GlgObject new_object;
```

Parameters

container

Specifies the container object.

index

Specifies the position inside the container at which the object will be replaced.

new_object

Specifies the new object.

This function returns GlgFalse if the specified index is invalid.

GlgSetResourceObject

Replaces a resource object:

```
GlgBoolean GlgSetResourceObject( object, resource_name, o_value )
   GlgObject object;
   char * filename;
   GlgObject o_value;
```

Parameters

object

Specifies the object whose resource is being set or replaced.

resource_name

Specifies a default attribute name of a resource that is being set or replaced.

o_value

Specifies a new resource object.

This function sets or replaces the resource object specified by the resource name. It may be used to attach custom properties (*CustomData* resource), history (*History* resource) and aliases (*Aliases* resource) to the object, or to replace them with new values. A container object may be used as the *o_value* to attach a list of properties or other objects. If objects have been drawn, they must be suspended with *GlgSuspendObject* before using *GlgSetResourceObject* and released with *GlgReleaseObject* when finished.

This function may also be used to set or replace values of other resources. For example, it may be used instead of the *GlgSetXform* function to set the object's transformation using the *Xform* resource name. However, *GlgSetResourceObject* provides direct manipulation capabilities and doesn't do any error checking, making the user responsible for the proper handling of objects.

The function returns *GlgTrue* if the operation was successful.

While this function provides the Extended API functionality, it may also be used with the Intermediate API for setting global configuration resources.

GlgSetXform

Adds or deletes transformations attached to an object.

```
GlgBoolean GlgSetXform( object, xform )
   GlgObject object;
   GlgObject xform;
```

Parameters

object

Specifies the object to which a transformation is to be attached.

xform

Specifies the transformation object to be attached to the object.

This function replaces the current object's transformation with a new one. The NULL pointer may be used as the value of the transformation parameter to just delete the existing transformation (use the *Xform* resource name to query the existence of the transformation with the *GlgGetResourceObject* function). The function returns *GlgTrue* upon successful completion, and *GlgFalse* for failure.

A transformation may be added only to a drawable object (such as a viewport, group, polygon, etc.) or to the object's attributes. Any attempt to add a transformation to a non-drawable object generates an error message.

The attributes of an object differ by their data type. It is important not to try to attach a transformation of an inappropriate type to these objects. For example, *GlgSetXform* fails if you try to attach a string transformation to a geometrical object.

A transformation may be added to an object only before it has been drawn. To add a transformation after the object has been drawn, use the *GlgSuspendObject* function. Use the *GlgReleaseObject* when done. Adding a transformation without using the *GlgSuspendObject* function after the object has been drawn generates an error message.

2.7 Real-Time Chart Functions

The following Standard and Intermediate API methods may be used to handle specific aspects of the GLG Real-Time Chart object:

Standard API

- **GlgGetSelectedPlot** returns the plot object corresponding to the legend item selected with the mouse, if any.
- **GlgGetNamedPlot** queries a chart's plot by its name.
- **GlgAddPlot** adds a plot to a chart.
- **GlgDeletePlot** deletes a plot from a chart.
- **GlgAddTimeLine** adds a vertical time line to a chart.
- **GlgDeleteTimeLine** deletes a vertical time line from a chart.
- **GlgAddAnnotation** adds an annotation object to a chart.
- **GlgDeleteTimeLine** deletes an annotation from a chart.
- **GlgClearDataBuffer** clears accumulated data samples of a real-time chart or one of its plots.
- **GlgGetChartDataExtent** queries the extent of the data accumulated in the chart's plots.
- **GlgSetLabelFormatter** attaches a custom label formatter to a stand-alone axis or an axis of a chart.
- **GlgSetLinkedAxis** links a chart's plot or a level line level with an axis for automatic rescaling.
- **GlgSetChartFilter** attaches a custom data filter to a chart's plot.
- **GlgUpdateChartTimeAxis** scrolls the chart forward to show all data samples up to the provided timestamp or index.
- **GlgUpdateChartState** triggers updating a chart's state after prefilling it with data using the quick mode.

Intermediate API

- **GlgCreateChartSelection** returns information about a chart's sample under the cursor.
- **GlgGetLegendSelection** returns a plot object corresponding to the legend item under the cursor.
- **GlgCreateTooltipString** creates and returns a tooltip string corresponding to the current cursor position on top of a chart or axis object.
- **GlgPositionToValue** returns a chart, plot or axis value corresponding to the cursor position.
- **GlgCreateDataSampleNode** creates a new data sample node to be used with *GlgAddDataSampleNode*.
- **GlgAddDataSampleNode** adds a data sample to a chart, is used to prefill a chart with a large number of samples.

The following sections provide detailed description of these methods.

Chart Functions of the Standard API

GlgAddAnnotation

Adds an annotation to a chart object.

```
GlgObject GlgAddAnnotation( object, resource_name, annotation,
                            position_x, position_y, add_box )
   GlgObject object;
   char * resource_name;
   GlgObject time_line;
   double position_x;
   double position_y;
   GlgBoolean add_box;
```

Parameters

object

Specifies a chart or a container object containing the chart.

resource_name

Specifies a resource name for accessing the chart inside the container. Use NULL when the *object* parameter specifies the chart.

annotation

Specifies an object ID of an annotation object to add. NULL may be used to create a new annotation.

position_x

Specifies an annotation's position relative to the X axis.

position_y

Specifies an annotation's position relative to the Y axis.

add_box

If set to *GlgTrue*, a text box will be added to the annotation's text object.

The function returns an object ID of the added annotation on success, or NULL on failure. If the *annotation* parameter is NULL, the returned annotation is referenced only by the chart's annotation array it has been added to and may be destroyed when the chart scrolls. The program should increase the annotation's reference count if it needs to keep a persistent reference to it.

The *NumAnnotations* attribute of the chart should be set to -1 in order to use this function.

GlgAddPlot

Adds a plot to a chart object.

```
GlgObject GlgAddPlot( object, resource_name, plot )
   GlgObject object;
   char * resource_name;
   GlgObject plot;
```

Parameters

object

Specifies a chart or a container object containing the chart.

resource_name

Specifies a resource name for accessing the chart inside the container. Use NULL when the *object* parameter specifies the chart.

plot

Specifies an object ID of a plot object. NULL may be used to create a new plot.

The function returns an object ID of the added plot on success, or NULL if it fails. If the *plot* parameter is NULL, the returned plot is referenced only by the chart's plot array it has been added to and may be destroyed when the number of plots changes. The program should increase the plot's reference count if it needs to keep a persistent reference to the plot.

GlgAddTimeLine

Adds a vertical time line to a chart object.

```
GlgObject GlgAddTimeLine( object, resource_name, time_line,
                          time_stamp )
   GlgObject object;
   char * resource_name;
   GlgObject time_line;
   double time_stamp;
```

Parameters

object

Specifies a chart or a container object containing the chart.

resource_name

Specifies a resource name for accessing the chart inside the container. Use NULL when the *object* parameter specifies the chart.

time_line

Specifies an object ID of a level line object to be used as a time line. NULL may be used to create a new time line.

time_stamp

Specifies the timestamp to position the time line at. This value will be assigned to the time

line's *Level* attribute.

The function returns an object ID of the added time line on success, or NULL on failure. The returned time line is referenced only by the chart's time line array it has been added to and may be destroyed when the chart scrolls. The program should increase the time line's reference count if it needs to keep a persistent reference to the time line.

The *NumTimeLines* attribute of the chart should be set to -1 in order to use this function.

GlgClearDataBuffer

Clears accumulated data samples of a real-time chart or one of its plots.

```
GlgBoolean GlgClearDataBuffer( object, resource_name )
   GlgObject object;
   char * resource_name;
   GlgObject plot;
```

Parameters

object

Specifies a chart, a plot or a container object containing the chart.

resource_name

If not NULL, specifies a resource name for accessing the chart or one of its plots inside the container specified by the *object* parameter. Use NULL when the *object* parameter specifies a chart or plot object.

For a *Chart* object, *GlgClearDataBuffer* discards data samples in all plots of the chart. For a *Plot* object, it discards only the data samples of that plot. The method returns *GlgTrue* on success

GlgDeleteAnnotation

Deletes an annotation from a chart.

```
GlgBoolean GlgDeleteAnnotation( object, resource_name, annotation,
                                    position_x, position_y )
   GlgObject object;
   char * resource_name;
   GlgObject annotation
   double position_x;
   double position_y;
```

Parameters

object

Specifies a chart or a container object containing the chart.

resource_name

Specifies a resource name for accessing the chart inside the container. Use NULL when the *object* parameter specifies the chart.

annotation

Specifies the annotation object to delete. If set to NULL, the annotation with the timestamp specified by the *position_x* and *position_y* parameters will be deleted.

position_x, position_y

Specify the position of the annotation to be deleted when the *annotation* parameter is NULL. The parameter is ignored if *annotation* is not NULL.

The function returns *GlgTrue* if the annotation was successfully deleted. The *NumAnnotations* attribute of the chart should be set to -1 in order to use this function.

GlgDeletePlot

Deletes a plot from a chart.

```
GlgBoolean GlgDeletePlot( object, resource_name, plot )
   GlgObject object;
   char * resource_name;
   GlgObject plot;
```

Parameters

object

Specifies a chart or a container object containing the chart.

resource_name

Specifies a resource name for accessing the chart inside the container. Use NULL when the *object* parameter specifies the chart.

plot

Specifies the plot to delete. The *GlgGetNamedPlot* method may be used to obtain a plot's object ID.

The function returns *GlgTrue* if the plot was successfully deleted.

GlgDeleteTimeLine

Deletes a time line from a chart.

```
GlgBoolean GlgDeleteTimeLine(object, resource_name, time_line,
                             time_stamp )
   GlgObject object;
   char * resource_name;
   GlgObject time_line;
   double time_stamp;
```

Parameters

object

Specifies a chart or a container object containing the chart.

resource_name

Specifies a resource name for accessing the chart inside the container. Use NULL when the *object* parameter specifies the chart.

time_line

Specifies the time line to delete. If set to NULL, the time line with the timestamp specified by the *time_stamp* parameter will be deleted.

time_stamp

Specifies the timestamp of the time line to be deleted when the *time_line* parameter is NULL. The parameter is ignored if *time_line* is not NULL.

The function returns *GlgTrue* if the time line was successfully deleted. The *NumTimeLines* attribute of the chart should be set to -1 in order to use this function.

GlgGetChartDataExtent

Queries the minimum and maximum extent of the data accumulated in a chart or in one of its plots.

```
GlgBoolean GlgGetChartDataExtent( object, resource_name, min_max,
                                  x_extent, visible_only )
   GlgObject object;
   char * resource_name;
   GlgMinMax * min_max;
   GlgBoolean x_extent;
   GlgBoolean visible_only;
```

Parameters

object

Specifies a chart, a plot or a container object containing the chart.

resource_name

Specifies a resource name to access the chart inside the container, or a plot inside the chart. Use NULL when the *object* parameter specifies the chart or its plot.

min_max

The receiver of the returned data extent. It is defined in the *GlgApi.h* file:

```
typedef struct _GlgMinMax
{
   double min, max;
} GlgMinMax;
```

x_extent

If *GlgTrue*, the function returns the X extent, otherwise it returns the Y extent.

visible_only

If *GlgTrue*, the function considers only visible data samples when calculating data extent, otherwise all samples are considered.

If the *object* and *resource_name* parameters point to a chart object, the data extent of all plots in the chart is returned. If *object* and *resource_name* point to a plot, the data extent of that plot is returned. The object hierarchy must be set up for this function to succeed. The function returns *GlgTrue* on success.

GlgGetNamedPlot

Given a name of a chart's plot, returns the plot's object ID. The function returns NULL if a plot with the specified name was not found.

```
GlgObject GlgGetNamedPlot( object, resource_name, plot_name )
   GlgObject object;
   char * resource_name;
   char * plot_name;
```

Parameters

object

Specifies a chart or a container object containing a chart.

resource_name

Specifies a resource name for accessing the chart inside the container. Use NULL when the *object* parameter specifies the chart.

plot_name

Specifies the plot name.

GlgGetSelectedPlot

Returns the plot object corresponding to the last legend item selected with the mouse, if any.

```
GlgObject GlgGetSelectedPlot( void )
```

If a legend item is selected with the mouse, the corresponding plot is stored internally and returned when the function is invoked. The selected plot is stored indefinitely until it is replaced by a new plot selection. Therefore, if the last mouse selection did not select any legend items, the function returns a previously selected plot, or NULL if no plots were previously selected.

GlgSetLinkedAxis

Associates a chart's plot or a level line with an Y axis for automatic rescaling, returns *GlgTrue* on success.

```
GlgBoolean GlgSetLinkedAxis( object, resource_name, axis_object,
                             axis_resource_name )
   GlgObject object;
   char * resource_name;
   GlgObject axis;
   char * axis_resource_name;
```

Parameters

object

Specifies a plot or a level line, or a parent object containing the plot or the level line.

resource_name

Specifies a resource name for accessing the plot or the level line inside the parent. Use NULL when the *object* parameter specifies the plot or the level line.

axis_object

Specifies the axis to link with, or a parent of the axis.

axis_resource_name

Specifies a resource name for accessing the axis inside the parent. Use NULL when the *axis_object* parameter specifies an axis.

GlgSetLabelFormatter

Attaches a custom label formatter to a chart axis or a stand-alone axis object, returns *GlgTrue* on success.

```
GlgBoolean GlgSetLabelFormatter( object, resource_name, formatter )
   GlgObject object;
   char * resource_name;
   GlgLabelFormatter formatter;
```

Parameters

object

Specifies a chart or a container object containing the chart.

resource_name

Specifies a resource name for accessing the chart inside the container. Use NULL when the *object* parameter specifies the chart.

formatter

Specifies a custom label formatter.

A label formatter is a function which is invoked every time a new label string needs to be generated. It can create and return a label string, or return NULL to use a default label string. When the created label string is no longer needed, it will be freed by the Toolkit using the *GlgFree* call. Therefore, it must be allocated using *GlgAlloc* or one of the GLG string utility functions, such as *GlgStrClone* or *GlgConcatStrings*. All custom label formatters have the following function prototype:

```
typedef char * (*GlgLabelFormatter)
            ( GlgObject axis, GlgLabelType label_type,
              GlgLong value_type, double value, time_t sec,
              double fractional_sec, void * reserved );
```

A custom label formatter is invoked with the following parameters:

axis

The axis object the formatter is attached to.

label_type

The type of an axis label to be generated: GLG_TICK_LABEL_TYPE for major tick labels, GLG_SELECTION_LABEL_TYPE for chart selection labels and GLG_TOOLTIP_LABEL_TYPE for chart tooltip labels.

A label formatter may be invoked by the chart selection or chart tooltip methods if an axis label is requested by the selection or tooltip formats.

value_type

The type of the axis value: GLG_TIME_VALUE or GLG_NUMERICAL_VALUE.

value

The label's X value or timestamp.

sec

An integer number of seconds since the Epoch corresponding to the label position (time axes only).

fractional_sec

A label's fractional number of seconds.

reserved

Is reserved for future use, must be NULL.

GlgSetChartFilter

Attaches a custom data filter to a chart's plot, returns *GlgTrue* on success.

```
GlgBoolean GlgSetChartFilter( object, resource_name, filter,
                              client_data )
   GlgObject object;
   char * resource_name;
   GlgChartFilter filter;
   void * client_data;
```

Parameters

object

Specifies a chart's plot, a chart or a container object containing the chart.

resource_name

Specifies a resource name for accessing the plot inside the container. Use NULL when the *object* parameter specifies the plot.

filter

Specifies a custom data filter.

client_data

Specifies data to be passed to the custom filter.

A data filter is a function which is invoked by a chart to aggregate the plot's data. A custom data filter has the following function prototype:

```
typedef GlgLong (*GlgChartFilter)
            ( GlgChartFilterCBReason reason,
              GlgChartFilterData * filter_data, void * client_data );
```

A custom data filter is invoked with the following parameters:

reason

The reason the filter is invoked.

filter_data

Contains data passed to the filter by the chart.

client_data

Contains client data passed to the filter by the application when the filter was attached to the plot.

The *src* subdirectory of the GLG installation contains source code of custom filter examples for various programming environments (*CustomChartFilter.c, CustomChartFilter.java* and *CustomChartFilter.cs*) that provide extensive self-documented examples of custom filters. Each example implements several types of custom filters (MIN_MAX, AVERAGE and DISCARD), and contains detailed explanation of all possible values of the *reason* parameter, as well as the fields of the *filter_data* structure.

GlgUpdateChartTimeAxis

Scrolls the chart forward to show all data samples up to the provided timestamp or index, returns *GlgTrue* on success.

```
GlgBoolean GlgUpdateChartTimeAxis( object, resource_name, filter,
                                   client_data )
   GlgObject object;
   char * resource_name;
   double time_stamp;
   boolean initialize;
```

Parameters

object

Specifies a chart to be scrolled, or a chart parent.

resource_name

Specifies a resource name for accessing the chart inside the chart parent. Use NULL when the *object* parameter specifies the chart.

time_stamp

Specifies a timestamp (or an index for a chart with an index X axis).

initialize

May be set to *GlgTrue* for initializing an empty sweep chart, positioning the sweep bar at the start of the plot instead of the plot end.

While a strip chart's X axis can be updated by simply setting its *EndValue* resource to the timestamp, a sweep chart needs to update the sweep bar and update the X axis only if needed.

The *GlgUpdateChartTimeAxis* function should be used to properly update sweep charts after using the quick mode of the *GlgAddDataSampleNode* function to supply a large number of data samples. The method handles both strip and sweep charts.

GlgUpdateChartState

Triggers updating a chart's state after prefilling it with data using the quick mode, returns *GlgTrue* on success.

```
GlgBoolean GlgUpdateChartState( object, resource_name, state )
   GlgObject object;
   char * resource_name;
   GlgChartState state;
```

Parameters

object

Specifies a chart to be scrolled, or a chart parent.

resource_name

Specifies a resource name for accessing the chart inside the chart parent. Use NULL when the *object* parameter specifies the chart.

state

Contains binary flags that specify the state to be updated:

- GLG_CHART_AUTOSCALE_STATE - updates chart's autoscale
- GLG_CHART_BUFFER_STATE - triggers trimming of the chart data buffer based on the chart's *BufferXSpan* and *BufferSize* attributes.
- GLG_CHART_SCROLLBAR_STATE - updates position of the chart's scrollbars, if any.

Multiple flags can be combined using the bitwise OR operator.

This function marks the selected chart states for updating. The actual update happens on the next invocation of the *GlgUpdate* or *GlgSetupHierarchy* functions.

Chart Functions of the Intermediate API

GlgAddDataSampleNode

Adds a data sample to a chart, is used to prefill the chart with a large number of samples:

```
GlgBoolean GlgAddDataSampleNode( plot, datasample_node, quick_mode )
   GlgObject plot;
   GlgDataSampleNode * datasample_node;
   GlgBoolean quick_mode;
```

Parameters

plot

A chart's plot to add datasample to.

datasample_node

A data sample node containing the data sample to be added. The *GlgCreateDataSampleNode* function may be used to create data sample nodes.

quick_mode

Enables a quick mode that bypasses chart autoscroll and autoscale logic for faster chart prefilling. It also disables all validity checks for increased performance.

When the last data sample is added, a chart with the SCROLL X axis can be scrolled to show the last added sample by setting *EndValue* of the chart's X axis to the timestamp of the last data sample.

For a chart with the SWEEP X axis, the *GlgUpdateChartTimeAxis* function can be used to scroll the chart.

The *GlgUpdateChartState* function may also be used to update the chart's autoscale and scrollbars when done.

This function is used to enhance performance when prefilling a chart with a large number of data samples and returns *GlgTrue* on success. For periodic updates, new data may be pushed into the plot's entry points using *GlgSetDResource*.

It is recommended that the initial development is done with the quick mode disabled, enabling it when all potential errors have been dealt with.

The RealTime Chart Demo provides a coding example of prefilling a chart with data.

GlgCreateDataSampleNode

Creates a data sample node structure to be used with the *GlgAddDataSampleNode* function:

```
GlgDataSampleNode * GlgCreateDataSampleNode( plot, extended )
   GlgObject plot;
   GlgBoolean extended;
```

Parameters

plot

An optional plot the data sample node will be added to. If provided, a new data sample node will be taken from the plot's cache, if possible.

extended

Specifies the type of the data sample to be created inside the node. If set to *GlgTrue*, an extended data sample (*GlgDataSampleExt*) is created, otherwise a data sample (*GlgDataSample*) is created.

The *GlgGetNodeDataSample* function can be used to get access to the node's data sample. The fields of the data sample structure must be filled before adding it to the plot.

GlgGetNodeDataSample

Returns the node's data sample. Depending on the type of the data sample contained in the node, returns either *GlgDataSample* or *GlgDataSampleExt* pointer that can be used to fill the data sample's fields before adding it to the chart.

```
GlgDataSample * GlgGetNodeDataSample( node )
   GlgDataSampleNode node;
```

Parameters

node

A data sample node.

GlgFreeDataSampleNode

Returns a data sample node to the plot cache for reuse, if possible.

```
void GlgFreeDataSampleNode( plot, node )
   GlgObject plot;
   GlgBoolean extended;
```

Parameters

plot

An optional plot the data sample node belongs to. If provided, the data sample node will be returned to the plot's cache. Otherwise, the node will be freed.

node

The data sample node to be freed.

GlgCreateChartSelection

Selects a chart's data sample closest to the specified position and returns a message object containing the selected sample's information.

```
GlgObject GlgCreateChartSelection( object, plot, x, y, dx, dy,
                                   screen_coord, include_invalid,
                                   x_priority )
   GlgObject object;
   GlgObject plot;
   double x, y;
   double dx, dy;
   GlgBoolean screen_coord;
   GlgBoolean include_invalid;
   GlgBoolean x_priority;
```

Parameters

object

Specifies a chart object. If the *plot* parameter is NULL, the function queries samples in all plots of the chart and selects the data sample closest to the specified position.

plot

Specifies an optional plot object. If it is not NULL, the function queries only the data samples in that plot.

x, y

Specify the position inside the chart. If the *screen_coord* parameter is set to *GlgTrue*, the position is defined in the screen coordinates of the chart's parent viewport. If *screen_coord* is set to *GlgFalse*, the *x* position is defined as an X or time value, and the *y* position is defined in relative coordinates in the range [0; 1] to allow the chart to process selection for plots with different ranges (0 corresponds to the *Low* range of each plot, and 1 to the *High* range).

When the mouse position is used, add GLG_COORD_MAPPING_ADJ to the x and y screen coordinates of the mouse for precise mapping.

dx, dy

Specify an extent around the point defined by the *x* and *y* parameters. The extent is defined in the same coordinates as *x* and *y*, depending on the value of the *screen_coord* parameter. Only the data samples located within the *dx* and *dy* distances of the specified position are considered for the selection.

screen_coord

Specifies the type of the *x* and *y* coordinates. If set to *GlgTrue*, the *x* and *y* parameters supply screen coordinates in the chart's parent viewport, otherwise they supply relative coordinates.

include_invalid

If set to *GlgTrue*, invalid data samples are considered for selection, otherwise invalid samples are never selected.

x_priority

If set to *GlgTrue*, the function selects the data sample closest to the specified position in the

horizontal direction with no regards to its distance from the specified position in the vertical direction. If it is set to *GlgFalse*, a data sample with the closest distance from the specified position is selected.

The information about the selected data sample is returned in the form of a message object described in the *Chart Selection Message Object* section on page 396. The function returns NULL if no data sample matching the selection criteria was found. The returned message object must be dereferenced using the *GlgDropObject* function when finished.

GlgCreateTooltipString

Creates and returns a chart or axis tooltip string corresponding to the specified position in screen coordinates:

```
char * GlgCreateTooltipString( object, x, y, dx, dy, format )
   GlgObject object;
   double x, y;
   double dx, dy;
   char * format;
```

Parameters

object

Specifies a chart or axis object for creating the tooltip.

x, y

Specifies a screen position on top of a chart or an axis. When using the cursor position, add GLG_COORD_MAPPING_ADJ to the cursor's X and Y screen coordinates for precise mapping.

dx, dy

Specify the maximum extent in screen pixels around the specified position.

format

Provides a custom format for creating a tooltip string and uses the same syntax as the *TooltipFormat* attributes of the chart and axis objects. If it is set to NULL or an empty string, the format provided by the *TooltipFormat* attribute of the chart or the axis will be used.

A special "<*single_line*>" tag may be prepended to the format string to force the function to remove all line breaks from the returned tooltip string. A *"<chart_only>"* tag may also be prepended to search only for a chart tooltip and disable search for the axes' tooltips when the function is invoked for a chart object.

If the *object* parameter specifies a chart, and the selected position is within the chart's data area, the function selects a data sample closest to the specified position and returns a tooltip string containing information about the sample. The *dx* and *dy* parameters define the maximum extent for selecting a data sample, and the *TooltipMode* attribute of the chart defines the data sample selection mode: X or XY. The function returns NULL if no data samples were found within the *dx* and *dy* distances of the specified position.

If the *object* parameter specifies an axis, or if the *object* parameter specifies a chart and the selected position is on top of one of the chart's axes, the function returns a tooltip string that displays the axis value corresponding to the specified position.

The returned string must be freed using the *GlgFree* function.

While the *Chart* object proves an integrated tooltip functionality that automatically displays the tooltip without any application involvement when a mouse is hovering over the chart, *GlgCreateTooltipString* can be used to generate a string containing information about the data sample under the current mouse position that is displayed as the mouse continuously moves over the chart. The *stripchart.c* file of the *GLG Real-Time Strip-Chart* demo demonstrates the use of the function to create a tooltip string that displays such cursor feedback information at the top of the chart.

GlgGetLegendSelection

Returns the plot object corresponding to the legend item at the specified position, if any. If no legend item is found at the specified position, NULL is returned.

```
GlgObject GlgGetLegendSelection( object, x, y )
   GlgObject object;
   double x, y;
```

Parameters

object

Specifies a legend object.

x, y

Specifies the position in screen coordinates of the viewport containing the legend.

GlgPositionToValue

Returns a value corresponding to the specified position on top of a chart or an axis object.

```
GlgBoolean GlgPositionToValue( object, resource_name, x, y,
                               outside_x, outside_y, value )
   GlgObject object;
   char * resource_name;
   double x, y;
   GlgBoolean outside_x, outside_y;
   double * value;
```

Parameters

object

If resource_name is NULL, specifies a chart, a plot or an axis object, otherwise it specifies a container object containing a chart, a plot or an axis.

resource_name

Specifies a resource path of a child chart, plot or axis object relative to the container object.

x, y

Specify a position in the screen coordinates of the parent viewport. When using the cursor position, add GLG_COORD_MAPPING_ADJ to the cursor's x and y coordinates for precise mapping.

outside_x

If set to *GlgTrue*, the function returns *GlgFalse* if the specified position is outside of the X extent of the chart or axis object.

outside_y

If set to *GlgTrue*, the function returns *GlgFalse* if the specified position is outside of the Y extent of the chart or axis object.

value

A pointer to the returned value.

For a plot, the function converts the specified position to a Y value in the Low/High range of the plot and returns the value through the *value* pointer.

For an axis, the function converts the specified position to the axis value and returns the value.

For a chart, the function converts the specified position to the X or time value of the chart's X axis and returns the value.

The function returns *GlgTrue* on success.

2.8 GIS Object Functions

The following Standard API methods may be used to handle user interaction with the GLG GIS object:

- **GlgGISConvert** performs coordinate conversion from the GIS coordinates of the GIS Object to the GLG coordinates of the drawing and vise versa.
- **GlgGISGetElevation** returns an elevation of a specified lat/lon point on a map.
- **GlgGISCreateSelection** returns information about GIS features located at a specified map location.
- **GlgGISGetDataset** retrieves the dataset object of a GIS object
- **GlmConvert** performs coordinate conversion from the GIS coordinates of a map to the GLG coordinates of the drawing and visa versa without the help of a GLG GIS Object.

GIS Function Descriptions

GlgGISConvert

Performs coordinate conversion from the GIS coordinates of the GIS Object to the GLG coordinates of the drawing and visa versa.

```
GlgBoolean GlgGISConvert( object, resource_name, coord_type,
            coord_to_lat_lon, in_point, out_point )
   GlgObject object;
   char * resource_name;
   GlgCoordType coord_type;
   GlgBoolean coord_to_lat_lon;
   GlgPoint * in_point;
   GlgPoint * out_point;
```

Parameters

object

If the *resource_name* parameter is NULL, specifies the GIS Object whose coordinate system to use for conversion. If the *resource_name* parameter is not NULL, specifies a parent of the GIS Object.

resource_name

If not NULL, specifies the resource name of the GIS Object inside the parent object specified by the *object* parameter.

coordinate_type

The type of the GLG coordinate system to convert to or from: *GLG_SCREEN_COORD* for screen coordinates or *GLG_OBJECT_COORD* for world coordinates.

coord_to_lat_lon

GlgTrue to convert from GLG coordinates to GIS longitude and latitude, *GlgFalse* to convert from GIS to GLG coordinates.

in_point

A pointer to the *GlgPoint* structure containing coordinate values to be converted.

out_point

A pointer to the *GlgPoint* structure that receives converted coordinate values.

The function returns *GlgTrue* if conversion succeeds. When converting from GLG to GIS coordinates, the Z coordinate is set to a negative GLG_GIS_OUTSIDE_VALUE value for points on the invisible part of the globe.

The *GlmConvert* function of may be used to perform coordinate conversion between GIS and GLG coordinates without the use of the GIS object.

GlgGISCreateSelection

Provides a high-level interface to the map server's *GlmGetSelection* function; returns a message object containing information about GIS features located at a specified position on the map.

```
GlgObject GlgGISCreateSelection( object, resource_name, layers,
            x, y, select_labels)
   GlgObject object;
   char * resource_name;
   char * layers;
   double x, y;
   GlgLong select_labels;
```

Parameters

object

If the *resource_name* parameter is NULL, specifies the GIS Object to query. If the *resource_name* parameter is not NULL, specifies a parent of the GIS Object.

resource_name

If not NULL, specifies the resource name of the GIS Object inside the parent object specified by the *object* parameter.

layers

A list of layers to query.

x, y

Specifies coordinates of a point in the screen coordinates of a viewport that contains the GIS object.

select_labels

Specifies the label selection mode, can have the following values (defined in the *GlmLabelSelectionMode* enum in the *GlmApi.h* file):

- GLM_LBL_SEL_NONE - GIS features' labels are not considered for the GIS selection.
- GLM_LBL_SEL_IN_TILE_PRECISION - enables labels to be considered for the GIS selection. For a faster selection, this option does not check labels that belong to GIS features in a different tile but extend to the current tile.
- GLM_LBL_SEL_MAX_PRECISION - enables labels to be considered for the GIS selection and performs selection with the maximum label precision.

The layer's settings have precedence over the *select_labels* parameter. The labels will be considered for selection only for the layers that do not override it by setting their LABEL SELECTION MODE = NONE in the layer's LIF file. If LABEL SELECTION MODE = INTILE, the labels will always be considered using the IN_TILE precision.

The function returns a message object containing information about the selected GIS features. The message object has to be dereferenced using the *GlgDropObject* function when finished.

If the GIS verbosity level is set to 2000 or 2001, extended information is also written into the error log file and printed to the terminal on Linux/Unix for debugging purposes.

Refer to the *GlmGetSelection* section on page 129 of the *GLG Map Server Reference Manual* for information on the structure of the returned message object as well as a programming example that demonstrates how to handle it.

GlgGISGetDataset

Returns a dataset object associated with the GIS object:

```
GlgObject GlgGISGetDataset( object, resource_name )
   GlgObject object;
   char * resource_name;
```

Parameters

object

If the *resource_name* parameter is NULL, specifies the GIS Object to query. If the *resource_name* parameter is not NULL, specifies a parent of the GIS Object.

resource_name

If not NULL, specifies the resource name of the GIS Object inside the parent object specified by the *object* parameter.

The function may be invoked after the GIS object has been setup to retrieve its dataset. The dataset may be used to dynamically change attributes of individual layers displayed in the GIS object. The program can use GlgSetResource methods for changing layer attributes. Refer to the *GLG Map Server Reference Manual* for information on layer attributes. The *GLG GIS Demo* provides an example of changing layer attributes programmatically.

GlgGISGetElevation

Returns an elevation of a lat/lon point on a map:

```
GlgBoolean GlgGISGetElevation( object, resource_name, layer_name,
            lon, lat, elevation )
   GlgObject object;
   char * resource_name;
   double lon, lat;
   double * elevation;
```

Parameters

object

If the *resource_name* parameter is NULL, specifies the GIS Object to query. If the *resource_name* parameter is not NULL, specifies a parent of the GIS Object.

resource_name

If not NULL, specifies the resource name of the GIS Object inside the parent object specified by the *object* parameter.

layer_name

The name of the layer with elevation data to query.

lon, lat

The lon/lat coordinates of a point on the map.

elevation

A pointer to the variable of a *double* type that will receive the returned elevation data.

The function returns *GlgFalse* if the point is outside of the map in the ORTHOGRAPHIC projection or if there is no elevation data for the specified location. If the function returns *GlgTrue*, the elevation value is returned in the units (such as meters or feet) defined by the elevation data file.

GlmConvert

A low level function that performs coordinate conversion from the GIS coordinates of a map to the GLG coordinates of the drawing and visa versa without the help of a GLG GIS Object. Since the function does not rely on a GIS Object being displayed and set up, it may be used before the drawing is rendered or without a drawing at all.

```
void GlmConvert( projection, stretch, coord_type, coord_to_lat_lon,
            center, extent, angle,
            min_x, max_x, min_y, max_y, in_point, out_point )
   GlgProjectionType projection;
   GlgBoolean stretch;
   GlgCoordType coord_type;
   GlgBoolean coord_to_lat_lon;
   GlgPoint * center, * extent;
   double angle;
   double min_x, max_x, min_y, max_y;
   GlgPoint * in_point, * out_point;
```

Parameters

projection

A GIS map projection, may have values of GLG_RECTANGULAR_PROJECTION or GLG_ORTHOGRAPHIC_PROJECTION.

stretch

Specifies if the map preserves the X/Y ratio (GlgFalse) or not (GlgTrue).

coordinate_type

The type of the GLG coordinate system to convert to or from: *GLG_SCREEN_COORD* for screen coordinates or *GLG_OBJECT_COORD* for world coordinates.

coord_to_lat_lon

GlgTrue to convert from GLG coordinates to GIS longitude and latitude, *GlgFalse* to convert from GIS to GLG coordinates.

center

A pointer to the *GlgPoint* structure containing lat/lon coordinates of the map center. The Z coordinate must be set to 0.

extent

A pointer to the *GlgPoint* structure containing X and Y extent of the map in the selected projection: in degrees for the rectangular projection or in meters for orthographic. The Z extent must be set to 0. Refer to the *GIS Object* section on page 83 of the *GLG User's Guide and Builder Reference Manual* for more details.

angle

A map rotation angle in degrees.

min_x, max_x, min_y, max_y

A map extent in the selected coordinate system. To get X/Y coordinates relative to the map image origin, use `min_x=0, min_y=0, max_x=image_width, max_y=image_height`.

in_point

A pointer to the *GlgPoint* structure containing coordinate values to be converted.

out_point

A pointer to the *GlgPoint* structure that receives converted coordinate values.

When converting from X/Y to lat/lon coordinates in the orthographic projection, the Z coordinate is set to a negative value for points on the invisible part of the globe or outside the globe area, and to zero for points on the edge of the globe. The coordinates of the returned point may be outside of the visible portion of the map for all projections.

2.9 GLG Installable Interface Handlers

GLG Installable Interface Handler Utilities assist an application developer with implementing functionality of complex user interactions usually encountered in editor-style applications. The utilities include a collection of methods as part of the GLG Intermediate API library and are available for the C/C++, Java, C#/.NET and JavaScript environments. The *GLG Diagram* demo provides an example of a custom diagraming editor application implemented using the GLG Installable Interface Handlers functionality.

The mechanism of interface handlers facilitates rapid application development for writing elaborate GUI with sophisticated application functionality for handling user interaction.

Overview

Installable Interface Handlers (referred to as interface handlers in the rest of this manual) provide a mechanism for handling user interaction where persistency is necessary to handle interrelated sequences of user actions. The interface handlers are hierarchical, allowing an application to handle nested operations, such as nested dialogs or chained operations. Internally, a stack is used to maintain a hierarchy of handlers, making it possible to handle arbitrary nested sequences, for example display a confirmation dialog based on the action of the parent dialog (such as parent dialog closing).

The use of the stack allows to easily pass control to the previous handler in the stack after the currently active handler is uninstalled and deleted from the stack. The handlers also automate the flow of control: if the currently active handler on the bottom of the stack is not interested in the event, it can uninstall itself from the stack and pass the event to the previous handler on the stack, which, in turn, can pass the event further to the next handler in the stack.

For example, a top-level handler can be used to handle events from toolbar icons and main pull-down menu, and a nested second-level handler can be installed to handle a popup dialog. A third-level handler can be used to confirm closing of the dialog managed by the second handler. When the dialog closing is confirmed, the third and then the second handlers are uninstalled (get removed from the stack), and the control returns to the main top-level handler.

Each handler maintains its own persistent data storage for intermediate data, which is automatically cleaned up when the handler is uninstalled. A variable number of dynamic parameters can be supplied to each handler to modify its behavior as needed. Parameters can be either optional or mandatory. Optional parameters allow the developer to extend a handler's functionality while maintaining backward compatibility.

User interaction events passed to an interface handler are encoded as integer *tokens* for efficiency and are passed to the currently active handler (the bottom handler currently stored on a stack).

The source code for each handler is provided by an application; the code handles a set of user interaction events identified by the tokens. A handler is typically designed to handle a particular set of tokens, for example tokens for an Apply or Cancel button for a specific dialog. If an unrecognized

token is encountered, the handler can either ignore the event to implement a modal dialog, or it can uninstall itself and pass the event to the parent handler, which, in turn, can pass it to its parent, and so on.

Interface handlers extend functionality of event handling callbacks and listeners (referred collectively as callbacks in the rest of this section). A disadvantage of a callback is that it does not provide data persistency: when a callback is invoked to handle a particular event, any changes to an application state have to be kept in an external global structure shared by all callbacks. A callback exits after processing each event and does not provide any means for storing intermediate interaction state in the callback itself. For example, if a dialog has several buttons, individual callbacks are invoked on each button press event, making it difficult to implement a single integrated event handler for the dialog as a whole.

GLG interface handlers address this problem by providing event handlers that support persistency and provide data storage for intermediate data that control user interaction. An interface handler stays active to process all events for a dialog or a page, until the handler is uninstalled when dialog closes or a page is switched to display another page. The interface handlers provide a flexible alternative to the Hierarchical State Machines (HMS) that are often used to handle the state of the user interface transitions. Unlike HMS, interface handlers allow a developer to extend the handler functionality by adding handler parameters and augment the handler source code based on the parameter values passed to it, which could be easier than adding new states to the state machine.

The **GLG Diagram Demo** provides examples of implementing several design patterns for handling various types of user interaction. Many other alternative options of using the interface handlers are possible, and an application can implement any desirable design pattern depending on the application requirements.

List of Functions

The following Installable Interaction Handlers functions are provided as a part of the GLG Intermediate API library:

- **GlgIHInit** initializes installable handler utilities.
- **GlgIHTerminate** terminates handler utilities.
- **GlgIHGlobalData** provides access to the global data storage.
- **GlgIHInstall** installs a handler.
- **GlgIHStart** starts a handler.
- **GlgIHResetup** restarts a handler.
- **GlgIHUninstall** uninstalls a handler.
- **GlgIHUninstallWithToken** uninstalls a handler and passes a token to the parent handler.
- **GlgIHUninstallWithEvent** uninstalls a handler and passes an event to the parent handler.
- **GlgIHGetCurrIH** returns the current handler.
- **GlgIHGetPrevIH** returns the current handler’s parent handler.

- **GlgIHGetFunction** returns the entry point of the current handler.
- **GlgIHGetPrevFunction** returns the entry point of the current handler's parent handler.
- **GlgIHGetType** returns a type of an interface event.
- **GlgIHGetToken** returns an event's token.
- **GlgIHCallCurrIH** passes an interface event to the current handler.
- **GlgIHCallCurrIHWithToken** passes a token to the current handler.
- **GlgIHCallCurrIHWithModifToken** modifies the event's token and passes it to the current handler.
- **GlgIHCallPrevIHWithToken** passes a token to the current handler's parent handler.
- **GlgIHCallPrevIHWithModifToken** modifies the event's token and passes it to the current handler's parent.
- **GlgIHPassToken** installs a handler and passes it a token; it can be used to implement pass-through handlers for processing global accelerators, stateless options, as well as managing floating stay-open dialogs.
- **GlgIHSetIParameter**
  GlgIHSetPParameter
  GlgIHSetSParameter
  GlgIHSetOParameter
  GlgIHSetDParameter
  GlgIHSetOParameterFromD
  GlgIHSetOParameterFromG
  set handler parameters of various data types.
- **GlgIHChangeIParameter**
  GlgIHChangePParameter
  GlgIHChangeSParameter
  GlgIHChangeOParameter
  GlgIHChangeDParameter
  change handler parameters.
- **GlgIHGetIParameter**
  GlgIHGetPParameter
  GlgIHGetSParameter
  GlgIHGetOParameter
  GlgIHGetDParameter
  queries handler parameters.
- **GlgIHGetOptIParameter**
  GlgIHGetOptPParameter
  GlgIHGetOptSParameter
  GlgIHGetOptOParameter
  GlgIHGetOptDParameter
  queries optional handler parameters.

Function Descriptions

To use any of the functions described in this chapter, the application program must include the function definitions from the GLG API header file. Use the following line to include those definitions:

```
#include "GlgApi.h"
```

GlgIHInit

Initializes installable handler utilities, must be invoked after *GlgInit* but before any installable handler methods.

```
void GlgIHInit()
```

GlgIHTerminate

Terminates installable handler utilities. All currently installed handlers must be uninstalled before invoking this method.

```
void GlgIHTerminate()
```

GlgIHGlobalData

Returns the global data container.

```
void GlgIHGlobalData()
```

GlgIHInstall

Creates a handler with the specified entry point and installs the handler.

```
GlgIH GlgIHInstall( GlgIHEntryPoint func )
```

The function creates the handler, adds it to a stack of handlers and returns the handler's ID. The stack of handlers is used to keep track of nested handlers; the last handler pushed onto the stack becomes the active handler, referred to as the *current handler* in the rest of this manual.

The *func* parameter specifies the entry point function of the new handler with the following function prototype:

```
void GlgIHEntryPoint( GlgIH ih, GlgIHCallEvent call_event )
```

The handler's entry point is invoked with the *call_event* parameter that provides the user interface event. The *ih* parameter provides the ID of the handler the entry point belongs to, and can be used to access parameters stored in the handler's data storage. The code of this function implements functionality of the handler. Parameters can be passed to a handler after is has been installed. An example of a handler code is shown in the description of the *GlgIHStart* function below.

GlgIHStart

Invokes the handler's entry point with the GLG_HI_SETUP_EVENT event, which allows the handler to perform any desired initialization, such as displaying a dialog associated with the handler, if any.

```
void GlgIHStart()
```

Parameters can be passed to a handler by storing them in the installed handler's data storage via *SetParameter* methods prior to invoking *GlgIHStart*, as shown in the following example:

```
GlgIHInstall( ConfirmIH )
GlgIHSetOParameter( GLG_IH_NEW, "ok_dialog", ok_dialog );
GlgIHSetSParameter( GLG_IH_NEW, "message", "OK to discard changes?" );
GlgIHStart();
```

The first argument of the *SetParameter* methods defines the handler to add parameters to. For convenience, a GLG_IH_NEW macro can be used to supply the ID of the just installed handler instead of using the handler ID returned by *GlgIHInstall*, as shown in the above example.

The following shows an example of a handler code that uses parameters from the above example to initialize the handler. The handler displays a confirmation dialog with a supplied message on start up and closes it when the handler is uninstalled. An optional *modal* parameter specifies if the dialog is modal.

```
void ConfirmIH( GlgIH ih, GlgIHCallEvent call_event )
{
   GlgObject ok_dialog;
   char * message;
   GlgCallEventType event_type;
   GlgIHToken token;

   /* Get the OK dialog ID provided as a parameter when the handler was
      installed.
   */
   ok_dialog = GlgGetOParameter( ih, dialog );
   event_type = GlgIHGetType( call_event );
   switch( event_type )
   {
    case GLG_HI_SETUP_EVENT:
      /* Display a dialog with the requested message. */
      message = GlgIHGetOptSParameter( ih, "message" );
      GlgSetSResource( ok_dialog, "DialogMessageString", message );
      GlgSetDResource( ok_dialog, "Visibility", 1. );
      GlgUpdate( ok_dialog );
      break;

    case GLG_MESSAGE_EVENT:
      token = GlgIHGetToken( call_event );
      switch( token )
      {
       case IH_OK:
       case IH_CANCEL:
         /* Uninstall the handler and pass selection to the parent. */
```

```
            GlgIHUninstallWithToken( token );
            break;

          default:
            if( GlgIHGetOptIParameter( ih, "modal_dialog", GlgFalse ) )
               /* Don't allow to leave in a modal mode.*/
               GlgBell( Viewport );
            else
               /* Uninstall the handler and pass event to the parent. */
               GlgIHUninstallWithEvent( call_event );
            break;
          }
        break;

        case GLG_CLEANUP_EVENT:
          /* Erase the dialog. */
          GlgSetDResource( ok_dialog, "Visibility", 0. );
          GlgUpdate( ok_dialog );
          break;
       }
   }
```

The IH_OK and IH_CANCEL tokens used in the above example are generated by the GLG Input callback that converts interface events to integer tokens for efficiency. The tokens are then passed to the currently active handler via the *GlgIHCallCurrIHWithToken* function call. Refer to the GLG Diagram demo for a complete source code example.

GlgIHResetup

Invokes the handler's entry point with the GLG_HI_RESETUP_EVENT event.

```
void GlgIHResetup( GlgIH ih )
```

GlgIHResetup can be used to reinitialize the handler by sharing some of the GLG_HI_SETUP_EVENT code, as shown in the following example:

```
void ConfirmIH( GlgIH ih, GlgIHCallEvent call_event )
{
   GlgObject ok_dialog;
   char * message;
   GlgCallEventType event_type;

   /* Get the OK dialog ID provided as a parameter when the handler was
      installed.
*/
   ok_dialog = GlgGetOParameter( ih, dialog );

   event_type = GlgIHGetType( call_event );
   switch( event_type )
   {
    case GLG_HI_SETUP_EVENT:
      /* Set the dialog's message to the message provided by the
        caller.
      */
```

```
        message = GlgIHGetOptSParameter( ih, "message" );
        GlgSetSResource( ok_dialog, "DialogMessageString", message );
        /* Fall through to popup the dialog. */

      case GLG_HI_RESETUP_EVENT: /* Popup the dialog again. */
        GlgSetDResource( ok_dialog, "Visibility", 1. );
        GlgUpdate( ok_dialog );
        break;

    ...
    }
}
```

GlgIHUninstall

Uninstalls the current handler (the last handler on the stack).

```
void GlgIHUninstall()
```

The handler's entry point is invoked with the GLG_CLEANUP_EVENT event prior to uninstalling to let the handler perform any required cleanup. After the handler is uninstalled, a previous handler in the stack (if any) becomes the active current handler.

Uninstalling a handler deletes its stored data and invalidates its ID; the ID should not be used after the handler has been uninstalled.

GlgIHUninstallWithToken

Uninstalls the handler using *GlgIHUninstall* and invokes the previous handler (which becomes current after *GlgIHUninstall* is invoked) with the specified token.

```
void GlgIHUninstallWithToken( GlgIHToken token )
```

GlgIHUninstallWithEvent

Uninstalls the handler using *GlgIHUninstall* and passes the event to the previous handler (which becomes current after *GlgIHUninstall* is invoked).

```
void GlgIHUninstallWithEvent( GlgIHCallEvent call_event )
```

If a handler receives an event that is not related to the handler's logic, this function can be used to uninstall the handler and pass the event to the previous handler.

GlgIHGetType

Returns event's type.

```
GlgCallEventType GlgIHGetType( GlgIHCallEvent call_event )
```

The type can be one of the following:

GLG_HI_SETUP_EVENT
received when the handler is started using *GlgIHStart*; is used to perform any required initialization.

GLG_HI_RESETUP_EVENT
received when the handler is reinitialized via *GlgIHResetup*.

GLG_CLEANUP_EVENT
received before destroying the handler when it is uninstalled; is used to perform any required cleanup.

GLG_MESSAGE_EVENT
a message event that is further identified by an associated token.

GlgIHGetToken

Returns a token associated with the GLG_MESSAGE_EVENT event.

```
GlgIHToken GlgIHGetToken( GlgIHCallEvent call_event )
```

Tokens are application-defined integer values used to uniquely identify each user interaction event. For example, IH_OK token can be associated with pressing a dialog's OK button, and IH_MOUSE_MOVED can identify a mouse move event that provides coordinates of the cursor. Integer values are used to allow efficient event processing using switch statements. An enum is usually used to define tokens, for example:

```
enum MyTokens
{
   IH_UNDEFINED_TOKEN = 0,
   IH_OK,
   IH_CANCEL
}
```

In a GLG application, Input and Trace callbacks are used to convert interface events to tokens, as shown in the GLG Diagram demo. The tokens are then passed to the currently active handler via the *GlgIHCallCurrIHWithToken* function call.

GlgIHCallCurrIHWithToken

Invokes the entry point of the current handler (the last handler on the stack) with the specified token.

```
GlgBoolean GlgIHCallCurrIHWithToken( GlgIHToken token )
```

Returns Glg*True* if the handler was uninstalled as a result of processing the token.

The function creates a call event object used to pass the token to the handler, and destroys the event object when done.

GlgIHCallCurrIHWithModifToken

Same as *GlgIHCallCurrIHWithToken*, except that it reuses the existing event by setting its token instead of creating a new call event to pass the token to the handler.

```
GlgBoolean GlgIHCallCurrIHWithModifToken( GlgIHCallEvent call_event,
                                                    GlgIHToken token )
```

Returns Glg*True* if the handler was uninstalled as a result of processing the token.

GlgIHCallCurrIH

Passes the event to the entry pointy of the current handler.

```
GlgBoolean GlgIHCallCurrIH( GlgIHCallEvent call_event )
```

Returns *GlgTrue* if the handler was uninstalled as a result of processing the event.

GlgIHCallPrevIHWithToken

Passes the token to the parent handler of the currently active handler. The parent handler is the handler preceding the current handler on the stack.

```
void GlgIHCallPrevIHWithToken( GlgIHToken token )
```

Please note that when the parent handler is invoked, the current handler is not the parent handler but the handler that invoked it. Therefore, the parent handler cannot uninstall itself by simply calling *GlgIHUninstall*.

GlgIHCallPrevIHWithModifToken

Same as *GlgIHCallPrevIHWithToken* but reuses the event object for efficiency, the same way as *GlgIHCallCurrIHWithModifToken*.

```
void GlgIHCallPrevIHWithModifToken( GlgIHCallEvent call_event,
                                              GlgIHToken token )
```

GlgIHGetCurrIH

Returns the ID of the current handler.

```
GlgIH GlgIHGetCurrIH()
```

GlgIHGetFunction

Returns the function used to implement the handler.

```
GlgIHEntryPoint GlgIHGetFunction( GlgIH ih )
```

The GLG_IH_CURR macro can be used as the *ih* parameter to access the current handler.

The following example demonstrates the use of *GlgIHGetFunction* to identify the currently active handler.

```
extern GlgIHEntryPoint MyIH;

if( GlgIHGetFunction( GLG_IH_CURR ) == MyIH )
   ...
```

GlgIHGetPrevIH

Returns the ID of the current handler's parent handler.

```
GlgIH GlgIHGetPrevIH()
```

GlgIHGetPrevFunction

Returns the function used to implement the parent handler of the currently active handler. The parent handler is the handler preceding the current handler on the stack.

```
GlgIHEntryPoint GlgIHGetPrevFunction()
```

GlgIHPassToken

Installs the specified handler and invokes its entry point with the specified token.

```
void GlgIHPassToken( GlgIHEntryPoint func, GlgIHToken token,
                     GlgBoolean uninstall )
```

If the *uninstall* parameter is *GlgTrue*, the handler is uninstalled after processing the token, unless the handler has already uninstalled itself. If the handler installs other handlers while processing the token, the last installed handler (the current handler) will be uninstalled instead of the handler the token is passed to.

This function is used to implement pass-through handlers for processing global accelerators, stateless options, as well as managing floating stay-open dialogs, such as the Edit Properties dialog shown on the GlgDiagram demo.

GlgIHSetIParameter
GlgIHSetDParameter
GlgIHSetSParameter
GlgIHSetOParameter
GlgIHSetPParameter
GlgIHSetOParameterFromD
GlgIHSetOParameterFromG

Creates a named parameter of a requested type: I (integer), D (double), S (string), O (GLG object) or P (pointer).

GlgIHSetOParameterFromD and *GlgIHSetOParameterFromG* create a GLG data object parameter containing double (D) or geometrical (G) data.

```
void GlgIHSetIParameter( GlgIH ih, char * name, GlgLong value )
void GlgIHSetDParameter( GlgIH ih, char * name, double value )
void GlgIHSetSParameter( GlgIH ih, char * name, char * value )
void GlgIHSetOParameter( GlgIH ih, char * name, GlgObject value )
void GlgIHSetPParameter( GlgIH ih, char * name, void * value )
void GlgIHSetOParameterFromD( GlgIH ih, char * name, double value );
void GlgIHSetOParameterFromG( GlgIH ih, char * name, double value1,
                              double value2, double value3 );
```

The *name* argument specifies the name of the parameter; this name can be used to query the value of the parameter using *GetParameter* functions. If a parameter with the specified name already exists, it will be replaced, which can change the type of the parameter associated with the name.

A parameter is added to the data storage of the handler specified by the *ih* argument. The GLG_IH_CURR macro can be used to specify the currently active handler, which can eliminate the need to pass the handler ID into all functions the handler may invoke.

The GLG_IH_GLOBAL macro can be used to add parameters to the global data storage instead of the data storage of the current handler.

When parameters are added, string parameters are cloned and object parameters are referenced. When the handler is uninstalled, all parameters stored in its data storage are automatically cleaned up: string parameters are freed and object parameters are dereferenced. If pointer parameters are used to store addresses of allocated memory, the memory should be freed, as shown in the following example.

```
case GLG_HI_SETUP_EVENT:
   ptr = GlgAlloc( sizeof my_struct );
   GlgIHSetPParameter( ih, "my_struct", ptr );
break;

...

case GLG_CLEANUP_EVENT:
   ptr = GlgIGGetPParameter( "my_struct" );
   GlgFree( ptr );
   break;
```

GlgIHChangeIParameter
GlgIHChangeDParameter
GlgIHChangeSParameter
GlgIHChangeOParameter
GlgIHChangePParameter

Same as corresponding SetParameter functions, except that the named parameter must exist. If the parameter does not exist, a GLG error is generated.

```
void GlgIHChangeIParameter( GlgIH ih, char * name, GlgLong value )
void GlgIHChangeDParameter( GlgIH ih, char * name, double value )
void GlgIHChangeSParameter( GlgIH ih, char * name, char * value )
void GlgIHChangeOParameter( GlgIH ih, char * name, GlgObject value )
void GlgIHChangePParameter( GlgIH ih, char * name, void * value )
```

GlgIHGetIParameter
GlgIHGetDParameter
GlgIHGetSParameter
GlgIHGetOParameter
GlgIHGetPParameter

Returns a value of a named parameter of a requested type: I (integer), D (double), S (string), O (GLG object) or P (pointer).

```
GlgLong GlgIHGetIParameter( GlgIH ih, char * name )
double GlgIHGetDParameter( GlgIH ih, char * name )
char * GlgIHGetSParameter( GlgIH ih, char * name )
GlgObject GlgIHGetOParameter( GlgIH ih, char * name )
void * GlgIHGetPParameter( GlgIH ih, char * name )
```

A GLG error is generated if a requested named parameter does not exist, or if its value cannot be converted to the requested type.

For string parameters, a returned string pointer is not allocated and should not be modified, as it points to the internal string stored in the data storage.

GlgIHGetOptIParameter
GlgIHGetOptDParameter
GlgIHGetOptSParameter
GlgIHGetOptOParameter
GlgIHGetOptPParameter

Same as the corresponding GetParameter functions, except that the named parameter is optional. If a parameter with the requested name does not exist, a default value provided by the *default_value* argument is returned.

```
GlgLong GlgIHGetOptIParameter( GlgIH ih, char * name,
                               GlgLong default_value )
double GlgIHGetOptDParameter( GlgIH ih, char * name,
                              double default_value )
char * GlgIHGetOptSParameter( GlgIH ih, char * name,
                              char * default_value )
GlgObject GlgIHGetOptOParameter( GlgIH ih, char * name,
                                 GlgObject default_value )
void * GlgIHGetOptPParameter( GlgIH ih, char * name,
                              void * default_value )
```

2.10 Custom Interaction Handlers

The Toolkit provides a number of stock interaction handlers, such as *GlgButton*, *GlgSlider*, etc. Custom interaction handlers may be developed to be used in both the GLG editors as well as application runtime. Custom interaction handlers are supported in all GLG programming environments (**C/C++, C#, Java** and **HTML5/JavaScript**).

Overview

A custom handler implements an application-specific logic for handling user interaction and is associated with a viewport in a GLG drawing the same way as the stock interaction handlers. In GLG editors, a custom handler is attached to a viewport or a light viewport by assigning the handler name as a value of the viewport's *Handler* attribute. To avoid error messages in case the custom handler is implemented only in the runtime application code, the custom handler name should start with the *$* character, for example *"$SampleHandler"*.

A custom interaction handler code implements a single handler entry point function that is invoked with different parameters to handle handler initialization and user interaction. The handler is installed by registering this function as a named handler using the *GlgAddHandler* function, which takes the handler name and its entry point as parameters:

```
GlgAddHandler( "$SampleHandler", SampleHandler );
```

This activates the custom handler in the application at run time. To activate the handler in GLG editors, the handler code can be added to the GLG Editor Custom Options DLL described in the *Custom Editor Options and Dialogs DLL* section on page 374 of the GLG Users Guide and Builder Reference Manual.

Sample source code of a custom interaction handler for all supported programming environments (C/C++, C#, Java and HTML5/JavaScript) is provided in the *src/SampleHandler* directory of the GLG installation.

The C sample code for the sample handler is integrated into the GLG Editor Custom Option DLL to make the sample handler active inside GLG editors.

To test the handler in the Graphics Builder, run the following script:

On LINUX/UNIX:

<glg_dir>/editor_extensions/custom_option_example/run_option_example

On WINDOWS:

<glg_dir>\editor_extensions\custom_option_example\run_option_example.bat

then load the *sample_handler.g* drawing into a Graphics Builder from one of the following locations:

<GLG_DIR>/editor_extensions/custom_option_example/sample_handler.g
<GLG_DIR>/src/SampleHandler/sample_handler.g

Start the prototype mode using the *Run, Start* menu option, then press the *Skip Command* button. Click on the buttons to interact with the handler: it will count the clicks, and will also trace and display mouse coordinates.

List of Functions

The following functions provide support for implementing custom interaction handlers and are provided as a part of the GLG Intermediate API library:

- **GlgAddHandler** registers a custom interaction handler.
- **GlgSetHandlerData** stores data used by the custom handler instance.
- **GlgGetHandlerData** retrieves stored data used by the custom handler instance.
- **GlgCreateMessage** creates a message object used to pass messages to the Input callback.
- **GlgSendMessageToParent** sends a message to the Input callback.
- **GlgHandlerGetNativeEvent** retrieves a native event that triggered handler invocation.

Function Descriptions

To use any of the functions described in this chapter, the application program must include the function definitions from the GLG API header file. Use the following line to include those definitions:

```
#include "GlgApi.h"
```

GlgAddHandler

Registers a custom interaction handler:

```
void GlgAddHandler( char * handler_name, GlgHandler func )
```

Parameters

handler_name

Specifies the handler name. The name may be assigned to the *Handler* attribute of a viewport or a light viewport in the GLG drawing. The name should start with the *$* character to avoid generating error messages in GLG editors if the handler assigned to the viewport is not activated in the editor via the GLG Editor Custom Option DLL described above.

func

Specifies the handler's entry point function. This function implements the handler's functionality and has the following function prototype:

```
void GlgHandler( GlgCallEvent * call_event )
```

The call event structure passed to the handler entry point provides information about the event that triggered handler invocation:

```
typedef struct _GlgCallEvent
{
   GlgCallEventType type;    /* Call event type, see below. */
   GlgObject object;         /* Handler's viewport. */
   GlgObject event_viewport; /* For NATIVE_EVENT, the event's
                                viewport. */
   GlgObject event_vl;       /* For NATIVE_EVENT, the event's light
                                viewport, if any, or the viewport
                                otherwise. */
   GlgObject message;        /* Message object for GLG_MESSAGE_EVENT.
                             */
   GlgLong flag;             /* On GLG_HI_SETUP_EVENT, the flag is
                                set to GlgTrue when the handler is
                                invoked before hierarchy setup. The
                                flag is set to GlgFalse when the
                                handler is invoked after the setup.
                                On GLG_DI_SETUP_EVENT, the flag is
                                set to GlgTrue on the first drawing
                                setup after the hierarchy setup. */
   /* Data for GLG_MESSAGE_ACTION, triggered by GlgSendMessage(). */
   char * message_name;
   GlgLong data1;
   GlgLong data2;
   GlgLong data3;
   GlgLong data4;
   GlgLong rval;      /* Used as the return value from
                         GlgSendMessage(). */
   GlgLong used;      /* The handler must set this flag if it handles
                         the message, otherwise an error message is
                         generated and GlgFalse is returned.*/
} GlgCallEvent;
```

The *type* field of the call event may have the following values contained in the *GlgCallEventType* enum:

GLG_HI_SETUP_EVENT
invoked before and after the hierarchy setup with different settings of the call_event flag.

GLG_DI_SETUP_EVENT
Invoked after the drawing setup triggered by the change of the handler's viewport or an object inside it.

GLG_CLEANUP_EVENT
Invoked on the handler reset.

GLG_MESSAGE_EVENT
Invoked to handle messages sent to an interaction handler by its children.

GLG_CHANGE_EVENT
Triggered by a change of an object inside the handler's viewport.

GLG_NATIVE_EVENT
Triggered by a native event.

GLG_MESSAGE_ACTION
Triggered by the *GlgSendMessage* call.

GlgSetHandlerData

Stores data used by the custom handler instance:

```
void GlgSetHandlerData( GlgObject viewport, void * data )
```

Parameters

viewport

Specifies the handler's viewport the data will be attached to.

data

Specifies data to be stored.

GlgGetHandlerData

Retrieves and returns stored data used by the custom handler instance:

```
void * GlgGetHandlerData( GlgObject viewport )
```

Parameters

viewport

Specifies the handler's viewport.

GlgCreateMessage

Creates a GLG message object used to pass messages to the Input callback.

```
GlgObject GlgCreateMessage( GlgObject viewport )
```

Parameters

viewport

Specifies the handler's viewport.

The message object's *Origin* resource will be set to the viewport's name, and the *Object* resource will be set to the viewport's object ID. The message object is usually stored to be reused for the duration of the handler lifetime and has to be dereferenced when the handler is cleaned up to prevent memory leaks.

GlgSendMessageToParent

Sends a message to the Input callback:

```
void GlgSendMessageToParent( GlgObject viewport, GlgObject message_obj,
                             char * action, char * sub_action )
```

Parameters

viewport

Specifies the handler's viewport.

message_obj

Specifies a GLG message object created using *GlgCreateMessage*.

action

The action to be set as the message's *Action* resource.

subaction

The subaction to be set as the message's *SubAction* resource.

The message is first sent to the handler of the viewport's parent (if any), which can either process and stop message propagation, or send the message to the *Input* callback. For example, when a spinner's *Increase* button is pressed, it sends a message to the spinner's handler, which increments the value and stops message propagation. If a stand-alone *Increase* button is pressed, the message is sent directly to the *Input* callback. This logic allows parent input handlers handle messages from their children's handlers.

GlgHandlerGetNativeEvent

Retrieves a native event that triggered handler invocation:

GTK version on Linux/Unix:

```
GXEvent * GlgHandlerGetNativeEvent( GlgCallEvent * call_event )
```

X11 version on Linux/Unix:

```
XEvent * GlgHandlerGetNativeEvent( GlgCallEvent * call_event )
```

On Windows:

```
MSG * GlgHandlerGetNativeEvent( GlgCallEvent * call_event )
```

Parameters

call_event

A pointer to the *call_event* structure the handler was invoked with.

The function return value is different depending on the platform (Unix/Linux or Windows), and, on Linux, also the version of the GLG library (GTK or legacy X11).

The native event can be used to retrieve the type of the event and the mouse coordinates if the handler needs to handle mouse and keyboard events. The code for handling native events is the same as the code in the Trace callback. The GLG Diagram Demo source code demonstrates handling of mouse events.

2.11 GLG C++ Bindings

C++ bindings allow you to use the GLG Toolkit with C++ applications as a C++ class library. The bindings are implemented in such a way that every GLG object may be used as a C++ object, and other C++ classes may be derived from it as needed. The source code for the C++ bindings is provided, allowing compiling and using the bindings with any C++ compiler. The underlying GLG library does not depend on a C++ compiler used and may be used with any of them.

Both C++ methods and C API functions can be used interchangeably in a C++ program. When C++ *GlgObjectC* objects are passed as parameters to the C API functions that take *GlgObject* parameters, the C++ *GlgObjectC* objects are automatically converted to the C *GlgObject* objects. Vise versa, when C *GlgObject* objects are passed as parameters to the C++ methods that take *GlgObjectC* parameters, they are automatically converted to the *GlgObjectC* object instances.

The C++ bindings are provided for the most commonly used GLG API functions, and the C functions can be used for the rest of the GLG API when needed.

If you want to write a cross-platform C++ program that can be compiled without changes in ether Linux/Unix or Windows environment, you may use *GlgObjectC* class, which uses the GLG Generic API and not depend on the windowing system. The *InitialDraw()* method of the class provides a **platform-independent API for creating a window** with a GLG drawing displayed inside it.

As with the GLG C API, the C++ bindings provide you with several ways of displaying a GLG drawing in a program. You can use native features of the corresponding platform, such as a *GtkmmGlgWidget* or *GlgWrapperC* widget class on Linux/Unix, or an MFC *CWnd* derived *GlgControlC* class on Windows.

The C++ API methods mimic the methods of the GLG C API. This chapter provides only a brief description of each API method; refer to the *Animating a GLG Drawing with Data Using the Standard API* chapter on page 67 for a detailed description of each method's functionality. Refer to the *Handling User Input and Other Events* chapter on page 115 for details of the Input callback. Refer to the *GLG Intermediate and Extended API* chapter on page 135 for a detailed description of the Intermediate and Extended API methods.

Standard, Intermediate and Extended API Macros

The C++ bindings for all GLG APIs (Standard, Intermediate and Extended) are implemented in a single source code file, *GlgClass.cpp*. The *GlgClass.h* include file provides declarations for all classes and methods used by the C++ bindings. To use methods of the Standard, Intermediate or Extended API, the application has to link with the corresponding GLG library.

The following preprocessor macros control the use of the GLG API libraries in the C++ bindings:

GLG_CPP_STANDARD_API

Activates methods of the Standard API and disables Intermediate and Extended API methods.

GLG_CPP_INTERMEDIATE_API

Activates methods of the Standard and Intermediate APIs and disables the Extended API.

GLG_CPP_EXTENDED_API

Activates methods of the Standard, Intermediate and Extended APIs.

To enable a required API, define one of the macros before including the *GlgClass.h* file. Passing a defined symbol to a compiler using the *-D* option is the most convenient way of defining a macro. In the Visual Studio environment, a preprocessor symbol can be defined in the project settings.

Only one of the API macros should be defined. If no API macro is defined, the Extended API will be activated by default, which can result in linking errors if the program is not linked with the Extended API library.

Handling of Constant Strings

Starting with the GLG version 3.3, strings passed as parameters to the GLG C++ methods are considered constant (`const char *`) by default to avoid compilation warnings. This is accomplished by the CONST_CHAR_PTR preprocessor macro which is defined as 1 by default.

The macro can be set to 0 for compatibility with the previous releases, or for performance optimization using writable strings where this option is available. The macro has to be defined before including the *GlgClass.h* file, in the same way as the API macros described above.

In the method prototypes listed below, the optional `const` modifier is not shown for string and string array parameters.

Qt and Gtkmm Integration

Examples of Qt and GTKMM integration are provided in the *integration* directory of the GLG installation. Each example contains the source code and all project files needed to build it.

For GTKMM, two examples are provided in the following subdirectories:

- *gtkmm3_glg_native* uses the native GTK version of the GLG library
- *gtkmm3_glg_x11* uses the legacy X11 version.

The *src* subdirectory of each example contains the source code of the *GtkmmGlgWidget* class.

Using Legacy X11 GlgWrapperC Widget

The GlgWrapperC class provides C++ bindings to the legacy X11 Motif Wrapper widget. These bindings are disabled by default. To enable them, the GLG_ENABLE_X11_WRAPPER preprocessor macro has to be defined before including the *GlgClass.h* file.

C++ API Files and Libraries

The C++ API consists of the following files:

GlgClass.h

Include file, located in the GLG include directory, declares all GLG C++ classes.

GlgClass.cpp

Source file, located in the GLG source directory (named "src"), has to be compiled with the project.

stdafx.h

Include stub file, is needed on Linux/Unix only to make the source code generic. It allows you to get around the special handling that Visual C++ provides for this file in case of precompiled headers.

No additional libraries are required for GLG C++ bindings.

Using GLG Objects

No matter which C++ API you use in your application, platform-specific or generic, you always use the main *GlgObjectC* C++ class for majority off the operations. If the GLG Generic API is used to display the drawing, the drawing is loaded as a *GlgObjectC* class. If a platform-specific widget wrapper is used to integrate the drawing, it provides a public variable or a method to query the drawing contained in the widget in the form of a *GlgObjectC* class.

The *GlgObjectC* class is the central class of the GLG C++ API. It keeps an object ID of the associated GLG object as one of its data members.

The *GlgObjectC* class has several constructors, allowing several ways of creating of the associated GLG object: by loading it from a file, loading from a generated memory image, or by referencing a named resource inside another GLG object.

There is also a default constructor with no arguments, which creates a *GlgObjectC* class with a *NULL* GLG object. The GLG object may be associated with this instance of the class later, by using several available *Load()* methods, which allow loading it from a file, memory image or associating a reference of some named resource of another object. If a *Load()* method is used for a class instance which already has an associated GLG object, it is dissociated from the previous object and associated with the new one.

The *GlgObjectC* class also provides an overloaded assignment operator, which associates the left hand side class instance with a reference to the same GLG object of the right hand class instance. Any previous associations are discarded.

There are type converters to and from *GlgObject* ID, which allows assigning the *GlgObject* ID to a class and using both the *GlgObject* ID and the instance of the *GlgObjectC* class interchangeably.

The *GlgObject* ID is used as a return value of some methods of the class, allowing you to avoid returning temporary instances of the class or class pointers, both of which may lead to memory leaks or dangling pointers. The *GlgObject* return value may then be assigned to an instance of the class as in the following example:

```
GlgObjectC car( "car.g" );
GlgObjectC wheel = car.GetResourceObject( "Wheel0" );
```

In this example, the object from "car.g" drawing file gets loaded and associated with a car class instance. Then the first wheel of the car gets associated with the wheel class instance. The above example may be rewritten using constructors only:

```
GlgObjectC car( "car.g" );
GlgObjectC wheel( car, "Wheel0" );
```

Notice that the wheel class instance is associated with an GLG object which is the part of another object. You cannot add that wheel object to the car object again, because it is already a part of it. You can use the *Copy()* method to create a new copy of a wheel object and associate that new copy with the class instance:

```
wheel.Copy();
wheel.SetResource( "Name", "SpareWheel" ); // Use distinct name
car.Add( wheel );
```

The same thing can also be accomplished by using a copy constructor:

```
{
   GlgObjectC spare_wheel( wheel );
   spare_wheel.SetResource( "Name", "SpareWheel" );
   car.Add( spare_wheel );
}
```

Automatic Referencing and Dereferencing

The *GlgObjectC* class provides automatic referencing and dereferencing of the associated GLG object, freeing the developer from keeping track of temporary objects. For example, the *spare_wheel* object in the above example gets automatically dereferenced in its destructor when it goes out of scope after being added to a car.

Some methods, such as *CreateTagList*, create and return a new GLG object (a list of tags in the case of *CreateTagList*). The return value of these methods has to be explicitly dereferenced using the *Drop* method to avoid memory leaks, as shown in the following example:

```
GlgObjectC tag_list = drawing.CreateTagList( True );
tag_list.Drop();
ProcessTags( tag_list );
```

In the above example, the tag list object created by *CreateTagList* gets extra referenced when it is assigned to the *tag_list* variable. Therefore, it is safe to dereference it by calling the *Drop* method right after the call to *CreateTagList*, even though the tag list is still used in the program. The tag list will be destroyed when the *tag_list* variable goes out of scope (unless the tag list is referenced inside the *ProcessTags* method by assigning it to a global variable to keep the tag list).

In addition to the explicit *Reference()* and *Drop()* methods , the GlgObjectC class also implements overloaded increment (++) and decrement (--) operators providing the same functionality. Note that calling the *Drop()* method explicitly without referencing the object first may cause destruction of the object and a crash due to an invalid GLG object ID.

Comparison Operators

The *IsNull()* method returns *TRUE* if there is no GLG object associated with the class instance (*NULL GlgObjectC* ID). There is also an overloaded logical negation operator which performs the same function: *!object* yields *TRUE* when the there is no associated GLG object.

Notice an interesting usage of double negation: *!!object* returns *TRUE* when there is a GLG object associated with the class instance.

The *Same()* method of the *GlgObjectC* class provides a logical equality operator: it returns *TRUE* if the instances of the class is associated with the same GLG object as the instance of the class passed as a parameter.

Using Input and Selection Callbacks

The input and selection callbacks are virtual methods of all three main GLG API classes: *GlgObjectC*, *GlgWrapperC* and *GlgControlC*. To use these callbacks, derive your class from any of these classes and provide your implementation of *Input()* and *Selection()* callbacks that overrides the ones of the base class. Use the *EnableCallbacks()* method of the base class to enable the callbacks. Refer to the programming examples described below for more details.

Miscellaneous Utility Classes

The *GlgSessionC* class handles initialization of the GLG Toolkit. One instance of this class has to be created to initialize the Toolkit. In some cases the initialization may be skipped, as describer later.

On Linux/Unix, the Toolkit is initialized automatically when a native wrapper widget class is created. On Windows, the toolkit is initialized automatically if the DLL version of the library is used, or when a *Create()* method of the *GlgControlC* class is invoked.

In both cases, the initialization may be skipped only if there are no other GLG calls were issued before invoking the *Create()* method. An explicit initialization is mandatory when the Generic C++ API is used. It is always safe to use explicit initialization in either case, which can also process any command-line options.

The *MainLoop()* method of the *GlgSessionC* class also provides a generic interface to the window system specific event loop.

The *GlgLinkC* class is derived from the *GlgObjectC* class and provides additional ICC functionality for communicating between GLG processes using link objects. The class provides methods for establishing a link to a GLG ICC viewport server. The rest of functionality for setting resources and updating through a link is inherited from it's base *GlgObjectC* class.

Programming Examples

The *examples_c_cpp* directory contains source code examples for both C and C++. On Windows, the *examples_mfc* directory contains MFC examples.

List of Classes and Methods in the GLG C++ Bindings

GlgSessionC

Provides an interface for initializing the GLG Toolkit.

Data Members:

GlgAppContext app_context;

Application context returned by *GlgInit()*.

Methods:

GlgSessionC(void);

Default constructor.

GlgSessionC(GlgBoolean initialized, GlgAppContext application_context,
*int argc = 0, char ** argv = NULL);*

Constructor: calls *GlgInit()*.

virtual ~GlgSessionC(void);

Destructor.

GlgAppContext GetAppContext(void);

Returns the application context.

GlgBoolean MainLoop(void);

Provides a generic interface to the event loop.

void Create(GlgBoolean initialized = False, GlgAppContext application_context = NULL,
*int argc = 0, char ** argv = NULL);*

Calls *GlgInit()*.

GlgObjectC

This is the main class of the GLG C++ bindings. It has the functionality of the regular and Extended API.

Data Members:

GlgObject glg_obj;

An object ID of associated GLG object.

GlgObject suspend_obj;

Suspension information object. Is not *NULL* if the object was suspended for editing.

Methods:

GlgObjectC(void);

Default constructor: Creates a null object.

*GlgObjectC(GlgObjectType type, char * name = NULL,*
GlgAnyType data1 = NULL, GlgAnyType data2 = NULL,
GlgAnyType data3 = NULL, GlgAnyType data4 = NULL);

Constructor: creates a new object of the type using default attributes.

*GlgObjectC(char * filename);*

Constructor: loads an object from a file.

GlgObjectC(long object_data[], long object_data_size);

Constructor: loads an object from a generated memory image.

GlgObjectC(GlgObjectC& object,
GlgCloneType clone_type = GLG_FULL_CLONE);

Copy constructor: creates a copy of the object. The type of copying is defined by a *clone_type* parameter. Default type is *GLG_FULL_CLONE*.

*GlgObjectC(GlgObjectC& object, char * resource_name);*

Constructor: associates with a named resource object inside another and references it.

GlgObjectC(GlgObject object);

Constructor: creates a C++ class from *GlgObject* to allow assigning *GlgObject* to a *GlgObjectC* object class:

```
GlgObjectC object = GetResourceObject( ... );
```

*GlgObjectC(GlgObjectC * object);*

Constructor: creates a C++ class from a GlgObjectC pointer. This constructor is needed by some C++ compilers for proper handling of temporary objects.

virtual ~GlgObjectC(void);

Destructor: Deletes or dereferences the GLG object.

operator GlgObject(void);

Type converter to *GlgObject*.

GlgObjectC& operator= (const GlgObjectC& object);

Assignment operator: associates the GLG object and references it.

GlgObjectC& operator++(void); // prefix ++

GlgObjectC& operator++(int); // postfix ++

GlgObjectC& operator--(void); // prefix --

GlgObjectC& operator--(int); // postfix --

Overloaded ++; --, both infix and postfix. Overloaded to reference/dereference.

void Reference(void);

void Drop(void);

Explicit reference/dereference.

*GlgBoolean Save(char * filename);*

Saves the associated GLG object into a file.

*GlgBoolean Load(char * filename, char * object_name = NULL);*

Explicit *Load()*. Replaces the associated GLG object with the loaded one. Two sets of overloaded load functions: one for loading the whole drawing or its named components, and another for extracting just the "$Widget" viewport.

Loads an object from a file. If *object_name* is not *NULL*, extracts and loads just that named resource of the object.

*GlgBoolean Load(void * object_data, long object_data_size, char * object_name = NULL);*

Loads an object from a generated memory image.

*GlgBoolean Load(GlgObjectC& object, char * object_name = NULL);*

Associates with an object inside another object and references it. If *object_name* is *NULL*, uses the parameter object itself.

*GlgBoolean LoadWidget(char * filename);*

Loads a "$Widget" viewport from a file.

*GlgBoolean LoadWidget(void * object_data, long object_data_size);*

Loads a "$Widget" viewport from a generated memory image.

GlgBoolean LoadWidget(GlgObjectC& object);

Loads a "$Widget" viewport from another object.

void LoadNullObject(void);

Dissociates and sets the GLG object ID to *NULL*.

*void Create(GlgObjectType type, char * name = NULL,*
GlgAnyType data1 = NULL, GlgAnyType data2 = NULL,
GlgAnyType data3 = NULL, GlgAnyType data4 = NULL);

Replaces the current object with a created object.

void Copy(GlgCloneType clone_type = GLG_FULL_CLONE);

Replaces the current object with a copy of it.

*GlgBoolean GetDResource(char * resource_name, double * value);*
*GlgBoolean GetResource(char * resource_name, double * value);*

Get value of a D (double) resource.

*GlgBoolean GetSResource(char * resource_name, char ** s_value);*
*GlgBoolean GetResource(char * resource_name, char ** s_value);*

Get value of an S (string) resource.

*GlgBoolean GetGResource(char * resource_name, double * x_value,*
*double * y_value, double * z_value);*
*GlgBoolean GetResource(char * resource_name, double * x_value,*
*double * y_value, double * z_value);*

Get value of a G (geometrical) resource.

*GlgBoolean GetDTag(char * tag_source, double * value);*
*GlgBoolean GetTag(char * tag_source, double * value);*

Get value of a D (double) tag.

*GlgBoolean GetSTag(char * tag_source, char ** s_value);*
*GlgBoolean GetTag(char * tag_source, char ** s_value);*

Get value of an S (string) tag.

*GlgBoolean GetGTag(char * tag_source,*
*double * x_value, double * y_value, double * z_value);*
*GlgBoolean GetTag(char * tag_source,*
*double * x_value, double * y_value, double * z_value);*

Get value of a G (geometrical) tag.

*GlgBoolean SetDResource(char * resource_name, double value);*
*GlgBoolean SetResource(char * resource_name, double value);*
*GlgBoolean SetDResourceIf(char * resource_name, double value, GlgBoolean if_changed);*
*GlgBoolean SetResource(char * resource_name, double value, GlgBoolean if_changed);*

Set value of a D (double) resource. If the *if_changed* parameter is *true*, the drawing will be updated only if the new resource value is different from the current one.

*GlgBoolean SetSResource(char * resource_name, char * s_value);*
*GlgBoolean SetResource(char * resource_name, char * s_value);*
*GlgBoolean SetSResourceIf(char * resource_name, char * s_value, GlgBoolean if_changed);*
*GlgBoolean SetResource(char * resource_name, char * s_value, GlgBoolean if_changed);*

Set value of an S (string) resource. If the *if_changed* parameter is *true*, the drawing will be updated only if the new resource value is different from the current one.

*GlgBoolean SetSResourceFromD(char * resource_name, char * format, double value);*
*GlgBoolean SetResource(char * resource_name, char * format, double value);*
*GlgBoolean SetSResourceFromDIf(char * resource_name, char * format, double value,*
GlgBoolean if_changed);
*GlgBoolean SetResource(char * resource_name, char * format, double value,*
GlgBoolean if_changed);

Set value of an S (string) resource from a numerical value using the specified C format. If the *if_changed* parameter is *true*, the drawing will be updated only if the new tag value is different from the current one.

*GlgBoolean SetGResource(char * resource_name,*
double x_value, double y_value, double z_value);
*GlgBoolean SetResource(char * resource_name,*
double x_value, double y_value, double z_value);
*GlgBoolean SetGResourceIf(char * resource_name,*
double x_value, double y_value, double z_value,
GlgBoolean if_changed);
*GlgBoolean SetResource(char * resource_name,*
double x_value, double y_value, double z_value,
GlgBoolean if_changed);

Set value of an G (geometrical) resource. If the *if_changed* parameter is *true*, the drawing will be updated only if the new resource value are different from the current one.

*GlgBoolean SetResourceFromObject(char * resource_name, GlgObjectC& object);*
*GlgBoolean SetResource(char * resource_name, GlgObjectC& object);*
*GlgBoolean SetResourceFromObjectIf(char * resource_name, GlgObjectC& object,*
GlgBoolean if_changed);
*GlgBoolean SetResource(char * resource_name, GlgObjectC& object, GlgBoolean if_changed);*

Set resource from object. If the *if_changed* parameter is *true*, the drawing will be updated only if the new tag value is different from the current one

*GlgBoolean SetDTag(char * tag_source, double value, GlgBoolean if_changed);*
*GlgBoolean SetTag(char * tag_source, double value, GlgBoolean if_changed);*

Set value of a D (double) tag. All tags with the specified *tag_source* will be set to the supplied value. If the *if_changed* parameter is *true*, the drawing will be updated only if the new tag value is different from the current one.

*GlgBoolean SetSTag(char * tag_source, char * s_value, GlgBoolean if_changed);*
*GlgBoolean SetTag(char * tag_source, char * s_value, GlgBoolean if_changed);*

Set value of an S (string) tag. All tags with the specified *tag_source* will be set to the supplied value. If the *if_changed* parameter is *true*, the drawing will be updated only if the new tag value is different from the current one.

*GlgBoolean SetSTagFromD(char * tag_source, char * format, double value, GlgBoolean if_changed);*
*GlgBoolean SetTag(char * tag_source, char * format, double value, GlgBoolean if_changed);*

Set value of an S (string) tag from a numerical value using the specified C format. All tags with the specified *tag_source* will be set to the supplied value. If the *if_changed* parameter is *true*, the drawing will be updated only if the new tag value is different from the current one.

*GlgBoolean SetGTag(char * tag_source,*
double x_value, double y_value, double z_value, GlgBoolean if_changed);
*GlgBoolean SetTag(char * tag_source,*
double x_value, double y_value, double z_value, GlgBoolean if_changed);

Set value of a G (geometrical) tag. All tags with the specified *tag_source* will be set to the supplied value. If the *if_changed* parameter is *true*, the drawing will be updated only if the new tag value is different from the current one.

*GlgBoolean HasResourceObject(char * resource_name);*

Checks if a named resource exists.

*GlgBoolean HasTagName(char * tag_name);*

*GlgBoolean HasTagSource(char * tag_source);*

Checks if a tag with the specified tag name or tag source exists.

GlgObject CreateTagList(GlgBoolean unique_tags);

Creates and returns a list of the object's tags. If *unique_tags* is set to *True*, only the first encountered tag of the highest priority will be reported for tags with the same tag source. Input tags are prioritized over output tags, and enabled tags over disabled tags.

Since the method creates a new object, its returned value must be explicitly dereferenced. This can be done by invoking the *Drop* method on the *GlgObjectC* object the return value is assigned to.

GlgObject CreateAlarmList(void);

Creates and returns a list of alarms attached to the object.

Since the method creates a new object, its returned value must be explicitly dereferenced. This can be done by invoking the *Drop* method on the *GlgObjectC* object the return value is assigned to.

GlgObject GetGlgObject(void);

Returns an object ID of the associated GLG object.

void SetGlgObject(GlgObject object);

Low-level object manipulation function: sets the associated GLG object.

GlgBoolean IsNull(void);

GlgBoolean operator!(void);

NULL check to use after Load/GetResourceObject/etc.

GlgBoolean Same(GlgObjectC& object);
GlgBoolean Same(GlgObject object);

Returns *TRUE* if the object refers to the same object ID as the parameter object.

void InitialDraw(void);

void SetupHierarchy(void);

void ResetHierarchy(void);

GlgBoolean Reset(void);

Other viewport-related GLG API functions.

GlgBoolean Sync(void);

Flushes all graphic requests and waits for their completion.

GlgBoolean Update(void);

Updates the viewport's graphic with the new data.

void EnableCallback(GlgCallbackType callback_type, GlgObject callback_viewport = NULL);

Enables selection and input callbacks of the object. If *callback_viewport* is not NULL, adds GLG callbacks to it, otherwise adds it to *this* object (which must be a viewport).

virtual void Input(GlgObjectC& callback_viewport, GlgObjectC& message);

*virtual void Select(GlgObjectC& callback_viewport, char ** name_array);*

*virtual void Trace(GlgObjectC& callback_viewport, GlgTraceCBStruct * trace_info);*

*virtual void Trace2(GlgObjectC& callback_viewport, GlgTraceCBStruct * trace_info);*

*virtual void Hierarchy(GlgObjectC& callback_viewport, GlgHierarchyCBStruct * hierarchy_info);*

Virtual callback functions to be overridden by a derived class.

*GlgBoolean SetZoom(char * resource_name, GlgLong type, double value)*

Performs a zoom or pan operation specified by the *type* parameter. If the *resource_name* parameter is *null*, the viewport itself is zoomed. Otherwise, the zoom operation will be performed on its child viewport specified by the *resource_name* parameter. The *value* parameter defines the zoom factor or pan distance. Refer to the *GlgSetZoom* section on page 109 for a complete list of all zoom and pan types.

*GlgBoolean SetZoomMode(char * resource_name, GlgObjectC * zoom_object,*
*char * zoom_object_name)*

Sets or resets a viewport's zoom mode by supplying a GIS or Chart object to be zoomed. In the Drawing Zoom Mode (default), the drawing displayed in the viewport is zoomed and panned. If the GIS Zoom Mode is set, any zooming and panning operation performed by the *GlgSetZoom* method will zoom and pan the map displayed in the viewport's GIS Object, instead of being applied to the viewport's drawing. In the Chart Zoom Mode, the content of the chart is zoomed and panned. If the *resource_name* parameter is *null*, the zoom mode of the viewport itself will be set. Otherwise, the zoom mode will be set for its child viewport specified by the *resource_name* parameter. If the *zoom_name* parameter is not *null*, it specifies the resource path of a GIS or Chart object relative to the *zoom_object* parameter, or relative to the viewport if the *zoom_object* parameter is *null*. The method may be invoked with *zoom_object=null* and *zoom_object_name=null* to reset the zoom mode to the default Drawing Zoom Mode.

*GlgBoolean ExportPostSrcript(char * filename="out.ps", double x=0., double y=0.,*
double width = 2000., double height = 2000.,
GlgBoolean portrait = True, GlgBoolean stretch = True,
GlgBoolean center = True);

Saves PostScript image of a viewport's drawing in a file.

*GlgBoolean SaveImage(char * resource_name, char * filename, GlgImageFormat format)*

Method for saving an image of the visible area of the drawing's viewport.

*GlgBoolean SaveImageCustom(char * resource_name, char * filename, GlgImageFormat format, GlgLong x, GlgLong y, GlgLong width, GlgLong height, GlgLong gap)*

Method for saving an unclipped image of the whole drawing.

*void MetaDraw(char * filename="out.meta", double x=-1000., double y=-1000., double width = 2000., double height = 2000., GlgBoolean portrait = True, GlgBoolean stretch = True);*

GLG Metafile support.

*GlgBoolean Print(CDC * dc, double x=0., double y=0., double width = 2000., double height = 2000., GlgBoolean portrait = True, GlgBoolean stretch = True, GlgBoolean center = True);*

WINDOWS ONLY: Native windows printing, GLG controlled page layout.

*void OnPrint(CDC * dc);*

WINDOWS ONLY: Native window printing, windows-controlled page layout.

*void OnDrawMetafile(CDC * dc);*

WINDOWS ONLY: Native window printing with clipping disabled.

*GlgAnyType SendMessage(char * resource_name, char * message, GlgAnyType param1, GlgAnyType param2, GlgAnyType param3, GlgAnyType param4)*

Sends a message specified by the *message* parameter. If the *resource_name* parameter is *null*, the message is sent to the object itself. Otherwise, the message is sent to the object's child specified by the *resource_name* parameter. The *param<n>* arguments define additional parameters of the message depending on the message type.

*GlgObject GetNamedPlot(char * resource_name. char * plot_name)*

Finds a chart's plot by name and returns it's object ID. If the *resource_name* parameter is NULL, a named plot of *this* chart is returned. Otherwise, the *resource_name* parameter points to a child chart of *this* object. The method returns NULL if a plot with the specified name has not been found.

*GlgObject AddPlot(char * resource_name, GlgObject& plot)*

Adds a plot to a chart object. If the *resource_name* parameter is NULL, the plot is added to *this* chart. Otherwise, the plot is added to a child chart of *this* object pointed to by the *resource_name* parameter. The *plot* parameter specifies the plot object to add; NULL can be used to create a new plot. The method returns the added plot object on success, or NULL on failure.

*GlgBoolean DeletePlot(char * resource_name, GlgObject& plot)*

Deletes a plot from a chart object. If the *resource_name* parameter is NULL, the plot is deleted from *this* chart. Otherwise, the plot is deleted from a child chart of *this* object pointed to by the *resource_name* parameter. The *plot* parameter specifies the plot object to delete. The method returns *True* if the plot was successfully deleted.

*GlgBoolean GetDataExtent(char * resource_name, GlgMinMax * min_max, GlgBoolean x_extent)*

Queries the minimum and maximum extent of the data accumulated in a chart or in one of its plots. If the *resource_name* parameter is NULL, the data extent of *this* chart or plot object is returned. Otherwise, the *resource_name* parameter points to a child chart or a child plot of *this* object. If *x_extent* is *True*, the X extent of the data is returned in *min_max*, otherwise the method returns the Y extent of the data. *GlgMinMax* is defined in the *GlgApi.h* file:

```
typedef struct _GlgMinMax
{
   double min, max;
} GlgMinMax;
```

For a chart object, the method returns the data extent of all plots in the chart. For a plot, the data extent of that plot is returned. The object hierarchy must be set up for this function to succeed. The method returns *True* on success.

Additional Standard API Methods

The *GetElement* and *GetSize* methods described in the *Intermediate and Extended API Methods* section are also available in the Standard API.

Extended and Intermediate API

There are two flavors of the Extended API: **GLG Intermediate API** and **GLG Extended API**. The Intermediate API provides all Extended API methods except for the methods for creating and deleting objects at run time. The list below describes methods of both APIs, with an availability comment for methods that are available only with the Extended API.

Intermediate and Extended API Methods

*GlgObject GetResourceObject(char * resource_name);*

Explicit *GetResourceObject()*. The return value may be assigned to a *GlgObjectC* class.

*GlgBoolean SetResourceObject(char * resource_name, GlgObjectC& o_value);*

Sets the new value of the object's attribute specified by the *resource_name* (Extended API only). It is used for attaching *Custom Property* objects and *aliases*.

*GlgObject GetTagObject(char * search_string, GlgBoolean by_name, GlgBoolean unique_tags, GlgBoolean single_tag, GlgTagType tag_type_mask)*

Finds a single tag with a given tag name or tag source, or creates a list of tags matching the wildcard. The *tag_type_mask* defines the type of tags to query: data tags or export tags. For export tags, tag type constants may be *OR*ed to query multiple subtypes of export tags.

The *search_string* specifies either the exact tag name, data tag source or export tag category to search for, or a search pattern which may contain the *?* and * wildcards. The *by_name* parameter defines the search mode. If set to *True*, the search is performed by matching the tag names, otherwise the search is done by matching the tag sources for data tags or matching the tag categories for export tags.

In the single tag mode, the first tag of the highest priority matching the search criteria is returned. Input tags are prioritized over output tags, and enabled tags over disabled tags. The *unique_tags* controls if only one data tag of the highest priority is included when several data tags with the same tag name or tag source exist in the drawing. The *unique_tags* flag is ignored in the single tag mode and when searching for export tags.

In the single tag mode, the method returns the first attribute that has a tag matching the search criteria. In the multiple tag mode, a list containing all attributes with matching tags will be returned. The return value may be assigned to a *GlgObjectC* class.

In the multiple tag mode, the method creates a new list object and its returned value must be explicitly dereferenced. This can be done by invoking the *Drop* method on the *GlgObjectC* object the return value is assigned to.

GlgObject QueryTags(GlgTagType tag_type_mask)

Returns a list of all tags of the specified tag type. Since the method creates a new list object, its returned value must be explicitly dereferenced. This can be done by invoking the *Drop* method on the *GlgObjectC* object the return value is assigned to.

*GlgObject GetAlarmObject(char * search_string, GlgBoolean single_alarm)*

Finds a single alarm with a given alarm label, or creates a list of alarms with alarm labels matching the specified wildcard. The *search_string* specifies either the exact alarm label or a search pattern which may contain the *?* and * wildcards.

In the single alarm mode, the method returns the first alarm object with an alarm label matching the search criteria. In the multiple alarm mode, a list containing all alarms with matching alarm labels will be returned.

In the multiple alarm mode, the method creates a new list object and its returned value must be explicitly dereferenced. This can be done by invoking the *Drop* method on the *GlgObjectC* object the return value is assigned to.

GlgBoolean SetXform(GlgObjectC& xform);

Adds a transformation to the object (Extended API only).

*GlgObject GetParent(GlgLong * num_parents);*

Gets a parent object or array of parents if *num_parents* > 1.

*GlgCube * GetBoxPtr(void);*

Returns 3D bounding box of an object.

GlgObject GetDrawingMatrix(void);

Returns a drawing matrix object used to render the object.

void SuspendObject(void);

void ReleaseObject(void);

Suspends or releases the object.

GlgObject CreateResourceList(GlgBoolean list_named, GlgBoolean list_default, GlgBoolean list_aliases);

Returns a list of resources of the object.

Since the method creates a new object, its returned value must be explicitly dereferenced. This can be done by invoking the *Drop* method on the *GlgObjectC* object the return value is assigned to.

GlgObject CreatePointArray(GlgControlPointType type);

Creates and returns an array containing all of an object's control points. The *type* parameter specifies the type of points to be returned.

Since the method creates a new object, its returned value must be explicitly dereferenced. This can be done by invoking the *Drop* method on the *GlgObjectC* object the return value is assigned to.

*GlgBoolean MoveObject(GlgCoordType coord_type, GlgPoint * start_point, GlgPoint * end_point);*

Moves an object by the specified vector.

GlgBoolean MoveObjectBy(GlgCoordType coord_type, double x, double y, double z);

Moves an object by the specified X, Y and Z distances.

*GlgBoolean ScaleObject(GlgCoordType coord_type, GlgPoint * center, double x, double y, double z);*

Scales an object relative to the center by the specified X, Y and Z scale factors. If the *center* parameter is *null*, the center of the object's bounding box is used.

*GlgBoolean RotateObject(GlgCoordType coord_type, GlgPoint * center, double x, double y, double z);*

Rotates an object around the specified center by the specified X, Y and Z angles. If the *center* parameter is *null*, the center of the object's bounding box is used.

GlgBoolean PositionObject(GlgCoordType coord_type, GlgLong anchoring, double x, double y, double z);

Positions an object at the specified location using the specified anchoring.

*GlgBoolean FitObject(GlgCoordType coord_type, GlgCube * box, GlgBoolean keep_ratio);*

Fits the object to the specified area.

GlgBoolean LayoutObjects(GlgObject sel_elem, GlgLayoutType type, double distance, GlgBoolean use_box, GlgBoolean process_subobjects)

Performs a requested layout operation. Refer to the *GLG Intermediate and Extended API* chapter on page 135 for more details.

*GlgBoolean ScreenToWorld(GlgBoolean inside_vp, GlgPoint * in_point, GlgPoint * out_point);*

Converts a point coordinates from the screen to the GLG world coordinate system. The *inside_vp* flag must be set to *GlgTrue* for viewport objects to convert to the world coordinate system inside the viewport. When converting the cursor position to world coordinates, add GLG_COORD_MAPPING_ADJ to the cursor's x and y screen coordinates for precise mapping.

*GlgBoolean WorldToScreen(GlgBoolean inside_vp, GlgPoint * in_point, GlgPoint * out_point);*

Converts a point coordinates from the GLG world to the screen coordinate system. The *inside_vp* flag must be set to *GlgTrue* for viewport objects to convert from the world coordinate system inside the viewport.

*GlgBoolean PositionToValue(char * resource_name, double x, double y,*
*GlgBoolean outside_x, GlgBoolean outside_y, double * value)*

Returns a value corresponding to the specified *x, y* position on top of a chart or an axis object. If the *resource_name* parameter is *null*, the value corresponding to *this* object is returned. Otherwise, the *resource_name* parameter points to a child chart, a child plot or a child axis of *this* object.

If *outside_x* or *outside_y* are set to True, the method returns NULL if the specified position is outside of the chart or the axis in the corresponding direction.

For a plot, the method converts the specified position to a Y value in the Low/High range of the plot and returns it via the *value* pointer.

For an axis, the function converts the specified position to the axis value and returns the value.

For a chart, the function converts the specified position to the X or time value of the chart's X axis and returns the value.

When using the cursor position, add GLG_COORD_MAPPING_ADJ to the cursor's x and y coordinates for precise mapping.

GlgObject CreateChartSelection(GlgObjectC plot, double x, double y, GlgBoolean screen_coord,
GlgBoolean include_invalid, GlgBoolean x_priority)

Selects a chart's data sample closest to the specified *x, y* position and returns a message object containing the selected sample's information. If *plot* contains a null GLG object, the method queries samples in all plots of the chart and selects the sample closest to the specified position. If *plot* contains a non-null GLG object, the method queries only the samples in that plot.

If the *screen_coord* parameter is set to *True*, the position is defined in the screen coordinates of the chart's parent viewport. If *screen_coord* is set to *False*, the *x* position is defined as an X or time value, and the *y* position is defined in relative coordinates in the range [0; 1] to allow the chart to process selection for plots with different ranges (0 corresponds to the *Low* range of each plot, and 1 to the *High* range). When the mouse position is used, add GLG_COORD_MAPPING_ADJ to the x and y screen coordinates of the mouse for precise mapping.

The *dx* and *dy* parameters specify an extent around the point defined by the *x* and *y* parameters. The extent is defined in the same coordinates as *x* and *y*, depending on the value of the *screen_coord* parameter. A data sample in a chart is selected only if its *x* and *y* coordinates differ from the specified position by amounts less or equal to the values of the corresponding *dx* or *dy* parameters.

If *include_invalid* if set to *True*, invalid data samples are considered for selection, otherwise they are never included. If *x_priority* is set to *True*, the method selects the data sample closest to the specified position in the horizontal direction, with no regards to its distance from the specified position in the vertical direction. If *x_priority* is set to *False*, a data sample with the closest distance from the specified position is selected.

The information about the selected data sample is returned in the form of a message object described in the *Chart Selection Message Object* section on page 396. The method returns NULL if no data sample matching the selection criteria was found.

Since the method creates a new object, its returned value must be explicitly dereferenced. This can be done by invoking the *Drop* method on the *GlgObjectC* object the return value is assigned to.

*char * CreateTooltipString(double x, double y, double dx, double dy, char * format)*

Creates and returns a chart or axis tooltip string corresponding to the specified *x, y* position in screen coordinates. The string can be used to provide cursor feedback and display information about the data sample under the current mouse position, as shown in the Real-Time Strip-Chart demo.

The *format* parameter provides a custom format for creating a tooltip string and uses the same syntax as the *TooltipFormat* attributes of the chart and axis objects. If it is set to NULL or an empty string, the format provided by the *TooltipFormat* attribute of *this* chart or axis object will be used. A special "<*single_line*>" tag may be prepended to the format string to force the method to remove all line breaks from the returned tooltip string.

If *this* object is a chart and the selected position is within the chart's data area, the method selects a data sample closest to the specified position and returns a tooltip string containing information about the sample. The *dx* and *dy* parameters define the maximum extent in pixels around the specified position for selecting a data sample, and the *TooltipMode* attribute of the chart defines the data sample selection mode: X or XY. The method returns NULL if no samples matching the selected criteria were found.

If *this* object is an axis, or if *this* object is a chart and the selected position is on top of one of the chart's axes, the function returns a tooltip string that displays the axis value corresponding to the specified position.

When using the cursor position, add GLG_COORD_MAPPING_ADJ to the cursor's x and y screen coordinates for precise mapping.

The returned string must be freed using the *GlgFree* function.

Attribute Objects

GlgBoolean ConstrainObject(GlgObjectC& to_attribute);

GlgBoolean UnconstrainObject(void);

Constrains or unconstrains the attribute object.

Matrix Objects

GlgObject InverseMatrix(void);

Creates and returns a new matrix. Since the method creates a new matrix object, its returned value must be explicitly dereferenced. This can be done by invoking the *Drop* method on the *GlgObjectC* object the return value is assigned to.

*void TransformPoint(GlgPoint * in_point, GlgPoint * out_point);*

Transforms point using the GLG matrix object associated with this class instance.

*void GetMatrixData(GlgMatrixData * matrix_data);*

Queries matrix coefficients.

*void SetMatrixData(GlgMatrixData * matrix_data);*

Sets matrix coefficients.

Container Functionality

GlgBoolean IsContainer(void);

Returns *TRUE* if the associated GLG object is a GLG container (viewport,

void SetStart(void);

Initializes container for traversing.

GlgObject Iterate(void);

Returns the next subobject of the associated container object.

GlgLong GetSize(void);

Returns the size of the associated container object.

GlgBoolean AddToTop(GlgObjectC& object);

GlgBoolean AddToBottom(GlgObjectC& object);

GlgBoolean AddAt(GlgObjectC& object, GlgLong index);

Add object to container (Extended API only).

GlgBoolean DeleteTopObject(void);

GlgBoolean DeleteBottomObject(void);

GlgBoolean DeleteObjectAt(GlgLong index);

GlgBoolean DeleteObject(GlgObjectC& object);

Delete this object from container (Extended API only).

GlgObject ReorderElement(GlgLong old_index, GlgLong new_index);

Reorder elements of the container.

GlgBoolean ContainsObject(GlgObjectC& object);

GlgObject GetElement(GlgLong index);

*GlgObject GetNamedObject(char * name);*

GlgLong GetIndex(GlgObjectC& object);

*GlgLong GetStringIndex(char * string);*

Search functions for finding elements of a container.

void Inverse();

Inverses the order of container's elements.

GlgControlC (Windows Only)

MFC *CWnd* derived class providing a window for displaying a GLG drawing.

Data Members:

GlgObjectC viewport;

An object ID of the associated GLG viewport object with the drawing.

Methods:

GlgControlC(void);

Default constructor: creates an empty control.

virtual ~GlgControlC(void);

Destructor. Destroys the window and dereferences the viewport object.

*void Create(char * drawing_file, CWnd * parent,*
*char * control_name = "GlgControl", CRect * in_rect = NULL, int IDC_id = 0);*

Creates a control's window and loads drawing from a file into it.

*void Create(void * drawing_image, long image_size, CWnd * parent,*
*char * control_name = "GlgControl", CRect * in_rect = NULL, int IDC_id = 0);*

Creates a control's window and loads drawing from a generated memory image.

*void Create(GlgObjectC& object, CWnd * parent,*
*char * control_name = "GlgControl", CRect * in_rect = NULL, int IDC_id = 0);*

Creates a control's window and uses the "$Widget" viewport of another GLG object as a drawing.

void InitialDraw();

Draws the graphics the first time.

GlgBoolean Update(void);

Updates the displayed graphics.

*void SetDrawing(char * filename);*

Changes the drawing, load a new drawing from a file.

*void SetDrawing(void * drawing_image, long drawing_image_size);*

Changes the drawing, load a new drawing from a generated memory image.

void SetDrawing(GlgObjectC& object);

Changes the drawing, use the "$Widget" viewport of another object as a new drawing.

virtual void EnableCallback(GlgCallbackType callback_type,
GlgObject callback_viewport = NULL);

Enables selection and input callbacks of the control. If *callback_viewport* is not *NULL*, adds GLG callbacks to it, otherwise adds it to the control's viewport.

virtual void Input(GlgObjectC& callback_viewport, GlgObjectC& message);

*virtual void Select(GlgObjectC& callback_viewport, char ** name_array);*

*virtual void Trace(GlgObjectC& callback_viewport, GlgTraceCBStruct * trace_info);*

*virtual void Trace2(GlgObjectC& callback_viewport, GlgTraceCBStruct * trace_info);*

*virtual void Hierarchy(GlgObjectC& callback_viewport, GlgHierarchyCBStruct * hierarchy_info);*

Virtual callback functions to be overridden by a derived class.

*void Print(CDC * pDC, DWORD dwFlags);*

Native Windows print method.

GlgObject GetViewport(void);

Returns an object ID of the control's viewport.

GlgWrapperC (Legacy X11 Version of the Toolkit on Linux/Unix Only)

This section describes a Motif-based wrapper widget for the **legacy X11 version** of the GLG library on Linux/Unix (*libglg.a*).

For the C++ GTKMM applications, a GTKMM-based *GtkmmGlgWidget* is also provided. Refer to the *integration/gtkmm3_glg_native* directory of the GLG installation for an example of the GTKMM-based *GtkmmGlgWidget* wrapper widget for Linux.

The *GlgWrapperC* class is a C++ wrapper around the *GlgWrapper* widget. It is a subclass of a *GlgObjectC* and inherits GLG API and GLG Extended API from it. Note that the *GetViewport()* method has to be called explicitly after the widget has been realized and before any API functions are used. The *GlgGetViewport()* method associates a viewport object with the instance of a class. While the widget is being realized, the widget creates a viewport. The viewport is set to *NULL* initially and its object ID has to be obtained from the widget after the widget has been realized. Since the widget may be realized by its parent, it is difficult to encapsulate *GetViewport*() functionality into the *GlgWrapperC* class so that it is transparent to the user. Trying to do so causes a lot of code duplication and will lead to problems in the future, so invoking this call is left as the responsibility of the user.

To activate C++ bindings for the legacy X11 version of the wrapper widget, define GLG_ENABLE_X11_WRAPPER macro when compiling the code.

Data Members:

Widget widget;

Methods:

GlgWrapperC(void);

Default constructor: creates an empty wrapper widget with no drawing.

virtual ~GlgWrapperC(void);

Destructor. Destroys the wrapper widget. Be careful to set the widget to *NULL* if it is destroyed by destroying its parent to avoid *XtDestroyWidget()* being called with an invalid widget ID.

*void Create(char * filename, Widget parent, char * widget_name = "GlgWrapper",*
long width = 400, long height = 300);

Creates the widget and loads drawing from a file into it.

*void Create(void * image_data, long image_data_size, Widget parent,*
*char * widget_name = "GlgWrapper", long width = 400, long height = 300);*

Creates a widget and loads drawing from a generated memory image.

*void Create(GlgObjectC& object, Widget parent, char * widget_name = "GlgWrapper",*
long width = 400, long height = 300);

Creates a widget and uses the "$Widget" viewport of another GLG object as a drawing.

The following collection of *SetDrawing()* methods change the drawing using a particular wrapper widget resource. The actual drawing displayed is still subject to the priority of a particular wrapper widget drawing resource. Setting the drawing by using one of the *SetDrawing()* functions may interfere with others.

*void SetDrawing(char * filename);*

Loads a new drawing from a file (uses *XtNglgDrawingFile* resource).

*void SetDrawing(void * image_data, long image_data_size);*

Loads a new drawing from a generated memory image (uses *XtNglgDrawingImage* resource).

void SetDrawing(GlgObjectC& object);

Uses the "$Widget" viewport of another object as a new drawing (uses *XtNglgDrawingObject* resource).

Widget GetWidget(void);

Returns a widget id.

GlgObject GetViewport();

Associates a viewport object of the wrapper widget with the instance of the class after the widget has been realized.

void EnableCallback(GlgCallbackType callback_type, GlgObject callback_viewport = NULL);

Enables selection and input callbacks of the control. If *callback_viewport* is not NULL, adds GLG callbacks to it, otherwise adds it to the widget's viewport.

virtual void Input(GlgObjectC& callback_viewport, GlgObjectC& message);

*virtual void Select(GlgObjectC& callback_viewport, char ** name_array);*

*virtual void Trace(GlgObjectC& callback_viewport, GlgTraceCBStruct * trace_info);*

*virtual void Trace2(GlgObjectC& callback_viewport, GlgTraceCBStruct * trace_info);*

*virtual void Hierarchy(GlgObjectC& callback_viewport, GlgHierarchyCBStruct * hierarchy_info);*

Virtual callback functions to be overridden by a derived class.

*void SetXtResource(String xt_ resource_name, char * res_value);*

*void GetXtResource(String xt_ resource_name, char ** res_value_ptr);*

Sets and gets Xt resources of string type (is primarily used with *XtNglgHResource** and *XtNglgVResource** resources).

Chapter 3

Using the ActiveX Control

3

Overview

The GLG ActiveX Control provides GLG Toolkit functionality and a complete GLG Programming API in the form of the Windows' ActiveX control. The ActiveX may be used in Visual Studio.NET with C++, C# and Visual Basic. It may also be used with other products that support embedded ActiveX controls.

The GLG ActiveX Control is implemented as a dynamic link library with the *.OCX* extension. When registered and loaded, this single *.ocx* file may be used to display an extensive set of GLG widgets and custom drawings that can be accessed by users and applications. There are several versions of the GLG ActiveX control in the commercial version of the Toolkit:

glg_std.ocx - includes GLG Standard API
glg_int.ocx - includes GLG Intermediate API
glg_ext.ocx - includes GLG Extended API

The demo and Community Edition of the GLG Toolkit contains a single *glg.ocx* that combines all three GLG APIs.

The flexibility of the GLG drawing architecture translates into flexibility for the GLG ActiveX Control. Any GLG drawing may be used as a custom control by loading it into the ActiveX control. For example, loading a custom process control drawing into the GLG ActiveX control makes it a custom control with the functionality of the loaded drawing. The GLG ActiveX Control includes properties and methods to dynamically change the resources of a GLG drawing as your application's data change, allowing you to bind your real-time data to the attributes of the drawing displayed by the control.

For example, in a business application you may bind real-time data like a current stock price, as well as the high and low values, to a graph to display a history of changes. In a chemical process monitoring application, you may bind the temperature and level of fluid in a tank to the color and height of the tank's filling in the drawing, to display these parameters graphically.

The GLG ActiveX Control also generates custom *Input*, *Select*, *Alarm* and other events. These events provide user feedback, allowing an application to react to user input or determine which graphical objects were selected with the mouse.

The ActiveX Control's API methods mimic the methods of the GLG API. This chapter describes specific details of using the GLG ActiveX Control and provides only a brief description of each API method; refer to the *Animating a GLG Drawing with Data Using the Standard API* chapter on page 67 for a detailed description of each method's functionality. Refer to the *Handling User Input and Other Events* chapter on page 115 for details of the Input callback. Refer to the *GLG Intermediate and Extended API* chapter on page 135 for details of the Extended API methods.

The *examples_csharp* and *examples_vbnet* directories contain source code examples of using the GLG ActiveX Control in the respective programming environments.

GLG ActiveX Control Properties

GLG ActiveX Control has the following properties:

General Page

DrawingFile

Specifies a GLG drawing to display in a control. The drawing file must exist at the time the control is created. This property has priority over the *DrawingImageFile* property described below.The drawing must contain a viewport object named *$Widget* with *HasResources* flag set to YES.

DrawingImageFile

Specifies a drawing to be used and stored in a control. The drawing file must exist at the time this property is set. and a copy of it is stored in the control. Saving the control with this property set saves a copy of the drawing with the control, allowing loading and using the control later with no separate drawing file. The *DrawingFile* property has higher priority: if both *DrawingFile* and *DrawingImageFile* properties are set, the control will use the drawing defined by the *DrawingFile* property, and only if loading that drawing fails, *DrawingImageFile* property is used. To discard the stored drawing, set this property to an empty string.

NOTE: the *DrawingImageFile* property doesn't support compressed drawings. Save drawings with the drawing compression option disabled for use them with this property.

DrawingURL

Specifies a URL address to load a drawing from a Web. The drawing is cached. If other drawing properties are used to define the drawing, they have higher priority. *DrawingURL* property has the lowest priority and is used when other drawing properties are not specified or can't be used. Local drawings may be loaded using this property by specifying "*file://*" at the beginning of the complete path name.

SetupDataURL

Specifies the URL used for the setup data script. The script has the standard *GLG script format*, described in the *The Data Generation Utility* section of the *GLG Programming Tools and Utilities* chapter. The script is invoked after the hierarchy has been setup and may access any resources of the drawing. The *H* properties of the control may be used to change resources before the hierarchy has been setup. The script is invoked only once when the drawing is loaded and initializes it with data. The script is not cached and is loaded asynchronously.

DataURL

Specifies the URL of the script to supply data. The script has the standard *GLG script format*, described in the *The Data Generation Utility* section of the *GLG Programming Tools and Utilities* chapter. The script is first invoked after the *SetupData* script and fills the control with data. If the *UpdatePeriod* property is set, the script at the *DataURL* location is invoked repeatedly with the interval defined by the *UpdatePeriod* property. This enables the ActiveX control to periodically reload data to update display with the latest data.

UpdatePeriod

Defines a periodic interval in seconds for running the *DataURL* script.

PrintFile

Specifies a filename for printing the current content of a drawing in the encapsulated PostScript format, which may later imported into most word processors. Setting this property saves a PostScript output into the specified file using a default page layout. The control's *Print* method described later may be used to control page layout.

InputEnabled

Controls whether input events are sent (default: TRUE). Input events are generated when a user activates sliders, dials, buttons or other input controls inside the GLG ActiveX Control. Generating input events is disabled if this property is set to FALSE.

SelectEnabled

Controls whether selection events are sent (default: TRUE). Selection events are generated when a user selects a graphical object inside the GLG ActiveX Control with the mouse. Generating selection events is disabled if this property is set to FALSE. Set this property to FALSE to increase performance for GLG ActiveX Controls which do not need selection feedback.

TraceEnabled

Controls whether trace events are sent (default: FALSE). Trace events are low level Windows events which may be used to trace the mouse position inside the ActiveX control, handle mouse and keyboard events, etc. Set this property to FALSE to increase performance for GLG ActiveX Controls which do not need trace events.

AlarmsEnabled

Controls whether alarm events are generated (default: FALSE). Alarm events are generated by alarm objects attached to dynamic attributes to monitor their values when the values go out of range.

SupressErrors

When set to TRUE (default value), suppresses the error message dialogs and recording errors in a "*glg_err.log*" log file. Set the property to FALSE to enable debugging GLG ActiveX Control problems.

UseOpenGL

Controls the use of the OpenGL driver (default: FALSE). When set to TRUE, the OpenGL driver is enabled for the viewports that have the OpenGLHint=ON. When set to FALSE, the OpenGL driver is disabled and the GDI driver is used for all viewports.

UseMapURL

Controls the usage mode of the map server (default: FALSE). When set to FALSE, the maps are rendered using local datasets defined by the *GISDataFile* property of the GIS Object. When set to TRUE, a web-based map server defined by the *GISMapServerURL* property of the GIS Object is used to render maps from a centralized web server, without installing map datasets on each client computer.

HProperties Page

HProperty0 - HProperty4

These are character strings that specify five pre-creation properties. These properties serve as entry points to the GLG drawing resource hierarchy and may be used for setting resources before the drawing's object hierarchy is created. They should be used for setting a drawing's H resources, which control the structure of a GLG drawing's hierarchy. For information about H resources, refer to the *H and V Resources* section in the *Integrating GLG Drawings into a Program* chapter. The syntax for the character strings is described in the *Dynamic Resource String Syntax* section of this chapter. These properties are persistent and may be used for customizing a drawing's appearance before saving the control.

The GLG ActiveX Control does not need five different dynamic properties of this type. In one sense, it would be adequate to have just one, and use it repeatedly. However, it is often useful to be able to set the initial values of several drawing resources, so multiple dynamic properties like these are provided. The same is true of the *V* properties described in the following section.

In the event that there are not enough GLG ActiveX Control dynamic properties for your purpose, you can use the GLG ActiveX Control's *Ready event* and call the GLG ActiveX Control's methods to set the initial resources. The GLG ActiveX Control's custom events and methods are described later in this chapter.

VProperties Page

VProperty0 - VProperty4

These are five character strings that specify five post-creation properties. These properties serve as entry points to the GLG drawing resource hierarchy and may be used for setting resources after the drawing's object hierarchy is created. They should be used for setting a drawing's V resources, which only control the values of attributes within the structure of a GLG drawing's hierarchy. For information about V resources, refer to the *H and V Resources* section in the *Integrating GLG Drawings into a Program* chapter. The syntax for the character strings is described in the *Dynamic Resource String Syntax* section of this chapter. These properties are persistent and may be used for customizing a drawing's appearance before saving the control.

DLinks Page

DLinkName0 - DLinkName4

Specify resource names to be used for accessing scalar (D) resources of a drawing.

DLinkValue0 - DLinkValue4

Specify double-precision values to be set for scalar resources whose names are defined by the corresponding *DLinkName* properties. Use these properties to access frequently used double resources, an entry point of a graph's data samples, for example. These properties are bindable and not persistent.

SLinks Page

SLinkName0 - SLinkName4

Specify resource names to be used for accessing string (S) resources of a drawing.

SLinkValue0 - SLinkValue4

Specify values to be set for string resources whose names are defined by the corresponding *SLinkName* properties. Use these properties to access frequently used string resources, an entry point of a graph's labels, for example. These properties are bindable and not persistent.

GLinks Page

GLinkName

Specify resource names to be used for accessing geometric (G) resources of a drawing (colors and control points defined by triplets of double values).

GLinkValueX, GLinkValueY, GLinkValueZ

Specify values to be set for a geometric resource whose name is defined by *GLinkName* property. These properties are bindable and not persistent.

Dynamic Resource String Syntax

`The H and V Properties are dynamic properties of the GLG ActiveX Control.` These properties contain character strings which in turn contain the names and values of GLG drawing resources. The strings have the following syntax:

```
<resource_name>  <resource_type>  <values>
```

<resource_name>

The name of the GLG drawing resource, including the complete path (omitting the *"$Widget"*). For example, *"XLabelGroup/XLabel4/String"* accesses the text string of the fifth label on the X axis in a graph widget.

<resource_type>

The type of the resource and can be one of the following:

- **d** Indicates the resource value is represented by a single floating point number (a scalar), like a line width, a font number or a value of a data sample
- **s** The resource value is a string, like a text string of a label text object
- **g** The resource value is a set of three floating point numbers, like a geometrical point with X, Y and Z coordinates or a color, in which case the three components represent the color's Red, Green, and Blue values.

<values>

A value or a set of values for the resource. The values given depend on the resource type.

The following strings are examples of specifying resources:

```
"DataGroup/DataSample3/Value d 2.5"
"DataGroup/DataSample5/Value d 4"
"XLabelGroup/XLabel4/String s April"
"DataGroup/DataSample6/FillColor g 0.5 0 0.9"
```

Persistency Support

The following properties are persistent and keep their values when the control is saved and loaded:

- *DrawingFile* (saves the file name only)
- *DrawingImageFile* (saves a copy of the drawing)
- *PrintFile* (saves the file name only)
- *InputEnabled*
- *SelectEnabled*
- *HProperty0-HProperty4* (the last property resource setting is stored)
- *VProperty0-VProperty4* (the last property resource setting is stored)

The remaining properties are not persistent and are initialized to their default values when loaded.

GLG ActiveX Control Events

GLG ActiveX Control has the following custom events:

Input(String Format, String Origin, String FullOrigin, String Action, String SubAction)

Input event is fired on user input, for example when a slider in the ActiveX control drawing is moved. It's parameters are identical to the resources of message object in the GLG API's *Input callback*. The last input message objects is stored by the ActiveX control, and its resources may also be accessed as resources by prepending "*$message/*" to the resource name. For example, "*$message/ValueX*" may be used to query the value of a slider using methods of the Standard API.

Input2(long message_obj)

Input2 is an alternative form of the *Input* event which provides a message object parameter containing all resources of the input event. The message object may be passed to the Extended API methods to query its resources. The *Input* event may be used with the Standard API methods. Refer to the *Appendix B: Message Object Resources* chapter on page 377 for more information.

Select(String selected_name, short button)

Selection event is fired when objects in the ActiveX control drawing are selected with a mouse click (button press). The parameters define the complete resource name of the selected object and the mouse button which was used for selection. If several closely positioned objects are selected with a mouse click, the callback is called repeatedly with a name of each selected object. The first and last call have "@@*SelectionStart*@@" and "@@*SelectionEnd*@@" as values of the selected_name parameter, allowing to detect the start and the end of a single mouse selection sequence of events. The name of the selected object and the mouse button used to select it may also be queried using the *GetSelectedName* and *GetSelectionButton* methods of the ActiveX control.

The Input2 event provides a more elaborate alternative for handling object selection using either object IDs or custom events and command actions attached to objects in the Graphics Builder.

Select2(long selection_array, short button)

Same as the *Select* event but invoked just once with an array of resource names of all selected objects. The *GetElementExt* method of the Extended API may be used to retrieve resource names of selected objects.

Trace(long top_viewport, long viewport, long message, long wParam, long lParam)

Trace2(long top_viewport, long viewport, long message, long wParam, long lParam)

Generated for all low-level windowing events such as mouse move, mouse click, button press, etc. May be used to trace the position of the mouse inside the drawing or handle native low-level events. The *top_viewport* parameter is the top-level viewport of the drawing. The *viewport* parameter is the viewport where the event occurred. The *message* parameter is the Windows' message number. The *wParam* and *lParam* are message-dependent data parameters. By default, trace events are activated for the top level viewport of the ActiveX control. The *SetTraceViewport* method may be used to activate trace events only for a child viewport of the control.

The Trace event is generated before the event has been dispatched for processing. The Trace2 event is generated after the event has been processed.

Alarm(long object, long alarm, String alarm_label, String action, String subaction, long reserved)

Generated by alarm objects based on the specified alarm condition. The *object* parameter is the resource whose value has changed. The *alarm* parameter is the alarm attached to the *data* object to monitor its value; it is the alarm that generated the alarm message. The *alarm_label* parameter supplies the *AlarmLabel* used to identify the *alarm object.* The *action* and *subaction* parameters provide details about the condition that caused the alarm; refer to the *Alarm Messages* section on page 218 of the *GLG User's Guide and Builder Reference Manual* for more information.

HCallback()

Generated after loading the control's drawing but *before* its hierarchy was set up. It may be used to initialize values of **H** resources, such as a number of instances in a series object or a number of datasamples in a graph.

VCallback()

Generated *after* the hierarchy setup but before the drawing is displayed for the first time. It may be used for initializing **V** resources, such as filling the graph with data for the initial appearance.

Ready()

Generated when the control finished loading the drawing and was painted for the first time.

ActiveX Control Methods

The methods of the ActiveX control's API allow you to animate control's drawing with application data from a VC++ or VB application.

The GLG ActiveX Control provides the following methods:

double GetDResource(String resource_name)
double GetDTag(String tag_source)

Return the value of the drawing's scalar resource or tag. Returns zero if the resource or tag was not found.

BOOL SetDResource(String resource_name, double dvalue)
BOOL SetDTag(String tag_source, double dvalue, BOOL if_changed)

Set the value of the drawing's scalar resource or tag. Returns TRUE if the resource or tag exists and the setting was successful, FALSE otherwise.

NOTE: Use *SetDResourceIfExt* with null object parameter for SetDResourceIf functionality.

String GetSResource(String resource_name)
String GetSTag(String tag_source)

Return the value of the drawing's string resource or tag. Returns the empty string ("") if the resource or tag was not found.

BOOL SetSResource(String resource_name, String svalue)
BOOL SetSTag(String resource_name, String svalue, BOOL if_changed)

Set the value of the drawings's string resource or tag. Returns TRUE if the resource or tag exists and the setting was successful; otherwise returns FALSE.

NOTE: Use *SetSResourceIfExt* with null object parameter for SetSResourceIf functionality.

BOOL SetSResourceFromD(String resource_name, String format, double dvalue)

Set the value of the drawing's string resource to a string obtained by converting the supplied double value according to the *format*. The *format* parameter is a C format string. Returns TRUE if the resource exists and the setting was successful; otherwise returns FALSE.

NOTE: Use *SetSResourceFromDIfExt* with null object parameter for SetSResourceFromDIf functionality.

*void GetGResource(String resource_name, double * x, double * y, double * z)*
*void GetGTag(String tag_source, double * x, double * y, double * z)*

Query the value of the drawing's geometrical resource or tag and return it via the *x*, *y* and *z* parameters.

double GetXResource(String resource_name)
double GetXTag(String tag_source)

double GetYResource(String resource_name)
double GetYTag(String tag_source)

double GetZResource(String resource_name)
double GetZTag(String tag_source)

Return the *X*, *Y*, or *Z* value of the drawing's geometric resource or tag. Returns zero if the resource was not found.

BOOL SetGResource(String resource_name, double dvalue1, double dvalue2, double dvalue3)
BOOL SetGTag(String tag_source, double dvalue1, double dvalue2, double dvalue3,
BOOL if_changed)

Set the value of the drawing's geometric resource or tag. For tags, all tags with the specified *tag_source* will be set to the supplied values. Returns TRUE if the resource or tag exists and the setting was successful.

NOTE: Use *SetGResourceIfExt* with null object parameter for SetGResourceIf functionality.

BOOL HasResourceObject(long object, String resource_name)

Returns TRUE if a named resource object exists, returns FALSE otherwise.

BOOL HasTagName(long object, String tag_name)
BOOL HasTagSource(long object, String tag_source)

Return TRUE if a tag with a specified tag name or tag source exists, returns FALSE otherwise.

long CreateTagList(long object, BOOL unique_tags)

Creates and returns a list of object's tags, or *null* if no tags were found. If *unique_tags* is set to TRUE, only the first encountered tag of the highest priority will be added to the list for tags with the same tag source. Input tags are prioritized over output tags, and enabled tags over disabled tags. If the parameter value is set to FALSE, all instances with the same tag source will be reported. The list must be destroyed using *DropObject* when it is no longer needed.

void UpdateGlg()

Forces GLG ActiveX Control to update the displayed drawing to reflect the new data. Several resource changes may be effected at once by invoking several resource setting methods and then invoking the update method once to display all changes at once. This also increases performance. Note, however, that setting some resources may interfere with setting other resources, and that it may be necessary to invoke either the *UpdateGlg* or *SetupHierarchy* methods in between setting these two kinds of resources. For example, before you know that a series has eight members, it makes no sense to query the eighth member. You must set the series *Factor* resource, execute the *UpdateGlg* method, and then query the series members.

void Update() (deprecated)

Same as the *UpdateGlg* method, deprecated due to a conflict with the .NET environment, where the *Update* method of the GLG ActiveX Control is shadowed by the object's *Update* method of the .NET framework. Use the *UpdateGlg* method instead.

BOOL SetAutoUpdateOnInput(BOOL update)

Enables or disables automatic updates on input events. Returns the previous setting of the parameter.

Automatic updates on input events are enabled by default but may be disabled to prevent slowing down scrolling and other operations of the real-time charts.

If automatic updates are turned off, input objects redraw themselves on user input event. However, an explicit *Update()* call has to be used to redraw any objects constrained to input objects and located outside of them.

When automatic updates are turned off, the application code also has to explicitly invoke *UpdateGlg()* for timer events (*Format*="Timer") to update objects in the drawing with the timer transformation attached.

BOOL SetZoom(long viewport, String resource_name, short type, double value)

Performs a zoom or pan operation specified by the *type* parameter. If the *resource_name* parameter is *null*, the viewport or the light viewport specified by the *viewport* parameter will be zoomed. Otherwise, the zoom operation will be performed on its child viewport specified by the *resource_name* parameter. If the *viewport* parameter is *null*, the viewport of the ActiveX control's drawing is used. The *value* parameter defines the zoom factor or pan distance. Refer to the *GlgSetZoom* section on page 109 for a complete list of all zoom and pan types.

In the Drawing Zoom Mode, if the method attempts to set a very large zoom factor which would result in the overflow of the integer coordinate values used by the native windowing system, the zoom operation is not performed and the method returns FALSE.

BOOL SetZoomMode(long viewport, String resource_name, long zoom_object, String zoom_object_name)

Sets or resets a viewport's zoom mode by supplying a GIS or Chart object to be zoomed. In the Drawing Zoom Mode (default), the drawing displayed in the viewport is zoomed and panned. If the GIS Zoom Mode is set, any zooming and panning operation performed by the *SetZoom* method will zoom and pan the map displayed in the viewport's GIS Object, instead of being applied to the viewport's drawing. In the Chart Zoom Mode, the content of the chart is zoomed and panned. If the *resource_name* parameter is *null*, the zoom mode of the viewport itself will be set. Otherwise, the zoom mode will be set for its child viewport specified by the *resource_name* parameter. If the *zoom_name* parameter is not *null*, it specifies the resource path of a GIS or Chart object relative to the *zoom_object* parameter, or relative to the viewport if the *zoom_object* parameter is *null*. The method may be invoked with *zoom_object=null* and *zoom_object_name=null* to reset the zoom mode to the default Drawing Zoom Mode.

*BOOL GISGetElevation(long object, String resource_name, String layer_name, double lon, double lat, double * elevation)*

Returns elevation of a *lat*/*lon* point on a map. If the *resource_name* parameter is *null*, the object parameter specifies the GIS object, otherwise it specifies the GIS object's parent and *resource_name* specifies the resource path of the GIS Object relative to the parent. The *layer_name* specifies the name of the elevation layer to query. The elevation value is returned via the *elevation* parameter. The method returns TRUE if the query was successful.

long GISCreateSelection(long object, String resource_name, String layers, double x, double y, int select_labels)

Returns a message object containing information about GIS features located at the specified position on the map. If the *resource_name* parameter is *null*, the *object* parameter specifies the GIS object, otherwise *object* specifies the GIS object's parent and *resource_name* specifies the resource path of the GIS Object relative to the parent. The *layers* parameter specifies a list of layers to query. The *x* and *y* parameters specify location on the map in the screen coordinates of the GIS object's parent viewport. The *select_label* parameter specifies the label selection policy. It can have the following values:

- GIS_LBL_SEL_NONE - disables label selection
- GIS_LBL_SEL_IN_TILE_PRECISION - enables label selection
- GIS_LBL_SEL_MAX_PRECISION - enables label selection with the maximum precision.

Refer to page 230 for more information on the label selection.

The function returns a message object containing information about the selected GIS features. Refer to the *GlmGetSelection* section on page 129 of the *GLG Map Server Reference Manual* for information on the structure of the returned message object as well as a programming example that demonstrates how to handle it.

long GISGetDataset(long object, String resource_name)

Returns a dataset object associated with the GIS object. If the *resource_name* parameter is *null*, the object parameter specifies the GIS object, otherwise it specifies the GIS object's parent and *resource_name* specifies the resource path of the GIS Object relative to the parent. The method may be invoked after the GIS object has been setup to retrieve its dataset. The dataset may be used to dynamically change attributes of individual layers displayed in the GIS object. The program can use SetResource methods for changing layer attributes. Refer to the *GLG Map Server Reference Manual* for information on layer attributes.

*BOOL GISConvert(long object, String resource_name, long coord_type, BOOL coord_to_lat_lon, double x1, double y1, double z1, double * x2, double * y2, double * z2)*

Converts coordinates between the GIS coordinate system of the GIS Object and GLG coordinate system of the drawing. If the *resource_name* parameter is *null*, *object* specifies the GIS Object whose coordinate system to use for conversion. If the *resource_name* parameter is not *null*, *object* specifies a parent of the GIS Object. The *resource_name* parameter specifies the resource name of the GIS Object inside the parent object specified by the *object* parameter. The *coord_type* parameter specifies the type of the GLG coordinate system type to convert to or from: *GLG_SCREEN_COORD* for screen coordinates or *GLG_OBJECT_COORD* for world coordinates. If the *coord_to_lat_lon* parameter is TRUE, coordinates are converted from GLG to GIS coordinate system. If it is FALSE, the coordinates are converted from GIS to GLG coordinate system. The *x1*, *y1* and *z1* parameters specify coordinates to be converted. The x2, y2 and z2 parameters point to the variables that will receive converted coordinate values. The function returns TRUE if conversion succeeds. When converting from GLG to GIS coordinates, the Z coordinate is set to a negative GLG_GIS_OUTSIDE_VALUE value for points on the invisible part of the globe.

long SendMessage(String resource_name, String message, long param1, long param2, long param3, long param4)

Sends a message specified by the *message* parameter. If the *resource_name* parameter is *null*, the message is sent to the object itself. Otherwise, the message is sent to the object's child specified by the *resource_name* parameter. The *param<n>* arguments define additional parameters of the message depending on the message type.

Refer to the *Input Objects* chapter on page 263 of the *GLG User's Guide and Builder Reference Manual* for a list of messages supported by each type of the available input handlers.

long SendMessageStr(String resource_name, String message, String param1, long param2, long param3, long param4)

Same as *SendMessage* but sends the first parameter as a string. It is intended for use in VB.net environment that does not provide type casts.

long ExportStrings(long object, String filename, long separator1, long separator2)

Writes all text strings defined in the drawing specified by the *object* parameter into a string translation file using requested separator characters. Refer to the *Localization Support* chapter on page 255 of the *GLG User's Guide and Builder Reference Manual* for information about the string translation file format. The method returns the number of exported strings or -1 in case of an error.

long ImportStrings(long object, String filename)

Replaces text strings in the drawing defined by the *object* parameter with the strings loaded from a string translation file. Refer to the *Localization Support* chapter on page 255 of the GLG User's Guide and Builder Reference Manual for information about the string translation file format. The method returns the number of imported strings or -1 in case of an error.

long ExportTags(long object, String filename, long separator1, long separator2)

Writes tag names and tag sources of all tags defined in the drawing specified by the *object* parameter into a file using requested separator characters. The method uses the same file format as the *ExportStrings* method. Refer to the *Localization Support* chapter on page 255 of the *GLG User's Guide and Builder Reference Manual* for information about the file format. The method returns the number of exported tags or -1 in case of an error.

long ImportTags(long object, String filename)

Replaces tag names and tag sources of tags in the drawing defined by the *object* parameter with information loaded from a tag translation file. The method uses the same file format as the *ImportStrings* method. Refer to the *Localization Support* chapter on page 255 of the *GLG User's Guide and Builder Reference Manual* for information about the tag translation file format. The function returns the number of imported tags or -1 in case of an error.

void Print(String filename, double x, double y, double width, double height,
BOOL portrait, BOOL stretch)

Saves encapsulated PostScript output with the current state of the drawing into the specified file. The *x*, *y*, *width*, and *height* parameters define the PostScript page layout: *xpos* and *ypos* define the coordinates of the upper left corner of the rectangle, *width* and *height* define its dimensions. All coordinates are specified in the world coordinate system, which maps a rectangle defined by the points (0;0) and (2000;2000) to a page.

As an example, the default page layout you get when you print a drawing by setting the *PrintFile* property puts the lower left corner of the drawing area at (*100;100*), while the dimensions are *1900* by *1900*. This makes the drawing about as large as it can be while still keeping a small border all the way around the page. The *portrait* parameter specifies portrait (TRUE) or landscape (FALSE) mode. If the *stretch* parameter is set to FALSE, the X/Y ratio of the drawing is preserved.

BOOL SaveImage(long object, String resource_name, String filename, long format)

Saves a JPEG or PNG image of the visible area of the viewport's drawing into a file. Returns TRUE if the image was successfully saved. This method is disabled in Secure mode if GLG_ACTIVEX_SAVE_FILE environment variable is not set. Both the JPEG (*format*=2) and

PNG (*format*=4) image formats are supported. The *resource_name* parameter specifies the resource name of a child viewport whose image to generate, or *null* to generate the image of the viewport itself. For a light viewport, the output will be generated for its parent viewport.

BOOL SaveImageCustom(long object, String resource_name, String filename, long format, long x, long y, long width, long height, long gap)

Saves a JPEG or PNG image of the specified area of the drawing into a file. Returns TRUE if the image was successfully saved. This method is disabled in Secure mode if GLG_ACTIVEX_SAVE_FILE environment variable is not set. Both the JPEG (*format*=2) and PNG (*format*=4) image formats are supported. The *resource_name* parameter specifies the resource name of a child viewport whose image to generate, or *null* to generate the image of the viewport itself. For a light viewport, the output will be generated for its parent viewport.

The *x, y, width* and *height* parameters specify the area of the drawing in screen pixels for generating the image. If both the *width* and *height* parameters are set to 0, the image of the whole drawing is generated, which may be bigger than the visible area of the viewport if the drawing is zoomed in. The *gap* parameter specifies the padding space between the extent of the drawing and the edge of the image when *width* and *height* equal 0. The viewport's zoom factor defines the scaling factor for objects in the drawing when the image is saved.

long GenerateImage(long object, String resource_name, long bitmap)

Same as the *SaveImage* method but returns the image as a bitmap. If the *bitmap* parameter is not NULL, fills and returns the bitmap provided by the parameter instead of creating and returning a new bitmap.

long GenerateImageCustom(long object, String resource_name, long bitmap, long format, long x, long y, long width, long height, long gap)

Same as the *SaveImageCustom* method but returns the image as a bitmap. If the *bitmap* parameter is not NULL, fills and returns the bitmap provided by the parameter instead of creating and returning a new bitmap.

void AboutBox()

Displays version number and other GLG ActiveX Control information in a box on the screen.

String GetSelectedName()

Returns the name of the last selected object reported by the *Select* event. It may be used to access the Select event information by using the control's methods in cases when there is no access to the events' parameters (scripting usage).

short GetSelectionButton()

Returns the mouse button which was pressed to generate the last *Select* event. Similar to *GetSelectedName*, it may be used to access the *Select* event information by using the control's methods.

BOOL GetModifierState(int modifier)

Returns the state of the modifier requested by the *modifier* parameter. Use 1 to query the state of the *Shift* key, 2 for the *Control* key and 3 for a double click: these values correspond to the elements of the *GlgModifierType* enum defined in the *GlgApi.h* file.

long CreateSelectionNamesExt(long top_vp, long selected_vp,
double x1, double y1, double x2, double y2)

ADVANCED: Given the top viewport of the drawing and a screen coordinates rectangle in a selected viewport, returns an array of names of all objects within the rectangle. This method may be used in the Trace event handler to implement custom selection logic based on the mouse events. Either a viewport or a light viewport can be used for the *top_vp* and *selected_vp* parameters.

long CreateSelectionMessage(long top_vp, long selected_vp,
double x1, double y1, double x2, double y2, long selection_type, long button)

ADVANCED: Given the top viewport of the drawing and a screen coordinates rectangle in a selected viewport, the method searches all objects inside the rectangle for the action with the specified selection trigger type attached to an object. Either a viewport or a light viewport can be used for the *top_vp* and *selected_vp* parameters.

The method returns a selection message for the first matching action it finds, and may be used in the Trace event handler to retrieve an action which would be triggered by a specified mouse event in the rectangular area. The selection message is a message equivalent to the message received in the input callback. The *selection_type* parameter may specify one of the following predefined selection types: GLG_MOVE_SELECTION, GLG_CLICK_SELECTION or GLG_TOOLTIP_SELECTION defined in the *GlgAPI.h* file.

void ChangeObject(long object, String resource_name)

Sends a change message to the object without actually changing the object's attributes, may be used to redraw the object.

If the *resource_name* parameter is *null*, the change message will be sent to the object specified by the *object* argument. Otherwise, the message will be sent to its child object specified by the *resource_name* parameter.

void SetBrowserObject(long browser, long object)

Sets the object for browsing with the GLG resource, tag or alarm browser widgets.

void SetBrowserSelection(long object, String resource_name, String selection, String filter)

Sets a new value of a browser's selection and filter text boxes for resource, tag, alarm and data browsers. Setting a new selection will display a new list of matching entries.

If the *resource_name* parameter is *null*, a new selection will be set for the browser supplied by the *object* argument. Otherwise, it will be set for its child object specified by the *resource_name* parameter.

BOOL SetEditMode(long viewport, String resource_name, BOOL edit_mode)

Sets the viewport's edit mode which disables input objects in the viewport while the drawing is being edited; returns TRUE on success. The *viewport* could be either a viewport or a light viewport. If the *resource_name* parameter is *null*, the edit mode of the viewport itself is set. Otherwise, the edit mode of its child viewport specified by the *resource_name* parameter will be set. If the *viewport* parameter is *null*, the viewport of the ActiveX control's drawing is used. The edit mode is global and may be set only for one viewport used as a drawing area. Setting the edit mode of a new viewport resets the edit mode of the previous viewport. The viewport's edit mode is inherited by its all child viewports. Returns TRUE on success.

void SetTraceViewport(long viewport, String resource_name)

Specifies the viewport for which the trace events will be activated. If the *resource_name* parameter is *null*, the trace events will be activated for the viewport itself. Otherwise, trace events will be activated for the child viewport specified by the *resource_name* parameter. If the *viewport* parameter is *null*, the viewport of the ActiveX control's drawing is used. The trace events will be generated only if the control's *TraceEnabled* property is set.

void SetFocus(long object, String resource_name)

Sets keyboard focus to the parent viewport of an object, or to the object itself if it is a viewport. If *resource_name* is *null*, the object is specified by the *object* parameter. If *resource_name* is not *null*, it specifies the child object of the *object* that will be used for setting focus.

void DropObject(long object)

Dereferences an object.

long GetStringID(String string)

Converts the *string* to a persistent GLG string and returns its handle. The handle has to be freed with *FreeStringID* method when done. This function provides conversion between the *String* and GLG string types for the .NET environment.

void FreeStringID(long string_id)

Frees the GLG string identified by its *string_id* handle in the .NET environment.

BOOL SetCursor(long viewport, long cursor_type)

Sets the Windows cursor type to be used by the *viewport*. If the *viewport* parameter is *null*, the cursor of the main viewport of the ActiveX Control is used. The *cursor_type* parameter specifies one of the predefined Windows cursor types. Returns TRUE on success.

void GlgGetMajorVersion()

void GlgGetMinorVersion()

Return major and minor version numbers of the GLG Toolkit.

long GetSelectedPlot()

Returns the plot object corresponding to the last legend item selected with the mouse, if any, or NULL if no plots were previously selected.

long GetNamedPlot(long object, String resource_name. String plot_name)

Finds a chart's plot by name and returns its object ID. If the *resource_name* parameter is *null*, a named plot of the chart specified by the *object* parameter is returned. Otherwise, the *resource_name* parameter points to a child chart of the *object*. The method returns 0 if a plot with the specified name has not been found.

long AddPlot(long object, String resource_name, long plot)

Adds a plot to a chart object. If the *resource_name* parameter is *null*, the plot is added to the chart specified by the *object* parameter. Otherwise, the plot is added to a child chart of the *object* pointed to by the *resource_name* parameter. The *plot* parameter specifies the object ID of the plot object to add; 0 can be used to create a new plot. The method returns the object ID of the added plot on success, or 0 on failure.

BOOL DeletePlot(long object, String resource_name, long plot)

Deletes a plot from a chart object. If the *resource_name* parameter is *null*, the plot is deleted from the chart specified by the *object* parameter. Otherwise, the plot is deleted from a child chart of the *object* pointed to by the *resource_name* parameter. The *plot* parameter specifies the plot object to delete. The method returns TRUE if the plot was successfully deleted.

long AddTimeLine(long object, String resource_name, long time_line, double time_stamp)

Adds a time line to a chart object. If the *resource_name* parameter is *null*, the time line is added to the chart specified by the *object* parameter. Otherwise, the time line is added to a child chart of the *object* pointed to by the *resource_name* parameter. The *time line* parameter specifies the level line object to be used as a time line; 0 can be used to create a new time line. The *time_stamp* parameter specifies the timestamp to position the time line at and is assigned to the time line's *Level* attribute. The method returns the added time line object on success, or 0 on failure. The returned time line is referenced only by the chart's time line array it has been added to and may be destroyed when the chart scrolls. The program should increase the time line's reference count if it needs to keep a persistent reference to the time line. The *NumTimeLines* attribute of the chart should be set to -1 in order to use this function.

BOOL DeleteTimeLine(long object, String resource_name, long time_line, double time_stamp)

Deletes a time line from a chart object. If the *resource_name* parameter is *null*, the time line is deleted from the chart specified by the *object* parameter. Otherwise, the time line is deleted from a child chart of the *object* pointed to by the *resource_name* parameter. The *time_line* parameter specifies the time line object to delete. If *time_line* is set to 0, the time line with the timestamp

specified by the *time_stamp* attribute will be deleted. If *time_line* is not 0, the *time_stamp* parameter is ignored. The method returns TRUE if the time line was successfully deleted. The *NumTimeLines* attribute of the chart should be set to -1 in order to use this function.

long AddAnnotation(long object, String resource_name, long annotation, double position_x, double position_y, BOOL add_box)

Adds an annotation to a chart object. If the *resource_name* parameter is *null*, the annotation is added to the chart specified by the *object* parameter. Otherwise, the annotation is added to a child chart of the *object* pointed to by the *resource_name* parameter. The *annotation* parameter specifies the annotation object to be added; 0 can be used to create a new annotation. The *position_x* and *position_y* parameters specify the annotation's position. If *add_box* is set to TRUE, a text box will be added to the annotation's text object. The method returns the added annotation object on success, or 0 on failure. The *NumAnnotations* attribute of the chart should be set to -1 in order to use this function.

BOOL DeleteAnnotation(long object, String resource_name, long time_line, double position_x, double position_y)

Deletes an annotation from a chart object. If the *resource_name* parameter is *null*, the annotation is deleted from the chart specified by the *object* parameter. Otherwise, the annotation is deleted from a child chart of the *object* pointed to by the *resource_name* parameter. The *annotation* parameter specifies the annotation object to delete. If *annotation* is set to 0, the annotation at the position specified by the *position_x* and *position_y* attributes will be deleted. If *annotation* is not 0, position parameters are ignored. The method returns TRUE if the annotation was successfully deleted. The *NumAnnotations* attribute of the chart should be set to -1 in order to use this function.

BOOL ClearDataBuffer(long object, String resource_name)

Clears accumulated data samples of a real-time chart or one of its plots. If *resource_name* is *null*, the *object* parameter supplies an object ID of the chart or plot object. If *resource_name* is not *null*, it supplies a resource name of a child chart or plot object inside the parent object specified by the *object* parameter. For a chart, *ClearDataBuffer* discards data samples in all plots of the chart. For a plot, it discards only the data samples of that plot. The method returns TRUE on success.

*BOOL GetChartDataExtent(long object, String resource_name, double * min, double * max, BOOL x_extent)*

Queries the minimum and maximum extent of the data accumulated in a chart or in one of its plots. If the *resource_name* parameter is *null*, the method returns the data extent of the chart or the plot specified by the *object* parameter. Otherwise, the *resource_name* parameter points to a child chart or a child plot of the *object*. If *x_extent* is TRUE, the X extent of the data is returned in *min* and *max* parameters, otherwise the method returns the Y extent of the data.

For a chart object, the method returns the data extent of all plots in the chart. For a plot, the data extent of that plot is returned. The object hierarchy must be set up for this function to succeed. The method returns TRUE on success.

BOOL SetLinkedAxis(long object, String resource_name, long axis_object, String axis_resource_name)

Associates a chart's plot or a level line with an Y axis for automatic rescaling, returns TRUE on success. If *resource_name* is *null*, associates the *object*, otherwise associates a child plot or a child level line of the *object* pointed to by the *resource_name* parameter. If *axis_resource_name* is *null*, associates with *axis_object*, otherwise associates with the child axis of *axis_object* pointed to by the *axis_resource_name* parameter.

BOOL UpdateChartTimeAxis(long object, String resource_name, double time_stamp, boolean initialize)

Scrolls the chart forward to show all data samples up to the timestamp or index specified by the *time_stamp* parameter, returns TRUE on success. If *resource_name* is *null*, *this* chart is scrolled, otherwise a child chart pointed to by the *resource_name* parameter is scrolled. Set *initialize=true* to initialize an empty sweep chart, positioning the sweep bar at the start of the plot instead of the plot end.

While a strip chart's X axis can be updated by simply setting its *EndValue* resource to the timestamp, a sweep chart needs to update the sweep bar and update the X axis only if needed. The *UpdateChartTimeAxis* method should be used to properly update sweep charts after using the quick mode of the *AddDataSampleNode* method to supply a large number of data samples. The method handles both strip and sweep charts.

BOOL UpdateChartState(long object, String resource_name, int state)

Triggers updating a chart's state after prefilling it with data using the quick mode, returns *true* on success. If *resource_name* is *null*, *this* chart is updated, otherwise a child chart pointed to by the *resource_name* parameter is updated. The state parameter contains binary flags that specify the state to be updated:

- CHART_AUTOSCALE_STATE=1 - updates chart's autoscale
- CHART_BUFFER_STATE=2 - triggers trimming of the chart data buffer based on the chart's *BufferXSpan* and *BufferSize* attributes
- CHART_SCROLLBAR_STATE=4 - updates position of the chart's scrollbars, if any.

Multiple flags can be combined using the bitwise OR operator.

The method marks the selected chart states for updating. The actual update happens on the next invocation of the *Update* or *SetupHierarchy* methods.

long GetNativeComponent(long object, String resource_name, short type)

Returns an ID of a native windowing system component used to render the viewport. If *resource_name* is *null*, a native component of the viewport specified by the *object* parameter is returned, otherwise a native component of a child viewport of the *object* pointed by the

resource_name parameter is returned. The *type* parameter defines the type of the native component to query, as described on page 86. The value of zero may be used to retrieve the handle of a viewport's window.

String GetObjectName(long object)

String GetObjectType(long object)

String GetDataType(long object)

Convenience functions for querying *Name* and *Type* properties of an object, as well as a *DataType* property of data and attribute objects.

Additional Standard API Methods

The *GetViewportExt*, *GetElementExt* and *GetSizeExt* methods described in the *Intermediate and Extended API Methods* section are also available in the Standard API.

Intermediate and Extended API Methods

The GLG Extended API provides methods that allows an application program to add objects on the fly or edit the drawing at run time, as well as query objects and resources contained in the drawing. This may be used to create sophisticated applications using C# or VB.NET.

The ActiveX Control provides two flavors of the Extended API: **Intermediate API** and **Extended API**. The Intermediate API provides all Extended API methods except methods for creating and deleting objects at run time.

The ActiveX Control's Extended API methods mimic the methods of the GLG Extended API. This chapter provides only a brief description of each method; refer to the *GLG Intermediate and Extended API* chapter on page 135 for a detailed description of each method's functionality.

The methods of the ActiveX control Extended API listed below have parameters identical to the corresponding methods of the C API, unless mentioned otherwise. A long value is used to represent an object id. This object id has meaning only inside a particular ActiveX control and can't be used interchangeably between several ActiveX controls. Refer to the *GlgApi.h* file for the values of the enumerated parameters of the methods. In most of the cases a null value may be used as a reasonable default.

There are several methods of Extended API which are identical to the corresponding methods of the regular API with the only difference of having an additional object parameter. For example,

```
SetDResource( resource_name, resource_value )
```

and

```
SetDResourceExt( object, resource_name, resource_value )
```

The Extended API method above allows setting resources of the specific object while the regular API method sets the resource of the main viewport of the drawing (viewport named *"$Widget"*).

If the null value is used as an object parameter of such Extended API methods, the object defaults to the main viewport of the drawing and the result will be identical to the corresponding regular API method invocation. This is also true even for the Extended API method which do not have corresponding regular API methods. For example, to add objects to the main viewport of the drawing using *AddObjectExt* method, you can use *null* as the first parameter, which results in using the main viewport of the drawing as a container. For the *Get/SetResource* methods of the Extended API passing *null* value for the object parameter allows using the methods even if the Extended API is disabled.

All Extended API methods have the *Ext* suffix, allowing to differentiate them from methods of the regular API.

The GLG ActiveX Control provides the following Extended API methods:

long AddAttachmentPointExt(long reference, double dx, double dy, double dz)

Adds an attachment point to a *reference* object and returns the added point. The world coordinate offsets for initial position of the attachment point relatively to the reference object's single control point are provided by the *dx, dy* and *dz* parameters. When the reference object is resized, the offsets will be automatically adjusted to maintain the point's position relatively to the object.

BOOL AddObjectToTopExt(long container, long object)

Adds the object to the beginning of the *container* (Extended API only). Returns *true* if the object was successfully added.

BOOL AddObjectToBottomExt(long container, long object)

Adds the object to the end of *container* (Extended API only). Returns *true* if the object was successfully added.

BOOL AddObjectAtExt(long container, long index)

Adds the object to the *container* at the *index* position (Extended API only). Returns *false* if *index* is invalid.

BOOL AddObjectExt(long container, long object, long access_type, long position_modifier)

Adds an object to the *container* at the position indicated by the *access_type* (BOTTOM, TOP or CURRENT) (Extended API only). If the access type is CURRENT, the object is added before or after the element indicated by the container's current position, as indicated by the *position_modifier* parameter (BEFORE or AFTER). Sets the container's current position to point to the added object. Returns *true* if the object was successfully added.

BOOL ContainsObjectExt(long container, long object)

Returns *true* if the object is contained in *container*.

long CloneObjectExt(long object, long clone_type)

Creates and returns a copy of an object according the clone_type (Extended API only).

BOOL ConstrainObjectExt(long from_object, long to_object)

Constrains an attribute object.

long ConvertViewportType(long object)

ADVANCED: Converts a viewport to a light viewport, or a light viewport to a viewport, returning a newly created object on success or *null* on error. The *object* parameter supplies the object to be converted. If a converted object is added to the drawing, the original object must be deleted, since both objects reuse the graphical objects inside the viewport and cannot be both displayed at the same time.

long CopyObjectExt(long object)

Creates and returns a copy of the object (Extended API only).

long CreateChartSelectionExt(long chart, long plot, double x, double y, BOOL screen_coord, BOOL include_invalid, BOOL x_priority)

ADVANCED: Selects a chart's data sample closest to the specified *x, y* position and returns a message object containing the selected sample's information. If *plot* is NULL, the method queries samples in all plots of the chart and selects the sample closest to the specified position. If *plot* is not NULL, the method queries only the samples in that plot.

If the *screen_coord* parameter is set to TRUE, the position is defined in the screen coordinates of the chart's parent viewport. If *screen_coord* is set to FALSE, the *x* position is defined as an X or time value, and the *y* position is defined in relative coordinates in the range [0; 1] to allow the chart to process selection for plots with different ranges (0 corresponds to the *Low* range of each plot, and 1 to the *High* range). When the mouse position is used, add GLG_COORD_MAPPING_ADJ to the x and y screen coordinates of the mouse for precise mapping.

The *dx* and *dy* parameters specify an extent around the point defined by the *x* and *y* parameters. The extent is defined in the same coordinates as *x* and *y* depending on the value of the *screen_coord* parameter. Only the data samples located within the *dx* and *dy* distances from the specified position are considered for the selection.

If *include_invalid* if set to TRUE, invalid data samples are considered for selection, otherwise they are never included. If *x_priority* is set to TRUE, the method selects a data sample closest to the specified position in the horizontal direction, with no regards to its distance from the specified position in the vertical direction. If *x_priority* is set to FALSE, a data sample with the closest distance from the specified position is selected.

The information about the selected data sample is returned in the form of a message object described in the *Chart Selection Message Object* section on page 396. The method returns *null* if no data sample matching the selection criteria was found. The returned message object must be dereferenced using the *DropObject* method when finished.

long CreateObjectExt(long object_type, String object_name,
long data1, long data2, long data3, long data4)

Creates and returns an object of a specified type (Extended API only). The object has to be explicitly added to the drawing in order to be displayed.

long CreatePointArrayExt(long object, long type)

Creates and returns an array containing all of an *object*'s control points. The *type* parameter specifies the type of points to be returned: control points (CONTROL_POINTS), or control and attachment points (CONTROL_AND_ATTACHMENT_POINTS).

long CreateSelectionExt(long top_vp, long selected_vp, double x1, double y1, double x2, double y2)

ADVANCED: Given the top viewport of the drawing and a screen coordinates rectangle in a selected viewport, returns an array of all objects within the rectangle. This method may be used in the Trace event handler to implement custom selection logic based on the mouse events. Either a viewport or a light viewport can be used for the *top_vp* and *selected_vp* parameters.

String CreateTooltipStringExt(long object, double x, double y, double dx, double dy,
String format)

ADVANCED: Creates and returns a chart or axis tooltip string corresponding to the specified *x, y* position in screen coordinates. The string can be used to provide cursor feedback and display information about the data sample under the current mouse position, as shown in the Real-Time Strip-Chart demo.

The *format* parameter provides a custom format for creating a tooltip string and uses the same syntax as the *TooltipFormat* attributes of the chart and axis objects. If it is set to NULL or an empty string, the format provided by the *TooltipFormat* attribute of the chart or the axis will be used. A special "<*single_line*>" tag may be prepended to the format string to force the method to remove all line breaks from the returned tooltip string.

If the *object* is a chart and the selected position is within the chart's data area, the method selects a data sample closest to the specified position and returns a tooltip string containing information about the sample. The *dx* and *dy* parameters define the maximum extent in pixels around the specified position for selecting a data sample, and the *TooltipMode* attribute of the chart defines the data sample selection mode: X or XY. The method returns NULL if no samples matching the selected criteria were found.

If the *object* is an axis, or if the *object* is a chart and the selected position is on top of one of the chart's axes, the function returns a tooltip string that displays the axis value corresponding to the specified position.

When using the cursor position, add GLG_COORD_MAPPING_ADJ to the cursor's x and y screen coordinates for precise mapping.

The returned string must be freed using the *FreeStringID* function.

BOOL DeleteObjectAtExt(long container, long index)

Deletes the object of the *container* at the *index* position (Extended API only). Returns *false* if *index* is invalid.

BOOL DeleteBottomObjectExt(long container)

Deletes the first object of this container (Extended API only). Returns *true* if the object was successfully deleted.

BOOL DeleteTopObjectExt(long container)

Deletes the first object of this container (Extended API only). Returns *true* if the object was successfully deleted.

BOOL DeleteThisObjectExt(long container, long object)

Deletes an object from *container* (Extended API only). Returns *true* if the object was successfully deleted.

BOOL DeleteObjectExt(long container, long reserved, long access_type, long position_modifier)

Deletes an object from a container (Extended API only).

long FindObjectExt(long container, long object, long search_type, long reserved)

Finds an object in the container according to the search type and returns it.

long FindMatchingObjectsExt(long object, long match_type, BOOL find_parents,
BOOL find_first_match, BOOL search_inside,
BOOL search_drawable_only, long object_type,
LPCTSTR object_name, LPCTSTR resource_name, long object_id)

Finds either one or all parents or children of the specified *object* that match the supplied criteria (Extended API only).

The *match_type* parameter specifies a bitwise mask of the object matching criteria using the following constants defined in the *GlgApi.h* file:

- GLG_OBJECT_TYPE_MATCH = 1 finds objects with an object type matching the type specified by the *object_type* parameter.
- GLG_OBJECT_NAME_MATCH = 2 finds objects with a name matching the name specified by the *object_name* parameter.
- GLG_RESOURCE_MATCH = 4 finds objects that have the resource specified by the *resource_name* parameter.
- GLG_OBJECT_ID_MATCH = 8 finds an object with the ID specified by the *object_id* parameter. This can be used to find out if the object supplied as the object parameter is a child or a parent of the object provided by *object_id*.

Multiple criteria can be ORed together. All of the criteria in the mask must be true in order for an object to match.

If *find_parents* is set to *true*, ancestors (parents, grandparents and so on) of the object are searched. Otherwise, the object's descendants (children, grandchildren and so on) are searched. If *find_first_match* is set to *true*, the search is stopped after the first match and the first matching object is returned, otherwise all matching objects will be returned.

If *search_inside* is set to *false*, the object's children or parents are not searched if the object itself matches. If *search_drawable_only* is set to *true* when searching descendants, only drawable objects are searched. For example, when this flag is set, a polygon object's attributes such as *FillColor* and *EdgeColor*, will not be searched, which can increase the search speed. Refer to the description of the *IsDrawableExt* method on page 301 for more information.

If *find_first_match* is set to *true*, the method returns the first found matching object. If *find_first_match* is set to *false,* a group containing all matching objects is returned. If no matching objects were found, 0 is returned.

double GetDResourceExt(long object, String resource_name)
double GetDTagExt(long object, String tag_source)

Return a value of an object's resource or tag.

String GetSResourceExt(long object, String resource_name)
String GetSTagExt(long object, String tag_source)

Return the value of a string resource or tag.

*double GetGResourceExt(long object, String resource_name, double * x, double * y, double * z)*
*double GetGTagExt(long object, String tag_source, double * x, double * y, double * z)*

Query the value of the drawing's geometrical resource or tag and return it via the *x*, *y* and *z* parameters.

double GetXResourceExt(long object, String resource_name)
double GetXTagExt(long object, String tag_source)

double GetYResourceExt(long object, String resource_name)
double GetYTagExt(long object, String tag_source)

double GetZResourceExt(long object, String resource_name)
double GetZTagExt(long object, String tag_source)

Return the value of the X, Y or Z coordinates of a G-type resource or tag.

long GetElementExt(long container, long index)

Returns the element of the *container* indicated by the *index*.

BSTR GetElementAsString(long container, long index)

Returns a string of the string list (*container)* indicated by the *index*. This method is intended for VB.net environment which does not provide type casts.

long GetIndexExt(long container, long object)

Returns the index of the first occurrence of the object in the container or -1 if the object wasn't found in the container.

long GetStringIndexExt(long container, String string)

Returns the index of the first occurrence of the string in the container or -1 if the string wasn't found in the container.

long GetLegendSelectionExt(long object, double x, double y)

Returns the plot object corresponding to the legend item at the specified position, if any. If no legend item is found at the specified position, NULL is returned. The *object* parameter specifies the legend object, and the *x* and *y* parameters specify position in screen coordinates of the viewport that contains the legend.

long GetNamedObjectExt(long container, String name)

Returns the named element of the *container* defined by the *name* parameter.

long GetNumParents(long object)

Returns the number of parents of an object. Used to determine a type of the return value of the GetParentsExt method.

long GetParentExt(long object)

Returns the objects' parent if object has one parent. If object has more than one parent, returns an array of parents. The type of the return value may be determined by using *GetNumParents* method.

long GetParentViewportExt(long object, BOOL heavy_weight)

Returns the parent viewport of a drawable GLG object specified by the *object* parameter. The heavy_weight parameter controls the type of a parent viewport to be returned. If set to *False*, the method returns the closest parent viewport regardless of its type: either a heavyweight or light viewport. If set to *True*, it returns the closest heavyweight parent viewport. For example, if the *object* is inside a light viewport, it will return the heavyweight viewport which is a parent of the light viewport.

long GetResourceObjectExt(long object, String resource_name)

Finds and returns an object ID of the object's attribute or named resource.

long GetTagObjectExt(long object, String search_string, BOOL by_name, BOOL unique_tags, BOOL single_tag, short tag_type_mask)

Finds a single tag with a given tag name or tag source, or creates a list of tags matching the wildcard. The *tag_type_mask* defines the type of tags to query: data tags or export tags. For export tags, tag type constants may be *OR*ed to query multiple subtypes of export tags.

The *search_string* specifies either the exact tag name, data tag source or export tag category to search for, or a search pattern which may contain the *?* and * wildcards. The *by_name* parameter defines the search mode. If set to *TRUE*, the search is performed by matching the tag names, otherwise the search is done by matching the tag sources for data tags or matching the tag categories for export tags.

In the single tag mode, the first tag of the highest priority matching the search criteria is returned. Input tags are prioritized over output tags, and enabled tags over disabled tags. The *unique_tags* controls if only one data tag of the highest priority is included when several data tags with the same tag name or tag source exist in the drawing. The *unique_tags* flag is ignored in the single tag mode and when searching for export tags.

In the single tag mode, the method returns the first attribute that has a tag matching the search criteria. In the multiple tag mode, a list containing all attributes with matching tags will be returned.

In the multiple tag mode, the method creates a new list object and its returned value must be explicitly dereferenced. The returned list must be dereferenced using *DropObject* when it is no longer needed.

long GetAlarmObjectExt(long object, String search_string, BOOL single_alarm, long reserved)

Finds a single alarm with a given alarm label, or creates a list of alarms matching the specified wildcard. The *search_string* specifies either the exact alarm label to search for or a search pattern which may contain the *?* and * wildcards. The *reserved* parameter is reserved for future use and must be set to 0.

In a single alarm mode, the function returns an object ID of the first alarm that has an alarm label matching the search criteria. In the multiple alarm mode, it returns an object ID of a group containing all alarms with matching alarm labels. The returned list must be dereferenced using *DropObject* when it is no longer needed.

long GetSizeExt(long container)

Returns the size of a container object.

long GetViewportExt(void)

Returns the object ID of the main viewport of the drawing (the viewport named *"$Widget"*)

BOOL IsDemoExt(void)

Returns TRUE if the a Free Non-Commercial version of the ActiveX Extended API is used.

BOOL IsDrawableExt(long object)

Returns *true* if the object is drawable. Drawable objects can be placed directly in a drawing area, have control points and a bounding box, have the visibility attribute and can contain custom properties, actions and aliases. For a chart object, the chart itself is drawable, but its plots are not.

void InverseExt(long container)

Inverses the order of components inside the container.

BOOL IsAtExt(long container, long position)

Checks the position of the iteration pointer (the current element of the *container*).

- The following lists the return values for different values of the *position* argument:
- for GLG_START, returns TRUE if the iteration pointer is positioned before the first element of the container
- for GLG_END, returns TRUE if the iteration pointer is positioned past the last element of the container
- for GLG_FIRST, returns TRUE if the iteration pointer is positioned at the first element of the container
- for GLG_LAST, returns TRUE if the iteration pointer is positioned at the last element of the container.

long IterateExt(long container)
long IterateBackExt(long container)

For the forward traversing, *IterateExt* advances the iteration pointer and returns the next element of *container*. For backward traversing, *IterateBackExt* decrements the iteration pointer and returns the previous element.

The *SetStartExt* method must be used to initialize the container for iterating before *IterateExt* is invoked the first time, and *SetEndExt* must be used to initialize the container for backward iteration with *IterateBackExt*. See page 163 for an example. The add, delete and search operations affect the iteration state and should not be performed on the container while it is being iterated.

To perform a safe iteration that is not affected by the add and delete operations, a copy of the container can be created using the *CloneObjectExt* method with the SHALOW_CLONE clone type. The shallow copy will return a container with a list of objects IDs, and it can be safely iterated while objects are added or deleted from the original container.

Alternatively, *GetElementExt* can be used to traverse elements of the container using an element index, which is not affected by the search operations on the container.

long LoadObjectExt(String filename)

Loads an object from a file and returns its object ID. The object has to be added to the drawing to be displayed.

long LoadWidgetFromFileExt(String filename)

Loads a viewport named *"$Widget"* from a file and returns its object ID. The viewport has to be added to the drawing to be displayed.

long LoadWidgetFromObjectExt(long object)

Finds a viewport named *"$Widget"* inside the object returns its object ID.

BOOL MoveObjectExt(long object, long coord_type, double start_x, double start_y, double start_z, double end_x, double end_y, double end_z)

Moves an object by the specified vector.

BOOL MoveObjectByExt(long object, long coord_type, double x, double y, double z)

Moves an object by the specified X, Y and Z distances.

BOOL ScaleObjectExt(long object, long coord_type, double center_x, double center_y, double center_z, BOOL around_center, double scale_x, double scale_y, double scale_z)

If *around_center* equals TRUE, scales an object relative to the specified center by the specified X, Y and Z scale factors, otherwise scales the object relative to the center of the object's bounding box.

BOOL RotateObjectExt(long object, long coord_type, double center_x, double center_y, double center_z, BOOL around_center, double x_angle, double y_angle, double z_angle)

If *around_center* equals TRUE, rotates an object around the specified center by the specified X, Y and Z angles, otherwise rotates the object relatively to the center of the object's bounding box.

BOOL PositionObjectExt(long object, long coord_type, long anchoring, double x, double y, double z)

Positions an object at the specified location using the specified anchoring.

BOOL FitObjectExt(long object, long coord_type, double x1, double y1, double z1, double x2, double y2, double z2, BOOL keep_ratio)

Fits the object to the specified area.

BOOL LayoutObjectsExt(long object, long sel_elem, int type, double distance, BOOL use_box, BOOL process_subobjects)

Performs a requested layout operation. Refer to the *GLG Intermediate and Extended API* chapter on page 135 for more details.

public void TransformObject(long object, long xform, long coord_type, long parent)

ADVANCED: Transforms all control points of an object with the transformation, calculating new control points values.

The *coord_type* specifies the coordinate system to interpret the transformation in. If SCREEN_COORD is used, the parameters of the transformation are considered to be in screen pixels. For example, this may be used to move an object by the number of screen pixels as defined by the mouse move. If OBJECT_COORD or PARENT_COORD is used, the transformation parameters are interpreted as GLG world coordinates.

The *parent* parameter specifies the object's parent and is required in the GLG_PARENT_COORD mode, or if the object is a G data value representing a control point. NULL may be passed in all other cases.

BOOL ScreenToWorldExt(long object, BOOL inside_vp,
double screen_x, double screen_y, double screen_z,
*double * world_x, double * world_y, double * world_z)*

Converts a point coordinates from the screen to the world coordinate system of the object.

If *object* is a viewport or a light viewport, *inside_vp* specifies which world coordinate system to use. If *inside_vp=TRUE,* the method will use the world coordinate system used to draw objects inside this (*object)* viewport. If *inside_vp=FALSE*, the method will use the world coordinate system used to draw this *object* viewport inside its parent viewport. The value of this parameter is ignored for objects other than a viewport or light viewport.

When converting the cursor position to world coordinates, add GLG_COORD_MAPPING_ADJ to the cursor's x and y screen coordinates for precise mapping.

BOOL WorldToScreenExt(long object, BOOL inside_vp,
double world_x, double world_y, double world_z,
*double * screen_x, double * screen_y, double * screen_z)*

Converts a point coordinates from the object's world coordinate system to the screen coordinate system.

If *object* is a viewport or a light viewport, *inside_vp* specifies which world coordinate system to use. If *inside_vp=TRUE,* the method will use the world coordinate system used to draw objects inside this (*object)* viewport. If *inside_vp=FALSE*, the method will use the world coordinate system used to draw this *object* viewport inside its parent viewport. The value of this parameter is ignored for objects other than a viewport or light viewport.

void TranslatePointOriginExt(long from_viewport, long to_viewport,
*double * x, double * y, double * z)*

Converts screen coordinates of a point in the viewport specified by the *from_viewport* parameter to the screen coordinates of the viewport specified by *to_viewport*. The *x, y* and *z* parameters specify the input values and return the output value.

*BOOL RootToScreenCoordExt(long viewport, double * x, double * y)*

converts screen coordinates relative to the root window to the screen coordinates in the specified *viewport*. The *x* and *y* parameters specify the input values and return the output value.

BOOL PositionToValueExt(long object, String resource_name, double x, double y,
*BOOL outside_x, BOOL outside_y, double * value)*

Returns a value corresponding to the specified *x, y* position on top of a chart or an axis object. Otherwise, the *resource_name* parameter points to a child chart, a child plot or a child axis of *this* object.

If *outside_x* or *outside_y* are set to TRUE, the method returns FALSE if the specified position is outside of the chart or the axis in the corresponding direction.

For a plot, the function converts the specified position to a Y value in the Low/High range of the plot and returns the value through the *value* pointer.

For an axis, the function converts the specified position to the axis value and returns the value.

For a chart, the function converts the specified position to the X or time value of the chart's X axis and returns the value.

When using the cursor position, add GLG_COORD_MAPPING_ADJ to the cursor's x and y coordinates for precise mapping.

The function returns TRUE on success.

BOOL PrintExt(long object, String filename, double x, double y,
double width, double height, BOOL portrait, BOOL stretch)

Saves a PostScript image of the specified viewport object into a file. This method is disabled in Secure mode if GLG_ACTIVEX_PRINT_FILE environment variable is not set. If *object* is a light viewport, the output will be generated for its parent viewport.

long ReferenceObjectExt(long object)

References the object and returns its object ID.

void ReleaseObjectExt(long object, long suspend_info)

Releases a previously suspended for editing object. The suspend_info parameter is a returned value of a previous call to *SuspendObjectExt*.

BOOL ReorderElementExt(long container, long old_index, long new_index)

Moves the container's element from the *old_index* to the *new_index* position. The function returns *false* if indexes are out of range.

void ResetHierarchyExt(long object)

Resets the hierarchy of the object.

BOOL SaveObjectExt(long object, String filename)

Saves object into a file. Returns TRUE if the object was successfully saved. This method is disabled in Secure mode if GLG_ACTIVEX_SAVE_FILE environment variable is not set.

BOOL SetDResourceExt(long object, String resource_name, double dvalue)
BOOL SetDResourceIfExt(long object, String resource_name, double dvalue, BOOL if_changed)
BOOL SetDTagExt(long object, String tag_source, double dvalue, BOOL if_changed)

Set a new value of a scalar resource or tag. For tags, all tags with the specified *tag_source* will be set to the supplied value. Returns TRUE if the value of the resource or tag was successfully changed.

BOOL SetElementExt(long container, long index, long new_object)

Replaces the element of the *container* indicated by the *index* with *new_object* (Extended API only).

BOOL SetGResourceExt(long object, String resource_name, double dvalue1, dvalue2, dvalue3)
BOOL SetGResourceIfExt(long object, String resource_name, double dvalue1, dvalue2, dvalue3, BOOL if_changed)
BOOL SetGTagExt(long object, String tag_source, double dvalue1, dvalue2, dvalue3, BOOL if_changed)

Set a new value of the G-type resource or tag. Returns TRUE if the value of the resource or tag was successfully changed.

BOOL SetResourceFromObjectExt(long object, String resource_name, long ovalue)

Sets a value of the data or matrix resource object defined by the *resource_name* to the value of the *ovalue* object (Extended API only). The resource types should match. Returns TRUE if the value of the resource was successfully changed.

BOOL SetResourceObjectExt(long object, String resource_name, long ovalue)

Sets a new value of the object's attribute specified by the *resource_name*. It is used for attaching *Custom Property* objects and *aliases*.

BOOL SetSResourceExt(long object, String resource_name, String svalue)
BOOL SetSResourceIfExt(long object, String resource_name, String svalue, BOOL if_changed)
BOOL SetSTagExt(long object, String tag_source, String svalue, BOOL if_changed)

Set a new value of a string resource or tag. For tags, all tags with the specified *tag_source* will be set to the supplied value. Returns TRUE if the value of the resource or tag was successfully changed.

BOOL SetSResourceFromDExt(long object, String resource_name, String format, double dvalue)
BOOL SetSResourceFromDIfExt(long object, String resource_name, String format, double dvalue, BOOL if_changed)
BOOL SetSTagFromDExt(long object, String tag_source, String format, double dvalue, BOOL if_changed)

Set a new value of a string resource or tag to a string obtained by converting the supplied double value according to the format. The format parameter is a C format string. Returns TRUE if the value of the resource or tag was successfully changed.

void SetStartExt(long container)
void SetEndExt(long container)

Initializes the container for the forward or backward traversing, should be called before using *IterateExt* and *IterateBackExt* methods.

BOOL SetViewportExt(long viewport)

Sets the new drawing of the ActiveX control. The setting the drawing by using *SetViewportExt* method has the highest priority and overrides all other drawing properties.

BOOL SetXformExt(long object, long xform)

Sets the object transformation to a constrained copy of the xform parameter (Extended API only). If the xform parameter is *null*, removes any object transformations. The *CloneObjectExt* method may be used in conjunction with *SetXformExt* to add unconstrained copies of a transformation to several objects. Returns True if a transformation was successfully added.

void SetupHierarchyExt(long object)

Sets up the object hierarchy of the object without repainting the object.

long SuspendObjectExt(long object)

Suspends object for editing. It must be called before adding a transformation or constraining attributes of the object whose object hierarchy has been setup (the object has been drawn). The method returns an object ID of the suspend_info object, which is used in a consequent call to the *ReleaseObjectExt* method.

BOOL UnconstrainObjectExt(long object)

Unconstrains the attribute object. Returns TRUE if the object was successfully unconstrained.

void UpdateExt(long object)

Updates the viewport object to display the latest resource settings. If *object* is a light viewport, update of its parent viewport will be performed.

BOOL TraceObjectExt(long object, BOOL state, BOOL is_widget, long top_parent)

Searches all parents of the traced *object* and highlights or unhighlights them based on the specified conditions. The parent search is performed until any of the specified conditions (*is_widget* or *top_parent*) is satisfied. The objects will be highlighted if the *state* parameter is TRUE and unhighlighted otherwise.

If *is_widget=TRUE*, the parent objects that have resource named *IsWidget* will be highlighted or unhighlighted depending on the *state* parameter. If *top_parent* is non-zero, the parent objects that are direct children of *top_parent* will be highlighted or unhighlighted. If no conditions are specified, all immediate drawable parents of the traced object will be highlighted or unhighlighted.

The function returns TRUE if any parents were highlighted.

The dependent parent objects are highlighted with an outline. The attributes of the outline are defined by the *GlgHighlightPolygon* global configuration resource.

BOOL SetAttachmentMoveModeExt(BOOL state)

Changes the attachment point move mode and returns the previous mode state. By default, the attachment point move mode is off, and moving the attachment point will move the reference object it is attached to. If the mode is on, moving the attachment point will change its position relatively to the reference object.

BOOL EnableAttachmentPointsExt(BOOL state)

Specifies if attachment points should be used in addition to the control points to determine object extents when the objects are aligned. Returns the previous setting.

BOOL DeleteTags(long object, long tag_type_mask)

Deletes all *object*'s data tags or all object's public properties. Returns TRUE on success. *type_type_mask* specifies a bitwise mask that defines the type of tags to be deleted: data tags (GLG_DATA_TAG), public properties (GLG_EXPORT_TAG) or both.

GLG ActiveX Control Security

By default, the GLG ActiveX Control is allowed to save files on a local file system. The GLG_ACTIVEX_SECURE_MODE environment variable may be set to True to activate a Secure mode, which prevents the control from saving files.

In the secure mode, a user can enable saving files to a file system using predefined filenames by setting the GLG_ACTIVEX_SAVE_FILE, GLG_ACTIVEX_IMAGE_FILE and GLG_ACTIVEX_PRINT_FILE environment variables described later in this document. This variable define filenames used for saving drawings, images and the PostScript files.

GLG ActiveX Control Environment Variables

Several environment variables may be used to modify GLG ActiveX Control's behavior. This is optional and not required for a typical GLG ActiveX Control usage.

The environment variables may be set using the system's Control Panel.

GLG_ACTIVEX_SECURE_MODE

Turns on the *Secure* mode, which prohibits the ActiveX control from saving arbitrary files in the file system.

GLG_ACTIVEX_SAVE_FILE

Provides a filename to be used for the ActiveX Control's *SaveObjectExt* method. In the *Secure* mode, saving is allowed only if this environment variable is set, in which case the *filename* parameter of the *Save* method is ignored and the name defined by GLG_ACTIVEX_SAVE_FILE will be used.

GLG_ACTIVEX_IMAGE_FILE

Provides a filename to be used for the ActiveX Control's *SaveImage* methods. In the *Secure* mode, saving is allowed only if this environment variable is set, in which case the *filename* parameter of the *SaveImage* method is ignored and the name defined by GLG_ACTIVEX_IMAGE_FILE will be used.

GLG_ACTIVEX_PRINT_FILE

Provides a filename to be used for the ActiveX Control's *Print* methods. In the *Secure* mode, *Print* methods are enabled only if this variable is set, in which case it provides a name of a PostScript file to be generated.

Chapter 4

Using the Java and C# Versions of the Toolkit

4

Introduction

The Java and C# versions of the GLG Toolkit provide Java and C# class libraries for native Java and C# applications. The class libraries are very similar, with majority of the methods, fields and interfaces being the same for both Java and C#.

This chapter describes general use of the GLG Java and C# class libraries. **Online documentation** for the Java and C# class library APIs provides detailed description of each API method.

Numerous **demos and examples** for both Java and C# are also provided in the following directories of the GLG installation:

- *DEMOS_JAVA*
- *DEMOS_C#*
- *examples_java*
- *examples_csharp.NET*
- *examples_csharp.NET_core*

Additional examples of using GLG C# library in the VB.NET and CLI environment are also provided in the *examples_VB.NET* and *examples_CPP_CLI* directories.

Class Library Overview

The Java and C# versions of the GLG Toolkit provide Java and C# class libraries that load and display GLG drawings in Java and C# applications. Both class libraries support all GLG Toolkit features and are available in all forms of the GLG API: **Standard**, **Intermediate** and **Extended**.

The Java and C# versions of the Toolkit use GLG drawings saved with the GLG Graphics Builder. The drawings imported from the Builder must be saved in ASCII format, either compressed or uncompressed. The same drawing may be used in both Java, C# and C/C++ versions of the Toolkit. The drawing can also be generated on the fly using the Extended API, which provides methods to create and copy objects at run-time.

Class Hierarchy

GlgObject is the main class of the Java and C# versions of the GLG Toolkit. The *GlgObject* class contains all the functionality available to GLG objects. Since all objects implement the *Get* and *SetResource* functions central to the GLG architecture, it's only natural that the *GlgObject* superclass is heavily used, making it a Superclass in a more general sense. Using one main superclass also makes it easier to deal with GLG objects by limiting the number of classes and methods one has to learn.

The GLG Toolkit also contains classes for different GLG objects: polygons, arcs, etc. However, only the constructors of these classes are used. Once the object is created, all further object manipulation is done by using the *GlgObject* methods.

The GLG Toolkit also provides classes such as *GlgPoint, GlgCube* and some other utility classes, mainly used to exchange information between different object methods.

Integrated Components for Java and .NET

The Java class library provides the ***GlgJBean***, a light-weight Swing-based Java bean component that seamlessly integrates GLG Toolkit into a Java environment, embedding Java-based GLG components into Java application frameworks and Java IDEs.

The C# class library provides the ***GlgControl*** component which integrates GLG Toolkit in the .NET and Visual Studio environment, providing properties and events that can be used in the Visual Studio Design Mode.

Both GLG components (*GlgJBean* in Java or *GlgControl* in C#) encapsulate a GLG viewport object that loads, displays and animates a GLG drawing. The components provide several properties that facilitate loading the drawing: **DrawingFile**, **DrawingURL**, **DrawingName** and **DrawingObject**.

Both components implement a subset of the GLG API provided by the *GlgObject* class, so that the API methods may be invoked directly on the component instead of its viewport object.

Two Ways to use the GLG API

There are two alternatives for most of the GLG API methods: the methods of the *GlgObject* class, and the methods of the *GlgJBean* or *GlgControl* integrated component.

While the *GlgObject* methods have to be invoked for individual objects, the component provides a central place for invoking methods on any object or any resource inside the component's drawing. The methods exposed by the GLG component are similar to the methods of the *GlgObject* class, and decision which methods to use depends on the complexity of the application's interaction with the drawing displayed in the component:

- If an application loads a GLG drawing, initializes it by setting a few resources and starts updating it with real-time data, the methods of *GlgJBean* or *GlgControl* are sufficient.
- If an application needs to perform elaborate custom actions, traverse objects in the drawing to determine the content of the drawing, or to perform other actions that require obtaining IDs of objects in the drawing, it may use the *GlgJBean* or *GlgControl* to integrate a GLG drawing in the program and then use methods of the *GlgObject* class for all further processing. After loading the drawing, the viewport object (a *GlgObject*) of the loaded drawing may be extracted by calling the component's *GetViewport* method.

In the rest of this chapter, the *GlgJBean* and *GlgControl* integrated components may be referred generically using the term *component*.

Events and Input Handling

GLG integrated components (*GlgJBean* in Java and *GlgControl* in C#) provide two mechanisms for handing user input: **listeners** or **callbacks**. In the C# environment, the *GlgControl* component also supports C# events.

Listeners can be added to a GLG component to handle events of different types, such as input or object selection. A listener must implement an interface to handle events of the corresponding type. For example, *GlgInputListener* must implement the *InputCallback* method.

The following listeners are supported for *GlgJBean* and *GlgControl* components:

- **GlgInputListener is** used to handle user interaction and object selection.
- **GlgSelectListener** provides a simplified name-based interface for handling object selection.
- **GlgHListener** and **GlgVListener** may be used for initializing the drawing's initial appearance.
- **GlgHierarchyListener** is used for initializing drawings displayed in the *SubWindow* objects or instances of the *SubDrawing* objects.
- **GlgReadyListener** is invoked after the drawing has finished initializing and has been drawn for the first time; it is often used to start timers used for animation.
- **GlgTraceListener** and **GlgTrace2Listener** may be used to process native events not handled by the listeners listed above.

By default, the GLG Component (*GlgJBean* or *GlgControl*) is used as a default listener for all event types and implements callback methods required by the listener interfaces: *InputCallback*, *ReadyCallback* and so on. This makes it more convenient to create custom components by subclassing *GlgJBean* or *GlgControl* and overriding some or all of its listener interfaces. An application can also add listeners to the component (via the *AddListener* method) to handle events without subclassing the component.

In addition to the GlgJBean or GlgControl listeners, event listeners may also be added to individual child viewports displayed in the component's drawing using methods of the *GlgObject* class. This allows an application to provide a custom input handler for each child viewport displayed in the drawing if needed. *Input, Select, Trace* and *Hierarchy* listeners may be added to individual GLG objects, while *H, V* and *Ready* listeners are supported only for the GLG component, as they are not needed at the *GlgObject* level.

In the C#/.NET environment, GlgControl events are also implemented for the convenience of using the control in the VisualStudio's design mode. The supported .NET event types match the types of the event listeners: **GlgInput**, **GlgSelect**, **GlgH**, **GlgV**, **GlgHierarchy**, **GlgReady**, **GlgTrace** and **GlgTrace2**.

Generic API

While the most common way to display a GLG drawing in a Java or .NET application is using integrated *GlgJBean* and *GlgControl* components described above, an application can also display a GLG drawing using only the methods of the *GlgObject* class as shown below.

Java:

```
GlgObject drawing =
   GlgObject.LoadWidget( "my_drawing.g", GlgObject.FILE );
drawing.AddListener( GlgObject.INPUT_CB, my_input_handler );
drawing.InitialDraw();
```

C#:

```
GlgObject drawing = GlgObject.LoadWidget( "my_drawing.g",
            GlgMediumType.FILE );
drawing.AddListener( GlgCallbackType.INPUT_CB, my_input_handler );
drawing.InitialDraw();
```

The *InitialDraw* method in the above code creates a window for the top-level viewport of the drawing and displays the drawing in it. A similar *DialogInitialDraw* method is often used to display popup dialogs.

Enums

Enums are used in the **C# version** of the class library. The enums are defined at the top level of the *GenLogic* package namespace. Refer to the online documentation for a list of all enums of the *GenLogic* namespace.

In the **Java version**, integer constants named the same way as enum elements are used. The constants are defined at the *GlgObject* class level. For convenience, enums are grouped in the online documentation using group names that match the corresponding C# enums. Refer to the online documentation for a list of all enums of the *genlogic.com* package.

Interfaces

Interfaces are used for event listeners, as well as for custom error and alarm handlers, label and tooltip formatters, and utility methods. Event listeners may be added to the *GlgJbean* or *GlgControl* components, as well to the individual viewport objects.

Event listeners note: Only a single event listener for each event type is enabled per an instance of a GLG object or a GLG component. The GLG component is a default listener for all events, and adding a listener to a component disables the component's callback for the corresponding event. Multiple listeners can be added to different child viewports inside the component's drawing, so that each viewport can have it sown event listener if required.

Using Threads in Java and C#/.NET

Threads are commonly used in Java and C# applications for getting data from various sources over a network. Using threads for querying data allows an application to continuously stream incoming data independently from the user interface.

GLG Toolkit uses Swing in Java and .NET graphics framework in C#. These graphical frameworks are single-threaded and do not allow asynchronous access to graphics. All access to graphics should be done only from the event dispatching thread in Java and the UI thread in C#. Since GLG uses native interface components and graphics for displaying GLG drawings, **all calls to the GLG API methods must be performed from the event dispatching thread as well**.

This applies to all GLG calls, including *GetResource* and *SetResource* methods. These methods are not simple getters and setters, but rather complex methods that traverse the GLG object hierarchy to perform the requested operation. This traversal operation is not thread-safe.

If an application uses a thread-safe timer to obtain new data, it can update GLG drawings by invoking GLG *SetResource* and *Update* methods directly from the timer callback.

If an application uses data threads for querying new data, the thread code can not invoke *SetResource* and *Update* methods directly, and instead should pass the accumulated data to the event thread where the data will be used to update the graphics. **In Java**, this can be accomplished by using the *SwingUtilities.InvokeLater* method. **In C#/.NET**, the data thread can use the *GlgControl*'s *Invoke* method or the *InvokeRequired* property to execute *SetResource* and *Update* calls on the UI thread.

An alternative way to implement multi-threaded design would be have a data thread continuously accumulating data, and use a thread-safe timer to periodically (several times per second) check if new data from the data thread are available. If the new data came in, the timer code can issue *SetResource* calls for all new data values and then invoke the *Update* method to redraw the drawing just once for optimum performance.

With either design, it is the application's responsibility to properly synchronize access to all data structures if data threads are used.

Error Handling

The Toolkit provides the ***GlgErrorHandler*** interface for custom error handling. A custom error handler can be installed using the *SetErrorHandler* method. The *Error* method of the error handler will be invoked with the error message string as a parameter when a GLG error is detected, such as trying to access a *null* or non-existing resource.

The default error handler emits an audio beep and displays the error message in the Java console or in a message dialog for C#/.NET. The error message includes a stack trace showing the location where the error occurred.

In the C#/.NET version of the class library, the default error handler also logs the error message in the *glg_error.log* log file. The location of the log file is determined by the GLG_LOG_DIR and GLG_DIR environment variables as described in the *Error Processing and Debugging* section on page 50. An application can use the *GlgObject.Error* method to signal errors or log messages into the *glg_error.log* log file.

Anti-aliasing and Point Coordinates Precision

In the Java version of the GLG class library, a global *GlgAntiAliasing* configuration resource, described on page 364 of the *Appendices* chapter, controls default anti-aliasing setting for all drawings in an application. The anti-aliasing behavior of individual viewports can be controlled at run time by the viewport's *AntiAliasing* attribute. The *AntiAliasing* attribute of individual polygons is ignored.

In the Java environment, point coordinates are passed as integer values, and the only available settings of the *GlgAntiAliasing* global configuration resource are ANTI_ALIASING_OFF and ANTI_ALIASING_INT.

In the C# version, the anti-aliasing behavior of polygons is controlled on per-object basis depending on the setting of the object's *AntiAliasing* attribute.

In the .NET environment, point coordinates are passed as double values and the ANTI_ALIASING_DBL setting of the attribute is supported in addition to ANTI_ALIASING_OFF and ANTI_ALIASING_INT. If an object's *AntiAliasing* attribute is set to ANTI_ALIASING_UNSET (default, displayed as INHERIT in the Builder), the anti-aliasing setting is inherited from the viewport the object is displayed in, or from the global *GlgAntiAliasing* configuration resource.

Installable Interface Handlers API

GLG Installable Interface Handler Utilities assist an application developer with implementing functionality of complex user interactions usually encountered in editor-style applications. The utilities include a collection of methods as part of the GLG Intermediate API library. The *GLG Diagram* demo provides an example of a custom diagraming editor application implemented using the GLG Installable Interface Handlers functionality.

Refer to the GLG Java or C# online documentation for descriptions of the interface handler methods for the respective APIs.

Graph Layout

The GLG Graph Layout component is provided as part of the GLG Extended API and is used for automatic layout of graphs containing nodes connected with edges. It implements both the spring embedder and circular graph layouts algorithms.

The spring embedder algorithm performs a number of iterations optimizing the graph layout and minimizing the number of node intersections. The circular layout is used for positioning nodes that are connected in circular chains.

The GLG Graph Layout Demo is an example of an application that creates and automatically lays out a graph containing the connected nodes of a network. The demo's source code may be used as an example of using the Graph Layout module.

As seen in the demo, GLG Graph Layout may be used together with the GLG graphical engine, using GLG to display the graph and its nodes. In this case, the GLG Graphics Builder may be used to define a palette of graphical objects containing icons for the graph's nodes.

The Graph Layout may also be used to lay out non-GLG objects, for example, widgets or controls representing an application's objects. In this case, the application uses the Graph Layout's relative node position output (in the range of 0 to 1) to position an application's objects on the page.

GLG Graphics Server

The GLG Graphics Server is used for server-side deployment of the GLG Toolkit in the JSP environment in Java and in the ASP.NET environment in C#. It provides a way to **web-deploy existing Java or C# code for legacy applications**. This an alternative to a more common web deployment of the GLG applications using the GLG JavaScript Library.

GLG Graphics Server for JSP / Java

A separate GLG class library for the JSP environment is provided in the *GlgServer.jar* file. This class library provides the GLG Extended API, and also supports headless operation as well as a multi-threaded access by the servlet threads. When used in a servlet, the *GlgObject* classes should be used instead of the GLG Bean. The *examples_jsp* directory of the GLG installation contains elaborate examples of the GLG servlets with the source code.

GLG Graphics Server for ASP.NET / C#

The Glg.NETServer.dll file provides the GLG DLL for the ASP.NET environment. In addition to providing the GLG Extended API, this DLL provides the *GlgHttpRequestProcessor* class used to implement custom HTTP handlers in the ASP.NET environment. The DLL also supports headless operation and multi-threaded ASP.NET handlers. The *examples_ASP.NET* directory of the GLG installation contains elaborate source code examples of custom GLG-based ASP.NET HTTP handlers.

Class Library Files for Java and C#/.NET

Java Jar Files

The following JAR files are available for the Java version of the Toolkit:

GlgCE.jar - a free **Evaluation and Community Edition** version, which includes all GLG APIs: Standard, Intermediate and Extended.

Glg2.jar - a commercial version of the **Standard API**. It is used to display the drawing, update it with dynamic data and handle user input.

GlgInt2.jar - a commercial version of the **Intermediate API**. It includes the Standard API and extends it with methods to manipulate objects in the drawing at run time and implement elaborate logic for handling user input.

GlgExt2.jar - a commercial version of the **Extended API**, includes all methods of the Standard and Intermediate APIs and adds methods for creating and adding objects to the drawing at run time.

GlgGraphLayout.jar provides the graph layout functionality.

GlgDemo.jar contains the class files of the GLG demos.

The following GLG Graphics Server JAR files are provided:

GlgServerCE.jar - a free **Evaluation and Community Edition** version
GlgServer.jar - a commercial version of the GLG Graphics Server for JSP.

Both Graphic Server JARs include all GLG APIs: Standard, Intermediate and Extended.

`C#/.NET DLLs`

The following DLLs are available for the C#/.NET version of the Toolkit:

Glg.NetCE.dll - a free **Evaluation and Community Edition** version, includes all GLG APIs: Standard, Intermediate and Extended.
Glg.Net.dll - a commercial version with the **Standard API**. It is used to display the drawing, update it with dynamic data and handle user input.
Glg.NetInt.dll - a commercial version with the **Intermediate API**. It includes the Standard API and extends it with methods to manipulate objects in the drawing at run time and implement elaborate logic for handling user input.
Glg.NetExt.dll - a commercial version with the **Extended API**, includes all methods of the Standard and Intermediate API and adds methods for creating and adding objects to the drawing at run time.

Glg.Net.GraphLayout.dll provides the commercial version of the graph layout library, while *Glg.Net.GraphLayoutCE.dll* can be used with the free Community Edition.

The following GLG Graphics Server DLLs provide all GLG APIs (Standard, Intermediate and Extended):

Glg.NetServerCE.dll - a free **Evaluation and Community Edition** version
Glg.NetServer.dll - a commercial version of the GLG Graphics Server for ASP.NET.

JurassicGLG.dll provides the JavaScript interpreter used by the above DLLs.

Chapter 5

Using the JavaScript Version of the Toolkit

5

Introduction

The JavaScript version of the GLG Toolkit provides an HTML5 JavaScript library that is used to deploy a GLG application on a web page in a browser. The library provides the same complete GLG API as the GLG C/C++, Java and C# libraries used to develop desktop applications. With the JavaScript API, these applications can be ported to HTML5 and deployed in a web browser with the client-side JavaScript.

The supported browser include **Firefox**, **Chrome**, **Safari** and **Microsoft Edge**, including both the desktop and mobile versions.

Standard, Intermediate and Extended APIs

The JavaScript API provides three levels of a programming interface:

- **Standard API** provides methods for loading drawings created in the Graphics Builder or HMI Configurator, animating them with dynamic data, as well as handling basic user interaction (selecting objects with the mouse, pressing buttons, etc.)
- **Intermediate API** provides additional functionality for manipulating objects in the drawing, handling elaborate user interaction, object selection and processing commands defined in the drawing.
- **Extended API** extends the Intermediate API with methods for creating objects programmatically at run time. It makes it possible to add and delete objects, create new drawings at run time, add custom properties and commands to objects, as well as implement other functionality available in the Graphics Builder.

JavaScript Library

The **GLG JavaScript Library** is a JavaScript object engine that renders GLG objects in the Web environment via HTML5 and JavaScript. The Toolkit takes care of the low-level details of the rendering process, such as optimized damage repair, screen-resolution independent coordinate conversion, transparent zooming and panning, double-buffering and other coding-intensive driver-level tasks.

The Toolkit provides a high-level graphical object model which allows an application developer to concentrate on the application logic, speeding up the development process and dramatically increasing the developer's productivity.

The Toolkit contains the Graphics Builder, a graphical editor used to create graphical objects and define their attributes and dynamic properties without tedious programming. The resulting drawing is then saved into a file and loaded into a web application on a web page using a provided GLG JavaScript library.

JavaScript API

Overview

The GLG JavaScript API methods mimic the corresponding methods of Java and C# GLG APIs described in the previous chapter, with minor differences related to the JavaScript syntax. This allows to share not only the drawings but also the programming logic between the desktop and Web versions of an application.

The **GLG JavaScript Online Documentation** contains descriptions of all JavaScript API methods. It may be found on the GLG web site at the following location:

http://www.genlogic.com/doc_html/start.html#JavaScriptDoc

Even though the JavaScript is a dynamically typed language, this documentation uses "class" and "type" terms to describe GLG methods and their parameters. The *GlgObject* "class" is an object that provides methods that can be applied to an instance of *GlgObject* of any type, such as a polygon or a viewport.

The majority of the GLG methods are listed in the *GlgObject* section of the documentation.

A few utility "classes", such as *GlgPoint*, *GlgCube*, *GlgTraceData*, etc., are used to pass data to and from the GLG object engine.

There are also several helper classes, such as *GlgInputListener*, *GlgTraceListener*, etc., that define type signature of various callback functions.

Using JavaScript API Methods

The vast majority of the JavaScript API methods are the same as the corresponding Java or C# methods of the GLG API. The only exception are the methods that initialize the GLG API and load the GLG drawing. The JavaScript demos and examples provide source code examples of using these methods.

The JavaScript API methods use the same arguments as their corresponding Java or C# equivalents. For example, the *SetDResource(resource_name, value)* method expects a string as the first parameter and a double value as the second parameter.

For methods that expect integer parameters, such as *GetElement(element_index)*, it is an error to pass a numerical value that is not an integer, such as 3.5. Such errors will be detected if a debugging version of the library (described on page 325) is used during development. If a production version of the GLG library is used, the error may be undetected and cause errors later in the program.

Static methods are invoked on a *GLG* handle obtained via a call to the *GlgToolkit* method at the beginning of the program:

```
// Get a handle to the GLG Toolkit library.
var GLG = new GlgToolkit();

// Load GLG drawing.
GLG.LoadWidgetFromURL( "process.g", null, LoadCB, null );
```

Instance methods are invoked on the object instance:

```
viewport.SetGResource( "FillColor", 0.7, 0.7, 0.7 );
```

Comparing GLG Objects

Some methods, such as *GetResourceObject*, return an instance of *GlgObject*. The returned object is actually a wrapper around the internal representation of the object used by the GLG object engine. As a result, two different wrapper instances may actually refer to the same internal *GlgObject*.

To avoid incorrect results, the *Equals* instance method should be used instead of the == operator in the JavaScript code to check if two GLG objects are the same:

```
var same = obj1.Equals( obj2 );
```

A static *ObjectsEqual* method can also be used to compare two objects, in case *obj1* may be null:

```
GLG.ObjectsEqual( obj1, obj2 );
```

Predefined Constants

For methods that use predefined integer constants, the constants are grouped using an "enum" concept, with the "enum" name listed as a “type” in the online documentation. For example, the *GetObjectType* method returns an integer object type using constants defined in the *GlgObjectType* "enum", which is an object containing GLG object type constants.

Another example is a WRAPPED_TEXT constant that is defined in the *GlgTextType* “enum” and may be used to set *TextType* of a GLG text object:

```
text.SetDResource( "TextType, GLG.GlgTextType.WRAPPED_TEXT );
```

The WRAPPED_TEXT constant shown above should be used instead of its numerical value to provide the required integer value.

All "enums" are defined as properties of the global *GLG* handle object, which makes it easier to find constants using auto-completion.

The complete list of all enums can be found at the end of the *GlgToolkit*.js* files and can also be accessed online at the following link:

http://www.genlogic.com/doc_html/javascript_doc/namespacemembers_enum.html

Asynchronous Load

In JavaScript, all operations that load any file from a server, such as a drawing, data or any other resource, are asynchronous. As a result, the GLG methods that load files, such as *LoadDrawingFromURL*, take a callback parameter. The callback will be invoked synchronously when the loading request has been completed. A JavaScript application that loads any file from a web server should be aware of the asynchronous nature of the load requests and use an appropriate design.

A simplified *LoadAsset* method of the GLG API may be used by an application instead of handing HTTP requests in JavaScript to load any type of data the application may need.

Additional Programming Notes

1. The *GlgObject* instances received as parameters of a callback function, such as the *Input* or *Trace* callback, are valid only inside the callback and the **assignment operator (=) cannot be used to store them in global variables** to be used outside of the callback (the only exception are *GlgObject* instances received as parameters of the load callbacks). **The static *GetReference* method must be used to store *GlgObject* in a global variable** in a way that allows it to be used outside of the callback.

2. To minimize garbage collection, the *ReleaseToCache* instance method may be used to collect temporary objects, such as *GlgPoint* or *GlgCube*, after they have been used. *ReleaseToCache* may also be invoked on temporary *GlgObject* instances, in which case it will release to the cache the object's wrapper instance without affecting the internal GLG object the wrapper refers to.

Deploying GLG JavaScript Library

JavaScript Library Layout

The GLG JavaScript Library consists of two parts: the GLG object engine and the API bindings. A debugging version of the library described below is also provided. Two versions of the API bindings are provided: a script version and a modular version.

Depending on the flavor of the GLG API, several versions of the library are provided:

Standard API

Glg.js	- GLG object engine
GlgDebug.js	- debugging version of the GLG object engine
GlgToolkit.js	- script version of the API bindings
GlgToolkit.mod.js	- modular version of the API bindings
GlgToolkitDebug.mod.js	- modular debugging version of the API bindings

Intermediate API

GlgInt.js	- GLG object engine

GlgIntDebug.js - debugging version of the GLG object engine
GlgToolkitInt.js - script version of the API bindings
GlgToolkitInt.mod.js - modular version of the API bindings
GlgToolkitIntDebug.mod.js - modular debugging version of the API bindings

Extended API

GlgExt.js - GLG object engine
GlgExtDebug.js - debugging version of the GLG object engine
GlgToolkitExt.js - script version of the API bindings
GlgToolkitExt.mod.js - modular version of the API bindings
GlgToolkitExtDebug.mod.js - modular debugging version of the API bindings

Community Edition

GlgCE.js - GLG object engine
GlgCEDebug.js - debugging version of the GLG object engine
GlgCEDebugSafari.js - debugging version of the GLG object engine for the Safari browser
GlgToolkitCE.js - script version of the API bindings
GlgToolkitCE.mod.js - modular version of the API bindings
GlgToolkitCEDebug.mod.js - modular debugging version of the API bindings
GlgToolkitCEDebugSafari.mod.js - modular debugging version of the API bindings for the Safari browser

The libraries are located in the *lib* directory of the GLG installation.

GLG Demo Note

GLG demos in the *DEMOS_HTML* directory use the demo version of the GLG libraries:

```
GlgDemo.js
GlgToolkitDemo.mod.js
```

To use the source code of any demo as a template for the application development and evaluation, use *GlgToolkitCE.mod.js* in the demo and example script files instead of *GlgToolkitDemo.mod.js*.

To run the demos, place the demo files on a web server, since most browsers will not allow running JavaScript demos from a local file system due to the security restrictions.

Deploying JavaScript Library in HTML

The GLG JavaScript library is deployed in the form of two files: Glg<V>.js and GlgToolkit<V>.js, where <V> is replaced with the type of the GLG API as described above. The library also uses the gunzip.min.js file for uncompressing GLG drawings saved in the compressed format.

We will use the Standard API examples in the rest of this chapter.

Using Scripts

If a non-modular version of an application script is used with the non-modular version of the GLG JavaScript library, the following three script files must be included in the html file before the application script as follows:

```
<script defer src="Glg.js"></script>
<script defer src="GlgToolkit.js"></script>
<script defer src="gunzip.min.js"></script>
<script defer src="application_script.js"></script>
```

The *application_script.js* file provides custom application code. To use the debugging version of the GLG library, use *GlgDebug.js* instead of *Glg.js*.

Using Modules

If a modular version of the application script is used, the application script needs to include only the modular version of the *GlgToolkit.mod.js* file, which will automatically include *Glg.js* and *gunzip.min.js* files from the same location as the *GlgToolkit.mod.js* file, as shown in the GLG demos and examples.

To use the debugging version of the GLG library, use *GlgToolkitDebug.mod.js* instead of *GlgToolkit.mod.js*.

Windows IIS Web Server Notes

Custom *application/x-glg* MIME type should be added for the GLG drawing files with the *.g* extension. To add a MIME type:

1. Start IIS Manager
2. Select *MIME Types*
3. Right-click, *Add*
4. In the *Add MIME Type* dialog, specify MIME type "*application/x-glg*",
   File name extension "*.g*"
5. If the project uses subdrawings that were saved using an *.sd* extension, the extension would also need to be assigned the *application/x-glg* MIME type using the above steps. The *.sd* extension is filtered by the IIS by default, resulting in the “File Not Found” error for the subdrawings. To disable filtering of the *.sd* extension, a *web.config* file containing the following lines needs to be placed in the root directory of the project:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.webServer>
    <security>
      <requestFiltering>
        <fileExtensions allowUnlisted="true">
          <remove fileExtension=".sd" />
          <add fileExtension=".sd" allowed="true" />
        </fileExtensions>
      </requestFiltering>
    </security>
  </system.webServer>
</configuration>
```

A sample *web.config* file is provided in the *src* directory of the GLG installation.

Debugging Version of the GLG JavaScript Library

The debugging version of the JavaScript library file can be used for debugging purposes during application development. While it executes slower than the highly optimized production library, the debugging library contains various verification checks that help catch errors at the application development stage.

The debugging library performs type checking of all method parameters and generates an exception if an incorrect parameter type is detected. It is recommended that the debugging library is used for the development, and a production library is used only for the deployment, after thorough testing of all code branches.

The *ThrowExceptionOnError* method shown in the demos and examples code may be used to stop execution on the GLG errors: the stack trace in the browser's debugger will show the location of the code that caused the GLG error.

By default, GLG errors, such as "Can't find resource", generate an error message on the browser console but do not stop execution of the program. Throwing an exception on error can help find the exact code in the application script that generated the error.

NOTE: The *GlgCEDebugSafary.js* file provided with the Community Edition can be used for debugging with the Safari browser on macOS.

Global Configuration Resources

GLG global configuration resources may be used to control some aspects of the GLG Toolkit run-time behavior. For example, *GlgDisableMouseButtonCheck* and *GlgDisableControlKeyCheck* resources may be used to adjust application behavior for mobile devices with a touch screen, and *GlgWrapNativeLabels* may be used to control label wrapping in native buttons and toggles. The *GlgSetNativeAttributes* resource may be used to obtain a more complete control over the native widgets' appearance inside a GLG drawing.

Refer to the *Appendix A: Global Configuration Resources* chapter on page 357 for a list of the global configuration resources.

Using HTML Cascading Style Sheets

HTML attributes of native widgets inside a GLG drawing, such as buttons, toggles, text boxes, etc.) may be controlled by the HTML style sheets using each widget class type. The browser's Inspector feature may be used to find the class assigned to a native widget of each type.

The HTML attributes of native widgets may also be set in the JavaScript code by using the *GetNativeComponent* method of a native widget's viewport to get the ID of the widget's HTML element and then using this ID to set the element's style attributes.

Deploying GLG Drawings Created in Different System Locales

JavaScript Encodings

Internally, JavaScript uses UTF-16 encoding to represent strings. When external strings are loaded in JavaScript, they are converted from an external encoding to UTF-16.

JavaScript supports a limited number of external encodings for importing strings, such as ASCII, Latin1 and UTF-8, and does not support all possible system locales for individual languages. Instead, it uses UTF-8 encoding for importing strings in different languages.

GLG JavaScript API uses Latin1 encoding as a default, which allows loading drawings containing both ASCII and Latin1 characters without a need to convert them to UTF-8. Drawings containing characters other than Latin1 stored in non-UTF-8 encodings need to be converted to UTF-8 as described below.

Using GLG Drawings in JavaScript

ASCII Drawings

GLG Drawings that contain strings with only ASCII characters do not require any special handling, since these characters are the same in the ASCII, Latin1 and UTF-8 encodings.

Latin1 Drawings

GLG drawings created in Latin1 locale and containing strings with Latin1 characters do not require any special handling either, since GLG JavaScript API uses Latin1 as a default encoding.

Non-ASCII Drawings Created on Linux/Unix in UTF-8 Locale

If GLG drawings containing non-ASCII characters (either Latin1 or in any other language) were created in the UTF-8 locale on Linux/Unix, they can be loaded into a GLG JavaScript program by either setting the *GlgDefaultEncodingName* global configuration resource to "utf8" to define a default encoding, or by specifying "utf8" as the encoding parameter value of the GLG drawing loading methods, such as *LoadWidgetFromURL*.

Non-Latin1 Drawings Created on Windows or on Linux in non-UTF-8 Locale

GLG drawings created in Latin1 locale do not require any special handling as described above.

If a GLG drawing containing characters in any other language was created in any system locale other than UTF-8, the strings have to be converted to UTF-8 by re-encoding them in UTF-8 and setting their *UTF8Encoding* flag. In the Graphics Builder or HMI Configurator, it may be done on per-string basis by clicking on the *DEF* toggle, which will change to *UTF*.

The GLG editors (Graphics Builder and HMI Configurator) also provide options for converting all strings in the drawing to UTF8. The *Options, Convert Strings to UTF8 on Save* menu option may be checked to automatically convert all string attributes (such as *TextString* of a text object) from the current locale to UTF8 when the drawing is saved, setting the *UTF8Encoding* flag of each string data object to YES. The option can be permanently activated using the *ConvertToUTF8OnSave* variable in the glg configuration file (*glg_config* or *glg_hmi_config*).

The *Arrange, UTF8 Conversions* menu provides options for converting all strings in the drawing from the current locale to UTF8 and from UTF8 to the current locale without saving the drawing.

The GLG Drawing File Conversion Utility (*gconvert*) may be used to convert a drawing, or all drawings in a directory tree. The *-convert-to-utf8* command-line option of the *gconvert* utility is used to convert all strings in a drawing to the UTF-8 encoding. The *-r* converter option may be used to recursively convert all drawing files in a directory tree. Refer to the the *Drawing File Conversion Utility* chapter on page 350 for more information.

The converter should be run in the system locale in which the drawing was created. All strings in the drawing will be converted to UTF-8 and marked as UTF-8-encoded in the converted drawing by setting their *UTF8Encoding* flag.

The UTF-8 conversion makes the strings independent from the encoding specified in JavaScript. As a result, the converted drawing may be loaded in the GLG JavaScript program using the default (or any other) encoding settings. When a drawing is loaded by the GLG JavaScript library, the strings with *UTF8Encoding=YES* are recognized as UTF-8 encoded and will be properly handled.

The UTF-8 conversion is performed on string attributes that are objects, such as the *TextString* attribute of a text object. String attributes that are not objects, such as object names, tag names and tag sources, will not be converted.

Chapter 6

GLG Programming Tools and Utilities

6

The GLG Toolkit includes three utility programs of some use to GLG application programmers:

datagen
Generates a variety of test data suitable for testing and prototyping GLG drawings.

gcodegen
Creates a memory image of a GLG drawing in the C language format. The output of the utility is a file with *.c* extension, containing the image of the drawing in a form of the C source code. The output file can be compiled into the program's executable, saving users from having to supply a separate drawing file.

gconvert
A GLG drawing file may have one of three different formats: Binary, ASCII, or Extended. The Binary files are the fastest to load, but the Extended and ASCII files are more portable. The *gconvert* program converts drawing files from one format to another. The utility can also be used to change resources of a drawing in a batch mode using the script command option.

These tools are described in more detail in this chapter.

The GLG Toolkit also provides support for **scripting**. Scripting may be used for creating drawings using a script as well as editing drawings in the batch mode. Scripting is only one of the ways to create a GLG drawing. The GLG Extended API may also be used to generate a drawing programmatically and provide full access to all GLG functionality at run time.

While this usage is not common, scripting could also be used to supply data for prototyping a drawing in the Builder.

The **GLG Script** format as well as the command-line options for using the GLG Builder for scripting are also described in this chapter.

6.1 The Data Generation Utility

The data generation utility is included with the Toolkit and can be used to supply animation data for a GLG drawing during development. It can generate random or incremental data, take data for animation from a file or execute a script.

The *datagen* utility is embedded into the GLG Builder, which invokes the utility to generate data for prototyping the drawing in the Run mode using the *$datagen* command in the *Run* dialog. *$datagen* instructs the Builder to use an internal version of the datagen utility.

The *datagen* utility can also be used in stand-alone mode with the *-print* option to write animation commands into a script file which can be later used with the GLG Builder. The *datagen* utility is a symbolic link to the GLG Builder executable and can be invoked in the following way:

```
datagen <datagen options>
```

On Windows, links are not supported, and the following command must be used:

```
GlgBuilder -datagen <datagen options>
```

Command Line Arguments and Options

Datagen requires several parameters, including the name and type of the resource to be set and the low and high range for that resource.

```
datagen  <options>  [data_set]  [data_set...]
```

<options>

are optional parameters applicable to all data sets and may include:

-sleep *num_secs*

Specifies the number of seconds to sleep after each update

-pack *number*

Allows supplying a few data iterations for one update. The update is performed only once per specified number of the data values sent. This option also affects generation of string data. If the option is specified, and string data is requested, only one value of the string type data is generated for each pack.

-argf *argfile*

Allows some or all of the *datagen* arguments to be specified in the text file specified by this option. The arguments read from the file are placed at the position in the command line where this option is located. Several files may be chained using this option.

-script *scriptfile*

Specifies a file to use as an input script. The script may have commands for setting specific resources and updating the drawing. The script commands are listed in the *GLG Script* section on page 334. If this option is used, the *data_set* parameters are ignored and may be omitted.

-print *scriptfile*

Writes animation commands into an output script file. The script file can be read with the *-script* option.

-restore

This option sets the value of any named resource to zero, ignoring the range and data set options.

Data Set Specification

Several data sets may be specified. Each data set specification is composed of a collection of one or more data set options, three parameters defining the type and range of the desired data set, and the name of a resource in the drawing. The data set options and parameters are used to generate a stream of data, and that data is applied to the specified drawing resource

The data set specification looks like this:

```
<data_set_options>  type  low_range  high_range resource_name
```

The data set arguments are defined as follows:

type

Specifies the type of the data to be supplied. It should correspond to the type of the resource the data will be assigned, which is specified by *resource_name*. The type may be one of the following:

d - a double-precision scalar value

g - a "geometrical" value consisting of a set of three scalar values

s - a string

Unless the *-datafile* or *-script* option is used, the string data produced by *datagen* are numerical strings formed by converting a double-precision scalar value into a string using the C-style *%d* format. Therefore, the difference between *low_range* and *high_range* should be more than 1 to obtain strings different from the low range value. The geometrical data generated by *datagen* has identical x, y, and z coordinates.

low_range

Specifies the lower limit of the data to be generated.

high_range

Specifies the upper limit of the data to be generated.

resource_name

Specifies the name of the resource (or tag if *-tag* dataset option is used) to which the generated data values are assigned. The actual type of the resource or tag should match the type specified by the *type* parameter. The resource name or tag source must be defined relative to the viewport whose server receives the *datagen* message.

Data Set Options

The data set options are applied only to the data set specification in which they are found. If no data set options are specified, datagen produces random data between the data limits. The data set options are:

-tag

The datagen will handle the *resource_name* parameter as a tag source.

-lin

-incr

Generates linear data. The data start at the range lower limit and increase monotonically until the upper limit is reached. After the upper limit is reached, the generated data start again from the lower limit. The increment added to the data between iterations is determined by the total range and the defined data period. The default period is 100 iterations.

-sin

Generates data using a sine function. The data start at the range lower limit and vary sinusoidally between the range limits. The oscillation period is measured in data iterations, and can be controlled with the *-period* option. The default period is 100 iterations.

-cos

Generates data using a cosine function. The data start at the range higher limit and vary sinusoidally between the range limits. The oscillation period is measured in data iterations, and can be controlled with the *-period* option. The default period is 100 iterations.

-int

Truncates generated values to integers.

-period *number*

Specifies a period of the sinusoidal or incremental data for the *-sin, -cos* or *-incr* option. The supplied number defines the number of iterations that compose one period. The default period is 100 iterations.

-surf

Generates data for animating a surface graph. The number of rows and columns has to be specified with the *-row* and *-column* options. The *-pack* option may be used to update a graph just once when the complete surface is generated. In this case a pack number should be greater than or equal to the number of rows times the number of columns.

-row *number*

Specifies the number of points in a row on a polysurface. This is one greater than the number of polygon patches in the row.

-col *number*

Specifies the number of points in a column on a polysurface. This is one greater than the number of polygon patches in the column.

-datafile *data_file*

Specifies the name of a data file. If such a file is specified, the data is read from this data file instead of being generated. The *low_range* and *high_range* parameters are ignored but still must be included in the data set specification. The format of the data file is outlined below.

Data File Format

The data file is used when testing a drawing requires data not easily generated with the available *datagen* data set options. The file contains ASCII data corresponding to the type of the resource. For a scalar resource the file should be an array of numbers separated by spaces or new lines. A scalar data file might look like this:

```
1 1.1 1.2 1.3 1.4 1.5
1.6 1.7 1.8 1.9 2 2.1
2.2 2.3 2.4 2.5 2.6 2.7
...
```

For a geometrical resource the file should be an array of numbers as well, but the number of values in the array should be a factor of 3, and there must be an integral number of points per line of data.

```
1 1 0
1.1 1.1 -0.1
1.2 1.2 -0.2
1.3 1.3 -0.3
...
```

There may be more than one geometrical value per line. The following is also a valid way to specify the same geometrical series shown above:

```
1 1 0 1.1 1.1 -0.1 1.2 1.2 -0.2
1.3 1.3 -0.3 1.4 1.4 -0.4 1.5 1.5 -0.5
...
```

For a string resource, the array should have strings separated by the new line character:

```
October 1996
November 1996
Deember 1996
January 1997
February 1997
March 1997
April 1997
...
```

6.2 GLG Script

Overview

The *GLG Script* serves a variety of functions in the Toolkit:

- used with the *-script* option of the *datagen* Data Generation Utility to prototype a drawing in the GLG Graphics Builder with custom data.
- used with the *-command* and *-script* options of the *gconvert* File Conversion Utility to edit a collection of drawings using the batch mode or create a drawing using a script.
- may be used to supply data for animating a drawing in the Builder's Run mode.

GLG Script includes a small set of commands which can be used to set the value of some resource, update the drawing, synchronize the connection or produce a PostScript output. A script file is an ASCII file with a single command on each line. The script provides Database Record Support extension to simplify usage of database records to supply data for animation.

The script also provides an extended command set for creating and deleting objects. These commands may be used for creating drawings using a script, or for editing drawings using the batch mode of the GLG Builder. The extended command set is available only in the Enterprise Edition of the GLG Graphics Builder. To create drawings programmatically at run time, use the GLG Extended API, which provides this functionality and more.

The syntax of the script commands is described below.

Standard Command Set

<*comment*>

Specifies a comment line. Any line that starts with the "#" character is interpreted as a comment. The "#" character can be preceded with spaces and tabs.

set_value

Sets the value of a particular resource.

```
set_value <resource_name> <resource_type> <value(s)>
```

The command arguments define the name, type, and new value of the resource. This syntax is the same as the syntax for the dynamic resources, described in the *Setting the Drawing Resources* section of the *Using the C/C++ version of the Toolkit* chapter on page 31.

The strings used for setting values of resources of the *S* (string) type can be surrounded with double quotes, in which case they may contain spaces as well as the new line characters for multi-line strings. The '\' escape character may also be used to define simple escape sequences, such as "\n", "\r", "\t", "\b", "\"" and "\\". Use *"$null"* to set the new attribute value to *null*.

Using set_value with the Extended Command Set

The "." and "/" symbols may be used in *resource_name* to reference the *current object* and the *container* selected with the *select_object* or *select_container* commands of the extended command set, respectively (see page 338). For example, *./FillColor* may be used to access the *FillColor* attribute of the *current object*, and */Visibility* may be used to access the visibility of the selected *container*.

Registers may be used in script commands instead of resource names. For example, *%3/FillColor* may be used to access the *FillColor* attribute of the object stored in register 3. The content of a register is set using the *select_object* command of the extended command set described on page 338.

The *$last_value* symbol may be used instead of the value parameters, to refer to the value(s) obtained by the last *get_value* or *get_tag* command of the extended command set.

set_tag

Same as *set_value* but uses a tag_source.

```
set_tag <tag_source> <tag_type> <value(s)>
```

update

Updates the drawing to reflect the new values of resources.

```
update
```

For more information about the update action, see the *GlgUpdate* section of the *Animating a GLG Drawing with Data Using the Standard API* chapter.

print

Print the server's viewport by producing a PostScript output and saving it into a specified file:

```
print <filename> <x> <y> <width> <height> <portrait> <stretch>
```

The *x*, *y*, *width*, and *height* parameters define the PostScript page layout. The origin is defined to be in the middle of the page, the left edge of the page is at -1000, and the right edge is at 1000. The top and bottom of the page are similarly defined to be at 1000 and -1000, respectively. The *x* and *y* parameters define the lower left corner of the drawing, while *width*, and *height* give the dimensions of the drawing area. As an example, the default page layout you get when you print a drawing by setting the *PrintFile* property puts the lower left corner of the drawing area at (*-900 -900*), while the dimensions are *1800* by *1800*. This makes the drawing about as large as it can be while still keeping a small border all the way around the page. The *portrait* parameter defines portrait (TRUE) or landscape (FALSE) mode. If the *stretch* parameter is set to FALSE, the aspect ratio of the drawing is preserved.

The generated PostScript output corresponds to the currently visible state of the server's viewport after the last update command or expose event. The PostScript output is saved on the server's side, so the filename should be defined for the file system of the server host machine. If the filename is defined relative to the current directory, the current directory of the process is assumed.

Database Record Support Commands

Database record support is a feature of a GLG script that allows using the output of database report generators as input data for GLG ActiveX control or as prototyping data for the GLG Graphics Builder.

If a regular GLG script is used, each supplied data value needs resource name and type information. Using database record support feature, the information for associating the value with a resource may be supplied just once in a separate setup script and the data file will contain just the raw data generated by a database report generator.

The database record support consists of the following additional GLG script commands:

create_record

Creates a record with a named defined by the *record_name* parameter:

```
create_record record_name
```

add_field

Adds a field to a record defined by the *record_name* parameter:

```
add_field
   record_name resource_name resource_type <separator>
```

resource_name parameter specifies a resource to be associated with the field and the *resource_type* parameter supplies the GLG resource type - *d* for double resources, *s* for string resources, and *g* for points and colors defined by 3 coordinates. A dummy field with "*u*" as a separator value may be used to update the drawing in the middle of reading records, in which case the resource_name and resource_type fields are ignored.

The *separator* parameter has to be enclosed in angle brackets and supplies a character used to separate the next field. If a space is used as a separator, spaces and tabulation characters may be used as actual separators.

delete_record

Deletes a record defined by the *record_name* parameter.

```
delete_record record_name
```

read_records

Starts reading records from a script:

```
read_records record_name
```

The format of the records is defined by the *record_name* parameter. Each record should start on a new line and should not continue to the next line. A dummy field with "*u*" as a separator value may be used to update the drawing in the middle of reading records.

end_read

Stops the process of reading records defined by the last read_records command.

read_one_record

Reads one records from a script.

```
read_one_record record_name
```

The format of the records is defined by the *record_name* parameter.

Database Record Support Example

The following is an example of a GLG script that reads data records containing a value and a label and updates the drawing after reading each record. The script consists of two parts: a header that describes the record structure, and a list of records with data. The following show an example of a header:

```
create_record graph_record
add_field graph_record DataGroup/EntryPoint d <,>
add_field graph_record XLabelGroup/EntryPoint s < >
add_field graph_record dummy u <,>
```

The last field is a dummy field which causes display to be updated after reading each record without waiting for all of them to be read.

The following shows the remaining part of the script containing data that may be generated by a database report generator:

```
read_records graph_record
0.1,Label 1
0.2,Label 2
0.3,Label 3
0.4,Label 4
0.5,Label 5
0.6,Label 6
0.7,Label 7
0.8,Label 8
```

```
0.9,Label 9
end_read
```

Extended Command Set

The extended command set includes commands for creating and deleting objects, properties and tags using script. The extended command set is available only with the Enterprise Edition of the GLG Graphics Builder and can be used to modify multiple drawings with a script in a batch mode using the *-script* option of the Drawing File Conversion utility. The utility can be used to apply the script to multiple files, and can also be used to process all files in a directory and all its subdirectories using the recursive mode. Refer to the *Drawing File Conversion Utility* chapter on page 350 for more information.

The extended command set provides a subset of the Extended API functionality in the scripting mode, making it possible to perform elaborate drawing modifications without a need to write a program. Alternatively, the GLG Extended API can be used at run time to modify the drawings or create drawings programmatically.

The extended command set introduces the notion of a *current object* and a selected *container*. The *current object* and *container* may be selected using the *select_object* and *select_container* commands, and then used by other script commands, providing a way to access them efficiently.

The "." and "/" symbols may be used in resource names to reference the *current object* and the selected *container*, respectively. For example, *./FillColor* may be used to access the *FillColor* attribute of the *current object*, and */Visibility* may be used to access the visibility of the selected *container*.

The script engine also provides 10 registers that can be used to store object IDs. The registers are referred in the script using the *%n* notation, where *n* is a number from 0 to 9 that specifies the register's index. Registers may be used in script commands instead of resource names. For example, *%3/FillColor* may be used to access the *FillColor* attribute of the object stored in register 3. The content of a register is set using the *select_object* command.

The *$last_value* symbol may be used instead of the value parameters of the *set_value* and *set_tag* command to refer to the value(s) obtained by the last *get_value* or *get_tag* command of the extended command set.

Quotes can be used to define strings that contain spaces, such as values of the S resources and tag comments. The *$null* symbol may be used to define a NULL string.

The following list contains the commands of the extended command set:

skip_if_no_resource

If no resource is found, suspends processing of the script's commands until the *skip_end* command is read or the end of the script is reached, whichever comes first:

```
skip_if_no_resource <resource_name>
```

The *resource_name* parameter specifies the resource name to be tested. Nested skip commands are supported.

skip_if_resource

If the resource is found, suspends processing of the script's commands until the *skip_end* command is read or the end of the script is reached, whichever comes first:

```
skip_if_resource <resource_name>
```

The *resource_name* parameter specifies the resource name to be tested. Nested skip commands are supported.

skip_end

Resumes processing of the script's commands:

```
skip_end
```

get_value

Queries the value(s) of a particular resource and stores them for later use:

```
get_value <resource_name> <resource_type>
```

The command arguments define the name and type of the resource. Use the *$last_value* symbol instead of the value parameters of the *set_value* or *set_tag* command to reference the values returned by the last *get_value* or *get_tag* command.

get_tag

Same as *get_value* but uses a tag source:

```
get_tag <tag_source> <resource_type>
```

select_object

Selects an object for later access:

```
select_object <resource_name> [destination]
```

The *resource_name* parameter defines the resource name of the object to be selected. Use the "*$null"* string as the *resource_name* to unselect the object at the targeted destination.

A *destination* parameter can be used to specify one of the registers that can store selected objects. Registers are specified using the *%n* notation, where *n* is a register's index in the range from 0 to 9. An object stored in a register can be referenced in other script commands, for example:

```
# Store the $Widget viewport in register 0
select_object $Widget %0
```

```
# Set the FillColor of the object stored in register 0
set_value %0/FillColor g 1 1 1

# Store an object from register 0 in register 3
select_object %0 %3
```

The "." string may be used as a destination to store the object as the *current object*, which can be referenced in other script commands as ".":

```
# Select the $Widget viewport as the current object
select $Widget .

# Set the FillColor of the current object
set_value ./FillColor g 1 1 1

# Store the current object in register 2
select_object . %2
```

The destination parameter is optional. If it is committed, the *current object* is used as a destination.

Note: The setting of the *current object* is volatile: some commands, such as *create*, *add_custom_property*, *add_public_property*, etc., store the created object or the added property as the *current object* for a possible later access. To persistently store an object for referencing it later in the script use registers described above.

The *current object* is also used as an implicit argument for other script commands (such as *reference* or *drop*).

select_container

Selects a *container* for adding or deleting elements:

```
select_container <resource_name>
```

The *resource_name* argument defines the resource name of the object to be selected as the *container*. Use *$null* to unselect the currently selected *container*. The *container* is used as an implicit argument by commands that add or delete objects.

The selected *container* can be referenced in other script commands by using the "/" symbol:

```
# Set visibilty of the currently selected container to 1
set_value /Visibility d 1
```

select_element

Selects an element of a container object, such as a group or a viewport, or selects a control point of a polygon:

```
select_element <resource_name> <element_index> [<destination>]
```

The command arguments define the resource name of the container object and the index of the element inside the container to select. The *destination* parameter can specify a register to store the selected element using the *%n* notation, where *n* is a register's index in the range from 0 to 9. If the "." string is used as the destination parameter, or the parameter is omitted, the selected element is stored as the *current object*.

The stored element can then be accessed using the *"."* or *"%n"* notation, depending on where it was stored.

load_object

Loads an object from a file and selects it:

```
load_object <filename>
```

The loaded object is stored as the *current object* and can be accessed using the "." symbol.

set_resource_object

ADVANCED: Sets a new value of the specified resource of the selected object:

```
set_resource_object <resource_name> <resource_name_of_new_value>
```

The *resource_name* parameter defines the name of the selected object's attribute. The *resource_name_of_new_value* parameter specifies the object to be used as a new value of the attribute. Use *"$null"* to set the new attribute value to *null*.

reference

Increases the reference count of the *current object*:

```
reference
```

This command uses the *current object* set by the *select_object* command as an implicit argument. If the *current object* is not set, an error is generated.

drop

Decreases the reference count of the *current object*:

```
drop
```

This command uses the *current object* set by the *select_object* command as an implicit argument. If the *current object* is not set, an error is generated.

add_alias

Adds a data tag to an object attribute:

```
add_data_tag <target> <from_name> <to_name>
```

The *target* parameter specifies the resource path of the object the alias is added to. The *from_name* and *to_name* parameters provide the corresponding attributes of the alias.

The added alias is stored as the *current object* for a possible later use.

add_custom_property

Adds a custom property to an object:

```
add_custom_propery <target> <property_name> <type> <value>
```

The *target* parameter specifies the resource path of the object the custom property is added to.

The *property_name*, *type* (*d, s* or *g*) and *value* parameters provide the corresponding attributes of the custom property. For custom properties of D and S types, the value parameter provides a double or string value, respectively. For custom properties of the G type, the value is supplied as a triplet of double values.

The added custom property is stored as the *current object* for a possible later use.

Refer to the *Drawing File Conversion Utility* chapter on page 350 for description of command-line options that control verbosity of the diagnostic output.

add_public_property

Adds a public property to an object attribute:

```
add_public_propery <target> <property_name> [<comment>]
```

The *target* parameter specifies the resource path of the object attribute the public property is added to.

The *property_name* and an optional *comment* parameters provide the corresponding attributes of the public property.

The added export tag is stored as the *current object* for a possible later use.

Note: The *-setup* command-line option of the utility must be used to set up the drawing hierarchy, which is needed to process this command in case when the target is specified using a named resource instead of a default resource name. Refer to the *Drawing File Conversion Utility* chapter on page 350 for description of this and other command-line options that control verbosity of the diagnostic output.

add_export_tag

Same as *add_public_property* but also allows specifying an export tag type:

```
add_export_tag <target> <tag_name> <type> [<comment>]
```

The *type* parameter specifies the export tag type and may be set to the following string values: *EXPORT*, *EXPORT_DYN* or *EXPORT_BOTH*. Refer to the *Drawing File Conversion Utility* chapter on page 350 for description of command-line options that control verbosity of the diagnostic output.

The added export tag is stored as the *current object* for a possible later use.

add_data_tag

Adds a data tag to an object attribute:

```
add_data_tag <target> <tag_name> <tag_source> <access_type>
             [<comment>]
```

The *target* parameter specifies the resource path of the object attribute the data tag is added to. The *tag_name*, *tag_source* and an optional *comment* parameters provide the corresponding attributes of the public property. The *access_type* parameter specifies the tag's access type and may be set to the following values: 0 for INPUT tags, 1 for INIT tags and 2 for OUTPUT tags.

The $null string can be used to specify NULL values for string parameters, such as *tag_name*, *tag_source* and *comment*. Refer to the *Drawing File Conversion Utility* chapter on page 350 for description of command-line options that control verbosity of the diagnostic output.

The added data tag is stored as the *current object* for a possible later use.

get_export_tag

Finds a named custom property of an object:

```
get_export_tag <target> <property_name>
```

The *target* parameter specifies the resource path of the object to search for, and *property_name* specifies the name of the custom property. If found, the custom property is stored as the *current object* accessible via the "." symbol. If a named custom property is not found, NULL is stored.

delete_custom_property

Deletes a custom property from an object:

```
delete_custom_property <target> <property_name>
```

The *target* parameter specifies the resource path of the object to delete the custom property from, and *property_name* specifies the name of the custom property to be deleted.

Refer to the *Drawing File Conversion Utility* chapter on page 350 for description of command-line options that control verbosity of the diagnostic output.

delete_public_property
delete_export_tag

Deletes a public property or an export tag from an object's attribute:

```
delete_public_property <target>

delete_export_tag <target>
```

The *target* parameter specifies the resource path of the object attribute to delete the public property or the export tag from.

Refer to the *Drawing File Conversion Utility* chapter on page 350 for description of command-line options that control verbosity of the diagnostic output.

delete_data_tag

Deletes a data tag from an object's attribute:

```
delete_data_tag <target>
```

The *target* parameter specifies the resource path of the object attribute to delete the data tag from.

Refer to the *Drawing File Conversion Utility* chapter on page 350 for description of command-line options that control verbosity of the diagnostic output.

create

Creates an object of the specified type, and stores it as the *current object* for later use:

```
create <object_type> <name> <parameters>
```

The command arguments define the type of object and its name, as well as any parameters required by some object types. The rest of the attribute values may be set using the script's *set_value* command after the object has been created. The following lists the supported object types and parameters:

polygon

Parameters:

```
<num_points> x1 y1 z1 x2 y2 z2 ... xn yn zn
```

Creates a polygon. Requires the number of points and a list of vertex values for polygon points. The rest of the polygon's attributes may be set using the *set_value* command.

parallelogram

Parameters:

```
x1 y1 z1 x2 y2 z2 x3 y3 z3
```

Creates a parallelogram. Requires the list of values of the parallelogram's three vertices. The rest of the parallelogram's attributes may be set using the *set_value* command.

rectangle

Parameters:

```
x y width height
```

Creates a rectangle; requires the rectangle's origin, width and height parameters. The rest of the rectangle's attributes may be set using the *set_value* command.

text

Parameters:

```
<text_sring>
```

Creates a text object. Requires a text string. The position and other attributes of the text object may be set using the *set_value* command.

image

Parameters:

```
<image_file>
```

Creates an image object. Requires the name of an image file. The position and other attributes of the image object may be set using the *set_value* command.

arc

Parameters: None

Creates an arc. The arc's parameters may be set using the *set_value* command.

rounded

Parameters:

```
x1 y1 z1 x2 y2 z2 x3 y3 z3
```

Creates a rounded rectangle (or ellipse, depending on the values of the object's attributes). Requires a list of values for the object's three vertices. The rest of the attributes may be set using the *set_value* command.

marker

Parameters: None

Creates a marker. The marker's position and other attributes may be set using the *set_value* command.

viewport

Parameters: None

Creates a viewport. The viewport's position and the rest of its attributes may be set using the *set_value* command. The viewport may be selected as a container in order to add other objects to it.

group

Parameters: None

Creates a group. The group may be selected as a container in order to add other objects to it.

gis

Parameters:

```
<image_file>
```

Creates a GIS Object. Requires the name of a data file. The position and other attributes of the GIS Object may be set using the *set_value* command.

rendering_attr

Parameters: None

Creates a rendering object containing extended rendering attributes, such as gradient. The rendering may be added to an object by using the *set_resource_object* script command.

box_attr

Parameters: None

Creates a text box attributes object. The text box may be added to a text object by using the *set_resource_object* script command.

The created object is stored as the *current object* that can be accessed using the "." symbol.

add_new

Same as *create* but also adds the created object to the selected *container*:

```
add_new <object_type> <name> <options>
```

This command has the same arguments as *create*; see description of the *create* command for details. The selected *container* is used as an implicit argument and must be set externally using the *select_container* command, otherwise an error is generated.

add_copy

Creates a strong copy of the *current object*, names it, adds it to the selected *container* and stores the copy as the *current object*:

```
add_copy <copy_name>
```

The command argument define the name of the copy. Both the *current object* and the selected *container* are used as implicit arguments and must be set, otherwise an error is generated.

The command may be used repeatedly to add a number of copies, so that *add_copy* may be used instead of *create* in cases when a number of objects with similar attributes have to be created. The attributes of individual copies may be set using the *set_value* command that uses the "." symbol (*current object*) in the *resource_name* parameter to reference the newly added copy.

The attributes of the copies may be constrained by setting the attribute's *Global* flag to GLOBAL before copying the object.

For example, the following script may be used to add 3 text labels to the drawing:

```
# Select the $Widget viewport as a container to add the labels to
select_container $Widget

# Create a text object and set its font type and size.
# Creating the text selects it for use in add_copy.
create text Label
set_value ./FontType 2
set_value ./FontSize 2
# Set TextType to FIXED
set_value ./TextType 1

# Add three copies of the text setting their position and label.
# Adding a copy stores it as the current object, accessed as "." later.
add_copy Label1
set_value ./String This is Label1
set_value Point -500 500 0

add_copy Label2
set_value ./String This is Label2
set_value Point -500 0 0

add_copy Label3
set_value ./String This is Label3
set_value Point -500 -500 0
```

copy

Creates a copy of an object using a specified clone type and stores the copy as the *current object*:

```
copy <clone_type> <resource_name>
```

The *resource_name* argument defines an object to copy. The *clone_type* parameter may have the following values:

f - full clone
w - weak clone
s - strong clone
c - constrained clone

Use the "."symbol (*current object*) in the *resource_name* parameter of other commands to reference the cloned object.

delete

Deletes the *current object* from the selected *container*:

```
delete
```

The deleted object is still stored as the *current object* and may be added to another container. Both the *current object* and selected *container* are used as implicit arguments and must be set, otherwise an error is generated.

constrain_object

ADVANCED: Constraints one object's attribute to another:

```
constrain_object <from_resource_name> <to_resource_name>
```

The *from_resource_name* parameter specifies the attribute to be constrained and must use default attribute name to reference the attribute. The *to_resource_name* parameter specifies the attribute to constrain to.

6.3 Code Generation Utility

The GLG Code Generation Utility is used to convert a drawing file in one of the GLG file formats into plain C code. The generated code represents a memory image of the drawing, which can be later compiled and linked with the program in order to have all necessary drawings included in the executable. This increases the size of the executable file but makes it unnecessary to supply additional files with an application.

The generated C code takes the form of an array, which is used by the *GlgLoadObjectFromImage* function. For more information about this function, see the *GlgLoadObjectFromImage* section of the the *GLG Intermediate and Extended API* chapter.

The Code Generation Utility is called *gcodegen* and requires three command line arguments:

```
gcodegen  in_file  out_file  variable
```

The arguments are defined as follows:

in_file

Specifies the name of the drawing file. This file may exist in any of the three GLG file formats (ASCII, binary, or extended).

out_file

Specifies the name of the file containing the generated code.

variable

Specifies the name of the program variable to be used for the array with produced data.

For example, the following command:

```
gcodegen bar1.g bar1.c bar1_data
```

produces a file called *bar1.c* containing the image of the *bar1.g* drawing in the following format:

```
/* Generated by the Generic Logic Toolkit */
unsigned long bar1_data[] =
            { <...data for the memory image are placed here...> };
long bar1_dataSize = sizeof( bar1_data );
```

The address of *bar1_data* array and the value of the *bar1_dataSize* variable are required to load a drawing from an image with the *GlgLoadObjectFromImage* function.

Code generation does not change the saved format of the drawing, so you must make sure the drawing file is in the correct format before running *gcodegen*. If the drawing was saved in the ASCII or EXTENDED format, the generated memory image is in that format, increasing the size of the produced code and making loading the drawing from the resulting image slower. Since the version of the linked memory image is guaranteed to match the version of the library after it has been linked correctly, it is safe to use the BINARY format, which makes loading drawings faster.

6.4 Drawing File Conversion Utility

The GLG Drawing File Format Conversion Utility is supplied with the Toolkit and can be used to convert a drawing file into a different saved format. It may also be used for setting resource values of GLG drawings in a batch mode using scripting commands.

The utility is a symbolic link to the GLG Builder executable and may be invoked in the following way:

```
gconvert <conversion utility options>
```

On Windows, symbolic links are not supported and the following command must be used:

```
GlgBuilder -convert <conversion utility options>
```

A GLG drawing can be saved in one of three different formats:

BINARY format is the fastest format to use for loading drawing files but not compatible between hardware platforms with different binary data representations.

ASCII format is slower to load than BINARY and is compatible between different hardware platforms but is not compatible between different versions of the Toolkit.

EXTENDED format is the slowest format to load but provides the ability to transfer drawing files between different versions of the Toolkit as well as between different hardware platforms.

In addition to converting the drawing file format, the utility can be used to export or import strings and tags, modify drawings using the supplied script, adjust font sizes, as well as change other drawing settings.

On Linux/Unix, error and information messages are displayed in the terminal. On Windows, the messages are stored in the *glg_error.log* file in the GLG installation directory. The GLG_LOG_DIR environment variable may be set locally to define a different directory for the generated *glg_error.log* file.

The utility is invoked in the following way:

On Linux/Unix: `gconvert <options> file [file...]`

On Windows: `GlgBuilder -convert <options> file [file...]`

The *file* argument specifies the name of the drawing file to be converted. By default, converted files are saved under their original names, overwriting the original files. The available options are as follows:

-a

Converts file into the ASCII format. This is the default if none of the conversion options (*-a*, *-b*, or *-e*) are specified.

-b

Converts file into the BINARY format.

-e

Convert file into the EXTENDED format, is available with the Enterprise Edition of the Graphics Builder.

-c

Compresses the converted drawing. This is the default if none of the compression options (*-c* or *-u*) are specified.

-u

Saves the converted drawing uncompressed.

-v

Prints the save format version of the converter and the specified drawings.

-r

If this option is specified, and the *file* argument is a directory, *gconvert* converts all the files in that directory, recursively descending into all subdirectories.

-p *pattern*

A regular expression pattern that can be used with the *-r* option for processing multiple files in the directory and all its subdirectories: only files with pathnames matching the pattern will be processed. The pattern can contain *?* and * wildcard characters for matching a single or multiple characters in a filename. On Linux/Unix, the special characters must be either escaped using the backslash character (\), or the pattern string should be surrounded with quotes.

Pattern matching is performed on a complete filename path, not just on the filename. For example, while the **.g* pattern will match files with the *.g* extension in any directory, the following pattern should be used to match files in any subdirectory with filenames starting with *button* and ending with the *.g* extension:

```
"*/button*.g"          (Linux/Unix)
*\button*.g            (Windows)
```

However, using the *button*.g* pattern instead of the above examples would not select any files, since it would not match a complete filename path that starts with the directory path.

The following pattern could be used to process button files in all subdirectories whose name starts with *controls* (i.e. *controls1, controls2, controls_misc*, etc.):

```
"*/controls*/button\*.g"          (Linux/Unix)
*\controls*\button*.g             (Windows)
```

-verbose-pattern-match
-vpm

Generates diagnostic information about files skipped due to a pattern mismatch.

-quiet

Suppresses information messages about processed files.

-o *out_file*

Defines the name of the output file. When this option is omitted, the converted file replaces the original file. This option is ignored if more than one input file is specified or if the *-r* option is

used.

-export-strings *strings_file*

Instead of converting the drawing file to a new format, save all text strings defined in the drawing into the specified strings file.

-import-strings *strings_file*

In addition to converting the drawing file, replace its strings with the corresponding strings from the specified string translation file.

-export-tags *tags_file*

Instead of converting the drawing file to a new format, save all tags defined in the drawing into the specified tags file.

-import-tags *tags_file*

In addition to converting the drawing file, replace tags defined in the drawing with the corresponding tags from the specified tags file.

-multiple-export-import
-mei

When this option is specified, the supplied *string_file* and *tag_file* parameters of the strings and tags export/import options are interpreted as suffixes to be added to the input filename to form the filename of the corresponding strings or tags file for each of the processed files. In the absence of this option, the *strings_file* and *tags_file* parameters are treated literally, and the same strings or tags file is used for all input files.

-convert-viewports

Converts all viewports in the drawing to light viewports, except for viewports with native widgets and dialogs (viewport with non-default settings of *WidgetType* or *ShellType* attribute).

-convert-native-viewports

Forces conversion of viewports with native widgets.

-skip-widget-viewport

Skips viewport conversion for viewports named *$Widget*.

-convert-light-viewports

Converts all light viewports in the drawing to viewports.

-command *"scripting_command"*

Defines the *set_value* or *set_tag* command in the *GLG Script Format* to be executed on each converted drawing, and is used to change a resource value of the converted drawing. Refer to the *GLG Script* chapter on page 334 for more information.

-script *script_file*

Defines the name of a script file in the *GLG Script Format* containing script commands to be executed on each converted drawing. It is used to change resource values of the converted drawing, as well as add or delete objects and properties from the converted file using advanced script commands. Refer to the *GLG Script* chapter on page 334 for more information.

-print-commands
-pc

Used with the -*script* option to print script commands as they are executed to assist debugging of the script errors.

-new

Used with the -*script* option in cases when the file doesn't exist and the drawing will be created by the script. The new drawing will be saved into a file specified by the -*o* option.

-setup

Used with the -*script* option to set up the hierarchy of the loaded drawing before executing the script. This may be required to process the *add_public_property* and *add_export_tag* script commands.

-skip-missing-targets
-smt

Used with the -*script* option to skip missing targets of the commands that add or delete properties or tags. In the absence of the option, error messages are generated for missing targets of these commands.

-skip-duplicate-properties
-sdp

Used with the -*script* option to skip duplicate properties or tags in the commands that add properties or tags. In the absence of the option, error messages are generated when trying to add duplicate properties or tags.

-skip-missing-properties
-smp

Used with the -*script* option to skip missing properties or tags in the commands that delete properties or tags. In the absence of the option, error messages are generated when trying to delete non-existent properties or tags.

-verbose-skip
-vs

Used with the -*script* option to provide diagnostic output about skipped commands without generating an error.

-font-size *int_value*

Sets the font size of all text objects to the supplied value.

-font-size-change *int_value*

Used to adjust the size of all text objects in the drawing by a specified integer value.

-min-font-size *int_value*

Minimum font size for scalable text objects.

-max-font-size *int_value*

Maximum font size for scalable text objects.

-text-scaling *int_value*

Text scaling type for the text objects, uses values of the *GlgScalingType* enum in *GlgApi.h*.

-set-utf8

Sets the *UTF8Encoding* flag of all S data objects in the drawing without re-encoding their strings.

-reset-utf8

Resets the *UTF8Encoding* flag of all S data objects in the drawing without re-encoding their strings.

-convert-to-utf8

Re-encodes strings of all S data objects in the drawing with an unset *UTF8Encoding* flag from the current locale's encoding to UTF8 and sets the flag.

-convert-from-utf8

Re-encodes strings of all S data objects in the drawing with an unset *UTF8Encoding* flag from UTF8 to the current locale and resets the flag.

-convert-all-from-utf8

Re-encodes strings of all S data objects in the drawing (regardless of the setting of the *UTF8Encoding* flag) from UTF8 to the current locale and resets the flag.

-fix-flat-arcs

Converts flat arcs into a form that will work with older GLG releases (version 4.1 and older) that didn't support flat arcs.

-palette-file *filename.pal*

Converts drawings listed in the specified GLG palette file.

For example,

```
gconvert *.g
```

converts all the GLG drawing files in the current directory (indicated with the "*.g*" suffix) into the ASCII format. The following command:

```
gconvert -b -r directory1
```

converts all the GLG files in the directory named "*directory1*" into the BINARY save format. All the GLG drawing files in any subdirectory will also be converted. The following command:

```
gconvert -o file2.g file1.g
```

converts the GLG drawing file named *file1.g* into the ASCII format and saves the new file under name *file2.g*.

The following command:

```
gconvert -command "set_value \$Widget/FillColor g 0. 0. 1." drawing.g
```

is an example of using the utility for setting resources of a drawing.

6.5 C/C++ Graph Layout Component

The GLG Graph Layout component is provided as part of the GLG Extended API and is used for automatic layout of graphs containing nodes connected with edges. It implements both the spring embedder and circular graph layouts algorithms.

The spring embedder algorithm performs a number of iterations optimizing the graph layout and minimizing the number of node intersections. The circular layout is used for positioning nodes that are connected in circular chains.

The GLG Graph Layout Demo is an example of an application that creates and automatically lays out a graph containing the connected nodes of a network. The demo's source code may be used as an example of using the Graph Layout module.

As seen in the demo, GLG Graph Layout may be used together with the GLG graphical engine, using GLG to display the graph and its nodes. In this case, the GLG Graphics Builder may be used to define a palette of graphical objects containing icons for the graph's nodes.

The Graph Layout may also be used to lay out non-GLG objects, for example, widgets or controls representing an application's objects. In this case, the application uses the Graph Layout's relative node position output (in the range of 0 to 1) to position an application's objects on the page.

Refer to the **online documentation** for detailed description of the graph layout functions.

Appendices

A

7.1 Appendix A: Global Configuration Resources

There are a small number of drawing features that can't be said to belong to any particular object in a GLG drawing. When a drawing is displayed in the GLG Graphics Builder, those drawing features, such as the file save format, are managed by the Builder. When a drawing is displayed in an application program, that program must assume the responsibility of managing those features. Rather than include more functions in the GLG API, the Toolkit provides **configuration resources** to control these features. These resources can be queried and modified with the same resource access functions you use to control any other feature of a drawing.

The path name of a configuration resource starts with *$config* followed by the configuration resource name. For example, *$config/GlgSaveFormat* is the name of the configuration parameter that controls the default save format. When configuration parameters are accessed in C, the object parameter of functions that query or set resource values may be NULL. In the object-oriented environment, any GLG object can be used to access global parameters.

The *glg_config* configuration file may be used to set global configuration resources for the GLG Graphics Builder, and the *glg_hmi_config* file may be used for HMI Configurator. At run time, the GLG API may be used to query or set these global resources in a program.

The configuration resources, their types and possible values, are listed in the following table. The table also lists environment variables which can be used as an alternative way of setting some global configuration resources for quick testing, or for the cases when the application code can not be accessed. The environment variables settings do not apply to the Java, C#/.NET and JavaScript versions of the Toolkit.

GLG Toolkit Configuration Resources

Name	Type	Default	Description (possible values)
GlgGridPolygon	polygon object	solid line, one pixel wide, adaptive color	Defines grid attributes such as a line width and a color (the grid spacing is controlled by an attribute of the viewport screen object). The polygon is not accessed directly, but its attributes are, for example: *$config/GlgGridPolygon/LineWidth* By default, the polygon's *EdgeColor* is set to (-1,-1, -1) that activates adaptive color mode controlled by the color scale transformation attached to the attribute. Setting *EdgeColor* to a valid RGB value will disable the adaptive mode and will use the specified grid color.
GlgArrowShape	polygon object	default arrow shape as seen in the Graphics Builder	3-point polygon defining the pixel dimensions and the shape of the arrowheads drawn by the rendering object. To change the arrowheads, change the coordinates of the polygon's points.

Name	Type	Default	Description (possible values)
GlgHighlightPolygon	polygon object	thin red line as seen in the Graphics Builder	Defines the line attributes of the outline used to highlight traced objects.
GlgTooltipErase-Distance	Double	5	Specifies the maximum distance in pixels the mouse can move without erasing the currently displayed tooltip. It is used to avoid erasing the tooltip on accidental small mouse movements. If a mouse moves over another object, the tooltip for the new object will be activated only when the current tooltip is erased due to the mouse move exceeding the tooltip erase distance. For environments other than Java, C# or JavaScript, the GLG_TOOLTIP_ERASE_DISTANCE environment variable may also be used to specify the erase distance.
GlgTooltipLabelColor	G - holds 3 RGB color values	(0;0;0) on Linux/Unix and in JavaScript, unset (-1;-1;-1) on Windows and in Java/C#	Specifies the tooltip label color. If unset, uses the default color inherited from the environment.
GlgTooltipBGColor	G - holds 3 RGB color values	(1;1;1) on Linux/Unix, (1;1;0.9) in JavaScript, unset (-1;-1;-1) on Windows and in Java/C#	Specifies the background color of the tooltip. If unset, uses the default color inherited from the environment.
GlgTooltipText-Alignment	String	*"left"*	Specifies the row alignment of a multi-line tooltip: *"left", "right"* or *"center"*. For environments other than Java, C# or JavaScript, the GLG_TOOLTIP_TEXT_ALIGNMENT environment variable can also be used and set to one of the following strings: LEFT, RIGHT or CENTER.
GlgTooltipFormatter-OnErase	Double	0	Controls behavior of the custom tooltip formatter. If set to 0, the formatter will be invoked before the tooltip is displayed. If set to 1, the formatter will also be invoked when the tooltip is erased. For environments other than Java, C# or JavaScript, the GLG_TOOLTIP_FORMATTER_ON_ERASE environment variable may also be used to specify the formatter behavior.

Name	Type	Default	Description (possible values)
GlgNativeTooltip (Java and C# only)	Double	GLG_TOOLTIP	Controls the type of the tooltips in the Java and C#/.NET environments; may be set to the following values: • GLG_TOOLTIP - uses GLG tooltips. In Java, this setting should be used to properly handle dynamically changing tooltips for buttons with the *GlgButton* and *GlgNButton* interaction handlers. • BOX_TOOLTIP - enables native tooltips of the box style. In Java, native tooltips are used only for buttons with the *GlgButton* and *GlgNButton* interaction handlers, and GLG tooltips are still used for object tooltips. • BALLOON_TOOLTIP (C# only) - enables C# balloon tooltips.
GlgButtonTooltipTimeout	Double	0.5	The button tooltip timeout in seconds. For environments other than Java, C# or JavaScript, the GLG_BUTTON_TOOLTIP_TIMEOUT environment variable may also be used to specify the timeout.
GlgMouseTooltipTimeout	Double	0.5	The "object mouse over" tooltip timeout in seconds. For environments other than Java, C# or JavaScript, the GLG_MOUSE_TOOLTIP_TIMEOUT environment variable may also be used to specify the timeout.
GlgTouchTooltipTimeout (JavaScript only)	Double	1.0	The "touch and hold" tooltip timeout in seconds for JavaScript deployed on mobile devices with a touch screen.
GlgDoubleClickTimeout	Double	0.5	The maximum time interval in seconds between two consecutive mouse clicks interpreted as a double-click. Two mouse clicks separated by a time interval greater than this interval will be reported as single clicks. For environments other than Java, C# or JavaScript, the GLG_DOUBLE_CLICK_TIMEOUT environment variable may also be used to specify the timeout.
GlgDoubleClickDelta	Double	5	The maximum distance in pixels between two consecutive mouse clicks interpreted as a double-click. Two mouse clicks separated by a distance greater than this value will be reported as single clicks. For environments other than Java, C# or JavaScript, the GLG_DOUBLE_CLICK_DELTA environment variable may also be used to specify the timeout.

Name	Type	Default	Description (possible values)
GlgURLTimeout (Java and C# only)	Double	-1	Defines the URL load timeout in seconds. If it is set to a value greater than zero, loading operations that use a URL as a source will timeout and generate an error after the timeout interval specified by this resource. It may be used in mission-critical applications to abort loading remote resources from URLs if it takes too long for the remote server to respond.
GlgZoomToButton	Double	1	The mouse button used by the ZoomTo operation.
GlgPanDragButton	Double	1	The mouse button used for panning and scrolling by dragging the drawing with the mouse.
GlgPickResolution	Double	5	Defines pick resolution for user input. When the mouse is within this many pixels of a point, it is considered to be on that point.
GlgChartCacheUse	Double	-1	If it is set to a value greater or equal than zero, it defines a global chart cache use setting for all charts. It overrides the setting of each plot's *CacheUse* attribute and provides a convenient way to test application performance with and without a cache. For environments other than Java, C# or JavaScript, the GLG_CHART_CACHE_USE environment variable may also be used to specify the global cache use setting.
GlgChartCacheExtraSamples	Double	10	Defines a number of extra data samples created when a cache is prefilled. The size of the preallocated cache will be equal to the chart's buffer size plus the number of extra data samples. For environments other than Java, C# or JavaScript, the GLG_CHART_CACHE_EXTRA_SAMPLES environment variable may also be used to specify the number of extra samples.
GlgFreeSDCache (except HTML5 JavaScript)	Double	1	Controls purging the subdrawing cache. If set to 1 (default), the cached subdrawing template is purged from the cache when the last instance that uses it is deleted. If set to 0, the cached template will never be purged, which may increase performance of some applications but may also increase memory consumption.
GlgFreeImageCache (except HTML5 JavaScript)	Double	1	Controls purging the image cache. If set to 1 (default), the cached image is purged from the cache when the last instance that uses it is deleted. If set to 0, the cached image will never be purged, which may increase performance of some applications but may also increase memory consumption.
GlgSaveFormat	String	*"ascii"*	Defines a default save format (*"binary"*, *"ascii"* or *"extended"*).

Name	Type	Default	Description (possible values)
GlgDefaultCharset (Java only)	String	*null*	Specifies the name of a Java charset that will be used as the default charset. If it is set to *null*, the platform's default charset will be used. The default charset controls string decoding in the loaded GLG drawings if the load method does not explicitly specify a charset name.
GlgDefaultEncoding (C# and JavaScript only)	Encoding	*null*	Specifies the name of a C# or JavaScript encoding that will be used as the default encoding. If it is set to *null*, the platform's default encoding will be used. The default encoding controls string decoding in the loaded GLG drawings if the load method does not explicitly specify an encoding. For the JavaScript, the following encodings are supported: *ASCII*, *UTF8* and *LATIN1 (ISO-8859-1)*.
GlgCompressFormat	String	*"uncompressed"*	Enables saved drawing compression (*"compressed"* or *"uncompressed"*).
GlgPSLevel	Double	2	The level of the PostScript output (1 or 2).
GlgSearchPath (C/C++ and ActiveX only)	String	GLG_PATH env. var. or NULL	The search path used to resolve not found relative file names. May be changed from a program. The GLG_PATH environment variable can also be used to define a search path. The search path can contain multiple directory entries separated by a colon on Linux/Unix or by a semicolon on Windows.
GlgCustomDataLib (C/C++ and ActiveX only)	String	NULL	The filename of a custom data browser DLL. The GLG_CUSTOM_DATA_LIB environment variable can also be used to define the DLL filename.
GlgLogLevel (all environments except Java and JavaScript)	Double	5	Controls logging of errors and warnings, may be set to the following values defined in the *GlgApi.h* file: GLG_DISABLE_LOGGING = 0 GLG_LOG_INTERNAL_ERRORS = 1 GLG_LOG_USER_ERRORS = 2 GLG_LOG_WARNINGS = 3 GLG_LOG_INFO_MESSAGES = 4 GLG_LOG_ALL = 5 For C/C++ and ActiveX environments, the GLG_LOG_LEVEL environment variable can also be used and set to a desired numerical value.

Name	Type	Default	Description (possible values)
GlgLabelAdjustmentXY	G (a triplet of X, Y and Z double values)	(2; 2; 0)	Defines an offset used to position labels of an angular axis based on their angular position. The X component defines an offset applied to the horizontal labels with the LEFT or RIGHT anchoring. The Y component defines an offset applied to the vertical labels with the TOP or BOTTOM anchoring. The value of the Z component must be 0. Absolute values greater than 1 define an offset in pixels. Absolute values less than 1 define a relative offset as a fraction of the font width (for the X offset) or the font height (for the Y offset). For C/C++ and ActiveX environments, the GLG_LABEL_ADJUSTMENT_X and GLG_LABEL_ADJUSTMENT_Y environment variables can also be used and set to a desired numerical values.
GlgSelectAllOnFocus	Double	0	May be used to override highlighting behavior of text boxes. If set to 0 (default), the text will be highlighted on gaining focus for all text boxes in the GLG editors, and also for the text boxes in the drawing that have their *SelectAllOnFocus* resource set. If set to 1, the text will always be highlighted on focus in all text boxes regardless of the setting of their *SelectAllOnFocus* resources. If set to -1, the text will not be highlighted.
GlgTabNavigation (all environments except Java and JavaScript)	Double	1	Controls tab navigation traversal order, may be set to the following values defined in the *GlgApi.h* file: GLG_TAB_NONE = 0 GLG_TAB_TEXT_BOXES = 1 GLG_TAB_BUTTONS = 2 GLG_TAB_TEXT_AND_BUTTONS = 3 For C/C++ and ActiveX environments, the GLG_TAB_NAVIGATION environment variable can also be used and set to a desired numerical value.
GlgNativeScrollbars	Double	0 for Qt and Gnome integrations 1 for all other environments	Enables (1) or disables (0) the use of native scrollbars used for integrated scrolling. If disabled, GLG viewport-based scrollbars will be used. For C/C++ and ActiveX environments, the GLG_NATIVE_SCROLLBARS environment variable can also be used and set to either True or False.

Name	Type	Default	Description (possible values)
GlgNativeScrollbar-Width	Double	Depends on the run time environment	Specifies the width of native scrollbars used for integrated scrolling. For C/C++ and ActiveX environments, the GLG_NATIVE_SCROLLBAR_WIDTH environment variable can also be used and set to a desired numerical value.
GlgScrollbarWidth	Double	15	Specifies the width of GLG viewport and light viewport scrollbars used for integrated scrolling. For C/C++ and ActiveX environments, the GLG_SCROLLBAR_WIDTH environment variable can also be used and set to a desired numerical value.
GlgHScrollbar	GlgObject	NULL	Specifies a custom horizontal scrollbar to be used for integrated scrolling.
GlgVScrollbar	GlgObject	NULL	Specifies a custom vertical scrollbar to be used for integrated scrolling.
GlgTitleBar	GlgObject	NULL	Specifies a custom title bar to be used for the Embedded Shell viewports.
GlgTitleBarHeight	GlgObject	NULL	Specifies the title bar height of the Embedded Shell viewports. For C/C++ and ActiveX environments, the GLG_TITLEBAR _HEIGHT environment variable may also be used to specify the title bar height.
GlgJavaScriptFile (all environments except JavaScript)	String	NULL	Specifies a global JavaScript file that contains definitions of global functions. For environments other than Java, C# or JavaScript, the GLG_JAVA_SCRIPT_FILE environment variable or the *-glg-java-script-file* command-line option may also be used to specify the file. In the JavaScript environment, a global JavaScript file can be included in HTML as a script file.
GlgJavaScriptArg-Check (C/C++ and ActiveX only)	Double	1 in the GLG editors 0 at run time	The setting of 1 activates a slower strict mode that performs additional check of the script expression's arguments. When set to 0, a relaxed mode is used for faster execution.
GlgAsyncImage-Loading (Java and C# only)	Double	0	Controls the global image and GIS map loading mode for the entire application. May be set to 0 for the synchronous loading mode, or to 1 for asynchronous loading. The *AsyncMode* attribute may be used to override the global setting for the image and GIS objects.
GlgChangeCursorOn-Grab	Double	1	If set to 1, slider and knob widgets will change cursor when they are moved or dragged with the mouse. Setting the resource to 0 disables cursor change. Alternatively (for environments other than Java, C# or JavaScript), the cursor change behavior may be controlled by setting the GLG_CHANGE_CURSOR_ON_GRAB environment variable to either *True* or *False.*

Name	Type	Default	Description (possible values)
GlgAntiAliasing	Double	GLG_ANTI_ALIASING_INT	In C/C++ and ActiveX, controls the default anti-aliasing setting of the OpenGL and Cairo renderers. In C# and JavaScript, controls the default anti-aliasing setting of the GDI+ or web browser renderer. The default setting is inherited by all polygons with the INHERIT (default) setting of the *AntiAliasing* attribute. The following values are supported: • GLG_ANTI_ALIASING_OFF - disables antialiasing. • GLG_ANTI_ALIASING_INT - enables antialiasing and maps vertices to integer pixel boundaries. • GLG_ANTI_ALIASING_DBL - enables antialiasing and uses double vertex coordinates. In C#, the above constants are elements of the *GlgAntiAliasingType* enum. In Java, the resource controls the default anti-aliasing setting and may be set to 1 to enable anti-aliasing, or to 0 to disable it. Alternatively (for environments other than Java, C# or JavaScript), the antialiasing may be disabled by setting the GLG_ANTI_ALIASING environment variable to *False*.
GlgDisableMouseButtonCheck	Double	0	If set to 1, disables the test of the mouse button number in the mouse click actions. It may be used to deploy applications on devices with a touch screen, so that actions that use the right mouse button will be activated on touch without checking the mouse button number. One example is deploying a GLG application on mobile phones using the GLG JavaScript API. Alternatively (for environments other than Java, C# or JavaScript), GLG_DISABLE_MOUSE_BUTTON_CHECK environment variable may be set to *True* to disable the mouse button check.

Name	Type	Default	Description (possible values)
GlgDisable-ControlKeyCheck	Double	0	If set to 1, disables the test of the Control key state in the mouse actions with the *ProcessArmed* condition. It may be used to deploy an application on devices with a touch screen and no physical keyboard, so that actions with the *ProcessArmed* condition will be activated on touch without checking the state of the Control key. Alternatively (for environments other than Java, C# or JavaScript), GLG_DISABLE_CONTROL_KEY_CHECK environment variable may be set to *True* to disable the Control key check.
GlgSliderTouchDrag (JavaScript only)	Double	1	May be set to 0 to disable dragging sliders and knobs with a touch and drag on mobile devices with a touch screen, in case when it interferes with scrolling the page via touch and drag.
GlgSetNative-Attributes (JavaScript only)	Double	1	If set to 1 (default), allows modifying attributes of native input objects in HTML5/JavaScript, such as wrapping text labels (according to the value of *GlgWrapNativeLabels*) and changing background color to the color specified by the viewport's *FillColor*. Setting the value to 0 disables any modifications and uses default color and label settings of the browser.
GlgWrapNativeLabels (JavaScript only)	Double	1	May be set to 0 to disable wrapping text labels in native buttons and toggles. The wrapping is enabled by default, unless it is disabled by *GlgSetNativeAttributes* set to 0.
GlgAdjustNative-LabelColor (JavaScript only)	Double	1	May be set to 0 to disable color adjustment for text labels in native widgets, such as buttons, toggles, lists, combo boxes, etc. If the background color of a native widget is dark, the color of the text label is set to white by default to improve label visibility. The color adjustment is enabled by default, unless it is disabled by *GlgSetNativeAttributes* set to 0.
GlgEnableMultiline (C/C++ on Windows only)	Double	1	May be set to 0 to disable wrapping text labels in native buttons and toggles on Windows. By default, the labels are wrapped using the BS_MULTILINE flag, which has a side effect of wrapping the label text if it doesn't fit the button, even if the label is not multi-line.
GlgCenterComboBox (C/C++ on Windows only)	Double	1	May be set to 0 to disable vertical centering of combo boxes inside their bounding boxed on Windows. If the combo box's height is smaller than the height of the bounding box defined by its points, the combo box will be centered vertically inside the bounding box. Alternatively, the GLG_CENTER_COMBO_BOX environment variable may be set to *false* to disable centering.

Name	Type	Default	Description (possible values)
GlgGtkFontSize (GTK version on Linux only)	String	NULL	Specifies the font size for the native input objects in the GTK version of the Toolkit on Linux. The font size string uses the CSS syntax common for both GTK and HTML, for example: *"20px"* or *"1.5em"*. If set to NULL, the default font size of the GTK theme will be used. Alternatively, the GLG_GTK_FONT_SIZE environment variable may be used to specify the font size.
GlgValidateUTF8 (GTK version on Linux only)	Double	1	Controls validation of string encoding in the GTK version on Linux. All strings passed to the GTK Cairo renderer and GTK native input objects must use UTF8 encoding to avoid undefined behavior and, by default, are validated to contain valid UTF8 strings. An error will be generated if validation fails. Setting *GlgValidateUTF8=0* disables the validation and should be done only if the application guarantees that all strings are properly encoded UTF8 strings. Alternatively, the GLG_VALIDATE_UTF8 environment variable may be set to *true* or *false* to control string validation.
GlgXResourceFile	String	NULL	Specifies location of the X resource file that can be used to define custom XFT font settings used by the native input objects in both the GLG editors and the C/C++ executables in the legacy X11 version of the Toolkit on Linux. Alternatively, the GLG_X_RESOURCE_FILE environment variable or the *-glg-x-resource-file* command-line option may be used to specify the resource file location. A sample *glg_x_resource_file* file with default settings is provided in the *src* directory of the GLG installation.
GlgNativeWidgetsXft	Double	1	May be set to 0 to disable the use of XFT fonts by native input objects for C/C++ executables in the legacy X11 version of the Toolkit on Linux for backward compatibility. Alternatively, the use of XFT fonts may be disabled by setting the GLG_NATIVE_WIDGETS_XFT environment variable to *false*, as well as using the *-glg-native-widgets-xft* command-line options.

Name	Type	Default	Description (possible values)
GlgDisableDB	Double	1 in GTK/Cairo, Java/Swing and JavaScript, 0 otherwise	Controls the use of double buffering. Setting it to 1 disables double buffering for viewports with *DoubleBuffering=ON* but leaving it enabled for viewports with *DoubleBuffering=ON (FORCED)*. Setting it to 2 disables it for all viewports. The 0 setting doesn't disable double buffering. The attribute is ignored for OpenGL viewports and is set to 1 for environments that implement native double buffering: GTK/Cairo, Java/Swing and HTML5/JavaScript. In the C/C++ environment, the GLG_DISABLE_DB environment variable may also be set to the above values to disable double buffering.
GlgMaxDBFactor	Double	1.25	Controls the threshold for disabling double buffering for exceedingly large children viewports to avoid system slow-down due to swapping at high zoom factors. If the value of *GlgMaxDBFactor* is non-negative, the viewport's double buffering is disabled if: $w * h > F * W * H$ where w and h are the pixel width and height of the viewport, W and H are the width and height of the display, and F is the value of *GlgMaxDBFactor*. If the value of *GlgMaxDBFactor* is negative, the double-buffering check is suppressed and the double-buffering is never disabled regardless of the viewport size. In JavaScript, Java/Swing and the GTK version of the C/C++ library, the GLG double buffering is disabled by default, since these environments employ double buffering on the system level. In the C/C++ environment, the GLG_MAX_DB_FACTOR environment variable may used to specify the *GlgMaxDBFactor value.*

Name	Type	Default	Description (possible values)
GlgOpenGLMode (C/C++ and ActiveX only)	Double	0 in GTK, 1 otherwise	Controls the use of the OpenGL renderer. If set to 1, the OpenGL renderer is used for the viewports with OpenGLHint=ON if the OpenGL renderer is available. Setting the resource to 0 suppresses the use of the OpenGL renderer. In the GTK version, the OpenGL renderer is disabled by default and can be selectively enabled for viewports that need 3D hidden surface removal by setting their *ForceOpenGL* attribute to YES. Alternatively, the OpenGL renderer may be enabled or disabled by setting the GLG_OPENGL_MODE environment variable to *true* or *false*, as well as using the *-glg-enable-opengl* and *-glg-disable-opengl* command-line options. The GLG_DISABLE_OPENGL or GLG_ENABLE_OPENGL environment variables may also be set to *true* to disable or enable the OpenGL renderer.
GlgOpenGLVersion (C/C++ and ActiveX only)	Double	300	Requests the OpenGL version to be used. The shader-based core OpenGL profile is used for OpenGL versions higher than 3.00, and the Compatibility profile is used for older OpenGL versions. The Core profile is used by default. If the graphics card doesn't support core profile, the driver will automatically revert to the compatibility profile. If the graphics card doesn't support OpenGL, the GDI driver will be used. Alternatively, the GLG_OPENGL_VERSION environment variable or the *-glg-opengl-version* <*NNN*> command-line option may be used to define the OpenGL version at run time.
GlgOpenGL-HardwareVendor *GlgOpenGL-HardwareRenderer* *GlgOpenGL-SoftwareVendor* *GlgOpenGL-SoftwareRenderer* (C/C++ and ActiveX only)	String	NULL	Read-only resources that provide an application access to the OpenGL vendor and renderer strings for the hardware and software OpenGL. These resources will be set after the viewport that uses a corresponding OpenGL driver is drawn the first time.

Name	Type	Default	Description (possible values)
GlgOpenGL-HardwareThreshold (C/C++ and ActiveX only)	Double	1	Controls the runtime mapping of the OpenGL priorities specified by a viewport's *OpenGLHint* attribute. All viewports with a non-zero value of the attribute, less than or equal to the threshold value, will be rendered using the hardware OpenGL renderer (if available). Viewports with attribute values between the *GlgOpenGLHardwareThreshold* and *GlgOpenGLThreshold* will be rendered using the software OpenGL renderer (if available). Alternatively, the GLG_OPENGL_HARDWARE_THRESHOLD environment variable or the *-glg-opengl-hardware-threshold <N>* command-line option may be used to define the threshold at run time. The hardware OpenGL renderer may be disabled by using the *-glg-disable-hardware-opengl* command-line option or by setting the GLG_DISABLE_HARDWARE_OPENGL environment variable to *true*. The software OpenGL renderer may be disabled by using the *-glg-disable-software-opengl* command-line option or by setting the GLG_DISABLE_SOFTWARE_OPENGL environment variable to *true*.
GlgOpenGLThreshold (C/C++ and ActiveX only)	Double	10	Controls the runtime mapping of the OpenGL priorities specified by a viewport's *OpenGLHint* attribute. All viewports with attribute values greater than the threshold value will be rendered using the GDI renderer. Alternatively, the GLG_OPENGL_THRESHOLD environment variable or the *-glg-opengl-threshold <N>* command-line option may be used to define the threshold at run time.
GlgOpenGLFull-Redraw (C/C++ and ActiveX only)	Double	-1	Controls what portion of the viewport will be redrawn when only a part of the viewport needs to be updated for hardware-accelerated OpenGL with double buffering. If set to 0, only the changed portion of the viewport will be redrawn. Setting the resource to 1 causes the whole viewport to be updated, which helps to eliminate flickering on non-NVidia graphics cards. The default -1 setting automatically selects redraw mode depending on the detected graphics card. Alternatively, a *True* or *False* setting of the GLG_OPENGL_FULL_REDRAW environment variable, or *-glg-opengl-enable-full-redraw* and *-glg-opengl-disable-full-redraw* command-line options may be used to define redraw policy.

Name	Type	Default	Description (possible values)
GlgGLXPixmapDB (C/C++ and ActiveX only)	Double	0	Controls the type of double buffering used by the hardware-accelerated OpenGL. If set to 0 (default), the double buffering will be performed using the OpenGL back buffer. If set to 1, the double-buffering will be performed by using an off-screen pixmap instead of the OpenGL back buffer. Alternatively, the GLG_GLX_PIXMAP_DB environment variable may be set to *True* or *False* to define the double buffering type.
GlgOpenGLNative-LineAA (C/C++ and ActiveX only)	Double	-1	Controls the anti-aliasing technique used by the core profile of the hardware-accelerated OpenGL. If set to 1, line anti-aliasing will be performed using the line anti-aliasing capabilities provided by the graphics card. These capabilities are optional and may be substandard or not implemented for graphics cards other than NVidia. If set to 0, shader-based anti-aliasing will be used. The default value of -1 detects if the graphics card supports line anti-aliasing and automatically selects the anti-aliasing type. Alternatively, the GLG_OPENGL_NATIVE_LINE_AA environment variable may be set to *True* or *False* to define the anti-aliasing type.
GlgOpenGLZSort (C/C++ and ActiveX only)	Double	2	Controls the number of passes used to eliminate pixel artifacts when drawing polygons with both fill and edges. If polygons have only fill or only edges, the resource can be set to 1 to use a single pass for increased performance. Setting the resource to 0 disables OpenGL hidden surface removal and resorts to the slower non-OpenGL depth-sorting technique. Alternatively, the GLG_OPENGL_ZSORT environment variable may be used to define the number of passes.
GlgOpenGLDepth-Offset (C/C++ and ActiveX only)	Double	100	Controls the depth offset used by the second pass of the hidden surface removal to offset edges of polygons. Alternatively, the GLG_OPENGL_DEPTH_OFFSET environment variable may be used to define the offset.

Name	Type	Default	Description (possible values)
GlgOpenMesaLibPath (C/C++ on Linux/Unix only)	String	NULL	Specifies the location of the *libOSMesa* library used for the software OpenGL. The library is located in the lib directory of the GLG installation and is found there by default if the GLG_DIR environment variable is set. Alternatively, the GLG_OPENGL_MESA_LIB_PATH environment variable, or *-glg-opengl-mesa-lib-path* command-line options may be used to define the library location.
GlgOpenTessLibPath (C/C++ on Linux/Unix only)	String	NULL	Specifies the location of the *libtess_util* library used by the software OpenGL renderer if the hardware OpenGL renderer is disabled. The library is located in the lib directory of the GLG installation. By default, it is searched for in the directory where the *libOSMesa* library was found. Alternatively, the GLG_OPENGL_MESA_LIB_PATH environment variable, or *-glg-opengl-tess-lib-path* command-line options may be used to define the library location.
GlgMapServerUsage (C/C++ and ActiveX only)	Double	0	Controls the use of Map Server in C/C++ and ActiveX versions of GLG and may have one of the following values defined in the *GlgApi.h* file: •GLG_LOCAL_MAP_SERVER - uses the Map Server from the GLG library . •GLG_REMOTE_MAP_SERVER - uses the Map Server from a web server defined by the GISMapServerURL attribute of the GIS Object. •GLG_AUTO_MAP_SERVER (default) - if the drawing was loaded from a URL, use a remote Map Server, otherwise use a local Map Server. May also be defined by setting GLG_MAP_SERVER_USAGE environment variable to the *"auto", "local"* or *"remote"* values.
GlgGISElevation Layer (GLG editors only)	String	NULL	Specifies the GIS elevation layer to be used for querying elevation in the map displayed in the GLG Builder or HMI Configurator.
GlgXftFonts (C/C++ on Linux/Unix only)	Double	1	Controls the use of the XFT fonts: setting the resource to 0 disables XFT fonts, and setting it to 1 enables XFT fonts. Alternatively, setting the GLG_XFT_FONTS environment variable to *true* or *false*, or using the *-glg-enable-xft-fonts* and *-glg-disable-xft-fonts* command-line options may be used to enable or disable XFT fonts.

Name	Type	Default	Description (possible values)
GlgDebugXftFonts (C/C++ on Linux/Unix only)	Double	0	If set to 1, enables diagnostic output for troubleshooting XFT fonts. The output is displayed in the terminal, and is also saved in the *glg_error.log* file. Alternatively, the GLG_DEBUG_XFT_FONTS environment variable may be set to *true*, or the *-glg-debug-xft-fonts* may be used.
GlgDebugFonts (C/C++ on Linux/Unix only)	Double	0	If set to 1, enables diagnostic output for troubleshooting font sets. Alternatively, the GLG_DEBUG_FONTS environment variable may be set to *true*, or the *-glg-debug-fonts* may be used. The output is displayed in the terminal, and is also saved in the *glg_error.log* file.
GlgDebugEncoding (C/C++ and ActiveX only)	Double	0	If set to 1, enables diagnostic output for troubleshooting string encoding conversions. Alternatively, the GLG_DEBUG_ENCODING environment variable may be set to *true*, or the *-glg-debug-encoding* command-line option may be used. The output is displayed in the terminal on Linux/Unix, and is also saved in the *glg_error.log* file.
GlgDefaultFontTable-File (except JavaScript)	String	NULL	The name of a drawing file containing a saved *FontTable* object to be used as a default font table. For C/C++ and ActiveX environments, it may also be defined by setting the GLG_DEFAULT_FONT_TABLE_FILE environment variable. The font table may be saved using the *Save Font Table* option of the viewport's *FontTable* object. The drawing can also contain a viewport object with a custom font table attached. In this case, the font table is searched using *$Fonttable* and then *$Widget/Fonttable* resource names. If still not found, the fonttable of a $*Widget* viewport is also considered.
GlgDefaultFontTable	GlgObject	NULL	The *FontTable* object to be used as a default font table. This resource can be used in a program that creates or loads a font table object at run time.
GlgDefaultFontFile (except JavaScript)	String	NULL	Specifies name of an ASCII file that contains a list of font names for the default font table. Each font in the file is listed on a separate line. For C/C++ and ActiveX environments, it may also be defined by setting the GLG_DEFAULT_FONT_FILE_NAME environment variable. The number of font types and sizes for the font table is specified using the *GlgDefaultNumFontTypes* and *GlgDefaultNumFontSizes* variables described below.

Name	Type	Default	Description (possible values)
GlgDefaultFontURL (Java and C# only)	String	NULL	Specifies a URL of an ASCII file that contains a list of font names for the default font table. Each font in the file is listed on a separate line. The number of font types and sizes for the font table is specified using the *GlgDefaultNumFontTypes* and *GlgDefaultNumFontSizes* variables described below.
GlgDefaultFontList (JavaScript only)	String	NULL	A string containing a list of font names for the default font table in the format of *GlgDefaultFontFile*. The number of font types and sizes for the font table is specified using the *GlgDefaultNumFontTypes* and *GlgDefaultNumFontSizes* variables described below.
GlgDefaultXftFontFile (except JavaScript)	String	NULL	Specifies name of an ASCII file that contains a list of XFT font names for the default font table. Each font in the file is listed on a separate line. For the C/C++ environment, it may also be defined by setting the GLG_DEFAULT_XFT_FONT_FILE_NAME environment variable. The number of font types and sizes for the font table is specified using the *GlgDefaultNumFontTypes* and *GlgDefaultNumFontSizes* variables described below.
GlgDefaultPSFont-File (except JavaScript)	String	NULL	Specifies name of the file that contains a list of PostScript font names for the default font table. For C/C++ and ActiveX environments, it may also be defined by setting the GLG_DEFAULT_PS_FONT_FILE_NAME environment variable. The number of font types and sizes for the font table in specified using the *GlgDefaultNumFontTypes* and *GlgDefaultNumFontSizes* variables described below.
GlgDefaultPSFont-URL (Java and C# only)	String	NULL	Specifies a URL of the file that contains a list of PostScript font names for the default font table. The number of font types and sizes for the font table in specified using the *GlgDefaultNumFontTypes* and *GlgDefaultNumFontSizes* variables described below.
GlgDefaultNumFont-Types	Double	12	Defines NumTypes attribute of the default font table. For C/C++ and ActiveX environments, it may also be defined by setting the GLG_DEFAULT_NUM_FONT_TYPES environment variable to the desired integer value.
GlgDefaultNumFont Sizes	Double	7	Defines NumSizes attribute of the default font table. For C/C++ and ActiveX environments, it may also be defined by setting the GLG_DEFAULT_NUM_FONT_SIZES environment variable to the desired integer value.

Name	Type	Default	Description (possible values)
GlgFontCharset (C/C++ and ActiveX only)	Double	0	Defines a charset to use for fonts with DEFAULT_CHARSET, providing a simple way to change the charset of all fonts in the default font table. May also be defined by setting GLG_FONT_CHARSET environment variable to the integer value of the desired font charset. If not set, the charset of the current system locale is used.
GlgMultibyteFlag (C/C++ and ActiveX only)	Double	0	Defines the value of the multi-byte flag to use for fonts with DEFAULT_CHARSET on Windows or for X11 core (XLFD) fonts in the legacy X11 version on Linux/Unix. It provides a simple way to change the multi-byte flag setting for all fonts in the default font table and may also be defined by setting GLG_MULTIBYTE_FLAG environment variable to the *"SINGLE_BYTE", "MULTI_BYTE"* or *"UTF8"* values. If not set, the setting matching the current system locale (SINGLE_BYTE or MULTI_BYTE) is used.
GlgDefaultColor-TableType	Double	1	Defines type of the default color table: RAINBOW (value of 1) or STANDARD (value of 2). For C/C++ and ActiveX environments, it may also be defined by setting the GLG_DEFAULT_COLOR_TABLE_TYPE environment variable to *"rainbow"* or *"standard"*.
GlgDefaultColor-Factor	Double	19	Defines *ColorFactor* of the default color table. For C/C++ and ActiveX environments, it may also be defined by setting the GLG_DEFAULT_COLOR_FACTOR environment variable to the desired integer value.
GlgDefaultNum-ColorGrades	Double	1	Defines *GradeHint* of the default color table. For C/C++ and ActiveX environments, it may also be defined by setting the GLG_DEFAULT_NUM_COLOR_GRADES environment variable to the desired integer value.
GlgDefaultDialog-Color	G - holds 3 RGB color values	(-1;-1;-1)	Specifies fill color for the DIALOG_AREA widget type. If it is set to (-1;-1;-1), the system color will be used as described below, otherwise the *FillColor* of the dialog widget will be used. On Windows, the system setting for the dialog color is used. In the GTK version on Linux, the GTK theme color is used.
GlgDefaultLabel-Color	G - holds 3 RGB color values	(-1;-1;-1)	Specifies color for *DefaultLabelColor* of the DIALOG_AREA widget type. If it is set to (-1;-1;-1), the system color will be used as described below, otherwise the value of the *DefaultLabelColor* will not be changed. On Windows, the system setting for the label color is used. In the GTK version on Linux, the GTK theme color is used.

Name	Type	Default	Description (possible values)
GlgIndexedColor-TableFile (except JavaScript)	String	NULL	A filename of a custom color palette drawing that defines a list of indexed colors. For environments other than Java, C# or JavaScript, it may also be defined via the GLG_INDEXED_COLOR_TABLE_FILE environment variable.
GlgIndexedColor-File (except JavaScript)	String	NULL	A filename of an ASCII file that defines a list of indexed colors. For environments other than Java, C# or JavaScript, it may also be defined via the GLG_INDEXED_COLOR_FILE environment variable.
GlgIndexedColor-Table (JavaScript only)	GlgObject	NULL	An object ID of a loaded custom color palette drawing that defines a list of indexed colors. This is the same drawing that is used for *GlgIndexedColorTableFile* in other environments.
GlgIndexedColor-List (JavaScript only)	String	NULL	A string containing a list of indexed colors in the format of *GlgIndexedColorTableFile.*
GlgDisableIndexed-Colors	Double	0	If set to 1, disables the use of indexed colors for compatibility with any possible special use of RGB values in older releases. For environments other than Java, C# or JavaScript, the GLG_DISABLE_INDEXED_COLORS environment variable may be set to *true* or *false* to control the use of indexed colors.
GlgIHArray	group object	an array holding default GLG interaction handlers	The array of GLG Interaction Handlers (*GlgButton*, *GlgSlider*, etc.) used to resolve interaction handlers names. It may be used by a program to add custom interaction handlers.
GlgCompatibility-Mode	Double	GLG_LATEST_RELEASE constant	Specifies compatibility mode. For example, when it is set to the GLG_PRE_2_9 constant, the release 2.9 changes which affect code compatibility with the older versions will be disabled. For C/C++ and ActiveX environments, this resource may also be defined by setting the GLG_COMPATIBILITY_MODE environment variable to the integer value corresponding to the desired constant.

Name	Type	Default	Description (possible values)
GlgExtSave42	Double	0	Controls compatibility mode for exporting drawings that have *Bound* subdrawing properties into earlier releases prior to release 4.3. If set to 1, saving the drawing using the EXTENDED save format will merge *Global* and *Bound* attributes into a single *Global* flag used in the earlier releases. Alternatively, this compatibility mode can also be enabled by using the *-glg-ext-save-42* command-line option or by setting the GLG_EXT_SAVE_42 environment variable to *true*. These options should only be used for exporting drawings to the pre-4.3 versions of the Toolkit.
GlgDisablePre35-ObjectEvents	Double	1	When set to 0, disables processing of the old-style (prior to v. 3.5) tooltips, custom events and mouse feedback, which may reduce the CPU load when the mouse moves over a large drawing. For C/C++ and ActiveX environments, this resource may also be defined by setting the GLG_DISABLE_PRE35_OBJECT_EVENTS environment variable to *True* or *False*.
GlmMaxIterations (C/C++ and ActiveX)	Double	10000	Defines the maximum number of internal iterations used to render filled polygons in the map server, which prevents rendering delays if the map is zoomed in too much. This parameter may need to be increased if large filled polygons (such as OSM country or state areas) are rendered with very high zoom levels. This value may be overridden by the MAX ITERATIONS parameter in the dataset's SDF file.

7.2 Appendix B: Message Object Resources

A message object is a GLG object used by the Toolkit to pass information about GLG input, object selection and other events to the *Input* callback. It is a group object containing data in the form of named resources. The *Input* callback code can query resources of the message object using the GLG GetResource methods. Refer to the *Handling User Input and Other Events* section of the *Using the C/C++ version of the Toolkit* chapter for more information on handling input events.

A message object is also used for returning multiple pieces of information from methods such as *GlgCreateChartSelection*, in which case the message does not include generic resources listed in the following section.

Generic Resources of the Message Object

All message objects received in the Input and other callbacks have the following named attributes:

Resource Name	Data Type	Description
Format	String	Defines the format of the message object and the resources present in it. It is defined by the event type and, for input messages, the type of the input handler which detected the input activity and sent the message. It may have values such as *Button*, *Slider*, *Knob*, *ObjectSelection* and *CustomEvent*.
Origin	String	Contains one of the following values: • The name of the viewport with an input handler that generated the input message • The name of the object that generated the custom event message • The name of the viewport that generated the window message • An empty string for object selection and other message types.
FullOrigin	String	Contains the full path name of the viewport that initiated an input or window message, or the full path name of the object that initiated a custom event message. This resource will be set to an empty string for other types of messages. For composite input objects, such as a spinner, *Origin* and *FullOrigin* can point to different objects. For example, when an *Increase* or *Decrease* button of the spinner is activated, the *Origin* resource contains the name of the button that was pressed, while the *FullOrigin* resource contains the full path to the spinner that contains the button. Another example when Origin and FullOrigin point to a different object is when a text object of a spinner gains or loses focus, generating corresponding events

Resource Name	Data Type	Description
Action	String	Describes the input action occurred. It may have values of *ValueChanged*, *Activated*, *MouseClick, MouseOver* and so on (see below). All input handlers send *Abort2* when they receive a middle mouse button click, and *Abort3* when they receive a right mouse button click.
SubAction	String	Describes the action in more details. For example, for the *ValueChanged* action it describes what caused the value change and may have values such as *Pick*, *Motion*, *Increase*, and so on.
Object	GlgObject	The viewport object that triggered the *Input* callback, or the object that triggered custom event. This resource is present in all messages except the *ObjectSelection* message.

Specific Resources of the Message Object

A message object can have other resources as well, depending on the type of the input handler (for input events) or the type of the message (for object selection and custom events).

In the case of an input event, the message object includes all handler resources of the widget. For example, the message object coming from a GLG slider object may also contain such resources as *ValueX*, *ValueY*, *Granularity*, *Increment*, *DisableMotion* and others. These resources may be queried from either the message object or the viewport object that generated the message.

Handler resources not present in the viewport are not present in the message object. For example, a one-dimensional slider uses only one of its *ValueX* and *ValueY* resources, and the second resource will be absent from the message object. Several handlers define resources that appear only in the message object. For example, the *GlgMenu* input handler defines *SelectedIndex* and other resources used by the handler during its operation. All such resources are defined in the sections below.

The following sections describe the format of message objects generated by various input handlers as well as object selection and custom events. Refer to the *Input Handlers* section of the GLG User's Guide for a complete list of resources of various GLG input handlers.

Slider Message Object

This message object is generated by the *GlgSlider* or *GlgNSlider* input handlers whenever a user acts to change the position of the slider or scrollbar controlled by these input handlers.

Slider Message Object Resources

Resource	Value
Format	*Slider*
Action	• *ValueChanged* indicates when a value change has resulted from a mouse click. • *GrabPointer* is issued when the mouse button is pressed over the slider active area, causing the mouse to assume control of the slider position. The *SubAction* is NULL. • *UngrabPointer* is issued when the mouse button is released, ending mouse control of the slider position. The *SubAction* is NULL. • *RepeatStart* and *RepeatEnd* are issued before and after the *ValueChange* messages that were generated by one of the slider's buttons (such as *Increase* or *Decrease*) and a button's repeated action was enabled by its *RepeatTimeout* resource.
SubAction	• *Click* when a value change has resulted from a mouse click • *Motion* when a value change has resulted from a mouse motion while the left mouse button is held down • *Increase*, *Decrease, PageIncrease* or *PageDecrease* when a value change has resulted from activating the corresponding button.
Object	The slider's viewport.

The *ValueX*, *ValueY*, *Granularity*, *Increment*, *DisableMotion* and other slider resources are also present in the message object.

Knob Message Object

This message object is generated by the *GlgKnob* input handler whenever a user acts to change the position of the knob.

Knob Message Object Resources

Resource	Value
Format	*Knob*
Action	• *ValueChanged* indicates when a value change has resulted from a mouse click • *GrabPointer* is issued when the mouse button is pressed over the slider active area, causing the mouse to assume control of the slider position. The *SubAction* is NULL. • *UngrabPointer* is issued when the mouse button is released, ending mouse control of the slider position. The *SubAction* is NULL. • *RepeatStart* and *RepeatEnd* are issued before and after the *ValueChange* messages that were generated by one of the knob's buttons (such as *Increase* or *Decrease*) and a button's repeated action was enabled by its *RepeatTimeout* resource.
SubAction	• *Click* when a value change has resulted from a mouse click • *Motion* when a value change has resulted from a mouse motion while the left mouse button is held down • *Increase*, *Decrease* , *PageIncrease* or *PageDecrease* when a value change has resulted from activating the corresponding button.
Object	The knob's viewport.

The *Value*, *Granularity*, *Increment*, *DisableMotion* and other knob resources are also present in the message object.

Button Message Object

This message object is generated by the *GlgButton* or *GlgNButton* input handlers whenever a user presses the button or toggle controlled by these handlers.

Button Message Object Resources

Resource	Value
Format	*Button*
Action	• *Activate* when activating push buttons without *OnState* resource or native push buttons. • *ValueChanged* when activating toggle buttons with *OnState* resource or native toggle buttons. • *RepeatStart* and *RepeatEnd* are issued before and after the *Activate* messages if the button's repeated action was enabled by its *RepeatTimeout* resource. • *Update* for various miscellaneous actions, in which case *SubAction* will indicate the action type.
SubAction	• *InState* when *InState* changes (*GlgButton*) • *PressedState* when *PressedState* changes (*GlgButton*) • *ArmedState* when *ArmedState* changes (*GlgButton*) • *Pressed* or *Released* (native push button with the *GlgNButton* handler) • *DelayTimerStart* when the delay timer starts (*GlgButton*) • *DelayTimerEnd* when the delay timer expires (*GlgButton*) • *DelayTimerCancel* when the delay timer gets cancelled because the button is released or the mouse is moved outside of the button before the delay interval expired (*GlgButton*) • *DelayState* when the value of the *DelayState* resource is changed (*GlgButton*)
Object	The button's viewport.

As with all the message objects, the button message object also includes the handler resources that are used in the widget, such as *PressedState*, *OnState* and others. Refer to the *Input Handlers* section of the GLG User's Guide for a complete list of resources of the *GlgButton* and *GlgNButton* input handlers.

Text Message Object

This message object is generated by the *GlgText* or *GlgNText* input handlers whenever a user enters text into the text input widget controlled by these input handlers.

Text Widget Message Object Resources

Resource	Value
Format	*Text*
Action	• *Activate* if the *Enter* key was pressed at the end of text input. • *ValueChanged* is generated for one of the following events: - if a key press changes the string displayed in a string text input - if the *Enter* key is pressed at the end of a numerical text input - if the value of a numerical text input has changed and the text input is about to lose focus. • *Focus* if the text input gained the input focus (*GlgNText* handler only). • *LosingFocus* if the text input lost the input focus (*GlgNText* handler only). • *Update* for any other key press or user action. This includes changing the current value displayed in a numerical text input, in which case the *ValueChanged* action will be generated when the user finishes entering a new value and either presses *Enter* or moves focus away from the text input.
SubAction	none
Object	The text widget's viewport.

The *TextString* resource and, for numerical inputs, *Value* resource are present in the message object even if they do not exist in the Text Widget's viewport, allowing the current values to be passed in the message. The *MaxLength* and, for numerical inputs, the *MinValue, MaxValue, EnforceRange, InputInvalid* and *ValueFormat* resources are also present in the message object.

Spinner Message Object

This message object is generated by the *GlgSpinner* input handlers whenever a user changes the spinner's value.

Spinner Widget Message Object Resources

Resource	Value
Format	*Spinner*
Action	• *ValueChanged* if the spinner's value has changed • *Update* for any other user interaction. • *Activate* if the spinner's Done button was pressed. The *Activate* action of the spinner's text input object is also propagated to the spinner. • *RepeatStart* and *RepeatEnd* are issued before and after the *ValueChange* messages that were generated by one of the spinner's or spinner slider's buttons (such as *Increase* or *Decrease*) and a button's repeated action was enabled by its *RepeatTimeout* resource.
SubAction	For *ValueChange* actions, indicates the origin of the action, such as *Slider*, *TextInput*, *Increase* or *Decrease* buttons. It is set to NULL otherwise.
Object	The spinner widget's viewport.

All resources of the spinner, such as *Value, MinValue*, *MaxValue*, *Increment*, etc. are also present in the message object.

List Message Object

This message object is generated by the *GlgNList* input handlers when a user selects items in a list widget controlled by this input handler.

List Widget Message Object Resources

Resource	Value
Format	*List*
Action	• *Select* if an item was selected or deselected. • *Update* for any other key.
SubAction	*DoubleClick* if the item was selected using a double-click, otherwise *NULL*.
Object	The list's viewport.

The *GlgSendMessage* method may be used to get extended selection information for a multiple selection list widget.

Option Message Object

This message object is generated by the *GlgNOption* input handlers when a user changes an option selection in an option menu or combox box widget controlled by this input handler.

List Widget Message Object Resources

Resource	Value
Format	*Option*
Action	• *Select* if a new option was selected. • *Update* for any other key.
SubAction	*none*
Object	The option widget's viewport.

Menu Message Object

This message object is generated by the *GlgMenu* input handler whenever a user presses a button in the menu.

Menu Message Object Resources

Resource	Value
Format	*Menu*
Action	*Activate*
SubAction	none
Object	The menu's viewport.

The *SelectedIndex* and *SelectedString* resources are present in the message object even if they do not exist in the menu viewport, allowing the menu to pass information about the selection with the message.

Clock and Stopwatch Message Object

The first message is generated by the clock and stopwatch widgets (*GlgClock* input handler) when they update the graphical representation of the clock. This is invoked automatically every second. If seconds are not displayed, the callback is invoked once per minute, instead.

The second message is only called from a stopwatch widget, whenever a user presses one of the control buttons.

Clock Widget Message Object Resources

Resource	Value
Format	*Clock*
Action	*Update*
SubAction	none
Object	The widget's viewport.

Stopwatch Widget Message Object Resources

Resource	Value
Format	*Clock*
Action	*Start*, *Stop*, or *Reset* depending on the corresponding stopwatch button.
SubAction	none
Object	The widget's viewport.

Either message object will also contain resources of *GlgClock* input handler such as *Sec*, *ValueSec*, and so on. For the Stopwatch Widget, the *Sec* resource is present in the message object even if it does not exist in the stopwatch's viewport.

Timer Message Object

This message object is generated automatically by the *GlgTimer* input handler upon every update of the timer output *Value*.

Timer Widget Message Object Resources

Resource	Value
Format	*Timer*
Action	*Update*
SubAction	none
Object	The timer's viewport.

The program should call the *GlgUpdate()* function to make the changes caused by the timer visible. The remaining resources of the message object (*Interval*, *Period*, *ActiveState*, and so on) can be used to change the timer's state and parameters. The timer handler has been superseded by the timer transformation in the release 2.8 of GLG.

Palette Message Object

This message object is generated by the *GlgPalette* input handler whenever a user selects one of the objects in the palette with the left mouse button, or changes the selection by dragging the mouse over the palette.

Palette Message Object Resources

Resource	Value
Format	*Palette*
Action	• *Activate* when a selection is made with a mouse click • *ValueChanged* when the selection is changed due to a mouse dragging.
SubAction	*Click* or *Motion*
Object	The widget's viewport.

The palette's message object also has a resource named *SelectedObject*. This resource refers to the object selected and is used to get values of the that object's attributes. For example, for a color palette, the *SelectedObject/FillColor* resource of the message object yields the chosen color.

Font Browser Message Object

This message object is generated by the *GlgBrowser* input handler whenever any input activity is detected in the font browser controlled by the input handler.

Font Browser Message Object Resources

Resource	Value
Format	*FontBrowser*
Action	• *Activate* when the *Done* button of the Font Browser is activated • *Cancel* when the *Cancel* button is pressed. • *Update* on any other activity in the Font Browser.
SubAction	none
Object	The widget's viewport.

All the resources of the font browser widget are also present in the message object, allowing the font selection to be passed in the message.

Browser Message Object

This message object is generated by the *GlgBrowser* input handler whenever any input activity is detected in the resource, tag, alarm or custom data browser controlled by the input handler.

Browser Message Object Resources

Resource	Value
Format	*Browser*
Action	• *Activate* when the *Done* button of the Browser is pressed. • *Cancel* when the *Cancel* button is pressed. • Select when the user clicks on a list entry. • *Update* on any other activity in the Browser.
SubAction	For the *Update* action, the *SubAction* may be *DoubleClick* when the user double-clicks to open another hierarchy level. Otherwise, this resource is not used.
Object	The browser's viewport.

The *Input* callback is called with *Activate* as the value of the *Action* resource when the *Done* button of the file browser is activated, *or* when the *Enter* key is pressed with the keyboard focus in the *Path* text widget *and* the string is a valid ID (resource name, tag name, etc.) of some element in the browser list.

Zoom Message Object

This message object is generated when a zoom or pan operation is performed, activated either by user interaction or programmatically.

Zoom Message Object Resources

Resource	Value
Format	*Zoom*
Action	• *Zoom* • *Pan* • *ScrollbarChange* Generated when an X or Y scrollbar is drawn or erased in the AutoPan mode.
SubAction	• *Left, Right, Up, Down, X, Y, x* or *y* for the *Pan* action. • *ZoomIn, ZoomOut* or *Reset* for the *Zoom* action. • *Set* for the fit to box *Zoom* actions (*'F'* and *'f'*). • *ResetX* or *ResetY* for corresponding *'N'* and *'n' Zoom* actions in the Chart Zoom mode. • *Start*, *End* or *Abort* for the ZoomTo mode of the *Zoom* action or the Dragging mode of the *Pan* action: notifies about the start, end or abort of the action. • *Drag* for the Dragging mode of the *Pan* action. Repeated messages are generated to notify about the drag action in progress. • *ZoomRectangle* for the ZoomTo mode of the *Zoom* action: notifies that zoom rectangle was created and is accessible . • *ZoomRectangleChange* for the ZoomTo mode of the *Zoom* action: notifies that the ZoomTo rectangle was resized by the mouse move. • *ZoomRectangleEnd* for the ZoomTo mode of the *Zoom* action: notifies that the ZoomTo rectangle is about to be destroyed and may be used to extract the final ZoomTo extent.
Object	The viewport that triggered the message.

The *Start* and *End* subactions are generated when the ZoomTo, custom zoom or dragging operation is started or finished. The *Reset* subaction is generated when the zooming state is reset using the 'n' or 'N' zoom action. The *Set* subaction is generated when the drawing is zoomed, paned or rotated using any zoom actions other than ZoomTo, custom zoom or *Reset*. The *ZoomRectangle* subaction indicates that the ZoomTo rectangle has been created and is accessible via the *GlgZoomRect* resource of the viewport in the ZoomTo mode. The *ZoomRectangleChange* and *ZoomRectangleEnd* subactions notify the application that the zoom rectangle is resized with the mouse or about to be destroyed at the end of the ZoomTo action.

Custom Event Message Object

This message object is generated by a GLG drawing when a mouse event triggers a custom event action (with *ActionType*=SEND_EVENT) attached to an object. Depending on the action's *Trigger*, an action is triggered when an object it is attached to is either clicked on with the mouse, the mouse moved over the object, or, for input objects, a matching input activity has occurred. Alternatively, the message may also be generated by the old-style (prior to v. 3.5) *MouseClickEvent* or *MouseOverEvent* custom properties attached to the object instead of an action object.

For actions with the MOUSE_CLICK and MOUSE_OVER triggers, two types of messages are generated: an activation message is generated when the action is activated by a mouse click or mouse over event, and a deactivation message is generated when the action ends due to the mouse button being released or the mouse moving away from the object.

Actions that are sensitive to the state of the *Control* key (actions with *ProcessArmed* set to a value other than NONE) may also be activated and deactivated by changes of the state of the *Control* key, in which case the *SubAction* parameter of the message in the *Input* callback will be set to *ArmedChange:*

- MOUSE_OVER (but not MOUSE_CLICK) actions with *ProcessArmed* set to ARMED_ONLY or UNARMED_ONLY are activated when the state of the *Control* key changes to a matching state while the mouse is over the object.
- MOUSE_CLICK actions get activated only on the mouse click, and only if the *Control* key is in a matching state at the time of the click.
- Active MOUSE_OVER and MOUSE_CLICK actions with *ProcessArmed* set to ARMED_ONLY or UNARMED_ONLY are deactivated if the *Control* key changes to a non-matching state.
- For MOUSE_OVER actions with *ProcessArmed*=ARMED_AND_UNARMED, an activation message with *SubAction=ArmedChange* is generated each time the state of the *Control* key changes while the mouse is over the object.

For the MOUSE_OVER actions, the deactivation message may not be generated if the mouse moves to another object, triggering an activation message for a MOUSE_OVER action attached to another object.

Messages generated when an action is activated contain information about the action, its *EventLabel* and the object the action is attached to.

Messages generated on action deactivation have an empty *EventLabel* and do not contain any additional information. If a program need this information, it can store it when the action is activated.

The viewport's *ProcessMouse* attribute has to include an appropriate *Move* and/or *Click* mask to enable custom event processing (except for actions with *Trigger*=INPUT). The *Input* callback can use the message object's resources to process the custom event. This message is invoked for any viewport object and does not require an input handler.

Custom Event Message Object Resources

Resource	Value
Format	*CustomEvent*
Action	• *MouseOver* when a MOUSE_OVER action is activated or deactivated. • *MouseClick* or *DoubleClick* when a MOUSE_CLICK action is activated, depending on the type of the click and the setting of the *ProcessDoubleClick* action attribute. • *MouseRelease* when a MOUSE_CLICK action is deactivated. • *InputObject* for actions with *Trigger*=INPUT.
SubAction	• *Armed* if the *Control* key was pressed when activating actions with *ProcessArmed* set to ARMED_ONLY or ARMED_AND_UNARMED. • *ArmedChange* if an action was activated or deactivated due to a change of the state of the *Control* key. • none (an empty string) for all other actions.
EventLabel	• When activating an action: the value of the action's *EventLabel* attribute, or the string value of the old-style (prior to v. 3.5) *MouseOverEvent* or *MouseClickEvent* properties if the custom event was generated by these properties. • When deactivating an action: none (an empty string).
ButtonIndex	• When activating MOUSE_CLICK actions: an index of the button that generated the custom event. • A value of 0 for all other actions.
ActionObject	• When activating an action: the action that generated the message. • NULL when deactivation an action, or when activating a custom event generated by the old (prior to v. 3 5) custom event properties.
Object	• When activating mouse actions: the top-level object that generated the custom event. For example, if a group object has a MOUSE_CLICK action attached, the group will be used as the *Object* resource when any object in the group is selected with the mouse. • When deactivating mouse actions: NULL. • For input actions (*Trigger*=INPUT): the input object the action is attached to.

Custom Event Message Object Resources

Resource	Value
OrigObject	• When activating mouse actions: the low-level object that generated the custom event. If a group object has a MOUSE_CLICK action attached, the *Object* resource is the group and *OrigObject* is the object inside the group that was actually selected with the mouse. • When deactivating mouse actions: NULL. • For input actions (*Trigger*=INPUT): the input object the action is attached to.

The *GetResourceObject* method may be used to get the object ID of the object and origin object from the message using the *Object* and *OrigObject* resource names:

```
GlgObject object = GlgGetResourceObject( message_object, "Object" );
```

The resources of the object and origin object may be accessed through the message object using the Standard API. For example, the *Object/FillColor* resource name may be used to access the *FillColor* attribute of the object as a resource of the message object:

```
GlgSetGResource( message_object, "Object/FillColor", 1., 0., 0. );
```

If the *ActionObject* resource of the message is not NULL, the action data may be accessed via the *ActionObject/ActionData* resource.

Command Message Object

This message object is generated by a GLG drawing when a command action is triggered by a mouse event or an input object. The viewport's *ProcessMouse* attribute has to include an appropriate *Move* and/or *Click* mask to enable processing of command actions (except for actions with *Trigger=INPUT*). The *Input* callback can use the message object's resources to process the command. This message is invoked for any viewport object and does not require an input handler.

Command Message Object Resources

Resource	Value
Format	*Command*
Action	• *MouseClick* for commands with *Trigger*=MOUSE_*CLICK*. • *MouseOver* for commands with *Trigger*=MOUSE_OVER. • *Input* for commands with *Trigger*=INPUT.
SubAction	• *Armed* when activating actions with *ProcessArmed*=ARMED_ONLY, or if the *Control* key was pressed when an action with *ProcessArmed*=ARMED_AND_UNARMED was activated. • none (an empty string) for all other actions.
EventLabel	The value of the action's *EventLabel* attribute.
ActionObject	The action that generated the message.
Object	• For commands activated by the mouse: the top-level object that generated the command. For example, if a group object has a MOUSE_CLICK command action, the group will be used as the *Object* resource when any object in the group is selected with the mouse. • For input object commands (*Trigger*=INPUT): the input object the action is attached.
OrigObject	• For commands activated by the mouse: the low-level object that originated the command. If a group object has a MOUSE_CLICK action attached, the *Object* resource is the group and *OrigObject* is the object inside the group that was actually selected with the mouse. • For input object commands (*Trigger*=INPUT): the input object the action is attached.

Tooltip Message Object

This message object is generated by a GLG drawing when a tooltip is activated or erased. The viewport's *ProcessMouse* attribute has to include *Tooltip* mask to enable object tooltips in the drawing (button tooltips do not require the mask and are always active). The *Input* callback can use the message object's resources to process the tooltip event. This message is invoked for any viewport object and does not require an input handler.

Tooltip Message Object Resources

Resource	Value
Format	*Tooltip*
Action	• *ObjectTooltip* when an object tooltip is activated, or when a non-button tooltip (object or special) is erased. • *ButtonTooltip* when a button tooltip is activated or erased. • *SpecialTooltip* when a chart or axis tooltip is activated.
SubAction	none
EventLabel	• The tooltip string for the tooltip display events • An empty string if the tooltip event is generated when the mouse moves away from the object and the tooltip is erased.
ActionObject	• The tooltip action that generated object tooltip. • NULL for old-style (prior to v. 3 5) and button tooltips, as well as when the message is generated on erasing a tooltip.
Object	• The top-level object that has a tooltip attached. For example, if a group object has a tooltip, the group will be used as the *Object* resource when a mouse hovers over any object in the group. • NULL if the message is invoked on erasing a tooltip.
OrigObject	• The low-level object the mouse is hovering at. If a group object has the custom events defined, the *Object* resource is the group and *OrigObject* is the object inside the group that is under the mouse. • NULL if the message is invoked on erasing a tooltip.

UpdateDrawing Message Object

This message object is generated by a GLG drawing when an action that changed the drawing has been triggered, such as an action with the SET_STATE, TRACE_STATE or TOGGLE_STATE action type. By default, when an action is triggered, it updates its parent viewport. If the action's *State* attribute is constrained to an object in another viewport, that viewport will not be automatically redrawn when the value of the *State* object changes. The *Input* callback can use this

message to update the top-level drawing in cases when the *State* attribute of the action is constrained to an object in another viewport. This message is invoked for any viewport object and does not require an input handler.

UpdateDrawing Message Object Resources

Resource	Value
Format	*UpdateDrawing*
Action	*Update*
SubAction	none

ImageLoad Message Object

This message object is generated by a GLG drawing in the Java, C# and JavaScript environments when an image with an asynchronous load type finishes loading its data. The *Input* callback can use this message to update the drawing. This message is invoked for any viewport object and does not require an input handler.

JavaScript Note: Image loading in JavaScript is inherently asynchronous. Static applications that do not use a continuous animation loop must handle the *ImageLoad* message in the Input callback to repaint images using the *Update* method once they have completely loaded.

ImageLoad Message Object Resources

Resource	Value
Format	*ImageLoad*
Action	*Update*
SubAction	none

TemplateLoad Message Object

This message object is generated by a GLG drawing in the JavaScript environment every time a template of a reference object gets loaded. The *Input* callback can use this message to update the drawing. This message is invoked for any viewport object and does not require an input handler.

The *GetPendingTemplates* and *GetPendingInstances* methods of the JavaScript API may be used to query the number of templates being loaded and the number of reference object instances waiting for their templates to be loaded.

TemplateLoad Message Object Resources

Resource	Value
Format	*TemplateLoad*
Action	*Update*
SubAction	none

Object Selection Message Object

This message object is generated by a GLG drawing when objects in the drawing are selected via the mouse over or mouse click events. The viewport's *ProcessMouse* attribute has include one or both of the *Click* and *Move* masks to enable object selection processing. The *Input* callback can use the message object's resources to process the object selection. This message is invoked for any viewport object and does not require an input handler.

Object Selection Message Object Resources

Resource	Value
Format	*ObjectSelection*
Action	• *MouseMove* when user moves the mouse over objects in the drawing. • *MouseClick* when the user clicks on the objects in the drawing.
SubAction	none
ButtonIndex	• The index of the button that generated the selection event. • 0 for selections generated by mouse over events.
SelectionArray	A group object containing object IDs of all objects on the lowest level of the hierarchy selected by the mouse event.

The objects selected by the mouse event may be queried from the *SelectionArray* using the *GetElement* method of the Extended API:

```
GlgObject selection_array =
   GlgGetResourceObject( message_object, "SelectionArray" );
for( i=0; i<GlgGetSize( selection_array ); ++i )
   GlgObject selected_object = GlgGetElement( selection_array, i );
```

The array contains the selected objects on the lowest level of the hierarchy. For example, if the object is part of a group, it will contain the object itself and not the group. The parents of the selected objects may be queried by using the selected object's *GetParent* method.

Chart Selection Message Object

This message object is returned by the *GlgCreateChartSelection* method and contains information describing the selected data sample. It does not have generic message object resources such as *Format* or *Action*, and contains only the resources listed in the following table.

Chart Selection Message Object Resources

Resource	Value
InputX	The X or time value corresponding to the specified input position.
InputY	The Y value corresponding to the specified input position.
SampleX	The X or time value of the selected data sample.
SampleY	The Y value of the selected data sample.
SampleValid	The value of the selected data sample's valid flag: 0 or 1.
InputXString	A string with the X or time value of the input position in the format of the X axis's label.
SampleXString	A string with the X or time value of the selected data sample in the format of the X axis's label.
SelectedPlot	An object ID of the selected plot.

Chart Message Object

This message object is generated every time a chart's cross-hair cursor is drawn or erased. When the mouse moves over the chart, a repeated draw message for each cross-hair cursor position is generated.

Chart Message Object Resources

Resource	Value
Format	*Chart*
Action	*CrossHairUpdate*
SubAction	*Draw* or *Erase*.
Object	The chart object that triggered the message.

Window Message Object

This message object is generated by a viewport object when it becomes visible for the first time or receives a message to delete or close its window. The *delete window* request is received when the user activates the window's controls to close the window. The *Input* callback gives the program a chance to react to the user's request. This message is invoked for any viewport object and does not require an input handler.

Window Message Object Resources

Resource	Value
Format	*Window*
Action	• *DeleteWindow* when user closes the viewport's window. • *FirstExposure* when the viewport's window is becoming visible for the first time.
SubAction	none
Object	The viewport that triggered the message.

7.3 Appendix C: GLG Object Attribute Table

The following lists the Default Attribute Names that may be used by a program to access attributes of various objects. The *generic attributes* are common attributes that may be used for objects of any type. The rest of the attributes exist only in certain types of objects and are listed under their corresponding object types.

Most of the attribute names are either self-explanatory or are described in the corresponding section the *GLG Objects* chapter of the GLG User's Guide. There are some attributes that are not described in the User's Guide and may be used only by a program to access some objects' internal structures. Such attributes may have an explanation next to the attribute name.

Generic Attributes

Type
: Object type.

Name

HasResources

Global

MoveMode

CoordFlag

HISetup
: Indicates if the object's hierarchy has been set up.

Xform
: An optional transformation object attached to the object.

Generic Attributes of Drawable objects

Visibility

HighlightFlag

History
: A group containing all history objects attached to the object, if any.

CustomData
: A group containing all custom properties attached to the object, if any.

NoCustomDataSetup
: A flag to disable setup of custom properties when complex objects are attached as properties.

Aliases
: A group containing all aliases attached to the object, if any.

Actions
: A group containing all actions and commands attached to the object, if any.

RenderingAttr
: The rendering object attached to the object, if any. Objects that have a rendering object attached, inherit the rendering object's attributes. In this case, rendering attributes may be queried directly from the object the rendering object is attached to.

HMIFlag
: A boolean flag that marks objects created in the HMI Configurator.

Polygon Attributes

EdgeColor

FillColor

LineWidth

LineType
FillType
OpenType
SelectionType
Shading
Controls shading of the polygon's fill and edges.
AntiAliasing
Array
A group object containing polygon points.

Rendering Object Attributes

GradientResolution
GradientType
GradientColor
GradientCenter
GradientLength
GradientAngle
ArrowType
ShadowOffset
ShadowColor
FillDirection
FillAmount

If an object has a rendering object attached, the object inherits all of the rendering object's attributes, so that they can be queried directly from the object.

Marker Attributes

MarkerType
MarkerSize
Point
Control point
EdgeColor
FillColor
AntiAliasing

Text Object Attributes

TextType
String
Point1, Point2, Point3
Control Points
TextColor
FontType
FontSize
MinFontSize
Anchoring
AnchorOffset
TextDirection
TextScaling
BoxAttr
Box Attributes object attached to the text object (if any).

CursorType
Cursor type (SCROLLED TEXT only).
CursorPosition
Cursor position (SCROLLED TEXT only).
TextStart
Start index of the first visible character (SCROLLED TEXT only).

The text object also inherits attributes from its attached Box Attributes object. These attributes may be queried directly from the text object.

Box Attributes Object Attributes

BoxOffset
AnchorOnBox

FillColor
EdgeColor
LineWidth
LineType
FillType
Generic resource names

BoxEdgeColor
BoxFillColor
BoxLineWidth
BoxLineType
BoxFillType
Box-specific resource names

Arc Attributes

Radius
StartAngle
EndAngle
AngleType
Center
Center control point.
Vector
Vector control point.
Resolution, ArcResolution
ArcFillType
FillType
OpenType
EdgeColor
FillColor
LineWidth
LineType
SelectionType
Shading
AntiAliasing
Polygon
The polygon object used to render the arc.

Parallelogram Attributes

ParallType
EdgeColor
FillColor
LineWidth
LineType
FillType
SelectionType
Shading
Controls shading of the object's fill and edges.
AntiAliasing
Polygon
The polygon object used to render the parallelogram.

Spline Object Attributes

SplineType
Resolution, SplineResolution
EdgeColor
FillColor
LineWidth
LineType
FillType
SelectionType
Shading
Controls shading of the object's fill and edges.
CPArray
A group containing the spline's control points.
Polygon
The polygon object used to render the spline.

Rounded Object Attributes

Radius1
Radius2
UnitType
Resolution, CornerResolution
EdgeColor
FillColor
LineWidth
LineType
FillType
SelectionType
Shading
Controls shading of the object's fill and edges.
AntiAliasing
Polygon
The polygon object used to render the object.

Image Object attributes

ImageType
Anchoring
ImageFile

EnableCache
AsyncMode
Java, C# and JavaScript only.
ImageLoaded
Java, C# and JavaScript only.
Point1, Point2
Control points.

GIS Object attributes

FillColor
GISDisabled
GISProjection
GISCenter
GISUsedCenter
The actual center value used by the map server (read-only).
GISExtent
GISUsedExtent
The actual extent value used by the map server (read-only).
GISAngle
GISStretch
GISDataFile
GISMapServerURL
GISLayers
GISVerbosity
GISDiscardData
AsyncMode
Java, C# and JavaScript only.
ImageLoaded
Java, C# and JavaScript only.
Point1, Point2
Control points.

Group and List Objects' Attributes

ZSort

Connector Object Attributes

EdgeType
EdgeDirection
EdgeColor
LineType
CPArray
A group containing the control points.
Rendering
The polygon or arc object used to render the connector.

Reference Object Attributes

ReferenceType
Source
SourcePath (previously *DrawingFile*)
A path for the drawing file or resource path for a palette object in the drawing.

ObjectPath
Resource path for the object in the template, may contain an origin resource path after a colon.
Point
Control point.
Reference
The template object.
Origin
The template's origin.
CloneType
EnableCache
KeepEditRatio (SubWindow only)
ContainerXform
The transformation used to scale or otherwise transform the reference's instance.
AttachmentArray
An optional array of the reference object's attachment points.
Bindings
An optional array of bindings for rebinding instance's attributes.

Series Object Attributes

Factor
Limit
CloneType
LogType
ZSort
Inversed
Persistent
Shift
Controls instance positioning. Changing the value of the attribute in the range of 0 to 1 may be used to animate the series by moving its instances along the path.
Path
The transformation object used as a path.
StartOrigin
The first point of the path transformation.
Origin
The template's origin point.
Template
The template object.
InstanceArray
A group containing template instances.

Square Series Object Attributes

Rows
Columns
Limit
ZSort
ColumnsFirst
CloneType
Persistent
KeepEditRatio
Point1, Point2, Point3
Control points.

Template
The template object.
InstanceArray
A group containing template instances.

Polyline Attributes

Factor
DrawMarkers
DrawLines
Segments
CloneType
Inversed
Path
The transformation object used as a path.
Polygon
The line or segment's template.
Point
The marker's template.
Points
A group containing marker instances.
InstanceArray
A group containing segment instances.

The polyline object also inherits attributes of its polygon template. These attributes may be accessed directly by querying the polyline.

Polysurface Attributes

Rows
Columns
DrawMarkers
DrawLines
ZSort
CloneType
Polygon
The surface segment's template.
Point1, Point2, Point3
Control points.
Point
The marker's template.
Points
A group containing marker instances.
InstanceArray
A group containing segment instances.

The polysurface object also inherits attributes of its polygon template. These attributes may be accessed directly by querying the polysurface.

Frame Object Attributes

FrameType
FrameFactor

CPArray
A group containing the frame's control points.
CNArray
A group containing the frame's constrained points.

Viewport Attributes

Point1, Point2
Control points.
EdgeColor
FillColor
LineWidth
Pan
ActivePan
Read-only attribute, contains a bit mask composed of the GLG_PAN_X and GLG_PAN_Y binary flags indicating which scrollbars are currently displayed.
GlgPanX
The horizontal scrollbar (viewport) of the integrated panning and zooming (if PanX is enabled).
GlgPanY
The vertical scrollbar (viewport) of the integrated panning and zooming (if PanY is enabled).
GlgPanSpacer
The spacer viewport of the integrated panning and zooming used in the lower right corner (if both PanX and PanY scrollbars are enabled).
ZoomEnabled
ProcessMouse
Enables mouse event processing.
Handler
A function object attached to the viewport as a handler.
ZSort
DisableInput
Disables input events in the viewport.
KeepEditRatio
OwnsInputCB
ScrollbarType
ZoomFromParent
JavaScriptFile
BaseWidth
A base width used for scaling text objects in the viewport.
ZoomFactor
Read-only attribute which may be queried to get the viewport's current zoom factor.
ZoomToMode
Read-only attribute which may be queried to get the current state of multi-state zoom and pan actions: GLG_ZOOM_TO_STATE or GLG_PAN_DRAG_STATE. It may also contain the GLG_X_STATE or GLG_Y_STATE binary flags ORed with the return value to indicate ZoomToX / ZoomToY actions or the direction of the pan dragging.
ZoomModeObject
Read-only attribute which may be queried to get the chart or GIS object used for the Chart or GIS Zoom Mode.
XYRatio
Read-only attribute which may be queried to get the X/Y ratio of the viewport's window.
InnerWidth

InnerHeight
Read-only attributes which may be queried to get the viewport's inner width and height.
Screen
The screen object associated with the viewport.
Light
Light object, if any.
Array
The group used by the viewport to contain the objects drawn in the viewport. The view and zoom transformations are attached to this group.

The viewport also inherits most of the screen object's attributes and all attributes of the light object attached to the viewport, so that they may be queried directly from the viewport.

Screen Object Attributes

OpenGLHint
A requested OpenGL mode.
OpenGL
The current OpenGL mode (read-only).
DoubleBuffering
HTMLBackground
CoordSystem
Stretch
PushIn
SpanX
SpanY
ShellType
WidgetType
ShadowWidth
AntiAliasing
Controls anti-aliasing for the screen's viewport. The default is ON.
Colortable
Fonttable
FonttableFile
A filename of a GLG drawing containing a font table to be used for the viewport's drawing.
ScreenName
Window title when the screen is used as a top-level window.
ExactColor
GridInterval
DefaultLabelColor
PreciseSize
ForceOpenGL
ScreenName
For screens of top-level viewports: Title to be used for the viewport's window.
XHint
YHint
WidthHint
HeightHint
For screens of top-level viewports with both of viewports control points set to (0,0,0): requested position and dimensions of the viewport on initial appearance.
Width
Height
The current width and height of the screen's viewport (read-only).

Widget
Returns the Widget ID of the native widget used to render the screen and viewport.
Shell
Returns the Widget ID of the highest level parent.
Drawable
The drawable used for rendering. Returns the ID of the double-buffering pixmap when double-buffering is enabled.
Display
Returns the Display pointer (legacy X11 version on Linux/Unix only).
DisplayName
Returns the display name.
TrueColor
Read-only attribute, is set to a non-zero value when the screen is displayed on a TrueColor display.

The screen also inherits most of the colortable object's attributes which may be queried directly from both the screen and viewport object.

Light Viewport Attributes

Point1, Point2
Control points.
Background
A *Line Attributes* object containing attributes used to render the light viewport's background.
FillColor
EdgeColor
LineWidth
LineType
FillType
Generic resource names for attributes inherited from *Background*.
ShadowWidth
CoordSystem
Stretch
PushIn
SpanX
SpanY
GridInterval
ZoomEnabled
ProcessMouse
Enables mouse event processing.
Handler
A function object attached to the viewport as a handler.
DisableInput
Disables input events in the viewport.
ZSort
KeepEditRatio
OwnsInputCB
ScrollbarType
ZoomFromParent
Clipping
BaseWidth
A base width used for scaling text objects in the viewport.

ZoomFactor
Read-only attribute which may be queried to get the viewport's current zoom factor.

ZoomToMode
Read-only attribute which may be queried to get the current state of multi-state zoom and pan actions: GLG_ZOOM_TO_STATE or GLG_PAN_DRAG_STATE. It may also contain the GLG_X_STATE or GLG_Y_STATE binary flags ORed with the return value to indicate ZoomToX / ZoomToY actions or the direction of the pan dragging.

ZoomModeObject
Read-only attribute which may be queried to get the chart or GIS object used for the Chart or GIS Zoom Mode.

XYRatio
Read-only attribute which may be queried to get the X/Y ratio of the viewport's window.

InnerWidth
InnerHeight
Read-only attributes which may be queried to get the viewport's inner width and height.

Array
The group used by the viewport to contain the objects drawn in the viewport. The view and zoom transformations are attached to this group.

The light viewport also inherits attributes of its background object, so that they may be queried directly from the light viewport.

Chart Object Attributes

Point1, Point2
Control points.

NumPlots
NumYAxes
NumLevels
NumTimeLines
NumAnnotations
CommonRange
AutoScroll
AutoScale
DrawGrid
Persistent
SortInput
BufferXSpan
BufferSize
YAxesOffset
TooltipMode
TooltipFormat
DrawOrder
DrawCrossHair

Plots
A group containing the chart's plots.

Levels
A group containing the chart's level objects.

TimeLines
A group containing the chart's time lines.

Annotations
A group containing the chart's annotations.

XAxis
The chart's X axis.
YAxisGroup
A group containing the chart's Y axes.
YAxis
The first Y axis of the chart.
Background
A *Line Attributes* object containing attributes of the chart's drawing area.
Grid
A *Line Attributes* object containing attributes of the chart's grid.
SelectionMarker
A *Marker* object used for annotating the selected data sample.
SweepBar
A *Line Attributes* object containing attributes of a sweep bar.
SweepPosition
The current sweep bar position in the units of the X axis.
CrossHair
A *Line Attributes* object containing attributes of the chart's cross-hair cursor.
DrawSelected
Controls drawing of the selection marker; may be reset to 0 at run time to erase the selection marker for a previous selection.
LegendContainer
The chart's legend, or the legend's parent viewport if the legend is placed in a separate viewport.
SelectedDataSample
A pointer to the *GlgDataSample* structure, may be used to query information about the selected sample at run time.

Plot Object Attributes

PlotType
Enabled
PlotAnnotation
YLow
YHigh
ValueEntryPoint
TimeEntryPoint
ValidEntryPoint
AutoScaleDelta
LinkedAxis
An object ID of the linked axis.
FilterType
FilterPrecision
FilterMarkers
IncludeZero
ExtendedData
CacheUse
RangeLock
NumSamples
Read-only resource for querying the number of accumulated samples.
EdgeColor
FillColor
Fill color of a bar plot.

FillType
Fill type of a bar plot.
LineWidth
LineType
Opacity
AntiAliasing
BarWidth
The width of bars in a bar plot.
BarYLow
The level of the bars' lower edge in a bar plot.
LineAttr
The *Line Attributes* object used to keep rendering attributes of the plot. The plot inherits the *EdgeColor*, *FillColor*, *LineWidth* and other line and bar rendering properties from this object.
Marker
Defines marker attributes.
Array
An internal group containing the plot's data samples. Data samples are represented by the *GlgDataSample* structure.
Counter
Keeps the incrementing current sample index for scrolling index charts.

Level Line Object Attributes

Level
YLow
YHigh
Enabled
RangeLock
EdgeColor
LineWidth
LineType
Opacity
AntiAliasing
LineAttr
The *Line Attributes* object used to keep rendering attributes of the level line. The level line inherits the *EdgeColor*, *LineWidth* and other line resources from this object.
LinkedAxis
An object ID of the linked axis.

Annotation Object Attributes

AnnotationType
PositionX
PositionY
AbsoluteX
AbsoluteY
YLow
YHigh
Enabled
RangeLock
LinkedAxis
An object ID of the linked axis.
LabelFormat

Label
The *Text* object used to render the annotation's label.
Marker
The *Marker* object used to render the annotation's marker.

Legend Object Attributes

Point1, Point2
Control points.
LayoutType
AutoLayout
Anchoring
RowAnchoring
MinRowSize
MaxRowSize
DisplayValue
ValueFormat
LineLength
MinLineWidth
BarHeight
XOffset
YOffset
XSpacing
YSpacing
LabelXOffset
LabelYOffset
LabelMaxWidth
LabelMaxHeight
Label
A *Text* object whose attributes are used to render the legend's labels.
Background
A *Line Attributes* object that defines rendering attributes of a background box drawn around the legend.

Axis Object Attributes

Point1, Point2
Control points.
AxisType
AxisPosition
Inversed
DrawOutline
RangeLock
Low (range axis only)
High (range axis only)
EndValue (scrolling axis only)
Span (scrolling axis only)
RulerStart (ruler axis only)
RulerScale (ruler axis only)
LabelFormat (non-time axis only)
TimeFormat (time axis only)
MilliSecFormat (non-time axis only)
MajorInterval
MinorInterval

AxisLabelString
LowOffset (non-time axis only)
HighOffset (non-time axis only)
RoundedPlacement
FixLeapYears (time axis only)
MajorTickSize
MinorTickSize
LabelOffset
LabelExtentRel
LabelExtentAbs
MajorOffset
TimeOrigin (relative time axis only)
TooltipFormat
AxisLabelOffset
AxisLabelPosition
AxisLabelAnchoring
Tick
A *Line Attributes* object that defines rendering attributes of the axis's minor and major ticks, as well as the attributes of the outline.
TickLabel
A *Text* object that defines rendering attributes used to draw the major tick labels.
AxisLabel
A *Text* object that used to draw the axis's label.
LabelFormatter
A pointer to the label formatting function associated with the axis.

Line Attributes Object

EdgeColor
LineWidth
LineType
FillType
FillColor
Opacity
AntiAliasing

Light Object Attributes

LightType
LightPoint
LightDirection
LightCoefficient
AmbientCoefficient

Colortable Object Attributes

ColortableType
ColorFactor
The requested number of colors.
NumColors
The actual number of colors.
GradeHint
The requested number of color grades.

NumGrades
The actual number of color grades.
PatternFactor
The requested number of dithering patterns.
NumPatterns
The actual number of dithering patterns.
RenderColor
The color to use for B&W rendering (when NumColors=1).
ColorCorrection
If set to True, increases the brightness of dark colors.
PastelLevel
A pastel color coefficient with a value between 0 and 1.

Font Table Object Attributes

NumFontTypes
NumFontSizes
FontArray
A group used to hold the font table's fonts

Font Object Attributes

FontName
XFontName
WinFontName
JavaFontName
FontPSName
MBFlag
Multi-byte flag.
FontCharset

Transformation Object Attributes

XformType
XformAttr1
The first attribute object (any type).
XformAttr2
The second attribute object (any type).
XformAttr3
The third attribute object (any type).
XformAttr4
The fourth attribute object (any type).
XformAttr5
The fifth attribute object (any type).
XformAttr6
The sixth attribute object (any type).
XformAttr7
The seventh attribute object (any type).
Matrix
The transformation's matrix object.
HasDynProperties
A boolean flag that controls display of predefined transformation properties.

Action Object Attributes

ActionType
Trigger
ProcessArmed
ProcessDoubleClick
MouseButton
Enabled
Tooltip
Tooltip string for tooltip actions.
StateObj
The object whose value is to be set for mouse feedback actions.
EventLabel
ActionData
Action data for SEND_EVENT actions.
Command
Command data for SEND_COMMAND actions.

Alias Object Attributes

Alias
The alias name.
ResPath
The resource path to associate with the alias.

History Object Attributes

ScrollType
VarName
EntryPoint
Inversed
RollBack
HistoryIndex
The index of the current scroll iteration.

Data Object Attributes

DataType
Value
The value of the data object, accessed using NULL as the resource name.
XfValue
The transformed value (value transformed with an xform attached to the data object; in world coordinates for control points).
UTF8Encoding (S data objects only)
AlwaysChanged
TagObject
An optional tag object attached to the data object. If a tag object is attached, the data object inherits the *TagName*, *TagSource*, *TagEnabled* and other tag attributes from the tag object.

Attribute Object Attributes

DataType
DataRole
Creation type resource that determines the type of transformations that will affect the attribute.

Value
The value of the attribute object, accessed using NULL as the resource name.
XfValue
The absolute value (in screen coordinates for drawable control points).
UTF8Encoding (S attribute object only)
AlwaysChanged
TagObject
An optional tag object attached to the attribute object. If a tag object is attached, the attribute object inherits the *TagName*, *TagSorce* and *TagComment* attributes from the tag object.
ExportTag
An optional export tag attached to the attribute object to define a public property.
DataObject
The base data object used for storing the attribute's value.

Tag Object Attributes

TagType
Identifies the tag as a data or export tag (OEM).
TagName
Used to persistently identify the tag when browsing.
TagSource (data tags only)
Defines the mapping to a database field.
Category (export tags only)
Defines the public property category.
TagComment
Contains any user-defined information.
TagEnabled (data tags only)
TagAccessType (data tags only)
May be used to designate the tag as output-only.

Function Object Attributes

FunctionName
The name of the function object (*GlgSlider*, *GlgButton*, etc.).

7.4 Appendix D: Global Parameters

The following lists names of global parameters used in the *GlgSetLParameter* and *GlgGetLParameter* functions:

- *"GlgReadOnlyStrings"* - controls if strings are writable (0) or read-only (1). Strings are read-only by default. May be changed only before the first call to *GlgInit*, and only if a static library is used.
- *"GlgFastAlloc"* - if set to 0, disables the GLG memory allocator and uses the system memory allocator. May be changed only before the first call to *GlgInit*, and only if a static library is used.
- *"GlgAllocBlockSize"* - controls the size of memory blocks requested from the system (1MB by default). May be changed only before the first call to *GlgInit*, and only if a static library is used.
- *"GlgMaxAllocSizeIndex"* - controls the maximum size of memory allocation requests handled by the GLG memory allocator. The maximum size is calculated by multiplying the value of the parameter by 8 (size of a double). Requests with the size greater or equal to the maximum size are passed to the system memory allocator. The default maximum size is 1KB. May be changed only before the first call to *GlgInit*, and only if a static library is used.
- *"GlgAllocBlockBytesLeft"* (read-only) - indicates the number of bytes left in the current memory block.
- *"GlgUsedAllocBlockCount"* (read-only) - contains the number of used memory blocks.
- *"GlgSpareAllocBlockCount"* (read-only) - contains the number of spare preallocated memory blocks.
- *"GlgSmallAllocCount"* (read-only) - contains the total number of memory allocations with sizes less than the maximum size controlled by the by *GlgMaxAllocSizeIndex*.
- *"GlgBigAllocCount"* (read-only) - contains the total number of memory allocations with sizes bigger than the maximum size.
- *"GlgAllocCount"* (read-only) - contains the total number of all memory allocations.
- *"GlgSmallAllocSize"* (read-only) - contains the combined size of all memory allocations with sizes less than the maximum size.
- *"GlgBigAllocSize"* (read-only) - contains the combined size of all memory allocations with sizes bigger than the maximum size.
- *"GlgAllocSize"* (read-only) - contains the combined size of all memory allocations.

Index

B

C

D

E

F

G

H

I

S

T