## Contents

1 Overview ........................................................................................................................... 1

1.1 Introduction ...................................................................................................... 1

1.2 WindML Architecture ...................................................................................... 2
  1.2.1 The SDK ............................................................................................... 3
  1.2.2 The DDK .............................................................................................. 3

1.3 WindML Distribution ...................................................................................... 5
  1.3.1 Prerequisites ........................................................................................ 5
  1.3.2 Documentation ................................................................................... 5

1.4 Customer Services ............................................................................................ 7

2 WindML Configuration ..................................................................................................... 9

2.1 Introduction ...................................................................................................... 9
  2.1.1 Comparison of Configuration Methods ......................................... 10

2.2 Basic Configuration .......................................................................................... 11
  2.2.1 Configuring a Graphics Device ........................................................ 12
  2.2.2 Configuring a Keyboard Device ...................................................... 12
  2.2.3 Configuring a Pointer Device ........................................................... 13
2.8.2 Adding WAVE File Support ............................................................. 47

3 2D Graphics Library .................................................................................................. 49

3.1 Introduction ...................................................................................................... 49

3.2 Drawing Primitives .......................................................................................... 50
3.2.1 Using the API ..................................................................................... 50
3.2.2 Example Code .................................................................................... 51

3.3 Text Rendering and Font Management ......................................................... 52
3.3.1 Using the API ..................................................................................... 53
3.3.2 Example Code .................................................................................... 56

3.4 Bitmap Management ........................................................................................ 57
3.4.1 Using the API ..................................................................................... 58
3.4.2 Example Code .................................................................................... 59

3.5 Cursor Management ........................................................................................ 62
3.5.1 Using the API ..................................................................................... 63
3.5.2 Example Code .................................................................................... 65

3.6 Batch Drawing .................................................................................................. 67
3.6.1 Example Code .................................................................................... 67

3.7 Graphics Contexts ............................................................................................ 68
3.7.1 Example Code .................................................................................... 72

3.8 Color Management ........................................................................................... 73
3.8.1 Color Types ......................................................................................... 74
3.8.2 Color Formats ..................................................................................... 75
3.8.3 Using the API ..................................................................................... 76
3.8.4 Example Code .................................................................................... 79
3.9 Double Buffering ................................................................. 82
3.9.1 Example Code ............................................................... 82

4 Event Services ........................................................................... 85
4.1 Introduction .............................................................................. 85
4.2 Event Service Components .................................................. 86
4.3 Using the API ............................................................................ 88
  4.3.1 WindML Events ................................................................. 88
  4.3.2 Event Handling ................................................................. 89
  4.3.3 Event Queues ................................................................. 89
4.4 Example Code ........................................................................... 90

5 Region and Window Management ........................................... 93
5.1 Introduction .............................................................................. 93
5.2 Regions and Clipping ............................................................. 93
  5.2.1 Clipping Regions ............................................................... 95
  5.2.2 Using the API ................................................................. 95
  5.2.3 Example Code ............................................................... 96
5.3 Window Management ............................................................. 97
  5.3.1 Containment and Z-Ordering .......................................... 97
  5.3.2 Region Management Algorithms - Painters and Clip Children . 99
  5.3.3 Event Routing ............................................................... 101
  5.3.4 Window Creation ............................................................ 103
  5.3.5 Window Drawing ............................................................. 104
  5.3.6 Window Event Processing ........................................... 106
  5.3.7 Input Event Routing ....................................................... 108
7.4.2 Registering Devices and Services .................................................... 130
7.4.3 Obtaining a Device ID ....................................................................... 131
7.4.4 Removing a Device from the Registry ............................................ 132
7.5 Overlay Surfaces .............................................................................................. 132
7.5.1 Creating an Overlay .......................................................................... 133
7.5.2 Accessing an Overlay ........................................................................ 135
7.5.3 Moving an Overlay ........................................................................... 136
7.6 Driver Information and Control .................................................................... 136
7.6.1 Querying a Driver with uglInfo() ................................................... 136
7.6.2 Controlling a Font Driver ................................................................. 138
7.6.3 Controlling an Input Driver ............................................................. 139
7.6.4 Setting the Display Mode ................................................................. 140

8 Graphics Device Drivers ................................................................................................. 143
8.1 Introduction ...................................................................................................... 143
8.1.1 The Generic Drivers .......................................................................... 144
8.1.2 The UGL Graphics Interface (UGI) API ......................................... 145
8.1.3 The Generic Driver API .................................................................... 147
8.2 Writing a Graphics Driver .............................................................................. 148
8.2.1 Initial Planning and Design ............................................................. 148
8.2.2 Creating the Graphics Driver Files .................................................. 150
8.2.3 Implementing the Device Creation Routine (xxxDevCreate()) .. 151
8.2.4 Implementing the Destroy Routine (xxxDevDestroy()) ............... 156
8.2.5 Implementing the Info Routine (xxxInfo()) .................................. 157
8.2.6 Implementing the Mode Get and Set Routines ............................. 159
8.2.7 Implementing the CLUT Get and Set Routines ............................ 164
8.2.8 Implementing the Frame Buffer Pixel Manipulation Routines .. 165
8.2.9 Packaging a Graphics Driver - For Third Party Developers 168

9 Advanced Graphics Drivers Topics ................................................................. 171
   9.1 Introduction .............................................................................................. 171
   9.2 Clipping ..................................................................................................... 171
   9.3 Double Buffering ...................................................................................... 174
   9.4 Cursor Management ................................................................................. 175
   9.5 Overlay Surfaces ....................................................................................... 176
   9.6 Acceleration ............................................................................................ 177
      9.6.1 Implementing Acceleration ...................................................... 178

10 Input Drivers ................................................................................................. 179
   10.1 Introduction ............................................................................................ 179
   10.2 The Input Driver API ............................................................................... 180
   10.3 Writing an Input Driver ........................................................................... 182
      10.3.1 Implementing the Initialize Routine (xxxDevInit()) .............. 183
      10.3.2 Implementing the Destroy Routine (xxxDestroy()) .............. 185
      10.3.3 Implementing the Format Routine (xxxFormatter()) .......... 186
      10.3.4 Implementing the Info Routine (xxxInfo()) ............................. 188
      10.3.5 I/O System Drivers ...................................................................... 189
      10.3.6 Packaging an Input Driver - For Third Party Developers .... 190

11 Audio Drivers ................................................................................................. 191
   11.1 Introduction ............................................................................................. 191
   11.2 Writing an Audio Driver ......................................................................... 193
      11.2.1 Writing the xxxSndDevCreate() Routine ............................... 193
11.2.2 DSP Device ......................................................................................... 197
11.2.3 Mixer Device ...................................................................................... 199

12 Font Drivers and Font Engines ...................................................................................... 201
12.1 Introduction ...................................................................................................... 201
12.2 Writing a Font Driver ...................................................................................... 202
  12.2.1 The Font Driver Creation Routine .................................................. 203
  12.2.2 The Font Driver Destroy Routine .................................................... 205
  12.2.3 Querying Font Driver Information ................................................. 205
  12.2.4 Creating a Font ................................................................................... 206
  12.2.5 Enumerating the Fonts on the System ............................................ 207
  12.2.6 Text Rendering ................................................................................... 207
12.3 Packaging a Font Driver - For Third Party Developers .............................. 209

13 Graphic Driver Extensions .............................................................................................. 211
13.1 Introduction ...................................................................................................... 211
13.2 Adding an Extension ....................................................................................... 211
13.3 Deleting an Extension ...................................................................................... 213
13.4 Example - Using the Video Extension ........................................................... 213
13.5 Implementing the Video Extension Driver API .......................................... 214

14 Portability ......................................................................................................................... 217
14.1 OS Abstraction Layer ...................................................................................... 217

15 WindML Source Conventions ............................................................................... 221
15.1 Introduction ...................................................................................................... 221
Contents

15.2 Directory Contents ........................................................................................................ 221
15.3 WindML Software Conventions ............................................................................... 227
  15.3.1 Directory Naming Conventions ........................................................................ 227
  15.3.2 File Naming Conventions ................................................................................ 228
  15.3.3 Routine Naming Conventions ......................................................................... 228
15.4 Working with WindML Source Code ....................................................................... 231

A Standard Drivers ........................................................................................................ 235
A.1 Introduction ............................................................................................................. 235
A.2 VGA Driver ............................................................................................................. 235
A.3 ARM Driver ............................................................................................................ 236
A.4 Chips and Technologies Driver ............................................................................ 238
A.5 Tvia IGS5050 Driver ............................................................................................ 239
A.6 MediaGX Driver ................................................................................................... 241
A.7 Motorola M821 Driver ......................................................................................... 242
A.8 PC BIOS Driver ..................................................................................................... 243
A.9 Q2SD Driver .......................................................................................................... 244
A.10 SimNT Driver ....................................................................................................... 245
A.11 SimSolaris Driver ............................................................................................... 246

B 2D Graphics API ........................................................................................................ 249

C Event API ................................................................................................................ 339

D Windowing API ......................................................................................................... 355
E  Extensions API ................................................................................................................. 383
F  UGI API ............................................................................................................................. 399
G  Font List ............................................................................................................................ 403
  G.1 Bitmap Fonts Supplied with WindML ................................................................. 403
Overview

1.1 Introduction

The Wind Media Library (WindML) supports multimedia applications running on embedded systems. WindML is designed to provide foundation graphics, video, and audio technology for a variety of operating systems and a framework within which to develop standardized custom device drivers.

WindML provides a high degree of architectural and hardware independence for application code. This portability results from WindML’s modular design, which isolates all hardware-specific functionality in a discrete set of API libraries.

The WindML API libraries provide a consistent interface to graphics hardware across a wide range of CPU architectures and operating systems. WindML also provides facilities to handle input devices and process events from input devices.

WindML has been implemented with the following design goals:

- **Simplicity.** Provide a robust set of graphics primitives, and basic video and audio functionality.
- **Hardware Portability.** Execute across a matrix of architectures.
- **OS Portability.** Be portable across RTOS environments.
- **Ease of driver development.** Provide developers a robust mechanism for developing custom drivers for their applications.
1.2 WindML Architecture

WindML consists of two components—a software development kit (SDK), and a driver development kit (DDK):

- The SDK component is used to develop an application. It provides a comprehensive API set in the areas of graphics, input handling, multimedia, font, and memory management, allowing developers to write hardware independent portable code for a variety of hardware platforms.

- The DDK component is used to implement drivers. It provides a complete set of reference drivers for common hardware configurations, and an API set that enable the developer to quickly bootstrap new drivers from supplied generic drivers.

The DDK is extensible and customizable. The complete source code base is shipped with the expectation that developers will do further customization.

WindML has a multi-layered architecture that abstracts functionality into different layers, see Figure 1-1.

Figure 1-1 Layered Architecture

Application

(graphics, audio, video, and event services)

Output Drivers

Input Drivers

Output Device (e.g., Monitor)

Input Device (e.g., Mouse)

VxWorks

SDK

(Hardware Independent Layer)

DDK

(Hardware Dependent Layer)

Hardware
1.2.1 The SDK

This layer defines an interface between the application code and the lower level hardware specific drivers, so that applications can be developed independently of the underlying hardware.

The SDK provides the following API sets:

- Resource initialization and cleanup routines for graphics chips
- Multimedia API
  - 2D Graphics
  - Region Management
  - Windowing
  - Color Management
  - Overlay Support
  - Video Support
  - JPEG Support
  - Audio
- Event Services
- Memory Management
- Extensions API
- Device Management

The API libraries and routines are fully documented online in HTML format. You can access the online documentation from the Tornado>Help>Manuals menu. The API routines are also documented in the appendices of this book.

1.2.2 The DDK

The DDK is the intermediate layer between the high level SDK and the hardware. It interfaces directly with the application’s target hardware devices, including monitors, video, audio, keyboard, mouse, and so on.

For most SDK APIs, an associated driver level structure and API is defined. For example, the SDK primitive uglRectangle(), which draws a rectangle to the
display device, accesses a driver level structure that contains a field called \textbf{rectangle}. This field is a function pointer that performs a rectangle drawing operation for the specific target hardware. Similar driver function pointers are available for most primitive drawing operations.

The following driver categories are defined within WindML:

- **graphics**
  
  This driver contains function pointers and variables used to allocate colors, perform primitive drawing operations, memory allocation, and overlay page management. Examples include VGA, BIOS, MediaGx, and IGS graphics drivers.

- **video**
  
  This driver is implemented as an extension to the graphics driver (it allocates space in the extension portion of the driver structure). It implements functionality for video start, stop, and streaming operations. An example of this type of extension to the graphics driver is the IGS video extension.

- **fonts**
  
  This driver contains function pointers and variables for font size and rendering operations. It uses the graphics driver for display information. Examples include BMF and AGFA font engines.

- **input**
  
  This driver contains function pointers and variables used to obtain and format input information. Examples include keyboard, mouse, and touchscreen drivers.

- **audio**
  
  Audio is not currently a true device driver, but rather implements the Open Sound System\textsuperscript{TM} (OSS) Free technology, which uses \texttt{open()}, \texttt{close()} and \texttt{ioctl()} calls.

Specific information relating to the creation and operation of these drivers is contained in later chapters.
1.3 WindML Distribution

It is not feasible for WindML to provide “out of the box” support for all available graphics hardware. WindML has been developed with the goal of supporting a diverse set of drivers that represent a cross-section of graphics driver types. Examples range from dedicated graphics processors, CPUs with on-chip LCD controllers, and various input drivers.

The WindML distribution consists of a documentation set and a software distribution. You can download the latest version of the WindML software and online documentation, and access the graphics driver repository from the following World Wide Web site:

http://www.windriver.com/csdocs/product/ugl

Access to this Web site is provided by your local Wind River Sales Representative.

1.3.1 Prerequisites

You should be familiar with the Wind River Tornado development environment and development tools, such as C compilers, debuggers, and build utilities.

Users of the DDK portion of WindML should have a working knowledge of low-level C programming and assembly languages, familiarity with device drivers, and general experience interfacing software with hardware.

1.3.2 Documentation


- WindML Programmer’s Guide (this book in hard-copy, PDF, and HTML)
- WindML Release Notes

Additional API library and routine documentation is available online in HTML format from the Tornado>Help>Manuals menu.
Documentation Conventions

This document adheres to certain conventions for cross references, path names, and typography.

Cross-References

Cross-references in this guide are to a named section elsewhere in the guide.

Path Names

In this manual, forward slashes are used as path name delimiters for both UNIX and Windows file names.

Typographical Conventions

This manual uses the conventions shown in Table 1-1 to differentiate various elements. Parentheses are always included to indicate a subroutine name, as in `printf()`.

Table 1-1 Font Usage for Special Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>files, path names</td>
<td>/etc/hosts</td>
</tr>
<tr>
<td>libraries, drivers</td>
<td>uglLib, nfsDrv</td>
</tr>
<tr>
<td>host tools</td>
<td>more, chkdsk</td>
</tr>
<tr>
<td>subroutines</td>
<td>uglColorAlloc()</td>
</tr>
<tr>
<td>boot commands</td>
<td>p</td>
</tr>
<tr>
<td>code display</td>
<td>main ()</td>
</tr>
<tr>
<td>keyboard input</td>
<td>make CPU=MC68040 ...</td>
</tr>
<tr>
<td>display output</td>
<td>value = 0</td>
</tr>
<tr>
<td>user-supplied parameters</td>
<td>name</td>
</tr>
<tr>
<td>constants</td>
<td>INCLUDE_NFS</td>
</tr>
<tr>
<td>C keywords, cpp directives</td>
<td>#define</td>
</tr>
<tr>
<td>named key on keyboard</td>
<td>RETURN</td>
</tr>
<tr>
<td>control characters</td>
<td>CTRL+C</td>
</tr>
</tbody>
</table>
1.4 Customer Services

For WindML technical support, contact the Customer Services Department at Wind River Systems:

Phone: (800) 545-WIND
Phone: (510) 748-4100
E-mail: support@windriver.com
WWW: http://www.windriver.com

You should receive detailed instructions on accessing the technical support Web site from your local WindRiver Sales Representative.
2 WindML Configuration

2.1 Introduction

The WindML library must be configured and built before it can be used for the first time. WindML configuration includes:

- the selection and hardware setup of an output driver
- the selection and hardware setup of input drivers, such as mouse, keyboard, or touchscreen
- the selection and hardware setup of an audio driver
- the fonts to be used by the application

In addition to configuration requirements, WindML can be customized to support a specific environment. Customizable elements include memory management features and custom hardware. After configuration, the WindML libraries must be compiled and linked with your application, and optionally, linked with the VxWorks image.

There are two ways to configure WindML:

- using the WindML configuration tool (available from the Tornado>Tools>WindML menu item)
  
  This is the primary method of configuring WindML and its associated drivers.

- by editing configuration header and source files directly (command line method) to perform customization that is not possible with the configuration tool
This allows you to perform customization that is not possible with the configuration tool. Typically, the configuration tool is sufficient for most situations.

The configuration method you use is determined by the extent of WindML customization needed for your target application. The standard configuration of WindML is that it includes one graphics device, one keyboard device, and one pointer device. The configuration tool allows you to configure this standard set of devices. If you use multiple devices, you need to do additional configuration by editing the configuration files directly.

If you are using WindML on a BSP that has not been enhanced to support WindML device drivers, you may also need to modify the BSP, see 2.5 VxWorks BSP Requirements, p.30.

Once, WindML has been configured, you can use the Tornado project facility to add WindML to the VxWorks image.

### 2.1.1 Comparison of Configuration Methods

The configuration methods are summarized in Table 2-1. For first-time installation and configuration, it is recommended that you use the configuration tool.

<table>
<thead>
<tr>
<th>Configuration Method</th>
<th>Selection Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration tool: GUI-based</td>
<td>Allows you to select device driver optional features.</td>
</tr>
<tr>
<td></td>
<td>Enforces valid configurations.</td>
</tr>
<tr>
<td></td>
<td>Allows you to select specific fonts.</td>
</tr>
<tr>
<td></td>
<td>Saves multiple configurations.</td>
</tr>
<tr>
<td></td>
<td>Only allows standard WindML configuration.</td>
</tr>
<tr>
<td>Configuration file editing</td>
<td>Allows non-standard configuration.</td>
</tr>
<tr>
<td></td>
<td>Allows you to modify device driver optional features.</td>
</tr>
<tr>
<td></td>
<td>Allows you to define custom devices.</td>
</tr>
<tr>
<td></td>
<td>Allows you to select specific fonts.</td>
</tr>
<tr>
<td></td>
<td>Allows you to use a custom memory manager.</td>
</tr>
</tbody>
</table>
2.2 Basic Configuration

Basic configuration implies that you are using the standard WindML configuration. The standard configuration uses the supported drivers provided with the WindML distribution and consists of:

- a single graphics device
- a maximum of one pointer device
- a maximum of one keyboard device
- a maximum of one audio device
- a supported font engine
- the default memory manager

If you are using unsupported hardware, or a configuration that is different to that listed below, you must do a non-standard configuration, see 2.7 Advanced Configuration, p.37.

If you use the configuration tool to define your WindML configuration, the tool generates the files used for compiling and linking WindML during the build process. If you do not use the tool, then you must edit the file `target/src/ugl/config/uglInit.h`.

NOTE: If you purchased WindML with a higher-level graphics client product, you should implement the configuration recommendations for that product.
2.2.1 Configuring a Graphics Device

To configure a graphics device, you must set the following basic configuration options:

- the type of graphics device, such as MediaGx
- the display resolution, that is, width and height of the display in pixels
- the frame buffer color depth, such as 4, 8, or 16 bit.
- the refresh rate of the display
- the type of output device, that is, CRT monitor or flat panel

 Scalable Graphics Driver Functionality

WindML provides additional functionality for graphics devices in the following areas (depending on the device hardware):

- software cursors
- overlay surfaces
- video
- JPEG
- alpha blending
- double buffers

Support for these features is scalable, you can include it or exclude it. However, these options require that the device driver be recompiled to produce a modified WindML archive.

2.2.2 Configuring a Keyboard Device

To configure a keyboard device, set the following configuration items:

- the type of the keyboard device
- the name of the device

The default name of the device is /keyboard/0.
2.2.3 Configuring a Pointer Device

To configure a pointer device, set the following configuration items:

- the type of the pointer device
- the name of the device

The default name for the pointer device is dependent on the type of the device:

- PS2 style pointer is /pointer/0
- Microsoft serial device is /tyCo/0
- Touch screen device is /touchscreen/0

2.2.4 Configuring Fonts

The font engine must be configured if your WindML application displays text. Font engine configuration is specific to the type of font engine. A font engine for bitmap fonts is included in the WindML distribution. Other font engines are available from third parties.

In general, you must:

- select the specific fonts that are to be used
- select configuration options specific to the font engine type

If you are configuring from the command line, you must modify the font engine-specific file in the target/src/ugl/config directory to define the included fonts and the characteristics of the font engine. The name of the file is uglFontengineCfg.c, where Fontengine is the name of the font engine being used. For example, for the bitmap font engine, this file is uglBmfCfgf.c.

Bitmap Font Engine

The configuration items for the bitmap font engine are:

- the size of the font cache
- the memory pool to be used for the font cache
- whether unicode (multi-byte) fonts are to be used
**Additional Font Engines**

WindML is designed so that other font engines can be added to it. These font engines may come from WindRiver, from a third party, or the user may develop their own font engine. The configuration options for these add-on font engines are provided by the supplier.

### 2.2.5 Configuring Audio

To configure an audio device, define these items:
- the audio device type
- the audio channel to use

### 2.2.6 Miscellaneous Configuration Items

The following items can also be configured for WindML:

- **Event Queue Size**
  
  This identifies the number of events that may be queued to a WindML application. The default size of the event queue is 100 events.

- **Memory Manager**
  
  WindML can use either a private memory pool or the VxWorks system memory pool. When a private memory pool is specified, all memory allocations performed by WindML are from the private memory pool. When a private memory pool is not used, then all memory allocations performed by WindML are from the VxWorks system pool.

- **Special Processor Requirements**
  
  Some processor types have special requirements. For example, the PowerPC has two memory models—PowerPC Reference Platform (PreP) and Common Hardware Reference Platform (CHRP). You must define the proper memory model when you are configuring a PowerPC-based system.

  Non-standard BSPs may dictate special requirements, for instance, a BSP which does not use the traditional PCI bus API.
2.3 Using the Configuration Tool

The WindML configuration tool can be used to define a standard configuration. It allows you to define any number of WindML configuration settings and save them in the following directory:

`target/h/ugl/config/configTool`

The tool presents a series of windows allowing you to configure different WindML components. It only allows you to select options that are applicable to a particular WindML configuration.

**From a Windows Host**

You can access the configuration tool on Windows by selecting `Tornado>Tools>WindML` from the Tornado menu. This displays the initial configuration window, Graphics, see Figure 2-1. From this window, you can open additional windows to configure input devices, fonts, audio, and set build options.

![Configurator on a Windows Host](image)
From a UNIX Host

The configuration tool on a UNIX host is launched by selecting WindML from the Launcher’s tool tray. The initial window is shown in Figure 2-2. From this window, you can open additional windows to configure input devices, fonts, audio, and set build options.

### 2.3.1 Defining a New Configuration

To define a new WindML configuration:

1. Enter a name for the configuration. For a Windows host, enter this in the field `Configuration File`. For a Unix host select `New Configuration` from `Configuration File`. This pops up a text entry dialog allowing you to enter the configuration name.

2. Select a processor type from the list of supported processors. After the processor is selected, the configuration fields will only provide options that are available for the processor.

**NOTE:** A processor type can have an appended suffix which indicates special processing. For instance, the PowerPC processor uses the PreP memory model by default, and this model is used when the PowerPC processor is chosen. However, the processor name PPC603-CHRP indicates a PowerPC processor using the CHRP memory model, and this should be the processor chosen to use the CHRP memory model.
3. Select the Graphics configuration page and set the following:
   - graphics device
   - output device type
   - color depth
   - resolution
   - refresh rate
   - optional components

4. Select the Input configuration page and select:
   - Pointer device type and device name
   - Keyboard device type, device name, and keyboard map
   - Size of the event key, that is, the number events from the pointer and keyboard that can be queued to the WindML application

5. From the set of installed font engines, select the font engine that you want to use. The UGL Bitmap font engine is the only engine provided in the WindML distribution.

   For the UGL Bitmap font engine, select:
   - Whether the application will use Unicode fonts
   - Size of the font cache
   - Specific fonts the application will use

   For other engines, consult their release notes.

6. In the Audio configuration page, select:
   - Type of audio hardware
   - Name of the audio device
   - Channel number on hardware to use

7. In the Miscellaneous configuration page, select:
   - Build options such as building with debug symbols and the library to build.
   - Whether a private WindML memory pool is to be used and if so, then the sizes of the memory pool.

After you have specified your configuration, save it by clicking on the Save button.
2.3.2 Modifying a Configuration

To modify a configuration, select the appropriate configuration entry in the Configuration File field. Any of the fields can then be modified as desired. After all the modification are made, save the configuration.

2.3.3 Deleting a Configuration

To delete a configuration, select the appropriate configuration entry in the Configuration File field and click on the Delete button.

2.3.4 Building WindML Libraries

Before building WindML, select the Miscellaneous page and select any or all of the following options:

Build VxWorks archive

This option will cause any WindML objects that are out of date in the objCpuToolvx directory to be rebuilt and archived into the libCpuToolvx.a archive. This is the archive that is used to build the VxWorks image from the BSP. A higher-level graphics product that uses WindML may link WindML objects from this archive.

Build WindML archive

This option will cause any WindML objects that are out of date in the objCpuToolUgl directory to be built and archived into the libCpuToolUgl.a archive. This is a standalone archive that only contains WindML objects.

Build WindML object

This option will cause any WindML objects that are out of date in the objCpuToolUgl directory to be built. Then, all SDK objects, device drivers, and the font engine as configured in the configuration tool are built into a downloadable object called lib/CpuTool.o which can be dynamically downloaded to the target.

Build Example Programs

This option performs the same operation as the Build WindML archive option with the addition that it also builds the example programs and links them against the libCpuToolUgl.a archive. The objects from the example programs are placed in the objCpuToolApps directory. The objects with the _ugl.o extension have been linked against the archive.
Build with debug causes the selected build options to build with the -g and -O0 compile options.

Once the build options have been selected, you can build the library by clicking the Build button. This opens a build window and performs the build.

### 2.3.5 Removing WindML Object Files

Clicking on the Clean button causes the objects selected by the build option to be removed. For example, if Build Examples Programs is selected, the objects in the objCpuToolUgl directory will be removed along with the libCpuToolUgl.a and the example program objects in the objCpuToolApps directory.

If the Build VxWorks archive option is selected, all WindML objects from the objCpuTool/vx directory will be removed, but the objects will not be removed from the libCpuTool/vx.a archive file.

### 2.4 Configuring on the Command-Line

In general, command-line configuration, that is directly editing source files, is not recommended. In most circumstances, you can specify a configuration using the configuration tool. Situations requiring the modification of source files usually involve configurations which use more than one graphics device, one pointing device, or one keyboard.

At a minimum, uglInit.h and the font configuration file for the selected font engine usually require modifications. In unusual situations, the uglInit.c file may be modified. Both of these files reside in target/src/ugl/config.

These files are described as follows:

**uglInit.h**

This file specifies the basic configuration of WindML, the device drivers to be used, how the device drivers are configured, and selects optional components.

**Font configuration files**

The font configuration files specify the configuration of the font engine which varies between individual font engines. The specific fonts to be used must be selected. The name of the font configuration file is uglFontengineCfg.c.
uglInit.c
This file controls the initialization and the de-initialization of WindML libraries. It contains the functions uglInitialize() and uglDeinitialize(). This file provides the functionality to handle a standard WindML system, consisting of, at most, one graphics device, one keyboard, one pointer, one font engine, and one audio device. Usually, the only reason that this file is modified is to handle more devices than a standard WindML system.

2.4.1 Editing the uglInit.h File
The uglInit.h file allows you to select the configuration for the WindML devices (graphics, keyboard, pointer, and audio), font engine, memory manager, and miscellaneous target/processor items. The file is split into sections covering each configuration item, as follows:

- Selection of device drivers
- Selection of font engine
- Input device configuration
- Graphics device configuration
- Audio device configuration
- Font engine configuration
- Memory manager configuration
- Miscellaneous target/processor configuration

Selecting the Device Drivers

The first section of the uglInit.h file selects the device drivers that are to be used. There is a subsection for each type of device that WindML supports. Each of the supported device drivers is identified within this file by an INCLUDE_* directive. To select a specific device driver, the appropriate INCLUDE_* directive is defined. All other INCLUDE_ directives within the respective sub-sections must be undefined.
The device driver selection section of the `uglInit.h` file is as follows:

```c
/* Specify the graphics device to use (Select 1) */
#undef INCLUDE_BIOS_GRAPHICS
#undef INCLUDE_CHIPS_GRAPHICS
#undef INCLUDE_CUSTOM_GRAPHICS /* User defined graphics device */
#undef INCLUDE_IGS_GRAPHICS
#undef INCLUDE_MEDIAGX_GRAPHICS
#undef INCLUDE_SA11XX_GRAPHICS
#undef INCLUDE_SIMULATOR_GRAPHICS
#undef INCLUDE_Q2SD_GRAPHICS
#undef INCLUDE_M821_GRAPHICS
#define INCLUDE_VGA_GRAPHICS

/* Specify the keyboard type (Select 1) */
define INCLUDE_PC_AT_KEYBOARD /* Standard PC AT style */
#undef INCLUDE_CUSTOM_KEYBOARD /* User defined keyboard device */
#undef INCLUDE_SIMULATOR_KEYBOARD /* Simulator keyboard device */

/* Specify the type of pointer device (Select 1) */
#undef INCLUDE_ASSABET_POINTER /* Assabet touchscreen */
#undef INCLUDE_CUSTOM_POINTER /* User defined pointer device */
#undef INCLUDE_MS_POINTER /* Microsoft serial mouse */
define INCLUDE_PS2_POINTER /* PS-2 type mouse */
#undef INCLUDE_SIMULATOR_POINTER /* Simulator pointer device */

/* Specify the audio hardware device (Select 1) */
#undef INCLUDE_IGS_AUDIO
#undef INCLUDE_CUSTOM_AUDIO /* User defined audio device */
```

In this example, the configuration consists of a VGA graphics device, a standard PC AT style keyboard, a PS-2 style pointer and no audio device.

**Selecting the Font Engine**

The next section of the file establishes which font engine is to be used. Each of the supported font engines is identified by an `INCLUDE_*` directive. To select a specific font engine the appropriate `INCLUDE_*` directive is defined. All other `INCLUDE_*` directives within the respective sub-section must be undefined. The font engine selection section of this file is as follows:

```c
/* Specify the font engine (Select 1) */
define INCLUDE_BMF_FONTS
```

In this example, the font engine used is the bitmap font engine, since `INCLUDE_BMF_FONTS` is defined.
Configuring Input Devices

WindML supports two general categories of input devices—keyboards and pointers. The pointer is a device that may point to a location on the display. It can be a traditional mouse or a trackball, touch screen, light pen, and so on. The pointer device can provide a location relative to the last position (as with a mouse) or it can provide an absolute position (as with a touch screen). A default set of device names is selected for the input devices, as follows:

Table 2-2  Device Names and Macros

<table>
<thead>
<tr>
<th>Device Type</th>
<th>Default Name</th>
<th>Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC keyboard</td>
<td>/keyboard/0</td>
<td>SYS_KEYBOARD_NAME</td>
</tr>
<tr>
<td>PS/2 mouse</td>
<td>/pointer/0</td>
<td>SYS_POINTER_NAME</td>
</tr>
<tr>
<td>Serial mouse</td>
<td>/tyCo/0</td>
<td>SYS_POINTER_NAME</td>
</tr>
<tr>
<td>Touch screen</td>
<td>/touchScreen/0</td>
<td>SYS_POINTER_NAME</td>
</tr>
</tbody>
</table>

These default device names can be changed by adding the appropriate macros listed above. For example, to change the serial mouse so that it uses /tyCo/1, add the following line to the uglInit.h file, before the line that includes the uglDepend.h file:

```c
#define SYS_POINTER_NAME "/tyCo/1"
```

```c
#include <ugl/config/uglDepend.h>
```

You can also define the type of keyboard mapping that is to be used. The mapping is selected by defining the desired mapping directive, as follows:

```c
/* Specify the keyboard key mapping (Select 1) */
#define INCLUDE_KMAP_ENGLISH_US
#undef INCLUDE_KMAP_ENGLISH_UK
#undef INCLUDE_KMAP_GERMAN
#undef INCLUDE_KMAP_ITALIAN
#undef INCLUDE_KMAP_FRENCH
#undef INCLUDE_KMAP_SWEDISH
#undef INCLUDE_KMAP_NONE
```

This mapping selects the English-US keyboard mapping.
Configuring Graphics Devices

The next section in the file describes the configuration of the graphics device. Elements that are configured for a graphics device are resolution, refresh rate, frame buffer format, and optional graphics device components.

This section of `uglInit.h` consists of:

```c
/*
 * Specify characteristics of the display
 */
#define UGL_DISPLAY_WIDTH 800
#define UGL_DISPLAY_HEIGHT 600
#define UGL_REFRESH_RATE 60

/*
 * When using a flat panel, select the flat panel type
 * (Defaults to a CRT monitor)
 */
#undef INCLUDE_UGL_KYOCERA_KCS057QV1AA /* Passive, 320x240x8 */
#undef INCLUDE_UGL_SHARP_LMPY385 /* Dual panel passive, 640x480x8 */
#undef INCLUDE_UGL_SHARP_LQ039Q2DS01 /* TFT, 320x240x(8 or 16) */

/*
 * Specify the frame buffer format (Select 1)
 */
#undef INCLUDE_UGL_MONO
#undef INCLUDE_UGL_GREYSCALE2
#undef INCLUDE_UGL_GRAYSCALE4
#undef INCLUDE_UGL_GRAYSCALE8
#undef INCLUDE_UGL_INDEXED4
#define INCLUDE_UGL_INDEXED8
#undef INCLUDE_UGL_ARGB4444
#undef INCLUDE_UGL_RGB565
#undef INCLUDE_UGL_ARGB8888

/*
 * Select graphics driver optional components
 */
#undef INCLUDE_UGL_ALPHA /* Alpha blending */
#undef INCLUDE_UGL_DOUBLE_BUFFERING /* Double buffering */
#define INCLUDE_UGL_JPEG /* JPEG extension */
#undef INCLUDE_UGL_OVERLAY /* Video overlay support */
#undef INCLUDE_UGL_SW_CURSOR /* Software cursor */
#undef INCLUDE_UGL_VIDEO /* Video extension */
```
This configuration specifies:

- a display resolution of 800 pixels wide and 600 pixels high
- a display refresh rate of 60 HZ
- the use of a CRT monitor
- a frame buffer format of indexed 8 bits per pixel
- the inclusion of the JPEG extension
- the inclusion of a software cursor

There is no error detection on the parameters that have been specified for a graphics device. Some drivers provide a default mode when the specified mode is invalid, other will fail to build. Therefore you should make sure that your specified configuration is valid.

Refer to the driver specific documentation for the selection of legal modes. Driver specific documentation is provided the text file *.txt in the directory 
\texttt{target/h/ugl/driver/graphics/device}, and also in appendix \textit{A. Standard Drivers} of this guide.

\textbf{Configuring Audio}

The audio driver is defined by the following parameters:

\begin{table}[h]
\centering
\caption{Audio Driver Parameters}
\begin{tabular}{|l|l|l|}
\hline
\textbf{Parameter} & \textbf{Macro} & \textbf{Default Value} \\
\hline
Name of audio device & SOUND_DEV_NAME & /sound \\
Hardware channel number & SOUND_CHANNEL & 0 \\
Hardware instance & SND_DEVICE_INSTANCE & 0 \\
Interrupt level & SND_DEVICE_INT_LEVEL & 0 \\
Interrupt vector & SND_DEVICE_INT_VECTOR & 0 \\
DMA engine & SND_DEVICE_DMA8 for 8 bit transfers and SND_DEVICE_DMA16 for 16 bit transfers & 0 \\
\hline
\end{tabular}
\end{table}
Refer to the audio driver documentation for further information on the configuration options (the xxx.txt file in the target/h/ugl/driver/audio directory, where xxx is the device name).

You can change the default configuration. For example, the following code uses the second occurrence of the audio board and uses hardware channel 1 for the audio:

```c
#define SOUND_CHANNEL 1
#define SND_DEVICE_INSTANCE 1
```

**Font Engine Configuration**

The primary font configuration mechanism is the uglFontengineCfg.c file. The configuration required by a font engine is very specific to the type of font engine. For example, the bitmap font engine requires that certain definitions be added to the uglInit.h file, other font engines may only need to be defined in the uglInit.h file.

This code illustrates setting up the bitmap font engine:

```c
/* Size of cache */
#define BMF_FONT_GLYPH_CACHE_SIZE UGL_BMF_GLYPH_CACHE_SIZE_MAX

/* Memory pool to use for glyph cache */
#define BMF_FONT_GLYPH_CACHE_MEM_POOL UGL_DEFAULT_MEM_POOL_ID

/* Include Unicode fonts */
#define INCLUDE_UGL_BMF_UNICODE
```

This sets the font cache to the maximum and uses the default memory pool and unicode fonts.

To configure the specific fonts required by an application, see 2.2.4 Configuring Fonts, p.13

**Configuring a Memory Manager**

The memory pool that WindML uses for all internal memory allocations can be either the VxWorks system pool or a private pool. The following segment of code shows the use of a private memory pool that is 0x100000 bytes in size.

```c
/* When INCLUDE_UGL_MEM_POOL is defined a private memory pool is * used for all memory allocations within UGL. When not defined, * the system memory pool is used for all allocations */
#define INCLUDE_UGL_MEM_POOL

/* The size of the memory pool in bytes */
#define UGL_MEM_POOL_SIZE 0x100000
```
**Miscellaneous Target/Processor Configuration**

WindML provides various compile time configuration options for specific target and processor configurations. An option may be provided because of the particular way in which a processor sets up its mapping, or because a BSP has non-standard features.

For instance, the PowerPC processor has two standard memory mapping schemes—the Prep (PowerPC Reference Platform) and the CHRP (Common Hardware Reference Platform). To select the CHRP memory mapping, the UGL_TARGET_CHRP macro must be defined, as follows:

```c
#define UGL_TARGET_CHRP
```

By default, the `uglInit.h` file is set up to use the CHRP memory model. When this macro is not defined, the PreP memory model is used.

### 2.4.2 Editing Font Configuration Files

To select specific fonts, edit the `uglFontengineCfg.c` file.

**Configuring the Bitmap Font Engine**

The bitmap font configuration file, `uglBmfCfg.c`, contains a data structure that defines the fonts that an application uses:

```c
extern const UGL_BMF_FONT_DESC uglBMFFont_Lucida_Sans_12;
extern const UGL_BMF_FONT_DESC uglBMFFont_Helvetica_Bold_12;
extern const UGL_BMF_FONT_DESC uglBMFFont_Lucida_Sans_8;
extern const UGL_BMF_FONT_DESC uglBMFFont_Courier_12;
const UGL_BMF_FONT_DESC * uglBMFFontData[] =
{&uglBMFFont_Lucida_Sans_12,
 &uglBMFFont_Helvetica_Bold_12,
 &uglBMFFont_Lucida_Sans_8,
 &uglBMFFont_Courier_12,
 NULL
};
```

The `uglBMFFontData` data structure is modified to contain the bitmap fonts that are required for the WindML application. In addition to placing the font in this data structure, the external reference for the font must be added to the list of external references. For example, to add a Courier Bold Oblique font with a pitch of 18, the following external reference must be added:

```c
extern const UGL_BMF_FONT_DESC uglBMFFont_Courier_Bold_Oblique_18;
```
Then the corresponding entry must be added to the data structure, as follows:

```c
&uglBMFFont_ Courier_Bold_Oblique_18,
```

The directory `target/src/ugl/fonts/bmf` contains the fonts that are available in the WindML distribution. Additional fonts can be added to this directory as required. See 2.7.4 Adding New Bitmap Fonts, p.45 for the procedure to add a new bitmap font.

**Configuring Other Font Engines**

Refer to the release notes provided with the specific font engine for configuration details.

**2.4.3 Building WindML Libraries**

When you have defined the configuration, the next step is to build the WindML library. As with building the WindML libraries with the configuration tool, there are four types of builds, as follows:

- building WindML object modules into a VxWorks archive
- building WindML object modules into a standalone WindML archive
- building WindML as a dynamically downloadable object file
- building WindML examples

The downloadable object file will contain the entire 2D layer and device drivers and font engine configured as specified in `uglInit.h` and the font configuration files.

**Building WindML into a VxWorks Archive**

To build WindML from the command line:

1. Go to the top level directory of the WindML sources `target/src/ugl`.
2. Use the appropriate make command.
An example command line for recompiling the WindML sources for the I80486 architecture is:

```bash
% cd $WIND_BASE/target/src/ugl
% make CPU=I80486
```

This causes all WindML source files that have changed to be recompiled. The object files that are generated are placed in the `lib/objCpuToolvx` directory. These object files become archived in the `libCpuToolvx.a` archive file.

You can provide additional command line arguments. This example compiles each source file with debug support enabled using compiler option `-g`:

```bash
% cd $WIND_BASE/target/src/ugl
% make CPU=I80486 ADDED_CFLAGS=-g
```

### Cleaning Object Files

There are four commands you can use to clean objects—`clean`, `aclean`, `rclean`, and `raclean`.

- **clean**
  
  To clean object files, the following command non-recursively removes the existing WindML object files from the `objCpuTool` directory, but not from the `libCpuToolvx.a` archive. For example:

  ```bash
  % make CPU=I80486 clean
  ```

  For example, if you type this command into the `target/src/ugl/2d` directory, all objects corresponding to source files in that directory will be removed, but not objects corresponding to source files in the directories beneath `2d`.

- **aclean**
  
  Use `aclean` to remove objects from the archive as well. For example:

  ```bash
  % make CPU=I80486 aclean
  ```

  This command does the same as `clean`, but it also removes the object module from `libCpuToolvx.a`.

- **rclean**
  
  This command does a recursive clean of all existing WindML object files from `objCpuToolvx`, but does not remove them from the archive. For example:

  ```bash
  % make CPU=I80486 rclean
  ```
raclean
This command recursively removes all existing WindML object files from
objcToolvx and also from the libCpuToolvx.a archive. For example:

% make CPU=I80486 raclean

Building WindML into a WindML Archive and a Downloadable Object

Like the default build described above, this process builds WindML object
modules and places them in the standard lib/objcToolvx directory. Unlike the
standard build, however, the object modules are then archived into a standalone
archive lib/libCpuToolUgl.a. The file uglCpuTool.o is a downloadable object
containing all of the 2D API and the device drivers and font engines configured as
specified in the uglInit.h file. This file is placed in the directory
lib/objcToolApps.

Once built, the uglCpuTool.o object module (containing the entire WindML library)
may be dynamically loaded/linked into the target at run-time using the target
shell’s ld command:

-> ld < uglI80486gnu.o

Again, there are several build commands:

ugl
To build the WindML standalone archive, type:

% make CPU=I80486 ugl

This recursively (that is, you do not have to be in target/src/ugl) builds the ugl
objects into the lib/objcToolUgl directory and archives them into the
libCpuToolUgl.a archive.

uglrclean
To recursively remove each object file from lib/objcToolUgl and delete the
entire lib/libCpuToolUgl.a archive, type:

% make CPU=I80486 uglrclean

uglclean
To non-recursively remove each object file for the current directory from
lib/objcToolUgl, but not the lib/libCpuToolUgl.a archive or the object
module from it, type:

% make CPU=I80486 uglclean
uglobj
To make a downloadable object containing all of the SDK and the device drivers and font engine as configured in `uglInit.h`, type:

```
% make CPU=I80486 uglobj
```
This results in an object called `lib/uglCpuTool.o`. You can use the Build WindML object option to product this file also.

ugldemo
This command produces the same results as the option Build Example Programs.

ugldemoclean
This removes the object files for the example programs only.

DRIVER="xxx"
This command can be used to build specific drivers, rather than all drivers. For example, the following command builds only the vga and mediagx drivers:

```
% make CPU=I80486 DRIVER="vga mediagx"
```
Generic driver code, such as non-graphics drivers (input, audio) will always be built.

## 2.5 VxWorks BSP Requirements

Typically, a VxWorks BSP must be modified to enable the graphics and input devices.

Due to the number of processors and target boards that VxWorks supports, a detailed description of all possible changes required to a BSP is not possible. A general discussion of the changes required follows.

⚠️ **CAUTION:** The INCLUDE_UGL macro must only be defined if you are using UGL 1.x. In the configuration file `configAll.h`, and in any other BSP-specific configuration file, you must not define the INCLUDE_UGL macro for this release of WindML.
2.5.1 Graphics Device Memory Mapping

A graphics device has two basic components:

- a frame buffer
- one or more controllers

The frame buffer is a block of memory that stores the image of the data displayed. Controllers may include graphics processors, RAMDACs, and clock chips. Depending on the graphics device, some of these controllers may not be required, or they may be integrated as a single controller.

The frame buffer and each of the controllers must be visible to the processor. Depending on the processor architecture, access to these controllers will be through memory mapped I/O, through the I/O space, or a combination of both.

To allow the processor to access a graphics device, two modifications must be made to the BSP:

- you must enable the physical mapping of the device
- you must enable the logical or virtual mapping

The exact changes required are dependent on the type of processor, the type of graphics device, and the bus architecture.

Physical Mapping

Physical mapping involves enabling the address mappers to decode the physical addresses where the graphics device resides. The mapping of the graphics controller depends on the type of the graphics controller, the processor architecture, and the VxWorks BSP.

WindML does not have any restrictions on the bus architecture that can used by the graphics device. The bus can be the CPU internal bus, ISA, PCI, AGP, VME buses, and so on. Configuration issues for some of the bus architectures are addressed below.

PCI Bus

The PCI specification allows graphics devices to be mapped to the PCI memory space using a standardized method.

On system startup, initialization software is responsible for allocating the required amount of PCI memory to each of the PCI devices. This initialization software can
be part of a manufacturer’s boot code or included within the VxWorks initialization sequence. Some manufacturer’s have a BIOS-like capability within firmware that will perform this mapping prior to transferring control to the VxWorks bootrom code.

For example, on X86 processors, the BIOS installed on the motherboard typically performs this mapping. This is also true for several Power PC processors. When the BSP provides support for the automatic configuration of the PCI bus, this capability can be used to map the graphics device into processor memory.

Unfortunately, this mapping of PCI devices is not done for all processors and VxWorks BSPs. When a PCI device is not mapped to PCI memory space, a modification to the sysLib.c file (in the target/config/arch directory) will be required to map the PCI graphics device into an available section of the PCI memory. When a safe area is found of sufficient size for the PCI device, then the base address (offset 0x10) of the PCI configuration header must be set to reflect this base address.

The following section of code shows how the Tvia IGS-5050 graphics device can be assigned a location:

```c
/* First the PCI bus must be probed to locate the device */
if (pciFindDevice (UGL_PCI_IGS_ID, UGL_PCI_IGS5050_CHIP_ID, instance, 
    &busno, &devno, &funcno) == OK)

    /* Device found, set the base address */
    pciConfigOutLong (busno, devno, funcno, 
        PCI_CFG_BASE_ADDRESS_0, baseAdrs);

    /* Enable I/O and memory access */
    pciConfigOutWord(busno, devno, funcno, PCI_CFG_COMMAND, 
        PCI_CMD_IO_ENABLE | PCI_CMD_MEM_ENABLE);

} else
    /* Error device not found */
```

A similar process can be followed to map any PCI graphics device into the processor’s memory, with some differences, as there may be PCI management differences among processors and BSPs.

**VME Bus**

For a VME bus, a VME mapper must be set up so the processor has visibility to the location at which the VME board resides. This step may include setting jumpers on the graphics boards to select the intended address. Refer to documentation for the VxWorks BSP and board specific hardware documentation for details on setting up the physical mapping.
Virtual Mapping

Virtual mapping enables the processors Memory Management Unit (MMU) to:

- perform the necessary virtual to physical mapping
- specify the caching policy appropriately

When the system architecture is PCI based and the mapping has been done by manufacturer's boot code (for example, the X86 BIOS), the location of the mapped device will be needed. This can be accomplished either by executing the function `pciHeaderShow()` or executing the utility `uglGraphicsMapShow()`.

NOTE: The `uglGraphicsMapShow()` utility can be downloaded from the WindML website.

The following shell session shows the output of a call to `uglGraphicsMapShow()`:

```bash
-> uglGraphicsMapShow
Probing PCI bus for graphics devices
Device Number 8
  Vendor = ATI (1002)
  Device = MACH64 (4354)
  Base address = 0xfb000000
  Size = 0x1000000
Device Number 14
  Vendor = IGS Tech (10ea)
  Device = 5050 (5000)
  Base address = 0xfe000000
  Size = 0x2000000
  value = 0 = 0x0
```

In this sequence, the hardware has two PCI graphics devices:

**ATI MACH64**

Device Number 8

The device number is the PCI device number, 8 in this case.

Vendor = ATI (1002)
Device = MACH64 (4354)

The first device is an ATI MACH64.
The number following the ATI is the PCI vendor number that is assigned to ATI.
The number following the MACH64 is the device ID that was assigned to the MACH 64 by ATI.
Base address = 0xfb000000
Size = 0x1000000

It is located at address 0xfb000000. It requires 0x1000000 space within the PCI memory map.

If the vendor or device ID is not known, these entries will be listed as unknown.

IGS5050
Device Number 14

The device number is 14.

Vendor = IGS Tech (10ea)
Device = 5050 (5000)

The second device is a IGS Technologies IGS5050.

Base address = 0xfe000000
Size = 0x2000000

It is mapped to the PCI at address 0xfe000000 and requires 0x2000000 space within the PCI map

The mapping is typically accomplished through modifications to the BSP’s sysLib.c file. This file has a data structure, called sysPhysMemDesc[], that defines the mapping of memory.

This data structure is an array of entries that define blocks of memory. Each entry has five fields: the physical address, virtual address, size of memory section, memory management enable masks, and memory management state values.

The data structure must be modified to include an entry that defines the mapping for the graphics device.

As an example, assuming the IGS Technologies, Inc. IGS5050 device is to be used with the configuration probed by the nglGraphicsMapShow() call in the previous example, the virtual mapping would be:

```c
/* linear video ram */
{
 (void *) 0xFE000000, /* physical mapping */
 (void *) 0xFE000000, /* virtual mapping */
 0x2000000, /* size of segment */
 VM_STATE_MASK_FOR_ALL,
 VM_STATE_FOR_IO
}
```
The size field must cover the entire device, which varies depending on the size of the frame buffer. The memory management mask and state fields must be set to mark the memory as valid, writable, and not cacheable.

**WARNING:** Memory must be marked as cache-inhibited (non cacheable) if the hardware does not have any cache coherency support.

### 2.5.2 Device I/O Configuration

As well as changes required to allow access to the graphics device memory:

- additional modifications may be required to enable certain devices
- new functions may need to be written to access device registers in the I/O space.

Device drivers are usually processor architecture specific. For example, a keyboard for the X86 processor will typically be attached to an I/O port which is accessed using the `sysInByte()`/`sysOutByte()` functionality of the VxWorks BSP. Typically, device drivers for Power PC architectures are similar to the X86 processor. Other processor architectures vary considerably, such as MIPs.

### 2.6 Configuring a Simulator to use WindML

You can develop and test a WindML application, without having a target, by using the VxWorks Simulator for NT (SimNT) or for Solaris (SimSolaris). The WindML 2D operations are supported on the simulator.

When WindML is used with a simulator, a window is displayed on the host, matching the size and color attributes specified in the configuration. For instance, a 640x480 pixel window with 256 colors will be displayed if that is what is set as the graphics device configuration.

The steps in the following sections must be done to configure the VxWorks simulators to work with WindML.
2.6.1 Configuring SimNT

The PC simulator vxWorks.exe can be built from the Tornado project facility, or from the command line. WindML must be configured using the configuration tool or by command line editing of uglInit.h.

To run WindML in the simulator, the file ugl_winLib.o must be included in the VxWorks image. This file is automatically included when statically linking WindML applications, or when WindML Components is selected in the project facility build options.

When building the simulator from the command line, and WindML applications are to be dynamically downloaded, ugl_winLib.o must be explicitly included in the BSP. This is accomplished using the MACH_EXTRA make macro as follows:

1. In the simpc BSP directory target/config/simpc, edit the Makefile and add the following line:

   MACH_EXTRA= %WIND_BASE%	arget\lib\objSIMNTgnuvx\ugl_winLib.o

2. Build the simulator image:

   make

Refer to the SimNT-specific documentation in A. Standard Drivers for a description of the capabilities of the driver and the operating modes it supports.

2.6.2 Configuring SimSolaris

The WindML driver for Solaris simulates the operation of the WindML 2D API by interfacing the WindML Device Driver layer with the Xlib layer of the underlying X Window System.

To obtain a working Solaris Simulator for WindML, you must:

1. Create a new objSIMSPARCSOLARISggnuUgl.a archive (if you have not already compiled one). This file contains the WindML APIs and underlying X11 driver file. WindML for Solaris may be configured using the configuration tool or the command line.

2. Recompile VxWorks to incorporate the new routine definitions found in the uhostLib.o and ugl_uhostLib.o files. The ugl_uhostLib.o and objSIMSPARCSOLARISgnuuUgl.a files must be specified as added modules. This can be added either from the project facility or from the command line as is appropriate for the specific build method.
If you are building with the project facility:

Using View>Properties on the Build page for your project, modify the EXTRA_MODULES macro as follows:

Add 
/usr/wind/target/lib/objSIMSPARCSOLARISgnuvx/ugl_uhostLib.o.

Modify the LIBS macros as follows:

Add /usr/wind/target/lib/libSIMSPARCSOLARISgnuUgl.a.

If you are building on the command line, add the following line to the make file for the SimSolaris BSP:

```
MACH_EXTRA = tyCoDrv.o \
$(TGT_DIR)/lib/objSIMSPARCSOLARISgnuvx/ugl_uhost.o \
$(TGT_DIR)/lib/libSIMSPARCSOLARISgnuUgl.a
```

3. Then build VxWorks using the project facility or by command line as appropriate.

Refer to the SimSolaris-specific documentation in A. Standard Drivers for a description of the capabilities of the driver and the operating modes it supports.

### 2.7 Advanced Configuration

Configurations can be specified for hardware suites that are not supported, fully or partially, in the WindML distribution. There are two categories of advanced configurations:

- unsupported devices
- supported devices which have a different access mechanism to that provided by WindML

Other advanced configuration items are hooking in a custom memory manager or adding custom fonts.
2.7.1 Customizing Driver Access Methods

The following discussion is limited to modification of access methods. Modifications beyond the simple modification of an access method requires an understanding of the device driver structure. For a description of the device structure, see 8. Graphics Device Drivers.

The access methods for a device driver consists of the mechanism to access ISA I/O space, interrupt vectors, and interrupt levels. Modification of the access mechanism is controlled by macros that are defined in the \texttt{uglInit.h} file. Table 2-4 defines the macros, how they are used, and their default values.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Purpose</th>
<th>Default Value for Processor Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSE_INT_LVL</td>
<td>Interrupt level for the PS-2 mouse.</td>
<td>X86 - 0xc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PowerPC860 - 76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PowerPC603 - 0x1c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SH-4 - 0xc</td>
</tr>
<tr>
<td>MSE_INT_VEC</td>
<td>Interrupt vector for the PS-2 mouse.</td>
<td>X86 - (0x20 + MSE_INT_LVL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PowerPC860 - INUM_TO_IVEC(76)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PowerPC603 - 0x1c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SH-4 - INUM_TO_IVEC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(INUM_IRL10)</td>
</tr>
<tr>
<td>KBD_INT_LVL</td>
<td>Interrupt level for the PC AT keyboard.</td>
<td>X86 - 0x1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PowerPC860 - 65</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PowerPC603 - 0x11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SH-4 - 0x1</td>
</tr>
<tr>
<td>KBD_INT_VEC</td>
<td>Interrupt vector for the PC AT keyboard.</td>
<td>X86 - (0x20 + KBD_INT_LVL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PowerPC860 - INUM_TO_IVEC(65)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PowerPC603 - 0x11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SH-4 - INUM_TO_IVEC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(INUM_IRL14)</td>
</tr>
<tr>
<td>COMMAND_8042</td>
<td>Command address for the keyboard/mouse controller.</td>
<td>X86 - 0x64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PowerPC - (UGL_ISA_BASE \mid 0x64)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SH-4 - 0xB04000C8</td>
</tr>
<tr>
<td>DATA_8042</td>
<td>Data address for the keyboard/mouse controller.</td>
<td>X86 - 0x60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PowerPC - (UGL_ISA_BASE \mid 0x60)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SH-4 - 0xB04000C0</td>
</tr>
</tbody>
</table>
The appropriated macros should be added to the `uglInit.h` file immediately prior to the line:

```c
#include <ugl/config/uglDepend.h>
```

The following code segment shows how the keyboard and mouse interrupt levels and vectors can be reassigned:

```c
#define MSE_INT_LVL 0x1
#define MSE_INT_VEC 0x11
#define KBD_INT_LVL 0x2
#define KBD_INT_VEC 0x12
#include <ugl/config/uglDepend.h>
#endif /* __INCuglinith */
```

These custom access methods will be used for both configuration methods—configuration tool and command line.

### 2.7.2 Adding a Custom Driver

Custom graphics and input device drivers can be hooked into WindML and configured using the command line mechanism. Section 8.2 Writing a Graphics Driver, p.148 describes the structure of the WindML device drivers.

To add a custom device driver to the configuration process, follow these steps:

1. Copy the `uglTemplateCustom.h` file in the `target/h/ugl/config` directory to the file `uglCustom.h` (or any other name) in the same directory.
2. Make the necessary changes to the `uglCustom.h` file to reflect the custom driver.
3. In the `uglInit.h` file, define the appropriate `INCLUDE_CUSTOM_`*` definitions to reflect the custom device drivers.

   /* Specify the graphics device to use (Select 1) */
   #undef INCLUDE_BIOS_GRAPHICS
   #undef INCLUDE_CHIPS_GRAPHICS
   #define INCLUDE_CUSTOM_GRAPHICS /* User defined graphics device */

   #undef INCLUDE_IGS_GRAPHICS
   #undef INCLUDE_MEDIAGX_GRAPHICS
   #undef INCLUDE_SA11XX_GRAPHICS
   #undef INCLUDE_SIMULATOR_GRAPHICS
   #undef INCLUDE_Q2SD_GRAPHICS
   #undef INCLUDE_M821_GRAPHICS
   #undef INCLUDE_VGA_GRAPHICS

4. In the `uglInit.h` file, define `UGL_CUSTOM_DRIVER_DEFS` to define the name of the configuration file created, including the path relative to the header file directory.

5. Build WindML using the `make` utility.

**Custom Graphics Driver**

A custom device driver (graphics, keyboard, or pointer) that is not in the directory structure of WindML can also be configured. A custom device driver for a graphics device requires the following items to be defined:

<table>
<thead>
<tr>
<th>Table 2-5</th>
<th>Custom Graphics Driver Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro</td>
<td>Description</td>
</tr>
<tr>
<td>UGL_GRAPHICS_NAME</td>
<td>String defining the graphics device name.</td>
</tr>
<tr>
<td>UGL_GRAPHICS_CREATE</td>
<td>The name of the function to initialize the device.</td>
</tr>
<tr>
<td>UGL_COLOR_DEPTH</td>
<td>The depth of the frame buffer (8 or 16).</td>
</tr>
<tr>
<td>UGL_MODE_FLAGS</td>
<td>The type of frame buffer</td>
</tr>
<tr>
<td></td>
<td>(UGL_MODE_DIRECT_COLOR or UGL_MODE_INDEXED_COLOR).</td>
</tr>
</tbody>
</table>
The following code segment shows how to add a new graphics device to WindML in the `uglCustom.h` file.

```c
/*
 * Set up for the custom graphics device
 * */
#if defined(INCLUDE_CUSTOM_GRAPHICS)
#define UGL_GRAPHICS_NAME "Custom graphics device"
#define UGL_GRAPHICS_INSTANCE 0 /* Device location */
#define UGL_GRAPHICS_IRQ 0 /* Device IRQ */
#define UGL_GRAPHICS_VECTOR 0 /* Device vector */
/* Initialization function */
extern UGL_UGI_DRIVER * uglCustom8BitDevCreate
 (UGL_ORD instance, UGL_UINT32 irq, UGL_UINT32 vector);
#define UGL_GRAPHICS_CREATE uglCustom8BitDevCreate
/* Display mode */
#define UGL_COLOR_DEPTH 8
#define UGL_MODE_FLAGS UGL_MODE_INDEXED_COLOR
#endif /* INCLUDE_CUSTOM_GRAPHICS */
```

In this example, the custom graphics device driver is initialized by the function `uglCustom8BitDevCreate()` and the device driver operates as an 8 bit indexed color driver.
**Custom Keyboard Driver**

When adding a custom keyboard device, the following parameters must be defined:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS_KEYBOARD_CREATE</td>
<td>This specifies the name of the function that creates the keyboard device driver. This parameter is not defined if the keyboard device is previously created by VxWorks (as is the case with the serial devices /tyCo/0, /tyCo/1, and so on.)</td>
</tr>
<tr>
<td>UGL_KEYBOARD_INIT</td>
<td>The name of the function to initialize the keyboard device and to hook it into the WindML event services.</td>
</tr>
<tr>
<td>SYS_KEYBOARD_NAME</td>
<td>The name of the keyboard device.</td>
</tr>
</tbody>
</table>

The following code segment shows how to add a new keyboard device to WindML in the uglCustom.h file.

```c
/* Custom keyboard */
#if defined INCLUDE_CUSTOM_KEYBOARD
#define INCLUDE_UGL_INPUT
#define SYS_KEYBOARD_NAME "/customKbd/0"

/* Name of the function to create the keyboard device */
extern STATUS customKbdDevCreate(char *name);
#define SYS_KEYBOARD_CREATE customKbdDevCreate

/* Name of the function to initialize the keyboard device */
extern UGL_INPUT_DEVICE_ID uglCustomKbdInit
    (char * name, UGL_EVENT_SERVICE_ID eventServiceId);
#define UGL_KEYBOARD_INIT uglCustomKbdInit
#endif /* INCLUDE_CUSTOM_KEYBOARD */
```

WindML uses a two stage initialization scheme for the keyboard driver. The first stage is to create the device and the second stage is to initialize the device.

When the device is already present, as is the case with the standard serial devices (/tyCo/0, /tyCo/1, and so on), no device create function is required. In the above example, a keyboard device create function is required which is called `customKbdDevCreate()`. The `uglCustomKbdInit()` function is responsible for initializing the device for use in WindML. The name of the device that is created and initialized is /customKbd/0.
**Custom Pointer Driver**

When adding a custom pointer device, the following parameters must be defined:

### Table 2-7 Custom Pointer Driver Definitions

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS_POINTER_CREATE</td>
<td>This specifies the name of the function that creates the pointer device driver. This parameter is not defined if the pointer device is previously created by VxWorks (as is the case with the serial devices /tyCo/0, /tyCo/1, and so on.)</td>
</tr>
<tr>
<td>UGL_POINTER_INIT</td>
<td>The name of the function to initialize the pointer device and to hook it into the WindML event services.</td>
</tr>
<tr>
<td>SYS_POINTER_NAME</td>
<td>The name of the pointer device.</td>
</tr>
</tbody>
</table>

The following code segment shows how to add a new pointer device to WindML in the `uglCustom.h` file:

```c
/* Custom pointer */
#ifdef INCLUDE_CUSTOM_POINTER
  #ifndef INCLUDE_UGL_INPUT
    #define INCLUDE_UGL_INPUT
  #endif /* INCLUDE_UGL_INPUT */
  #define SYS_POINTER_NAME "/customPtr/0"
  /* Name of the function to create the pointer device */
  extern STATUS customPtrDevCreate(char *name);
  #define SYS_POINTER_CREATE customPtrDevCreate

  /* Name of the function to initialize the pointer device */
  extern UGL_INPUT_DEVICE_ID uglCustomPtrInit
    (char * name, UGL_EVENT_SERVICE_ID eventServiceId);
  #define UGL_POINTER_INIT uglCustomPtrInit
  #endif /* INCLUDE_CUSTOM_POINTER */

As with the keyboard driver, WindML uses a two stage initialization scheme for the pointer driver. The first stage is to create the device and the second stage is to initialize the device. When the device is already present, as is the case with the standard serial devices (/tyCo/0, /tyCo/1, and so on), no device create function is required. In the above example, a keyboard device create function is required which is called `customPtrDevCreate()`. The `uglCustomPtrInit()` function is responsible for initializing the device for use within WindML. The name of the device that is created and initialized is `/customPtr/0`.
Custom Audio Driver

When adding a custom audio device, the following parameters must be defined:

Table 2-8 Custom Audio Driver Definitions

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SND_DEVICE_INSTANCE</td>
<td>The occurrence of the audio device. For PCI type devices this is the location of the device on the bus, where the first device is 0, the second is 1, and so on. For non-PCI type devices, this may be a physical base address for memory mapped devices or it may be an ISA port address.</td>
</tr>
<tr>
<td>SND_DEVICE_INT_LEVEL</td>
<td>The interrupt level that the audio device uses.</td>
</tr>
<tr>
<td>SND_DEVICE_INT_VECTOR</td>
<td>The interrupt vector that the audio device uses.</td>
</tr>
<tr>
<td>SND_DEVICE_DMA8</td>
<td>The port that will be used for 8 bit transfers.</td>
</tr>
<tr>
<td>SND_DEVICE_DMA16</td>
<td>The port that will be used for 16 bit transfers.</td>
</tr>
</tbody>
</table>

The following code segment shows how to add a new audio device to WindML in the uglCustom.h file.

```c
/*
 * Set up for the custom audio device
 */
#ifdef INCLUDE_CUSTOM_AUDIO
#define SND_DEVICE_INSTANCE 220 /* instance of device */
#define SND_DEVICE_INT_LEVEL 5 /* interrupt level */
#define SND_DEVICE_INT_VECTOR 5 /* interrupt vector */
#define SND_DEVICE_DMA8 1 /* 8 bit DMA controller */
#define SND_DEVICE_DMA16 5 /* 16 bit DMA controller */

/* Creation function */
extern STATUS customSndDevCreate (char *devName, int channel, int instance,
                                 int intLevel, int intVector, int dma8, int dma16);
#define SOUND_DEV_CREATE customSndDevCreate
#endif /* INCLUDE_CUSTOM_AUDIO */
```

In this example, the audio device creation function is called `customSndDevCreate()`. Its instance, or access location is 220 and it uses an interrupt level and vector of 5. For DMA transfers, it uses port 2 for 8 bit transfers and port 5 for 16 bit transfers.
2.7.3 Using a Custom Memory Manager

WindML allows you to assign a custom memory manager. You must specify functions to:

- create a memory pool
- destroy a memory pool
- allocate a block of memory (malloc and calloc functions)
- reallocate a previously allocated memory block
- free a previously allocated memory block

These functions are assigned in `uglInit.h` immediately before the line:

```c
#include <ugl/config/uglDepend.h>
```

This is an example of a custom memory manager definition:

```c
#define UGL_MEM_POOL_CREATE_RTN myMemPoolCreate
#define UGL_MEM_POOL_DESTROY_RTN myMemPoolDestroy
#define UGL_MEM_ALLOC_RTN myMemAlloc
#define UGL_MEM_CALLOC_RTN myMemCalloc
#define UGL_MEM_REALLOC_RTN myMemRealloc
#define UGL_MEM_FREE_RTN myMemFree
```

These custom memory manager functions will be used for both configuration methods—configuration tool and command line.

2.7.4 Adding New Bitmap Fonts

Additional fonts can be added to the set of available bitmap fonts. The directory `target/src/ugl/fonts/bmf` provides all the bitmap font images. Any font that is in this directory is automatically available to the configuration process. The font may be referenced in the `uglBmfCfg.c` file when building WindML from the command line. They are also available in the WindML configuration tool. The format of these bitmap font description files are WindML specific. There are utility programs to convert BDF fonts to the WindML format.
2.8 Project Facility

The Tornado project facility allows you to include or exclude WindML functionality into your VxWorks build. For general guidelines on using the Tornado project facility, see the *Tornado Getting Started Guide* and the *Tornado User’s Guide: Projects*—this manual emphasizes only those project facility features relevant to the setup of WindML.

When WindML is installed, a WindML components folder is created in the Tornado project facility’s component hierarchy, as shown in Figure 2-3.

This folder has two sub-folders, 2D graphics and Audio components. Note that a folder displayed with a checkbox icon contains options that are either mutually exclusive or inclusive.

To use the project facility to include a WindML component, follow these steps:

1. Select a component.
2. Click the right mouse button, popping up a context menu.
3. Select the Include command.
You can exclude components in the same manner, if they have already been included. When you click the right mouse button, the context menu displays an Exclude command instead.

2.8.1 Linking the 2D Library

There are two ways in which the 2D library can be linked with the VxWorks image:

- Link the entire 2D layer into the VxWorks image (complete 2D library)
  This allows the subsequent download of a WindML application after the target is booted. This mode allows any WindML application be dynamically downloaded since the VxWorks/WindML image contains the entire WindML 2D functionality.

- Link only the necessary 2D functionality (necessary 2D library)
  This mode requires that WindML is built into VxWorks, in addition to the WindML application. This results in a smaller VxWorks/WindML image.

 NOTE: If you are already linking a higher-level graphics product such as Zinc or PersonalJWorks, you should select necessary 2D library, not complete 2D library.

2.8.2 Adding WAVE File Support

Audio functions to support audio files formatted as WAVE files can be included in the VxWorks image. A WindML application subsequently downloaded will be able to play audio files.
3

2D Graphics Library

3.1 Introduction

The 2-D API can be divided into the following functional areas, which are examined in greater detail in this chapter:

- **Drawing Primitives.** These routines provide basic drawing primitives, including lines, rectangles, ellipses, polygons, pixels.

- **Text Rendering and Font Management.** Text can be rendered using different fonts. You can manage the fonts used by the application, specify fonts, request exact or approximate fonts, and use fonts for text rendering.

- **Bitmap Management.** WindML provides an extensive bitmap management API, providing color, monochrome, and transparent bitmaps. Also device-independent bitmaps and device-dependent bitmaps.

- **Cursor Management.** The cursor management API allows the application to manage and control the behavior of the cursor or mouse. This includes the functionality to maintain and dynamically switch between several bitmaps that are associated with the cursor.

- **Batch Drawing.** This API allows you to optimize a series of low-level drawing requests. This ensures that drawing is done efficiently—for instance, by removing affected cursors only once from the display, instead of performing repeated hide/show requests at the beginning and end of each sequential drawing operation.

- **Graphics contexts (GCs).** Graphics contexts are data structures which contain information relevant to the current drawing operation (for example, line width, fill pattern, clip regions). The 2-D API provides routines to create and
destroy graphics contexts, and set the fields of the graphics context data structure. A graphics context must be set up for a specific output device before any drawing operations can take place.

- **Color Management.** The color management API provides routines and macros allowing basic and customized color management to be performed depending on the application requirements.

- **Double Buffering.** The double buffering API allows the application to manage multiple off screen buffers for flicker-free off-screen rendering. Note that double buffering is not supported by all drivers.

### 3.2 Drawing Primitives

Basic drawing is accomplished with simple geometry primitives, including `uglEllipse()`, `uglLine()`, and `uglPolygon()`. With the WindML API, you can draw lines, rectangles, ellipses, pixels, and polygons. These routines form the basic building blocks of WindML drawing. They are summarized in Table 3-1.

**Table 3-1 Basic Drawing Primitives**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uglEllipse()</code></td>
<td>Draws a ellipse, arc, or pie slice.</td>
</tr>
<tr>
<td><code>uglLine()</code></td>
<td>Draws a line.</td>
</tr>
<tr>
<td><code>uglPixelSet()</code></td>
<td>Draws a pixel.</td>
</tr>
<tr>
<td><code>uglPixelGet()</code></td>
<td>Gets the color of a pixel.</td>
</tr>
<tr>
<td><code>uglPolygon()</code></td>
<td>Draws a polygon.</td>
</tr>
<tr>
<td><code>uglRectangle()</code></td>
<td>Draws a rectangle.</td>
</tr>
</tbody>
</table>

### 3.2.1 Using the API

All of these routines use the information in the current graphics context to select foreground and background colors, line thicknesses, and so on. Refer to 3.7 Graphics Contexts, p.68 for more details on graphics contexts.
3.2.2 Example Code

This code segment shows some basic drawing operations:

```c
int windMLExampleBasic (void)
{
    UGL_GC_ID gc;
    . . .

    /*
    * Create a graphics context. Default values are set during
    * the creation.    */
    gc = uglGcCreate(devId);

    /* ---------- Draw a line. ---------- */
    uglBatchStart(gc);
    uglForegroundColorSet(gc, colorTable[WHITE].uglColor);
    uglLine(gc, 0, 0, displayWidth - 1, displayHeight - 1);
    uglBatchEnd(gc);

    /* ---------- Draw a rectangle. ---------- */
    uglBatchStart(gc);
    uglFillPatternSet(gc, patternBitmap);
    uglForegroundColorSet(gc, colorTable[WHITE].uglColor);
    uglBackgroundColorSet(gc, colorTable[GREEN].uglColor);
    uglLineStyleSet(gc, UGL_LINE_STYLE_SOLID);
    uglLineWidthSet(gc, 7);
    uglRectangle(gc, displayWidth / 8, displayHeight / 8,
                 displayWidth / 4, displayHeight / 4);
    uglBatchEnd(gc);

    /* ---------- Draw an ellipse. ---------- */
    uglBatchStart(gc);
    uglForegroundColorSet(gc, colorTable[BLUE].uglColor);
    uglBackgroundColorSet(gc, colorTable[RED].uglColor);
    uglCenterEllipse(gc, 400, 80, 600, 160, 0, 0, 0, 0);
    uglBatchEnd(gc);
    . . .
}
```
3.3 Text Rendering and Font Management

The 2D text rendering and font management API provides a portable method of drawing textual information to the display. Font management allows the programmer to specify various types of font families (Helvetica, Courier, Times Roman) and to combine font attributes; such as bold, italic, and point size to produce a specific type of font appropriate for presentation in the target application. The text rendering routines use created fonts to draw the text information on the display.

The 2D font API is listed in Table 3-1.

Table 3-2  Font Management Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglFontCreate()</td>
<td>Creates a font.</td>
</tr>
<tr>
<td>uglFontDestroy()</td>
<td>Destroys a font.</td>
</tr>
<tr>
<td>uglFontMetricsGet()</td>
<td>Retrieves font metric data for a font.</td>
</tr>
<tr>
<td>uglFontFind()</td>
<td>Finds a font, using a font descriptor structure.</td>
</tr>
<tr>
<td>uglFontFindString()</td>
<td>Finds a font, specifying the parameters in a string.</td>
</tr>
<tr>
<td>uglFontFindFirst()</td>
<td>Starts enumerating the fonts available for a font driver.</td>
</tr>
<tr>
<td>uglFontFindNext()</td>
<td>Continues enumerating fonts available for a font driver.</td>
</tr>
<tr>
<td>uglFontFindClose()</td>
<td>Finishes the font enumeration process.</td>
</tr>
<tr>
<td>uglFontDriverInfo()</td>
<td>Retrieves or send information to a font driver.</td>
</tr>
<tr>
<td>uglFontSet()</td>
<td>Set the current font set in the graphics context.</td>
</tr>
<tr>
<td>uglFontGet()</td>
<td>Gets the current font set in the graphics context.</td>
</tr>
<tr>
<td>uglFontRotationAngleSet()</td>
<td>Sets the rotation angle of a font (if rotation is supported by the font driver).</td>
</tr>
<tr>
<td>uglFontSizeSet()</td>
<td>Sets the pixel size of a font (for scalable fonts only).</td>
</tr>
<tr>
<td>uglFontWeightSet()</td>
<td>Sets the weight of a font (for scalable fonts only).</td>
</tr>
</tbody>
</table>

The text rendering routines are listed in Table 3-3.
3.3.1 Using the API

Initializing a Font Driver

Before fonts can be rendered to the screen, the font driver and font engine must be initialized. A font driver is responsible for interfacing with the font engine. The font engine is responsible for creating and rendering the fonts to the screen. The 2D text and font management API is a portable interface to different font drivers and font engines. Typical embedded systems will only include a single font driver and corresponding font engine. However, it is possible for a system to contain multiple font engines.

Usually the font driver and font engine are initialized or created in the `uglInitialize()` routine and deinitialized or destroyed in the `uglDeinitialize()` routine along with other drivers. The UGL_FONT_DRIVER_ID returned from the font driver creation routine is required by some of the 2D API font routines. This ID can be obtained from the device driver registry by the application at any time. See 7. Resource Management for more details on the device registry, `uglInitialize()`, and `uglDeinitialize()`.

When the font driver has been created, it may need to be configured before it can be used properly. All run-time configuration of a font driver is done through the `uglFontDriverInfo()` routine. The following code segment shows how `uglFontDriverInfo()` is used to specify an upper-left text origin, instead of the driver’s default baseline positioning.

```c
textOrigin = UGL_FONT_TEXT_UPPER_LEFT;
uglDriverFind (UGL_FONT_ENGINE_TYPE, 0, (UGL_UINT32 *)&fontDrvId);
uglFontDriverInfo(fontDrvId, UGL_FONT_TEXT_ORIGIN, &textOrigin);
```

### Table 3-3 Text Rendering Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uglTextDraw()</code></td>
<td>Draws text using the current font, starting at location (x,y).</td>
</tr>
<tr>
<td><code>uglTextDrawW()</code></td>
<td>Draws text (double-byte characters) using the current font.</td>
</tr>
<tr>
<td><code>uglTextSizeGet()</code></td>
<td>Gets the width and height of a text string.</td>
</tr>
<tr>
<td><code>uglTextSizeGetW()</code></td>
<td>Gets the width and height of a text string (double-byte characters).</td>
</tr>
</tbody>
</table>
Run-time configuration options may be font driver specific. Consult the font driver’s documentation for further details. Also consult the font driver’s documentation for any compile-time configuration and scalability options that may be available.

Creating Fonts

Each font must be created before it can be used to render text to the screen. Creation of fonts is a 2-step process:

1. Find an available font that nearest matches the desired font attributes.
   Finding a nearest match is important to conserve system memory requirements. Many embedded systems only provide a limited selection of fonts and the set of available fonts might not be known by the application at run-time. The nearest matching process provides a means for the application to find a font at run-time that most closely matches the particular font attributes. This can be done with the uglFontFind() or uglFontFindString() routines. Note that if a particular font is known to be available, the nearest matching step can be skipped.

2. Create the font found from the nearest matching process.
   Once a font is known to be available, it must be created before it can be used. Note that the application is required to destroy all fonts that it creates. Fonts are created and destroyed by using the uglFontCreate() and uglFontDestroy() routines.

Drawing Text

Once a font has been created, you must set its font ID (UGL_FONT_ID) in the graphics context before you can use it. Only one font ID can be set in a graphics context at a time.

The routine uglTextDraw() (and its double-byte equivalent, uglTextDrawW()) is used to draw text on the screen, using the current font in the graphics context.

NOTE: uglTextDraw() and uglTextDrawW() do not wrap text on the display or bitmap. Any text extending beyond the bounds of the display or bitmap will be clipped.
Information on text string dimensions is important for applications that must present and manage text in a consistent manner. You can use the `uglTextSizeGet()` routine to determine the dimensions that the string will have when drawn on the screen using the current font.

**NOTE:** WindML deals in pixel sizes, not in point sizes, for fonts. This is because the 2D font API is independent of any particular screen type or size. It is the application’s responsibility to translate from pixel sizes to point sizes if necessary.

### Font Types

There are 2 general categories of font glyphs:

- scalable
- bitmapped

Scalable fonts can usually be sized and/or rotated at runtime without the need for a separate creation step. This can be done through the `uglFontSizeSet()`, `uglFontWeightSet()`, and `uglFontRotateSet()` functions. Bitmapped font engines do not usually support these routines.

### Encoding

WindML provides a set of bitmapped fonts that are organized according to the ISO8859-1 standard for 8-bit character representations and according to the UNICODE standard for 16-bit character representations. They are not organized according to local code pages (e.g. Code Page 437).

The WindML font API does not do any encoding translation, so programmers must ensure the construction of textual information is done according to the underlying capabilities of the font driver (which is ISO8859-1 for default WindML bitmapped fonts).
3.3.2 Example Code

This code segment shows how a font is specified text rendered to the display using that font:

```c
int windMLExampleTextIO (void)
{
    UGL_FONT_DRIVER_ID fontDrvId;
    UGL_FONT_DEF fontDef;
    char *text = "WindML text input and output example.";

    /*
    * This function allows us to find a font from the set of available
    * fonts. A search parameter string is used to specify the font we
    * want. Incomplete specifications leave the function to make the
    * choice. Here we specify Lucida, but haven't specified a size, so
    * the font function will choose one for us. There are several para-
    * meters that can be used. They are semicolon delimited.
    */
    uglFontFindString(fontDrvId, "familyName=Lucida", &fontDef);

    /*
    * Now that we have found a font and it is specified in fontDef,
    * we can "create" the font. What is accomplished here is dependent
    * upon the specific font engine being used. This step could
    * include conversion of the font data from a platform independent
    * format to a platform dependent format; similar to a DIB being used
    * to create a DDB. For the BMF font engine a glyph cache and other
    * resources are configured.
    */
    if ((fontBanner = uglFontCreate(fontDrvId, &fontDef)) == UGL_NULL)
    {
        printf("Font not found. Exiting.\n");
        return(0);
    }

    /* Set color combination for the banner. */
    uglForegroundColorSet(gc, colorTable[YELLOW].uglColor);
    uglBackgroundColorSet(gc, colorTable[BLUE].uglColor);

    /* Set the font we acquired earlier for the banner. */
    uglFontSet(gc, fontBanner);

    /*
    * Draw the text! The GC contains the bitmap that we want to render to.
    * In this case it will be the frame buffer. We are locating the text
    * at the upper left corner of the screen (0,0).
    */
    uglTextDraw(gc, 0, 0, -1, text);
}
```

56
3.4 Bitmap Management

The 2D bitmap management API provides a portable mechanism for creating and rendering monochrome, color, and transparent bitmap images to the display.

WindML supports these three types of bitmaps in the following manner:

- **Standard color bitmaps**
  These bitmaps are rectangular and can have one to thousands of colors.

- **Transparent Bitmaps**
  These bitmaps can be used to create different non-rectangular images. They have two components:
  - a full color component
  - a transparent component where the background shows through.

- **Monochrome Bitmaps**
  These bitmaps are typically used for fonts. They are bitmaps in the true sense of the word. Each bit corresponds to one pixel with a value of 1 or 0.
  - A '1' indicates that the pixel will be turned on.
    
    This means that the color of the pixels is determined by the foreground color setting in the graphics context.
    
    - A '0' indicates that the pixel will be turned off.
      
      This means that the color will be determined by the background color in the graphics context.

Bitmaps are an important part of a graphical application that can consume a large portion of CPU and bus time. As such, proper management and planning of bitmap usage is critical during application development. WindML's 2D bitmap API provides flexible bitmap management which allows an application to balance portability and performance.

Bitmaps manipulated by the routines listed in Table 3-4.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ugl_bitmap_blt()</td>
<td>Performs a block image transfer of a bitmap.</td>
</tr>
<tr>
<td>ugl_bitmap_create()</td>
<td>Creates a DDB.</td>
</tr>
</tbody>
</table>
Device Independent and Dependent Bitmaps

WindML defines a portable bitmap representation called a *Device Independent Bitmap*, or DIB. There are two versions of DIBs:

- a standard color DIB (UGL_DIB), which is used for standard bitmap type operations
- a monochrome DIB (UGL_MDIB), which used where only a single bit type or color operation is required.

The standard color DIB typically defines each bitmap pixel with an associated RGB value. This portable color is converted, at run-time, into a color recognized by the underlying graphics driver.
The monochrome DIB defines each pixel as either on (1) or off (0). Typically, each bitmap pixel is defined as a 1-bit value. This value is then used to indicate whether a particular pixel will be rendered to the display, or whether the pixel will be transparent.

Transparent bitmaps are created by combining a monochrome and color DIBs, where the monochrome DIB provides transparency information that is applied to the color DIB.

Most bitmap creation routines require the use of monochrome or color DIBs.

WindML also defines a hardware specific bitmap representation called a Device Dependent Bitmap, or DDB. There are three types of DDBs:

- a standard color DDB (UGL_DDB_ID), which defines a color bitmap according to the underlying color system (e.g. RGB565, ARGB888)
- a monochrome DDB (UGL_MDDB_ID), which defines a monochrome bitmap according to the most efficient graphics driver representation for transparency
- a transparent DDB (UGL_TDDB_ID), which combines a color DDB with a monochrome DDB.

These bitmap representations optimize features of the graphics driver to ensure blt operations are performed in an optimal manner. Because graphics vendors define a number of different optimization techniques, no particular DDB representation can be assured.

All bitmap rendering operations require the creation of a color, monochrome, or transparent DDB.

### 3.4.2 Example Code

This code segments creates a two bitmaps, a regular DIB and a transparent DIB, and bitblits them to the display:

```c
UGL_UINT8 transparentData[] =
{
    #define O WHITE,
    #define B BLUE,
    #define Y LIGHTGRAY,
    #define G GREEN,
    B B B B B B O O O B B B B B B
```
#undef G
#undef Y
#undef B
#undef O

UGL_UINT8 transparentMask[] =
{
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF,
0xF}
/*
* colorData is a linear array of UGL_COLOR elements which we
* indicate in the DIB header using UGL_DEVICE_COLOR_32.
*/
(UGL_COLOR *)transDib.pImage = colorData;
transDib.colorFormat = UGL_DEVICE_COLOR_32;

/*
* Since we already derived the "direct" representation of the
* color from the colorTable, we are not using a clut in the
* DIB.  We indicate that we have a direct representation in
* the image data using UGL_DIB_DIRECT.
*/
transDib.clutSize = 0;
transDib.pClut = UGL_NULL;
transDib.imageFormat = UGL_DIB_DIRECT;

/*
* Set up the dimensions of the color and monochrome bitmaps.
* Typically the stride will be equal to the width, but for special
* cases (such as creating bitmaps from sub-sections of a larger DIB)
* it can be larger.
*/
transDib.width = transDib.height = transDib.stride = 16;
transMdib.width = transMdib.stride = transMdib.height = 16;

/*
* The monochrome DIB is just bits (see transparentMask); nothing
* more to set.
*/
transMdib.pImage = transparentMask;

/*
* When the bitmaps are created, the DIB format is converted to the
* hardware frame buffer format.  Knowledge of the frame buffer's
* format is not required in order to manipulate, draw to, or blit
* bitmaps using the UGL 2-D API.
*/
transBitmap = uglTransBitmapCreate(devId, &transDib, &transMdib,
UGL_DIB_INIT_DATA, 0, UGL_NULL);
testBitmap = uglBitmapCreate(devId, &transDib, UGL_DIB_INIT_DATA,
0, UGL_NULL);

/* Blit the images to the screen */
uglBatchStart(gc);
ClearScreen(gc);

/*
* UGL provides a function for "writing" a DIB to a bitmap (in fact, this
* is used in the creation of the bitmap).  Writing includes the
* conversion of the DIB's format to that of the bitmap.  This operation
* is typically slower than blits.  Since the frame buffer's format
* defines the format of bitmaps, the frame buffer can be considered a
* bitmap itself. The frame buffer bitmap is identified using
* UGL_DISPLAY_ID.
*/
uglBitmapWrite(devId, &transDib, 0,0,15,15,UGL_DISPLAY_ID, 100,
                displayHeight - 132);

/*
* Blitting can only be done between bitmaps. We specify the rectangular
* area within the originating bitmap to be blitted, and the destination
* position. When the GC is created its "default bitmap" is set to the
* screen bitmap, UGL_DISPLAY_ID. Since we have not changed it to another
* bitmap, we can blit to the screen by using the UGL_DEFAULT_ID bitmap.
*/
uglBitmapBlt(gc, testBitmap,0,0,15,15, UGL_DEFAULT_ID, displayWidth / 2,
              displayHeight / 2 - 50);

uglBitmapBlt(gc, transBitmap,0,0,15,15,UGL_DEFAULT_ID, displayWidth / 2,
              displayHeight / 2 + 15);
}

3.5 Cursor Management

A cursor is a special bitmap image that is created by an application and is
positioned on the screen by a pointing device. The cursor bitmap image supports
254 colors, in addition to transparent and inverted pixels.

An application can manage a cursor in two different ways:

- It can let WindML process the pointing device input and automatically update
  the cursor position.

  This method is easy for the application programmer to implement, as they do
  not have to take responsibility for moving the cursor.

- It can keep cursor rendering separate from the pointing device's input.

  In this case, the application must move the cursor in response to the position
  changes of the pointing device. This provides the application with greater
  control over the functioning of the cursor and pointing device.

The WindML cursor functions available to the application programmer are listed
in Table 3-5.
3.5.1 Using the API

Software and Hardware Cursors

A hardware cursor is an implementation of cursor technology that puts a graphic cursor on a different drawing plane from other primitive drawing operations. Hardware cursor implementations greatly enhance general drawing operations, because a hardware cursor does not need to be hidden while performing drawing operations, a feature required for software cursor implementations. However, most graphics chips do not currently support hardware cursors.

\[\text{NOTE: Consult the driver documentation for any hardware limitations such as image size.}\]

The underlying WindML DDK defines support for both a software and hardware cursor interface, but the SDK cursor API does not distinguish between hardware and software cursors. It is recommended you use the batch functions `uglBatchStart()` and `uglBatchEnd()` to minimize the time required to perform cursor management on drawing operations, unless you are guaranteed that the target system implements hardware cursors.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uglCursorInit()</code></td>
<td>Initializes a cursor.</td>
</tr>
<tr>
<td><code>uglCursorDeinit()</code></td>
<td>Deinitializes a cursor.</td>
</tr>
<tr>
<td><code>uglCursorBitmapCreate()</code></td>
<td>Creates a cursor bitmap.</td>
</tr>
<tr>
<td><code>uglCursorBitmapDestroy()</code></td>
<td>Destroys a cursor bitmap.</td>
</tr>
<tr>
<td><code>uglCursorImageSet()</code></td>
<td>Sets the image displayed by the cursor.</td>
</tr>
<tr>
<td><code>uglCursorImageGet()</code></td>
<td>Gets the image currently set for a cursor.</td>
</tr>
<tr>
<td><code>uglCursorMove()</code></td>
<td>Moves the position of a cursor.</td>
</tr>
<tr>
<td><code>uglCursorPositionGet()</code></td>
<td>Gets the current position of a cursor.</td>
</tr>
<tr>
<td><code>uglCursorOn()</code></td>
<td>Turns on a cursor.</td>
</tr>
<tr>
<td><code>uglCursorOff()</code></td>
<td>Turns off a cursor.</td>
</tr>
</tbody>
</table>
Hot Spots

Most images are positioned on a display using their (0,0) origin. This is often inappropriate for cursor images. The part of the image that defines the point (or what the user perceives as the part of the image that points to—or is placed on—something on the display) may not be at the (0,0) origin, see Figure 3-1.

The *hot spot* in a cursor image allows the application to offset the cursor image on the display. This allows the cursor image to be centered on a particular feature within the bitmap image.

For instance, the center of this crosshair image is at location (12,13) within the cursor bitmap. The coordinates (12, 13) are set as the hot spot coordinates. A call to `uglCursorMove( )` with a position of (200,200) would place the bitmap at (188, 187). This causes the center of the crosshairs to be at (200,200), as desired.

Constraining and Freeing the Cursor

The cursor can be constrained by setting a rectangle for constraining the pointing device using the `UGIInputDeviceInfo( )` function. If the application is processing the pointing device input itself, then it may constrain the cursor by constraining the input device or by its own means.
Here is an example of setting the pointing devices constraints:

```c
UGL_RECT  rect;
rect.left = newMinX;
rect.top = newMinY;
rect.right = newMaxX;
rect.bottom = newMaxY;
uglInputDeviceInfo (inSvcId, ptrDevNum,
    UGL_DEVICE_SET_SCREEN_CONSTRAINT, &rect);
```

When a cursor bitmap is no longer needed, the cursor should be destroyed using `uglCursorBitmapDestroy()`. At the end of program execution, the cursor should first be de-initialized and then the cursors bitmap should be destroyed.

To change the cursor image without corrupting the rendering of the cursor, do not free the cursor’s current bitmap until a new one is selected into the cursor.

Animating the cursor can be accomplished by setting different cursor bitmaps into the cursor in succession.

### 3.5.2 Example Code

This code segment creates a cursor, sets up its hot spot, and sets it as the default cursor.

```c
/*
 * The actual image data for the cursor.  This could be set up
 * as an array of numbers, or the like, but a more graphical
 * representation can be had using the #defines.
 */

UGL_UINT8 cursorData[] =
{
    #define B 0,
    #define Y 1,
    #define T 255,
    #define I 254,
int windMLExampleCursor (void)
{
    UGL_CDIB  cursorDib;
    UGL_CDDB_ID cursorBitmap;
    . . .

    /* Assemble a cursor DIB (CDIB). Note that hardware cursors often have
     * a limitation on their dimensions, and other characteristics.
     * We are centering the “hot spot” on the cursor image. This means that
     * when the cursor is moved to x,y the pixel at 16,16 in the cursor
     * image is located at x,y. */
    cursorDib.width = cursorDib.height = cursorDib.stride = 32;
    cursorDib.hotSpot.x = cursorDib.hotSpot.y = 16;
    cursorDib.pImage = cursorData;
    cursorDib.clutSize = 2;
    cursorDib.pClut = cursorClut;

    /* Initialize the cursor. Set its max size and initial position. */
    uglCursorInit (devId, 32, 32, displayWidth / 2, displayHeight / 2);

    /* Use the DIB to create a cursor type bitmap. */
    cursorBitmap = uglCursorBitmapCreate(devId, &cursorDib);

    /* Set the cursor type bitmap, cursorBitmap, as the cursor’s bitmap.
     * By changing the image to different images the cursor can be
     * animated. Do not free a bitmap that is being used by the cursor.
     * So, if animating, select the next image bitmap into the cursor, and
     * then free the old one. */
3.6 Batch Drawing

In addition to the basic drawing methods described above, WindML supports two routines that add simple batch capabilities to drawing operations: `uglBatchStart()` and `uglBatchEnd()`. Batch drawing provides the following benefits:

- use of semaphores to ensure the integrity of drawing operations
- minimizes screen flickering
- effective use of system resources

Whenever a drawing operation is performed, system resources are reserved and cursors are hidden and shown on the display.

3.6.1 Example Code

The following code draws a line and a rectangle with text on the display:

```c
DrawScreenItems(UGL_GC_ID gc)
{
    uglBatchStart(gc);
    uglLine(gc, 10, 70, 100, 10);
    uglRectangle(gc, 50, 50, 149, 79);
    uglText(gc, 4, 4, "This is the sample text");
    uglBatchEnd(UGL_GC_ID gc);
}
```

The calls to `uglBatchStart()` and `uglBatchEnd()` optimize the drawing of information to the display, therefore keeping software cursor flickering and system resource use to a minimum.
3.7 Graphics Contexts

The graphics context contains information about drawing characteristics. Most drawing operations can only be done by specifying a valid graphics context to determine the characteristics of how the operation is performed. The drawing characteristics stored in a graphics context include:

- information for drawing primitives, such as background and foreground colors, line style and width, fill patterns
- the default bitmap
- clipping and view port dimensions
- raster modes
- the font for text rendering (described in 3.3 Text Rendering and Font Management, p.52)

A graphics context is directly associated with a graphics or display device ID. The application can create multiple graphics contexts on each graphics device. Information on how to obtain the graphics device ID from the device registry can be found in 7. Resource Management.

Graphics Context Creation Routines

The routines used to create, copy, and destroy a graphics context are listed in Table 3-6.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglGcCreate()</td>
<td>Creates a graphics context.</td>
</tr>
<tr>
<td>uglGcDestroy()</td>
<td>Destroys a graphics context.</td>
</tr>
<tr>
<td>uglGcCopy()</td>
<td>Copies a graphics context.</td>
</tr>
</tbody>
</table>

Drawing Primitive Routines

The routines which set the display information for graphics primitives, such as foreground and background colors, line width and style, and fill patterns are listed in Table 3-7.
Default Bitmap

The destination of a drawing operation is determined by the setting of the default bitmap in the graphics context. The default bitmap may point to the physical display or an off-screen color bitmap. The physical display is identified by UGL_DISPLAY_ID. A color bitmap is UGL_DDB_ID. Bitmap operations can specify the default bitmap as the source or destination using the UGL_DEFAULT_ID macro.

The routines used to get and set the default bitmap are listed in Table 3-8.

### Table 3-8 Default Bitmap Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglDefaultBitmapSet()</td>
<td>Sets the default bitmap for a graphics context.</td>
</tr>
<tr>
<td>uglDefaultBitmapGet()</td>
<td>Gets the default bitmap for a graphics context.</td>
</tr>
</tbody>
</table>

Clipping and View Ports

In addition to containing information on drawing characteristics, the graphics context stores information about the clip region, clip rectangle, and view port:
the **clip rectangle** is defined as the rectangular area in the default bitmap to which drawing operations are restricted

- the **clip region** is an abstract area within the default bitmap defined by the union of a set of rectangles to which drawing operations are restricted

- the **view port**, similar to the clip rectangle, is defined as the rectangular area in the default bitmap to which drawing operations are restricted, with the addition that all coordinates (including the clip region and clip rectangle) are relative to the upper left corner of the view port rectangle.

The actual clipping area is defined by the intersection of the clip rectangle, clip region, and view port, where the coordinates of both the clip rectangle and clip region are relative to the upper left corner of the view port rectangle.

For example, in Figure 3-2, the display area is 640 x 480 pixels. The view port is defined to start at an \((x, y)\) location of \((30, 30)\) relative to the top left of the display. The view port is 200 pixels wide and 100 pixels high.

The intersection of the clip region and clip rectangle (indicated by the clip area) is specified using an origin relative to the view port, at an \((x, y)\) location of \((30, 20)\). The absolute coordinates of the clip area, relative to the upper left corner of the physical display, are \((60, 50)\).

**Figure 3-2  Display Regions and Coordinates**

When drawing routines are called, the coordinates specify an area within the view port. In other words, the coordinates in the drawing operation are relative to the coordinates of the upper left corner of the view port, not the upper left corner of the physical display. Once the coordinates have been transposed relative to the view port, the drawing is then clipped to both the rectangular boundary defined
by the view port and the boundary defined by the union of the clip region and clip rectangle.

Given the display, view port, and clipping area in Figure 3-2, the `uglLine()` and `uglText()` calls would result in the rendering in Figure 3-3.

![The Clipped Result](image)

**NOTE:** Setting the view port will always cause the clip rectangle's dimensions to match the dimensions of the view port rectangle. Any previous clip rectangle information is lost and must be set by the application after the view port is modified, if needed. The information in the clip region is left unchanged when the view port or clip rectangle is changed.

The routines which set clipping regions, rectangles, and viewports for a graphics context are listed in Table 3-9.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uglClipRectSet()</code></td>
<td>Sets the clipping rectangle of a graphics context.</td>
</tr>
<tr>
<td><code>uglClipRectGet()</code></td>
<td>Gets the clipping rectangle of a graphics context.</td>
</tr>
<tr>
<td><code>uglClipRegionSet()</code></td>
<td>Sets the clipping region of a graphics context.</td>
</tr>
<tr>
<td><code>uglClipRegionGet()</code></td>
<td>Gets the clipping region of a graphics context.</td>
</tr>
<tr>
<td><code>uglViewPortSet()</code></td>
<td>Sets the view port of a graphics context.</td>
</tr>
<tr>
<td><code>uglViewPortGet()</code></td>
<td>Gets the view port of a graphics context.</td>
</tr>
</tbody>
</table>
Raster Modes

Raster mode information for a particular operation is also set in the graphics context. The raster mode of a graphics context determines how a new drawing operation impacts the existing image on the display.

The core set of graphics context routines are listed in Table 3-10.

Table 3-10  Raster Mode Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglRasterModeSet()</td>
<td>Sets the raster mode of a graphics context.</td>
</tr>
<tr>
<td>uglRasterModeGet()</td>
<td>Gets the raster mode of a graphics context.</td>
</tr>
</tbody>
</table>

3.7.1 Example Code

This code example shows some basic drawing being done to a graphics context:

```c
int windMLExampleBasic (void)
{
    UGL_GC_ID gc;
    ...

    /*
    * Create a graphics context. Default values are set during
    * the creation.
    */
    gc = uglGcCreate(devId);

    /* -------------- Draw a line. -------------- */
    uglBatchStart(gc);
    uglForegroundColorSet(gc, colorTable[WHITE].uglColor);
    uglLine(gc, 0, 0, displayWidth - 1, displayHeight - 1);
    uglBatchEnd(gc);

    /* -------------- Draw a rectangle. ----------- */
    uglBatchStart(gc);
    uglFillPatternSet(gc, patternBitmap);
    uglForegroundColorSet(gc, colorTable[WHITE].uglColor);
    uglBackgroundColorSet(gc, colorTable[GREEN].uglColor);
    uglLineStyleSet(gc, UGL_LINE_STYLE_SOLID);
    uglLineWidthSet(gc, 7);
    uglRectangle(gc, displayWidth / 8, displayHeight / 8,
                 displayWidth / 4, displayHeight / 4);
    uglBatchEnd(gc);
```
3.8 Color Management

WindML's color management API is designed to allow portable application development. This means that a single application can be written that will run in multiple display modes, or on multiple types of display devices. In addition, the API allows you to optimize applications for specific display modes or architectures.

There are two general categories of display modes or architectures:

- **direct color**
  
  In direct color modes, all color data is contained within the frame buffer.
  
  For example, in RGB565 mode, each pixel in the frame buffer is represented by 16 bits of data: 5 bits each for the red and blue components of the color and 6 bits for the green component.

- **indexed color**
  
  In indexed color modes, each pixel in the frame buffer is represented by an index value. Each index value specifies an entry in a color lookup table (CLUT) which contains the color information for the corresponding pixel. Because the size of the CLUT is typically much less than the number of available colors, this restricts the number of colors that may be displayed simultaneously.

The color management API is listed in Table 3-11.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglColorAlloc()</td>
<td>Allocate colors to be used on a device.</td>
</tr>
<tr>
<td>uglColorFree()</td>
<td>Frees colors that are currently in use on a device.</td>
</tr>
</tbody>
</table>
3.8.1 Color Types

WindML provides three color type definitions:

- **UGL_ARGB**
  
  The **UGL_ARGB** color type is a 32-bit value with 1 byte each for alpha-channel, red, green, and blue components. The alpha-channel component is found in the most significant byte, and blue is found in the least significant byte.

  The alpha-channel byte defines the translucency of a color. A value of 255 defines a color that is completely opaque, while a value of 0 defines a color that is completely transparent. Alpha-channel colors may be used in a frame buffer, or an overlay, to determine how images are mixed on the display. The alpha-channel component is ignored on graphics drivers that don’t support alpha channels.

  The red, green, and blue bytes define intensities between 0 and 255 for their respective components of color.

- **UGL_RGB**

  A color type of **UGL_RGB** is the same as **UGL_ARGB**, but it has no alpha component.

- **UGL_COLOR**

  The **UGL_COLOR** color type is a 32-bit value with an internal color representation that is determined by the graphics driver. This color representation typically matches the color representation used by a graphics device’s frame buffer. Applications do not usually need to know the color representation used within **UGL_COLOR** values.

  Table 3-12 lists utility macros that create **UGL_ARGB** and **UGL_RGB** color values, and the color component values of these color types.

  The file `target/h/ugl/uglclr.h` contains additional color utility macros.
3.8.2 Color Formats

The following color formats are defined by WindML:

- **UGL_ARGB8888** - Specifies a 32 bit format with 8 bits for alpha channel, 8 bits for red, 8 bits for green, and 8 bits for blue. Alpha channel is represented in the most significant bits, and blue is represented in the least significant bits.

- **UGL_RGB888** - Specifies a 24 bit format with 8 bits for red, 8 bits for green, and 8 bits for blue. Red is represented by the first byte, green by the second byte, and blue by the third byte.

- **UGL_RGB565** - Specifies a 16 bit format, with 5 bits for red, 6 bits for green, and 5 bits for blue. Red is represented in the most significant bits, and blue is represented in the least significant bits.

### Table 3-12 Color Macros

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UGL_MAKE_ARBG()</strong></td>
<td>Generates a UGL_ARGB value from alpha, red, green, and blue components.</td>
</tr>
<tr>
<td><strong>UGL_MAKE_RGB()</strong></td>
<td>Generates a UGL_RGB value from red, green and blue components.</td>
</tr>
<tr>
<td><strong>UGL_RGB_RED()</strong></td>
<td>Returns a value, between 0 and 255, of the red component of a UGL_RGB color.</td>
</tr>
<tr>
<td><strong>UGL_RGB_GREEN()</strong></td>
<td>Returns a value, between 0 and 255, of the green component of a UGL_RGB color.</td>
</tr>
<tr>
<td><strong>UGL_RGB_BLUE()</strong></td>
<td>Returns a value, between 0 and 255, of the blue component of a UGL_RGB color.</td>
</tr>
<tr>
<td><strong>UGL_ARGB_ALPHA()</strong></td>
<td>Returns a value, between 0 and 255, of the alpha component of a UGL_ARGB color.</td>
</tr>
<tr>
<td><strong>UGL_ARGB_RED()</strong></td>
<td>Returns a value, between 0 and 255, of the red component of a UGL_ARGB color.</td>
</tr>
<tr>
<td><strong>UGL_ARGB_GREEN()</strong></td>
<td>Returns a value, between 0 and 255, of the green component of a UGL_ARGB color.</td>
</tr>
<tr>
<td><strong>UGL_ARGB_BLUE()</strong></td>
<td>Returns a value, between 0 and 255, of the blue component of a UGL_ARGB color.</td>
</tr>
</tbody>
</table>

---
- **UGL DEVICE COLOR** - Specifies an internal format used by the driver. The size and type of information contained in a UGL DEVICE COLOR is dependent on the underlying definition used by the driver. For example, a 565 RGB color system may represent color values in a compressed 16 bit integer format, or because of compiler register optimizations, may still store the value as a 32 bit integer, even though only 16 bits of information are used.

- **UGL DEVICE COLOR_32** - Specifies a format is similar to UGL DEVICE COLOR, but is always stored as a 32 bit value. For example, a 565 RGB capable graphics driver may store the actual underlying RBG value as a 16 bit integer, but specifying UGL DEVICE COLOR_32 will force the underlying driver to communicate with SDK routines through 32 bit values.

WindML provides the ability to convert colors between various types and formats. This functionality is provided by the `uglColorConvert()` routine. All conversion possibilities are not necessarily provided by `uglColorConvert()`. See the reference API documentation for more information about which conversions are available.

### 3.8.3 Using the API

#### Direct CLUT Management

Sometimes, you may want your application to have direct access to a device's CLUT. For example, the CLUT can be manipulated to achieve special effects such as blinking and fading or animations. You can directly access a device's CLUT using the routines `uglClutSet()` and `uglClutGet()`.

These routines should be used with caution, as they do not update the use count or free list assigned to the colors in the CLUT. It is possible to populate the CLUT with these routines, but not recommended as other WindML routines (such as the bitmap creation routines) may use the CLUT internally. These routines will not be aware that any colors have been allocated and will fail.

In direct color modes, these routines have no effect and return `UGL_STATUS_ERROR`.

Under normal circumstances, applications should use the `uglColorAlloc()` and `uglColorFree()` routines.
Indexed CLUT Management

In indexed modes, it can be difficult to manage a device’s color lookup table to handle the requirements of multiple images that must be displayed simultaneously, or multiple applications that must share a single display. There are two strategies that can be used to handle these situations:

- color cubes and dithering
- dynamic color allocation

Color Cubes and Dithering

A color cube algorithm can be used to generate a good spectrum of colors which can be written to the CLUT. The red, green, and blue color components of color each represent an axis on the cube. After the color cube is allocated, the CLUT remains static. Requested colors are assigned the nearest matching color in the color cube, and dithering can then be used to achieve maximum image quality.

The following code segment sets the color model of an application:

```c
void setUglColorModel (void)
{
    UGL_ORD i;

    /* Check the color model and space */
    if (uglColorModel == UGL_CMODEL_DIRECT &&
        uglColorSpace == UGL_CSPACE_RGB &&
        uglColorDepth > 8)
    {
        /* No additional work is needed */
        return;
    }
    else if (uglColorSpace == UGL_CSPACE_RGB &&
             uglColorDepth > 4)
    {
        /* INDEXED-RGB color model */
        /* Determine the optimal color cube */
        UGL_ORD r, g, b;
        if (uglColorDepth == 8)
            maxRed = 4, maxGreen = 9, maxBlue = 6;
        else
        {
            maxRed = maxGreen = maxBlue = 4;
            while ((maxRed + 1) * (maxGreen + 1) *
                   (maxBlue + 1) < uglAssignedColors)
                maxRed++, maxGreen++, maxBlue++;
        }

        /* Allocate the colors in the cube */
        for (i = 0, r = 0; r < maxRed; r++)
            for (g = 0; g < maxGreen; g++)
```
for (b = 0; b < maxBlue; b++, i++)
{
    rgbMap[i].red = r * 0xff / (maxRed - 1);
    rgbMap[i].green = g * 0xff / (maxGreen - 1);
    rgbMap[i].blue = b * 0xff / (maxBlue - 1);
    clrMap[i] = uglColorAlloc(root_gc, -1,
        UGL_MAKE_RGB(rgbMap[i].red,
        rgbMap[i].green,
        rgbMap[i].blue));
}

else if (uglColorSpace == UGL_CSPACE_RGB)
{
    /* Set up the optimal color table */
    RGBTuple color[] =
    { /* Depth == 1 */
        { 0, 0, 0 },
        { 255, 255, 255 },
    /* Depth == 2 */
        { 128, 128, 128 },
        { 192, 192, 192 },
    /* Depth == 3 */
        { 0, 0, 255 },
        { 0, 255, 0 },
        { 255, 0, 0 },
    /* Depth == 4 */
        { 0, 0, 128 },
        { 0, 128, 0 },
        { 0, 128, 128 },
        { 128, 0, 0 },
        { 128, 0, 128 },
        { 128, 128, 0 },
        { 128, 128, 128 },
        { 255, 0, 255 },
        { 255, 255, 0 }
    };

    /* Allocate the colors in the table */
    for (i = 0; i < uglAssignedColors; i++)
    {
        rgbMap[i] = color[i];
        clrMap[i] = uglColorAlloc(root_gc, -1,
            UGL_MAKE_RGB(rgbMap[i].red,
            rgbMap[i].green,
            rgbMap[i].blue));
    }

    else if (uglColorSpace == UGL_CSPACE_GRAY ||
        uglColorSpace == UGL_CSPACE_MONO)
    {
        /* INDEXED-GRAY and INDEXED-MONO color models */
        int color;
        for (i = 0; i < uglAssignedColors; i++)
        {
            /*...*/
        }
    }
}
Dynamic Color Allocation

Colors are allocated when needed, and freed when no longer required. When an image needs to be displayed, the colors required by that image must first be allocated. When the image is no longer displayed, these colors should then be freed. As the CLUT becomes full—as images and/or applications request colors—nearest color matching occurs, and used counts increase. As images and applications are closed, used counts decrease, and more colors become available.

For an example of this type of index color management, see the example program in `target/src/ugl/examples/basics/wexbasic.c`.

Palette Management

An application or application manager takes complete control of the CLUT using `uglClutSet()`. Color arbitration is then under complete control of the application and colors can be used however it sees fit. In this case, a palette manager might give priority to images in the foreground while images in the background lose image quality.

### 3.8.4 Example Code

This code segment sets up a CLUT and associated colors:

```c
color = i * 0xff / (uglAssignedColors - 1);
rgbMap[i].red = color;
rgbMap[i].green = color;
rgbMap[i].blue = color;
clrMap[i] = uglColorAlloc(root_gc, -1,
        UGL_MAKE_RGB(rgbMap[i].red,
                      rgbMap[i].green,
                      rgbMap[i].blue));
}
}
```

/*
 * The color table is where we define the colors we want
 * to have available. The format is an array of
 * ARGB values paired with their allocated uglColor. As
 * of this writing, we don't need to worry about Alpha
 * ("A") values unless we are using video.
 */
struct _colorStruct
{
    UGL_ARGB rgbColor;
    UGL_COLOR uglColor;
}

colorTable[] =
{
    { UGL_MAKE_ARGB(0xff, 0, 0, 0), 0 }, /* The color table uses ARGB's */
    { UGL_MAKE_ARGB(0xff, 0, 0, 168), 0 }, /* (see uglColorAlloc). */
    { UGL_MAKE_ARGB(0xff, 0, 168, 0), 0 }, /* Initialize alpha to 255 for */
    { UGL_MAKE_ARGB(0xff, 0, 168, 168), 0 }, /* now (opaque). */

    { UGL_MAKE_RGB(168, 0, 0), 0 }, /* UGL_MAKE_RGB takes care of */
    { UGL_MAKE_RGB(168, 0, 168), 0 }, /* the alpha for us. */
    { UGL_MAKE_RGB(168, 84, 0), 0 },
    { UGL_MAKE_RGB(168, 168, 168), 0 },
    { UGL_MAKE_RGB(84, 84, 84), 0 },
    { UGL_MAKE_RGB(84, 84, 255), 0 },
    { UGL_MAKE_RGB(84, 255, 84), 0 },
    { UGL_MAKE_RGB(84, 255, 255), 0 },
    { UGL_MAKE_RGB(255, 84, 84), 0 },
    { UGL_MAKE_RGB(255, 84, 255), 0 },
    { UGL_MAKE_RGB(255, 255, 84), 0 },
    { UGL_MAKE_RGB(255, 255, 255), 0 }
};

/* Label the colors we defined */

#define BLACK (0)
#define BLUE (1)
#define GREEN (2)
#define CYAN (3)
#define RED (4)
#define MAGENTA (5)
#define BROWN (6)
#define LIGHTGRAY (7)
#define DARKGRAY (8)
#define LIGHTBLUE (9)
#define LIGHTGREEN (10)
#define LIGHTCYAN (11)
#define LIGHTRED (12)
#define LIGHTMAGENTA (13)
#define YELLOW (14)
#define WHITE (15)

int windMLExampleBasic (void)
{
    . . .

    /*
    * Initialize colors. UGL maintains a Color Look-Up Table (CLUT)
    * for devices that do not represent colors directly. Essentially
    * some hardware is only able to represent a subset of colors at
* any given time. To manage which colors will be available for rendering, UGL uses color allocation. If the hardware is able to represent colors directly, then the uglColorAlloc() function still works, but it is then essentially a no-op.
* We have set up for 16 colors, so here we will "allocate" them within UGL's CLUT (sometimes referred to as a "palette"). Colors can also be de-allocated, or freed, with uglColorFree.
* Since the ARGB's are intermingled with the UGL_COLORs in the colorTable, we must allocate each color individually.
* If the ARGB's had a contiguous array of ARGBs, and likewise for the UGL_COLORs, a single uglColorAlloc call could be made.
* (see the windows example).
*/

uglColorAlloc(devId, &colorTable[BLACK].rgbColor, UGL_NULL, &colorTable[BLACK].uglColor, 1);
uglColorAlloc(devId, &colorTable[BLUE].rgbColor, UGL_NULL, &colorTable[BLUE].uglColor, 1);
uglColorAlloc(devId, &colorTable[GREEN].rgbColor, UGL_NULL, &colorTable[GREEN].uglColor, 1);
uglColorAlloc(devId, &colorTable[CYAN].rgbColor, UGL_NULL, &colorTable[CYAN].uglColor, 1);
uglColorAlloc(devId, &colorTable[RED].rgbColor, UGL_NULL, &colorTable[RED].uglColor, 1);
uglColorAlloc(devId, &colorTable[MAGENTA].rgbColor, UGL_NULL, &colorTable[MAGENTA].uglColor, 1);
uglColorAlloc(devId, &colorTable[BROWN].rgbColor, UGL_NULL, &colorTable[BROWN].uglColor, 1);
uglColorAlloc(devId, &colorTable[LIGHTGRAY].rgbColor, UGL_NULL, &colorTable[LIGHTGRAY].uglColor, 1);
uglColorAlloc(devId, &colorTable[DARKGRAY].rgbColor, UGL_NULL, &colorTable[DARKGRAY].uglColor, 1);
uglColorAlloc(devId, &colorTable[LIGHTBLUE].rgbColor, UGL_NULL, &colorTable[LIGHTBLUE].uglColor, 1);
uglColorAlloc(devId, &colorTable[LIGHTGREEN].rgbColor, UGL_NULL, &colorTable[LIGHTGREEN].uglColor, 1);
uglColorAlloc(devId, &colorTable[LIGHTCYAN].rgbColor, UGL_NULL, &colorTable[LIGHTCYAN].uglColor, 1);
uglColorAlloc(devId, &colorTable[LIGHTRED].rgbColor, UGL_NULL, &colorTable[LIGHTRED].uglColor, 1);
uglColorAlloc(devId, &colorTable[LIGHTMAGENTA].rgbColor, UGL_NULL, &colorTable[LIGHTMAGENTA].uglColor, 1);
uglColorAlloc(devId, &colorTable[YELLOW].rgbColor, UGL_NULL, &colorTable[YELLOW].uglColor, 1);
uglColorAlloc(devId, &colorTable[WHITE].rgbColor, UGL_NULL, &colorTable[WHITE].uglColor, 1);

...
3.9 Double Buffering

Double buffering is used primarily to eliminate the flickering that can accompany a set of drawing operations. Using the double buffering API, the programmer can first draw a set of objects to an off-screen page or buffer and then, when the drawing operation is complete, set the off-screen page to become visible on the physical display. Pages or buffers are always the same size as the physical display or screen.

The double buffering API can be used to create and manage multiple off-screen buffers. The page where drawing takes place can be set independently of the visible page. When the default bitmap member of the graphics context is set to UGL_DISPLAY_ID, subsequent drawing takes place on the active draw page even if the draw page is not currently set to be the visible page.

The double buffering API is listed in Table 3-13.

Table 3-13 Double Buffering API

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglPageCopy()</td>
<td>Copies one double buffer page to another.</td>
</tr>
<tr>
<td>uglPageCreate()</td>
<td>Create a page.</td>
</tr>
<tr>
<td>uglPageDestroy()</td>
<td>Destroy a page.</td>
</tr>
<tr>
<td>uglPageDrawGet()</td>
<td>Get the active drawing page.</td>
</tr>
<tr>
<td>uglPageDrawSet()</td>
<td>Set the active drawing page.</td>
</tr>
<tr>
<td>uglPageVisibleGet()</td>
<td>Get the active display page.</td>
</tr>
<tr>
<td>uglPageVisibleSet()</td>
<td>Set the visible display page.</td>
</tr>
</tbody>
</table>

3.9.1 Example Code

```c
void pageDemo(void)
{
    ...
    /* Draw an ellipse to an off-screen page, then set the page to be visible on the display */
    page1 = uglPageCreate (devId);
```
/* Initialize graphics context */
uglDefaultBitmapSet (gc, UGL_DISPLAY_ID);
uglForegroundColorSet (gc, colorTable[RED].uglColor);
uglBackgroundColorSet (gc, colorTable[GREEN].uglColor);

uglPageDrawSet (devId, page1);
uglEllipse (gc, 400, 80, 600, 160, 0, 0, 0, 0);
uglPageVisibleSet (devId, page1);
... }
4.1 Introduction

WindML provides a scalable and extensible mechanism for communicating input events from input drivers and external events to the application. Its scalability allows a minimal footprint for simple input event retrieval. Its extensibility enables it to provide services to high level systems such as application tasks. The developer accesses this functionality using the Event Service API.

During `углInitialize()` (uglInitialize()), an event service is created to handle events for one or more input devices.

An event service performs three major tasks:

- it gets raw input data from an input device and wraps this data into a WindML event
- it passes the WindML event to the event handler for further processing (frontend processing, cursor movement, event routing, and so on)
- it passes the WindML event to an application queue

A high level diagram of an event service is shown in Figure 4-1.

While it is possible to have multiple event services in operation at one time, typically there will only one event service in operation per session. That is, one event service processes the events coming from all input devices.
4.2 Event Service Components

An event service consists of several components:

- The input task

  The input task is created when an event service is created. The input task pends on activity from the input devices associated with the event service using `select()`. When the input device has raw data available, the input task wakes up and calls the input driver associated with the input device to process and package the raw data into a WindML Event. Once the WindML Event has been processed, the event is passed to the event handler.

- the input driver

  The input driver is invoked by the input task when it detects activity on an input device. There is one input driver for each input device. The input driver
reads raw data from the input device, formats it as necessary, and assembles the processed data into a WindML event. The API for input drivers is described in 10. Input Drivers.

- The event handler

The event handler use any callbacks provided by the application to handle or further process the WindML event. The event handler will pass the event to a callback only if the callback has registered for that particular event type. Once the callback processing is complete, the event handler passes the event to the appropriate event queue. Each event service has one event handler. Events can be passed to the event handler from external sources also.

The routine in the event service API are shown in Table 4-1.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglEventHandlerCreate()</td>
<td>Creates an event handler.</td>
</tr>
<tr>
<td>uglEventHandlerDestroy()</td>
<td>Destroys an event handler.</td>
</tr>
<tr>
<td>uglEventHandler()</td>
<td>Causes an event handler to handle an event.</td>
</tr>
<tr>
<td>uglEventCallbackAdd()</td>
<td>Adds a callback to an event handler.</td>
</tr>
<tr>
<td>uglEventCallbackRemove()</td>
<td>Removes a callback from an event handler.</td>
</tr>
<tr>
<td>uglEventCallbackChain()</td>
<td>Chains a callback to an event handler.</td>
</tr>
<tr>
<td>uglEventServiceCreate()</td>
<td>Creates an event service.</td>
</tr>
<tr>
<td>uglEventServiceDestroy()</td>
<td>Destroys an event service.</td>
</tr>
<tr>
<td>uglEventHandlerGet()</td>
<td>Gets an event service’s event handler.</td>
</tr>
<tr>
<td>uglEventHandlerSet()</td>
<td>Sets an event service’s event handler.</td>
</tr>
<tr>
<td>uglEventQCreate()</td>
<td>Creates an event queue.</td>
</tr>
<tr>
<td>uglEventQDestroy()</td>
<td>Destroys an event queue.</td>
</tr>
<tr>
<td>uglEventPost()</td>
<td>Posts an event to an event service.</td>
</tr>
<tr>
<td>uglEventGet()</td>
<td>Gets an event from an event queue.</td>
</tr>
<tr>
<td>uglInputEventGet()</td>
<td>Gets an event from an event service.</td>
</tr>
</tbody>
</table>
4.3 Using the API

4.3.1 WindML Events

WindML encapsulates all of the information it requires to process an event in an event structure. The base event structure, UGL_EVENT, is a generic event structure. Typically, more specific events are derived for particular classes of input devices. Event structures are defined in the following files:

target/h/ugl/uglevent.h

target/h/ugl/uglinput.h

All events contain an event header as their first field, which stores the type and category of event. This structure is shown as follows:

```c
typedef struct ugl_event_header
{
    UGL_INT32 type;    /* type of event */
    UGL_INT32 category; /* category of event */
    UGL_ID objectId;  /* object id */
} UGL_EVENT_HEADER;
```

The fields are described as follows:

**type**

The **type** field specifies the type of an event. This must be an exact type. Currently, two event types are pre-defined, **UGL_EVENT_TYPE_KEYBOARD** and **UGL_EVENT_TYPE_POINTER**. Positive value events types are reserved for WindML definitions. Negative values may be used to define additional application event types.

**category**

The **category** field defines a category of events. Currently, only one event category is defined by WindML, **UGL_EVENT_CATEGORY_INPUT**. Positive event category values are reserved for WindML definitions. Negative values may be defined by applications.

**objectId**

The **objectId** field is used for event routing, in conjunction with windowing or some other high-level operation.
The UGL_EVENT_INPUT structure which encapsulates an input event is shown as follows:

```c
/* UGL input event */
typedef struct ugl_input_event
{
    UGL_EVENT_HEADER header; /* event identifier */
    UGL_TIMESTAMP timeStamp; /* time event queued */
    UGL_POS x;              /* current X position */
    UGL_POS y;              /* current Y position */
    UGL_UINT32 modifiers; /* keyboard modifier keys and */ /* pointer button states */
    union
    {
        UGL_EVENT_KEYBOARD keyboard; /* keyboard data */
        UGL_EVENT_POINTER pointer; /* pointer data */
    } type;
} UGL_INPUT_EVENT;
```

Extra information is stored that is specific to input events, and also to the specific type of input event. Derived event structures can be downcast to the base event type.

### 4.3.2 Event Handling

When the input task has assembled a complete input event, it posts the event to the event service. The event handler associated with the event service takes the event and may perform further processing on it. Ultimately, it returns an event to the input task which then posts it to an event queue.

Event handlers use callbacks to process an event, see Figure 4-2. The specific callback invoked depends on the type and category of the event. You can also chain additional callbacks for an event type. In this situation, all the callbacks chained for a particular event will be called sequentially.

Figure 4-2 shows an event handler with a list of two callbacks for two event types. Event types with the value 2 are handled by three chained callbacks.

Additional processing performed by callback functions might be Front End Processing (FEP) or cursor manipulation.

### 4.3.3 Event Queues

The application is responsible for creating and destroying any event queues that it requires. If multiple queues are created, the application is responsible for
Event queues can be created and destroyed using `uglEventQCreate()` and `uglEventQDestroy()`. Events are retrieved from a queue by calling `uglEventGet()`. The `EVENT_SERVICE_ID` is required to create a queue and can be obtained from the device registry described in 7.4 Device Driver Registry, p.129.

### 4.4 Example Code

The code below shows a routine that assigns a callback to an event service and reads the events from an event queue. The callback routine handles moving the mouse cursor.

```c
UGL_EVENT_SERVICE_ID eventServiceId;
UGL_EVENT_Q_ID qId;

int demo (void)
{
    UGL_EVENT_HANDLER_ID eventHandlerId;
    ...

    /* obtain the input service identifier */
    uglDriverFind (UGL_EVENT_SERVICE_TYPE, 0, (UGL_UINT32 *)&eventServiceId);
```
qId = uglEventQCreate (eventServiceId, 100);
uglEventHandlerGet (eventServiceId, &eventHandlerId);
uglEventCallbackAdd (eventHandlerId, UGL_EVENT_TYPE_POINTER,
                    UGL_EVENT_CATEGORY_INPUT, uglCallbackCursorMove);

/* The mouse will move as the user moves it until a key is pressed */
mouseMove();
uglEventCallbackRemove(eventHandlerId, UGL_EVENT_TYPE_POINTER,
                       UGL_EVENT_CATEGORY_INPUT, uglCallbackCursorMove);
uglEventQDestroy (eventServiceId, qId);
...

void mouseMove(void)
{
    while(1)
    {
        status = uglEventGet (qId, &event, sizeof (event), UGL_WAIT_FOREVER);
        if (status != UGL_STATUS_Q_EMPTY)
        {
            if (event.header.type == UGL_EVENT_TYPE_KEYBOARD)
            {
                UGL_INPUT_EVENT * pInputEvent = (UGL_INPUT_EVENT *)&event;
                if (pInputEvent->modifiers & UGL_KEYBOARD_KEYDOWN)
                    break;
            }
            ...
        }
    }
}
5

Region and Window Management

5.1 Introduction

WindML provides a region management API and a windowing API. You can use the region management API to define clip regions into which you can draw using the 2D Graphics API. If your application is more sophisticated, or if you will be sharing the display between multiple threads or tasks, you can use the windowing API. The windowing API lets you create, display, and manipulate windows and to process events destined for those windows. The approach you take depends on the requirements of your application. This chapter looks at the region and windowing APIs and discusses the different methods of event handling used by each API.

5.2 Regions and Clipping

WindML defines regions in two functional areas:

- as a drawing area or areas, used in combination with clipping, where you can use the 2D graphics API
- as a part of windowing

This section only describes regions management outside of the windowing API.
Regions are most often used for clipping. A region is the area enclosed by a group of rectangles. The region shown in Figure 5-1 is composed of six rectangles.

![Region of Six Rectangles](image)

A region is always optimized to enclose the specified area with a minimum number of rectangles. The region management API employs a sophisticated algorithm to recycle rectangles in such a way that memory allocations are minimal.

The region management API is listed in Table 5-1.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglRegionCreate()</td>
<td>Creates a region.</td>
</tr>
<tr>
<td>uglRegionDestroy()</td>
<td>Destroys a region.</td>
</tr>
<tr>
<td>uglRegionCopy()</td>
<td>Copies a region.</td>
</tr>
<tr>
<td>uglRegionRectInclude()</td>
<td>Includes a rectangle in a region.</td>
</tr>
<tr>
<td>uglRegionRectExclude()</td>
<td>Excludes a rectangle from a region.</td>
</tr>
<tr>
<td>uglRegionIntersect()</td>
<td>Finds the intersection of two regions.</td>
</tr>
<tr>
<td>uglRegionEmpty()</td>
<td>Empties the contents of a region.</td>
</tr>
<tr>
<td>uglRegionMove()</td>
<td>Moves a region.</td>
</tr>
<tr>
<td>uglRegionUnion()</td>
<td>Finds the union of two regions.</td>
</tr>
</tbody>
</table>
5.2.1 Clipping Regions

Clipping regions restrict the area in which drawing operations may take place. A graphics context stores a clipping rectangle and a region.

The intersection of the clipping rectangle and the clipping region determines the active clipping area. In addition, the clipping rectangle is always constrained to the graphics context's view port and all operations are performed relative to this view port's coordinate system. (See 3.3.6 Graphics Contexts for more information about clip rects and view ports.)

The clipping and view port routines are listed in Table 5-2.

Table 5-2 Clipping Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglClipRectGet()</td>
<td>Gets the clipping rectangle from a GC.</td>
</tr>
<tr>
<td>uglClipRectSet()</td>
<td>Sets a clipping rectangle into a GC.</td>
</tr>
<tr>
<td>uglClipRegionGet()</td>
<td>Gets the clipping region from a GC.</td>
</tr>
<tr>
<td>uglClipRegionSet()</td>
<td>Sets a clipping region into a GC.</td>
</tr>
<tr>
<td>uglViewPortGet()</td>
<td>Gets the view port from a GC.</td>
</tr>
<tr>
<td>uglViewPortSet()</td>
<td>Sets a view port into a GC.</td>
</tr>
</tbody>
</table>

5.2.2 Using the API

A graphics context’s clip region may be set to NULL, in which case only the clipping rectangle is involved in clipping operations. (A clipping rectangle is automatically set by WindML when a view port is specified. A NULL clip region encloses all space.)

By default, a graphics context’s clip rectangle is equal to the entire display or default bitmap and its clip region is NULL.

NOTE: A clipping rectangle is copied into the graphics context, while only the ID of a clipping region is copied. For this reason, care must be used to avoid destruction or modification of a region that is use by a graphics context.

View ports can be instead of windows if you are guaranteed that they will never overlap.
5.2.3 Example Code

A graphical meter is displayed on part of the display, with its dimensions stored in the structure `meterRect`. It has its own task to update its display. The view port for the meter task’s graphics context, `meterGc`, can be set up as follows:

```c
/* Set the view port to be used for the meter */
uglViewPortSet (meterGc, meterRect.left, meterRect.top,
                meterRect.right, meterRect.bottom);
```

The routine `uglViewPortSet()` automatically sets the clip rect of the graphics context to `meterRect`. Drawing primitives for the meter task can now use the top left corner of the viewport as the (0, 0) origin.

A second example assumes that the graphical meter is part of a larger control panel on the display, and that the control panel and the meter both have their own drawing tasks. The dimensions of the control panel are defined in `cPanelRect`, and its graphics context is `cPanelGc`. To ensure that the control panel does not draw over the meter, the control panel uses the following code to set its clip region:

```c
/* Set the clip region to be used for the control panel */
UGL_REGION_ID cPanelRegionId;
cPanelRegionId = uglRegionCreate();
uglRegionRectInclude (cPanelRegionId, cPanelRect.left, cPanelRect.top,
                      cPanelRect.right, cPanelRect.bottom);
uglRegionsRectExclude (cPanelRegionId, meterRect.left, meterRect.top,
                      meterRect.right, meterRect.bottom);
uglClipRegionSet (cPanelGc, cPanelRegionId);
```

The final example uses the windowing APU. It assumes that the control panel is drawn in a window which can be overlapped and therefore obscured by other windows. To ensure that the control panel does not redraw itself if it is not visible, its clip region is set to the intersection of the control panel’s region and the visible region of the window. In the example below, `windowId` is the ID of the window containing the control panel:

```c
/* set the clip region to be used for the control panel */
UGL_REGION_ID drawRegionId;
UGL_REGION_ID visibleRegionId;
drawRegionId = uglRegionCreate();
visibleRegionId = winVisibleRegionGet (windowId);
uglRegionIntersect (cPanelRegionId, visibleRegionId, drawRegionId);
uglClipRegionSet (cPanelGc, drawRegionId);
```
5.3 Window Management

Windows allow multiple applications with graphical interfaces to coexist on one display. A window is a rectangular object that can be programmatically moved, sized, overlapped, and contained within another window. Windows typically are responsible for doing their own drawing, redrawing themselves as windows are rearranged, and processing events that are sent to them.

5.3.1 Containment and Z-Ordering

Three factors determine how and when windows need to redraw themselves:
- containment
- z-ordering
- visible and dirty regions

Containment

Containment refers to the parent-child relationship between windows. A containment hierarchy refers to the tree of parent-child relationships between a group of windows. A root window lies at the base of a containment hierarchy and generally occupies an entire display.

A window becomes the child of another window when it is attached to that window using the `winAttach()` routine. This action also places the child window in the containment hierarchy. A window can be a child of a window and also a parent to other windows. Figure 5-2 displays a single containment hierarchy of windows.

The diagram on the right shows the parent-child relationship.

The child window is always constrained by its parent. While a child may be positioned partially or completely outside its parent window, it is only visible where it overlaps with the parent. In Figure 5-2, notice how the display area of child window E is clipped to the area of its parent window, 2.

Usually windows belong to a single containment hierarchy, but multiple hierarchies can exist to support multiple displays or other specialized purposes.
While containment ensures that child windows are drawn on top of parent windows, a further ordering applies between sibling windows—the z-order.  

The z-order determines which windows are obscured when sibling windows overlap and occupy the same area on the display. Each window has a z-order that is assigned to it when it is attached to a parent. The z-ordering of a window can be changed using the routine `winZPosSet()`. If you want to make a partially obscured child window redraw itself on top of all other child windows, you can call `winRaise()`. This routine raises the child to the top of the z-order.

All parent windows maintain a list of their child windows. This list is updated as child windows are attached to the parent using `winAttach()`. The order of a child window in the child window list determines its z-order. The last window attached to a parent has the highest z-order (because it is the last child stacked onto the parent).

**NOTE:** The order of a window in WindML’s internal list is the opposite of the order on the display. The top most window on the display (z-order of 1) is actually the last element of the list.

---

1. The term z-order refers to the placement of windows along an imaginary z-axis that extends out from the display, perpendicular to the display itself.
Visible and Dirty Regions

Each window maintains two regions—a visible region and a dirty region. The visible region of a window is the area of that window that is not obscured by any sibling window or any higher level window (that is, parent, grandparent, and so on). The dirty region of a window is that area of the window that needs to be redrawn—because, for instance, windows have been rearranged so that a previously obscured area now becomes visible. By default, both visible and dirty regions include the regions of any child windows, but the child window regions may be excluded depending on the type of region management selected for that window.

Figure 5-3 displays the visible and dirty regions of a window.

Figure 5-3 Example of Visible and Dirty Regions

In this example, Window 1 and Window 2 are siblings. Window 3 is a child of Window 1. The shaded area in the left hand image is the area that Window 1 is responsible for drawing—that is, its visible region. That area excludes the areas of Window 3 and Window 2, as they perform their own drawing and are on top of Window 1.

In the example on the right hand side, Window 1 has been raised above Window 2. The dark region—Window 1’s dirty region—must be redrawn by Window 1.

5.3.2 Region Management Algorithms - Painters and Clip Children

By default, new windows are managed using the painters region management algorithm. Alternatively, you can use the clip children algorithm, which can be specified on a per window basis. The routine winClipModeSet() allows you to set
the algorithm used by windows. The routine `winClipModeGet()` gets the clipping mode being used by a window.

**Painters Algorithm**

The painters algorithm assumes that dirty windows will be redrawn bottom to top just as a painter would apply paint. If two colors are to be placed in the same area, the bottom coat (bottom-most window) is applied first, then the next color (next window) is applied over the top of the first.

This algorithm has the advantage of familiarity for traditional desktop-like user interfaces and often seems faster. Because the painters algorithm is simple, it also can be significantly faster than the clip children algorithm for typical, broad window hierarchies (that is, hierarchies that are not very depth, but are many children wide.)

**Clip Children Algorithm**

The clip children algorithm may be appropriate for very deep window hierarchies where painting windows from back to front may be more expensive than the computations required to track complex regions and clip drawing to them. Also, clip children works best in situations where a window should not be erased by its parent.

**Guidelines for Selecting an Algorithm**

These are some guidelines for selecting an appropriate region management algorithm for your windows:

- If your window hierarchy is broad but not deep, choose painters (the default) for best performance. If you are unsure how to characterize your window hierarchy, use this algorithm as a best guess. It yields better performance in most typical applications.
- If your window hierarchy is deep but not broad, choose clip children for best performance.
- If your application approximates a traditional desktop GUI, choose painters to provide a user experience similar to desktop windowing systems.
- If your windows must not erase their children, perhaps because this flicker effect could be too distracting, choose clip children.
- If your child window is being drawn by a different task than its parent, choose clip children to ensure that the child window does not draw out of order or disappear for too long.
5.3.3 Event Routing

All root windows have an associated event router that receives events from an event service and routes them based on its window hierarchy. To accomplish this, an event router makes an association between a root window, a display device, and an event service. This association is formed in three steps:

1. When an event router is created (with the routine `winEventRouterCreate()`), it associates a display with an event service, usually created with `uglInitialize()`. This association is stored in a data structure called an application context.

2. A root window is created and associated with the application context.

3. The application that created the event router then associates the event router with the root window (with the routine `winRootSet()`).

This relationship ensures the following:

- The root window and any windows in its containment hierarchy will be displayed on the display device.
- The root window and any windows in its containment hierarchy will receive input events from that event service. The event router also provides the information necessary for input events to be routed to the appropriate application event queue.

Windowing applications in the same containment hierarchy, that is, displaying on one display, require only one event router. This event router is normally created in its own task, which also creates a cursor, and processes events for the associated root window.

When an event router is used by more than one application, the event router should be registered by calling `uglDriverRegister()`. This allows applications to access the event router using `uglDriverFind()`.

Event Router Callbacks

When an event router is created, it automatically adds two input hooks to its associated event service:

- the event router input hook itself

  This hook determines which window an input event belongs to. It then sets the event’s `objectId` field to the Id of the window and routes the event to the window’s application event queue. See 5.3.6 Window Event Processing, p.106 for more details.
the cursor move input hook

This callback moves a cursor on the associated display when a pointer device reports movement.

Application Contexts

An application context stores common information required by an application—essentially global data. The information stored in the application context, which each window has a pointer to, includes:

- an event queue
- an event router
- an display device
- an event service
- an event processing thread

In addition, the application context keeps track of dirty windows (that is, windows with regions that need to be redrawn). The windowing system uses this dirty window list to generate draw events which it sends to the dirty windows at appropriate times.

The application context also provides security for thread-safe operations. For instance, a window may only be destroyed by the same task that processes events for that window. The application context can enforce this because it knows which thread processes the window’s events.

All windows require an application context ID when they are created. The application context itself is created by the `winAppCreate()` routine, which takes an event router ID as a parameter. You can pass NULL in as the event router ID, in which case `winAppCreate()` will look for a registered event router. Typically only one event router exists, which is instance zero, and this is what `winAppCreate()` uses.

**NOTE:** A single application can have more than one application context, although this would not be a typical situation. If one window in the application needs to have a higher priority than the other windows in the application, you could create a separate application context for that window and associate it with a higher priority thread or task to draw the window.
Setting up the Association

The following code segment shows how an event router is typically created:

```c
/* create event router and register it*/

eventRouterId = winEventRouterCreate (displayId, eventServiceId);
uglDriverRegister (UGL_EVENT_ROUTER_TYPE, 0, (UGL_UINT32)eventRouterId);

/* create app context, and root window */
appId = winAppCreate (eventRouterId, 100);
rootWindowId = winCreate (appId, 0, 0, modeInfo.width, modeInfo.height);

/* associate event router with root window */
winRootSet (eventRouterId, rootWindow);
```

5.3.4 Window Creation

A window is created using the `winCreate()` routine. This routine takes the window’s position, size, and an application context as parameters. The position specifies the location of the top-left corner of the window relative to its parent or to the display, if the window will be the root window.

The `winAttach()` routine attaches a window to another window as a child. Until a window is either attached to another window, or set as the root window, it will not appear on the display. Similarly, the associated application thread will not receive events for that window. `winAttach()` takes the ID of a parent window as a parameter. If this is NULL, the window becomes the root window.

You can also set the z-order of the window when you attach it, by identifying the window above your window in the z-order. If you supply NULL for this parameter, then the window is placed at the top of the z-order.

The following code shows the creation and attachment of a window:

```c
/* create an application context */
appId = winAppCreate (UGL_NULL, 100);

/* create hello world and attach it to the display as root */
uglTextSizeGet (fontId, &textWidth, &textHeight, -1, "Hello World!");
helloWindowId = winCreate (appId, 0, 0, textWidth + 100, textHeight + 100);
winAttach (UGL_NULL_ID, helloWindowId, UGL_NULL_ID);
```
5.3.5 Window Drawing

You draw to a window using the 2D Graphics API routines, as you would in a non-windowing application. However, the graphics context is set up differently when you are drawing to a window:

– The clipping region and the view port are used to clip all drawing to the visible area of the window.
– The view port is used to convert all coordinates to be relative to the top-left corner of the window.

This is accomplished by setting the view port to the window’s rectangle (obtained using the `winRectGet()` routine), and by setting the clip region to the window’s visible region (obtained using `winVisibleRegionGet()`).

Synchronous and Asynchronous Drawing

Window drawing can be done synchronously or asynchronously. Synchronous drawing happens when an application explicitly performs a drawing operation. For instance, if an application receives real-time data, it immediately sets up a graphics context and draws the data graphically.

Asynchronous drawing happens in response to an event of type `WIN_EVENT_TYPE_DRAW`. This event is sent to an application when a window is exposed. The exposure of the window can result from a number of events:

– the window is attached to the containment hierarchy
– a child window is detached from the window
– the window is raised above another window
– the window is moved from off screen or from behind another window
– the window is manually invalidated because of a call to `winRectInvalidate()` or `winRegionInvalidate()`.

For asynchronous drawing, only the area of a window that has been exposed needs to be updated. This area is stored as the window’s dirty region. You should set the graphics context’s clip region to the dirty region, rather than the visible region, as drawing takes less time and excessive flickering is prevented.
Locking Windows during Drawing

Windows must be locked during drawing operations so that they are not overwritten by operations performed by another task. When a window is locked, it cannot be moved, sized, or obscured. The routines `winLock()` and `winUnlock()` lock and unlock a window for drawing.

Instead of locking a window directly, you can use two additional routines to simplify the drawing process—`winDrawStart()` and `winDrawEnd()`.

The `winDrawStart()` routine does the following:
- locks the window so that it cannot be moved, sized, or obscured.
- sets the view port such that coordinate (0, 0) specifies the top left corner of the window.
- sets the clip region to the window’s visible region. Alternatively, if the window’s `cliptoDirtyRegion` parameter is TRUE, it sets the clip region to the dirty region.
- calls `uglBatchStart()` to optimize the drawing process.

The `winDrawEnd()` routine does the following:
- clears the window’s dirty region if the `clearDirtyRegion` parameter is TRUE.
- calls `uglBatchEnd()` to end the batch drawing
- unlocks the window

The following code draws a rectangle and the text “Hello World” asynchronously.

```c
/* process events */
do {
    /* get an event for this application */
    winEventGet (appId, &event, UGL_WAIT_FOREVER);
    switch (event.header.type) {
    /* draw hello world window */
    case WIN_EVENT_TYPE_DRAW:
    {
        UGL_WINDOW_ID windowId = (UGL_WINDOW_ID)event.header.objectId;
        UGL_RECT windowRect;
        winDrawStart (windowId, gc, UGL_TRUE);
        ....
```
/* get the window's rectangle */
winRectGet (windowId, &windowRect);

/* draw the window background */
uglRectangle (gc, 0, 0, UGL_RECT_WIDTH (windowRect) - 1,
UGL_RECT_HEIGHT (windowRect) - 1);

/* draw the window text */
uglTextDraw (gc, windowRect.left + 50, windowRect.bottom - 50,
-1, "Hello World!");
winDrawEnd (windowId, gc, UGL_TRUE);
break;

} while (event.header.type != WIN_EVENT_TYPE_APP_EXIT);

Note that winRectGet() returns the position of the window relative to its parent. The coordinates passed to uglRectangle() must be relative to the window itself, so that coordinate (0, 0) is the left-top corner of the window. An alternative method for obtaining the window’s rectangle and converting it to the coordinates need when drawing the window’s background is shown below:

/* get the window’s rectangle, adjusted with left-top at (0, 0) */

winRectGet (windowId, &windowRect);
UGL_RECT_MOVE_TO (windowRect, 0, 0);

/* draw the window background */

uglRectangle (gc, windowRect.left, windowRect.top,
windowRect.right, windowRect.bottom);

5.3.6 Window Event Processing

Non-windowing WindML applications use the uglEventGet() routine to retrieve events from the event queue. Windowing applications use a different routine—winEventGet(). This routine gets the next event from the application’s event queue. The objectId field of the event identifies the window to which it belongs.

The following code segment shows a typical event processing loop for a windowing application. The application’s main thread goes into this loop after it has created an application ID, and its basic windows:
/* process events */

do{
    /* get an event for this application */
    winEventGet (appId, &event, UGL_WAIT_FOREVER);

    switch (event.header.type)
    {
        /* process the event */
        
    }
} while (event.header.type != WIN_EVENT_TYPE_APP_EXIT);

If the event queue is empty and no events are waiting to be serviced, the application ID is checked to see if any windows belonging to the application have dirty regions. If a dirty window is found, a draw event is returned for the dirty window. Otherwise, the thread or task is pended until an event is placed in the queue, a window becomes dirty, or the specified timeout period expires.

Because draw events are only returned when an application’s event queue is empty, asynchronous drawing occurs only when an application’s event processing task would otherwise be idle. This behavior is important to allow time critical events to be processed quickly and to prevent excess drawing when a window becomes dirty multiple ways in a short period of time, or becomes dirty but is later hidden before a redraw is requested.

**NOTE:** `winEventGet()` can only be called by the same thread that created the application ID passed to it. In addition, `winDestroy()` can only be called by the same thread that created the application ID passed to `winCreate()`. This ensures that events are handled in the proper order, and that applications do not get events for windows that have been destroyed.

The following events are typically processed by windows:

**UGL_EVENT_TYPE_KEYBOARD**
This event is received by the window that has grabbed the keyboard input whenever a key is pressed or released. If no window has grabbed the keyboard, then the root window receives this window. For more information on grabbing, see 5.3.7 Input Event Routing, p.108.

**UGL_EVENT_TYPE_POINTER**
This event is received by the window that has grabbed the keyboard input whenever a pointer device is moved, or a pointer key is pressed. If no window
has grabbed the keyboard, then the window under the cursor receives the event. The following sample code raises a window when it clicked on by a pointer device:

```c
/* raise window when left pointer button is pressed */

case UGL_EVENT_TYPE_POINTER:
    if (UGL_BUTTON1_DOWN (&event))
        winRaise (windowId);
    break;
```

**WIN_EVENT_TYPE_DRAW**

This event is received by an application when the application has windows with dirty regions and its event queue is empty. The application should respond to this event by drawing the window identified by the `objectId` member of the event header. See 5.3.5 Window Drawing, p.104 to see how this event is normally processed.

Applications can also define their own events. Application events must always be given an event type with a negative value. See 4. Event Services for more information on the event structures and event type values.

### 5.3.7 Input Event Routing

Input event routing consists of two steps:

- determining which window an input event belongs to
- routing the event to the corresponding applications’s event queue.

This routing is performed by the input task of an event service. The input task uses an event router input hook that was attached to the event service. Event routers are discussed in more detail in 5.3.3 Event Routing, p.101.

Normally, events are routed the following ways:

- keyboard events are routed to the root window
- pointer events are routed to the window under the cursor.

You can change this routing by grabbing a keyboard or pointer device, or by replacing the event router hook. When a window has grabbed an input device, all input events generated by that device are routed to the window regardless of the pointer location. When the window ungrabs (that is, releases) the input device, the default routing of events from that device resumes.

The routines listed in Table 5-3 can be used to grab or ungrab an input device:
Using the API

The following guidelines should help you to successfully handle keyboard and pointer grabbing in a multi-tasking environment:

- Typically, a window should grab the keyboard when it becomes active or gains focus, for example, in response to a pointer click. This window then receives all keyboard input until another window becomes active and grabs the keyboard away from the first window.

- A pointer can be grabbed when one or more of its buttons is depressed. For example, you could grab a pointer during a pointer button down-click, and ungrab the pointer on the subsequent up-click.

Input can be grabbed to a window even if another window has already grabbed the device. However, a device can only be ungrabbed by the window that grabbed it last.

Example Code

This code demonstrates how pointer grabbing can be used to drag a window.

```c
case UGL_EVENT_TYPE_POINTER:
{
    /* grab pointer events when left or right button is pressed */
    if (UGL_BUTTON1_DOWN (&event) || UGL_BUTTON2_DOWN (&event))
        winPointerGrab (windowId);
    /*
    * Move the window when pointer is dragged with left or
    * right button pressed
    */
```
if (UGL_BUTTON1_DRAG (&event) || UGL_BUTTON2_DRAG (&event))
{
    UGL_INPUT_EVENT *pInputEvent = (UGL_INPUT_EVENT *)&event;
    winMove (windowId, pInputEvent->type.pointer.dx,
             pInputEvent->type.pointer.dy);
}

/* ungrab pointer when left or right button is released */
if (UGL_BUTTON1_UP (&event) || UGL_BUTTON2_UP (&event))
    winPointerUngrab (windowId);
break;

Note that UGL_BUTTONx_DOWN(), UGL_BUTTONx_DRAG(), and UGL_BUTTONx_UP() are utility macros found in the windowing demo.
6 Multimedia

6.1 Introduction

This chapter discusses the support provided by WindML in the following areas:

- **Video**. These routines provide support for video adaptors attached to a display.
- **Audio**. These routines provide support for audio files.
- **JPEG**. These routines allow you to manipulate JPEG images.

6.2 Video

The video API provides support for video adapters attached to an output display. An adapter may have multiple ports and use various video encodings such as NTSC, PAL, SECAM, and so on. The video extension is implemented as an extension to a graphics driver. The functionality provided by the video extension allows an application to do the following:

- identify the video encoding/formats the video adapter supports
- establish a configuration for the video adapter
- display video to a region on a display
- lock and unlock the video adapter (reserve a port for exclusive use)
- set and get video port attributes
The complete video API is listed in Table 6-1.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglVideoInit()</td>
<td>Initializes the video support.</td>
</tr>
<tr>
<td>uglVideoAdapterInfo()</td>
<td>Obtains/sets video adapter information.</td>
</tr>
<tr>
<td>uglVideoPortLock()</td>
<td>Locks a video port for exclusive access.</td>
</tr>
<tr>
<td>uglVideoStreamPut()</td>
<td>Writes a continuous stream of video to a specified region on the display.</td>
</tr>
<tr>
<td>uglVideoStillPut()</td>
<td>Writes a single video frame to a specified region on the display.</td>
</tr>
<tr>
<td>uglVideoStop()</td>
<td>Stops video on a video port.</td>
</tr>
<tr>
<td>uglVideoPortUnlock()</td>
<td>Unlocks a video port.</td>
</tr>
</tbody>
</table>

### 6.2.1 Overlays and Video

The video API requires that an overlay surface be present. The incoming video is stored in the overlay surface. The visibility of the incoming video on the overlay surface is defined by the configuration of the overlay. For details on setting up an overlay and controlling its visibility and positioning, see 7.5 Overlay Surfaces, p.132.

For example, the overlay surface can be configured to use the display mode color key. In this display mode, video will be displayed anywhere in the primary frame buffer that is set to the specified color (within the overlay area). In the areas of the primary frame buffer where the color in the overlay region is not the color key, then graphics will be shown instead of video.

The video extension attempts to display incoming video in the entire overlay area. The dimensions of the incoming video can be specified using the `uglVideoStreamPut()` and `uglVideoStillPut()` routines. By adjusting the dimensions of the incoming video, you can give the effect of zooming the video image in and out.

For example, when the dimensions of the incoming video are reduced by one half from the previous setting, the effect will be that the video image displayed will be zoomed in by a factor of two, since the size of the overlay surface was not changed.
6.2.2 Video Extension Configuration

The uglVideoAdapterInfo() routine provides general control of the video adapter. You can use this function to perform the following operations:

- Identify the ports that are available and the characteristics of a port. Characteristics include the available video modes (NTSC, PAL, SECAM), and the maximum width and height permitted.

- Set the mode. This includes specification of the video mode (NTSC, PAL, SECAM), the overlay surface to be used, and whether the input video stream is interlaced.

- Retrieve the current configuration.

- Adjust the attributes of the video (hue, brightness, saturation, and contrast).

- Retrieve the current attributes of the video.

- Determine if there is a video signal present.

In general, these are the steps followed when using the video extension:

1. Initialize the video extension using uglVideoInit().
2. Determine the presence of the video port that you need using uglVideoAdapterInfo().
3. Reserve the port for exclusive use using uglVideoPortLock().
4. Create an overlay surface of the required size and pixel format (see 7.5 Overlay Surfaces, p. 132).
5. Configure the video port using uglVideoAdapterInfo().
6. Start the display of video using uglVideoStreamPut().

You can perform various operations on the video such as resizing, repositioning on the display, capturing as a JPEG image, or saving as a bitmap.

Some of these operations require overlay manipulation. For example, you must move the overlay to reposition the image on the display.
6.2.3 Capturing a Video Image

You can capture a video image to:

- a bitmap
  The contents of the video image can be read using the routine `uglBitmapRead()`. Since the video image does not reside in the primary frame buffer, the device ID for the `uglBitmapRead()` must be the overlay ID rather than the actual graphics device ID. When the bitmap is read, the YUV422 format that is used for video will be converted to RGB.

- a JPEG file
  By attaching the JPEG extension to the overlay, you can capture the video image as a JPEG image. Again, the device ID must be the overlay ID rather than the actual graphics device ID.

Video Extension Example Program

The program `uglvdemo.c` in the directory `target/src/ugl/demo` is an example of the use of the video extension. This program shows you how to:

- Configure the video extension and create the overlay surface
- Modify the video attributes
- Freeze the video
- Zoom the video image
- Capture the video image to a JPEG file

6.3 Audio

The audio component of WindML is accessed by an application using the standard I/O routines—`open()`, `close()`, `read()`, `write()`, and `ioctl()`. This API is compatible with the Open Sound SystemTM (OSS) Free format.
An audio driver uses two sub-devices to support the various functions required of a sound system. The devices are:

- **DSP (Digital Signal Processor)**
  
  The DSP provides the digital to analog conversion required to playback a digitized sound. It also provides an analog to digital conversion allowing audio to be recorded.

- **Mixer**
  
  The mixer device is responsible for controlling playback and recording levels and mixing the sound with other audio sources.

Typically, the device name for a single audio device is /sound. If the audio device driver is named /sound, then the DSP and the mixer are referenced as /sound/dsp and /sound/mixer, respectively.

### 6.3.1 The DSP Device

Double buffering is supported by the DSP device. When an audio stream is being played, the hardware plays from one buffer while the processor is filling a second buffer with data. With most audio hardware, playback of the second buffer automatically starts as soon as the first buffer completes. The hardware continues this process until the audio stream is completed. Therefore, as long as there is more data to be played, the processor must fill these buffers quickly enough to prevent an underrun situation.

An audio block is the total amount of data that the device driver can buffer at one time. This block is composed of fragments which correspond to the size of each individual audio buffer. The size of an audio fragment can be modified through the audio API to accommodate a specific audio application and improve system performance. The audio application should size the data in each write() operation so that it corresponds to a multiple of the fragment size. The driver can then place each new write at the start of the next available fragment.

In addition to reading and writing audio streams using the read() and write() functions, the ioctl() control functions configure and control the size of the audio blocks and fragments, see Table 6-2.
Table 6-2  *ioctl* Control Functions for the DSP Device

<table>
<thead>
<tr>
<th>Control Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNDCTL_DSP_SYNC</td>
<td>Pends until the current operation has been completed. For example, if an audio stream is being played when <em>ioctl</em> is called, the <em>ioctl</em> call will not complete until the audio stream has finished.</td>
</tr>
<tr>
<td>SNDCTL_DSP_SPEED</td>
<td>Sets the data rate at which the audio stream is recorded or played. The speed is represented as the number of samples per second.</td>
</tr>
<tr>
<td>SNDCTL_DSP_STEREO</td>
<td>Identifies the audio stream as being mono (0) or stereo (1)</td>
</tr>
<tr>
<td>SNDCTL_DSP_GETBLKSIZE</td>
<td>Obtains the size of the audio block.</td>
</tr>
<tr>
<td>SNDCTL_DSP_SETFRAGMENT</td>
<td>Sets the fragment size to the value specified in the argument.</td>
</tr>
<tr>
<td>SNDCTL_DSP_SAMPLESIZE</td>
<td>Sets the size of an audio sample as the number of bits per sample.</td>
</tr>
<tr>
<td>SNDCTL_DSP_CHANNELS</td>
<td>Identifies the number of channels that are present within the audio sample (expands on the SNDCTL_DSP_STEREO).</td>
</tr>
<tr>
<td>SNDCTL_DSP_GETOSPACE</td>
<td>Obtains the amount of space in bytes that is available in the output buffer. The information is returned in the AUDIO_BUF_INFO data structure. This data structure identifies the fragment size, total number of fragments, fragments available, and the amount of storage available in bytes.</td>
</tr>
<tr>
<td>SNDCTL_DSP_SETFMT</td>
<td>Sets the audio format.</td>
</tr>
<tr>
<td>SNDCTL_DSP_GETFMTS</td>
<td>Obtains the available audio formats in a bit-mapped long word.</td>
</tr>
<tr>
<td>SNDCTL_DSP_NONBLOCK</td>
<td>Does not block on <em>read()</em> or <em>write()</em> operations.</td>
</tr>
</tbody>
</table>
The audio formats are identified with a bit-mapped long word for the `SNDCTL_DSP_SETFMT` and the `SNDCTL_DSP_GETFMTS` operations. The formats are listed in Table 6-3:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFMT_MU_LAW</td>
<td>Logarithm mu-law audio encoding.</td>
</tr>
<tr>
<td>AFMT_A_LAW</td>
<td>Logarithm a-law audio encoding.</td>
</tr>
<tr>
<td>AFMT_IMA_ADPCM</td>
<td>A 4:1 compressed format where a 16-bit audio sequence is represented using an average of 4 bits per sample.</td>
</tr>
<tr>
<td>AFMT_U8</td>
<td>Unsigned 8 bit samples.</td>
</tr>
<tr>
<td>AFMT_S16_LE</td>
<td>Signed 16 bit samples organized as little endian.</td>
</tr>
<tr>
<td>AFMT_S16_BE</td>
<td>Signed 16 bit samples organized as big endian.</td>
</tr>
<tr>
<td>AFMT_S8</td>
<td>Signed 8 bit samples.</td>
</tr>
<tr>
<td>AFMT_U16_LE</td>
<td>Unsigned 16 bit samples organized as little endian.</td>
</tr>
<tr>
<td>AFMT_U16_BE</td>
<td>Unsigned 16 bit samples organized as big endian.</td>
</tr>
<tr>
<td>AFMT_MPEG</td>
<td>MPEG MP2/MP3 audio format.</td>
</tr>
</tbody>
</table>

A device driver may not support all of the above audio formats. An attempt to set an audio format that is not supported results in the `ioctl()` control function returning `ERROR`.

When a specific audio format is not specified, a default of `AFMT_U8` is used for 8 bit samples and `AFMT_S16_LE` for 16 bit samples.

### 6.3.2 The Mixer Device

All communication with the mixer device is done through the `ioctl()` routine. The mixer has various functional components that are controlled using `ioctl()`, see Table 6-4:

**READ** or **WRITE** in the command dictates whether the control data is to be read or written to the mixer device. Note that the mixer device does not allow audio data to be read or written.
The values that are read or written are within the range of 0 to 100, where 0 is minimum setting for the control. When the mixer supports stereo, then the values are encoded such that the lower 8 bits represent the left channel and bits 8 - 15 represent the right channel.
The following code segment sets the volume:

```c
int md; /* Mixer file descriptor */
int volume = (100 << 8) | 100; /* maximum left and right */
/* set the volume to maximum */
md = open("/sound/mixer", O_RDWR, 0);
ioctl (md, SOUND_MIXER_WRITE_VOLUME, &volume);
```

Not all of the operations in Table 6-4 may be present in the mixer hardware or supported by the driver. To determine the capabilities of the mixer device, the driver issues an `ioctl` with the `SOUND_MIXER_DEVMASK` command. This call returns a mask identifying the capabilities of the mixer, see Table 6-5.

<table>
<thead>
<tr>
<th>Supported Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOUND_MASK_VOLUME</td>
<td>Set when the mixer supports a volume control.</td>
</tr>
<tr>
<td>SOUND_MASK_BASS</td>
<td>Set when the mixer supports a bass control.</td>
</tr>
<tr>
<td>SOUND_MASK_TREBLE</td>
<td>Set when the mixer supports a treble control.</td>
</tr>
<tr>
<td>SOUND_MASK_SYNTH</td>
<td>Set when the mixer has a synthesizer.</td>
</tr>
<tr>
<td>SOUND_MASK_PCM</td>
<td>Set when the mixer has a PCM output.</td>
</tr>
<tr>
<td>SOUND_MASK_SPEAKER</td>
<td>Set when the mixer is connected to a PC speaker.</td>
</tr>
<tr>
<td>SOUND_MASK_LINE</td>
<td>Set when the mixer has an audio in line.</td>
</tr>
<tr>
<td>SOUND_MASK_MIC</td>
<td>Set when the mixer has a microphone input.</td>
</tr>
<tr>
<td>SOUND_MASK_CD</td>
<td>Set when the mixer is connected to a CD.</td>
</tr>
<tr>
<td>SOUND_MASK_RECLEV</td>
<td>Set when the mixer has a global recording level setting.</td>
</tr>
</tbody>
</table>

### 6.3.3 WAVE File Handling

An audio library is provided to support audio WAVE files. The routine `wavHeaderRead()` reads and parses the WAVE file header to determine the sample rate, sample size, and number of samples.

The program `wavPlay.c` in the directory `target/src/ugl/demo` shows you how to use the audio device to play a WAVE file.
6.4 JPEG

WindML JPEG support is provided as a graphics driver extension. The JPEG API allows you to:

- Read an image from a DDB (Device Dependent Bitmap) and store it as a JPEG image. The JPEG image must be a file.
- Read a JPEG image and write it to a DDB.

The source of the JPEG image may be either a data file or a data buffer.

The JPEG API is listed in Table 6-6.

Table 6-6 JPEG Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglJpegInit()</td>
<td>Initializes the JPEG extension.</td>
</tr>
<tr>
<td>uglJpegFromDDB()</td>
<td>Creates a JPEG file from the contents of a DDB (bitmap).</td>
</tr>
<tr>
<td>uglJpegToDDBFromFile()</td>
<td>Decompresses a JPEG image from a file and write to a bitmap.</td>
</tr>
<tr>
<td>uglJpegToDDBFromData()</td>
<td>Decompresses a JPEG image from a data buffer and write to a bitmap.</td>
</tr>
<tr>
<td>uglJpegModeGet()</td>
<td>Gets the compress/decompress parameters of a JPEG image.</td>
</tr>
<tr>
<td>uglJpegModeSet()</td>
<td>Sets the compress/decompress parameters of a JPEG image.</td>
</tr>
</tbody>
</table>

6.4.1 Initialization

Before you can use the JPEG extension, you must initialize it using the `uglJpegInit()` routine. The initialization process verifies that the JPEG extension is installed for the specified graphics device and readys the extension for use. The `uglJpegInit()` routine returns an `UGL_JPEG_ID` which must be used as a parameter for all subsequent JPEG routine calls.
6.4.2 Compression/Decompression Parameters

After the JPEG extension is initialized, then the compression/decompression parameters can be modified. The higher the quality of a JPEG image, the longer it takes to process the image. You can adjust the parameters to provide a trade-off between quality and processing time.

The parameters are:

Quality
The quality of the JPEG image can be adjusted from a quality of 0 to 100. The default setting for quality is 75.

Smoothing
The smoothing factor of the image can be adjusted from zero smoothing (0) to maximum smoothing (100). The default setting for smoothing is 0, that is, image smoothing is not performed.

Scaling
You can scale an image during the decompression process. The supported scaling levels are 1:1, 1:2, 1:4, and 1:8. The default scaling is 1:1.

6.4.3 Writing a JPEG Image to a DDB

A JPEG image can be written to a DDB (Device Dependent Bitmap) for display or manipulation, if the image is present in a data buffer or a data file. The file can be on a locally mounted file system, a remote file system, a pipe, or a network socket. The steps involved in writing the JPEG image to the DDB are as follows:

1. Set the mode (that is, the decompression parameters) using the `uglJpegModeSet()` routine. For example, the image may be scaled for the DDB.

2. Convert the JPEG image to a bitmap image and write it to the DDB using the `uglJpegToDDBFromData()` routine or the `uglJpegToDDBFromFile()` routine, as appropriate. Optionally, you can create a DDB to receive the JPEG image before you call these routines.

The DDB is passed to either routine as a pointer to a `UGL_DDB_ID`. This pointer may point to one of three values:

- an ID for a DDB you created with the `uglBitmapCreate()` routine
- the macro `UGL_DISPLAY_ID`
This specifies the frame buffer (the display itself). The rectangular area is relative to the top-left corner of the display. The image transfer is clipped to the bounds of the display.

- NULL

This indicates that a DDB should be created of sufficient size such that the entire JPEG image can be accommodated.

When writing a JPEG image to a DDB, the complete image or just part of the image can be written to the DDB. You can specify the area of the JPEG image to be written to the DDB by passing to the routines a pointer to a rectangle (UGL_RECT). This source rectangle may be set to NULL to indicate that the entire JPEG image is to be written to the DDB. The following code segment shows you how to write a JPEG image to the display from a file:

```c
/* initialize the JPEG extension */
jpegId = uglJpegInit (pDeviceId, &jpegVersion);
/* open the file to receive the image */
fp = fopen("capture.jpg" , "r");
/* use default decompression parameters, so just start the read */
uglJpegToDDBFromFile(jpegId, fp, pDisplayId, UGL_NULL, 0, 0);
/* close the file */
fclose (fp);
```

In this code segment, the JPEG image that is contained in the file `capture.jpg` is written to the graphics display (pDisplayId points to UGL_DISPLAY_ID) at the top left corner of the display (0,0). Since the source rectangle is set to UGL_NULL, the entire JPEG image is written to the display.

**NOTE:** When the source of the JPEG image is a data buffer, the entire JPEG image must be present within the buffer.

### 6.4.4 Saving a Bitmap as a JPEG Image

A bitmap (DDB) can be captured and saved as a JPEG file. The file must previously have been opened for writing. The file can be a file on a locally mounted file system, a remote file system, a pipe, or a network socket. The entire DDB or just part of the DDB can be captured and written to the file.

The process of reading a DDB and creating a JPEG image from the DDB as follows:

1. Set the mode (compression parameters) using the `uglJpegModeSet()` routine.
2. After the mode is set, convert the DDB to a JPEG file using the `uglJpegFromDDB()` routine.

The following code segment shows you how to save an image on the display to a file:

```c
/* initialize the JPEG extension */
jpegId = uglJpegInit (pDeviceId, &jpegVersion);

/* open the file to receive the image */
fp = fopen("capture.jpg", "w");

/* set compression parameters */
jpegMode.quality = 80;
jpegMode.smooth = 40;
jpegMode.scale = 1;
uglJpegModeSet(jpegId, &jpegMode);

/* save the image as a JPEG image */
uglJpegFromDDB (jpegId, UGL_DISPLAY_ID, 0, 0, 150, 100, fp);

/* close the file */
fclose (fp);
```

The above code segment saves the specified rectangular area to the `capture.jpg` file using a quality of 80, a smoothing of 40, and a scaling of 1:1.
7

Resource Management

7.1 Introduction

The application programmer must be aware of various WindML resources and the manner in which these resources are managed. In WindML, resource management refers to the establishment of a resource, the subsequent control of the resource, and the deletion of the resource.

Resources that need to be created or managed are:

– general WindML resources, such as devices and event queues, which need to be initialized
– memory management
– the driver registry
– overlay surfaces

In addition, drivers can be queried and controlled.
### 7.2 General Resources

#### 7.2.1 Initialization

WindML provides a single function, `uglInitialize()` which an application must call to perform basic WindML initialization. An application must explicitly call `uglInitialize()` before using any other WindML function call. This routine initializes WindML with the configuration specified by the configuration tool or specified by the `uglInit.h` file (command line configuration specification). It sets up the following general WindML resources:

1. **Graphics device.** The device is created and configured for the requested mode (color depth, resolution, output device type and refresh rate).
2. **Input devices.** The keyboard and pointer devices are created and set to the mode that was specified in the configuration. Keyboard type and key mapping for the keyboard configuration and the pointer type (PS-2, Microsoft serial, and so on) for the pointer configuration is set up.
3. **Event service queue.** The default event service queue is created and set to the size specified by the configuration.
4. **Audio device.** The audio device is created and initialized.
5. **Font engine.** The specified font engine is created and initialized according to the specified configuration.
6. **Memory manager.** If a private WindML memory pool is to be used, then that memory pool is created. The WindML private memory manager function mappings are established.

Each of these resources, with the exception of the memory manager, is placed in the device registry so that the application can look up the resource handle.

#### 7.2.2 Cleanup

When a WindML application terminates, it must delete any resources that it created, and then call the function `uglDeinitialize()` to delete the general resources.
7.3 Memory Management

Memory management is essential to graphical applications, enabling them to:

- use memory efficiently

  Graphical applications are memory intensive. They allocate and free memory frequently which leads to memory fragmentation and can degrade the performance of more critical parts of the application. The memory management API allows the developer to set up a default memory pool to be used exclusively by WindML, thereby isolating the memory used by the graphical system from the memory used by the rest of the application. The application can also use this API to replace the memory manager provided by the OS.

- optimize memory usage

  Many graphics devices have display memory that is not used for the visible display; that is off-screen display memory. This extra memory can be used for the storage of frequently accessed graphical information such as bitmaps. Since this memory is usually on the same internal bus as the graphics processor, the use of this memory for accessing frequently used bitmap data can dramatically improve the performance of the graphics system. Special device memory pools can be created by the application in this display memory to allow fast direct access by graphics processors.

By default, WindML uses the standard memory manager provided by the operating system. If you want, you can configure WindML to use an alternative memory manager, see 2. WindML Configuration.

7.3.1 Memory Pools

WindML always creates a default memory pool when it is initialized. The default memory pool is defined in WindML configuration. It may be either the VxWorks system pool or a private memory pool. It is referenced as UGL_DEFAULT_MEM.

A WindML graphics device can create another memory pool when the graphics device is initialized. The graphics device allocates any remaining display memory to a device memory pool after the visible frame buffer is allocated from the display memory. The macro UGL_VIDEO_MEM is used to reference this memory pool. The size of this memory pool is variable, depending on the size of the display memory and the display resolution and color depth. There will be no device memory pool if the entire display memory is required to support the visible display.
The double buffering and overlay functionality use this memory pool. After the necessary allocations are made from this pool to accommodate any double buffering or overlay support, the remainder may be used for handling pixel data that is frequently moved from off-screen to on-screen and vice versa. You should exercise care using the device memory pool as it is limited in size.

Only use the memory pool to store pixel data so that you can take advantage of the acceleration features of the graphics device. For example, if an application creates a bitmap that will be frequently transferred to the display, it can create it as follows:

```c
frequentDDB = uglBitmapCreate (devId, pDib, UGL_DIB_INIT_DATA, 0, UGL_VIDEO_MEM);
```

**NOTE:** Movement of data from the device memory pool to the visible display area of the frame buffer by the main processor (rather than the graphics processor), can adversely effect performance. This is because the main processor has to access the display memory twice—once to read from off-screen memory and again to write to the on-screen memory.

The memory API routines are listed in Table 7-1.

### Table 7-1 Memory Management Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uglMemDefaultPoolSet()</code></td>
<td>Sets the default memory pool.</td>
</tr>
<tr>
<td><code>uglMemDefaultPoolGet()</code></td>
<td>Gets the default memory pool.</td>
</tr>
<tr>
<td><code>uglMemDevicePoolCreate()</code></td>
<td>Creates a device memory pool.</td>
</tr>
<tr>
<td><code>uglMemDevicePoolDestroy()</code></td>
<td>Destroys a device memory pool.</td>
</tr>
</tbody>
</table>

### 7.3.2 Memory Macros

The header file `target/h/ugl/uglmem.h` contains the definitions for the memory management API, see Table 7-2.

### Table 7-2 Additional Memory Macros

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>UGL_MALLOC</code></td>
<td>Malloc from default memory pool.</td>
</tr>
<tr>
<td><code>UGL_CALLOC</code></td>
<td>Calloc from default memory pool.</td>
</tr>
</tbody>
</table>
Device Driver Registry

WindML maintains a device driver registry to store information about the device drivers and other services that are configured and initialized in the system.

When WindML is initialized, using the `uglInitialize()` function, each WindML device driver and service is placed in a driver registry. To use a device or service, the application must query the driver registry for a device driver ID or service ID.

The driver registry API routines are listed in Table 7-3.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uglDriverFind()</code></td>
<td>Finds a device driver or service within the driver registry.</td>
</tr>
<tr>
<td><code>uglDriverRegister()</code></td>
<td>Adds a device driver or service to the driver registry.</td>
</tr>
<tr>
<td><code>uglDriverUnRegister()</code></td>
<td>Removes a device driver or service from the driver registry.</td>
</tr>
</tbody>
</table>

Typically, drivers and services are only added to the device driver registry by the `uglInitialize()` function when WindML is initialized. Likewise, registered devices and services are removed when the `uglDeinitialize()` function is called.
7.4.1 Device Types and Instances

Each device or service supported by WindML is assigned a device type, see Table 7-4.

<table>
<thead>
<tr>
<th>Device Types</th>
<th>Device Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>UGL_EVENT_SERVICE_TYPE</td>
<td>Event services</td>
</tr>
<tr>
<td>UGL_KEYBOARD_TYPE</td>
<td>Keyboard</td>
</tr>
<tr>
<td>UGL_POINTER_TYPE</td>
<td>Pointer—mouse or touchscreen</td>
</tr>
<tr>
<td>UGL_DISPLAY_TYPE</td>
<td>Graphics display</td>
</tr>
<tr>
<td>UGL_SOUND_TYPE</td>
<td>Sound device</td>
</tr>
<tr>
<td>UGL_FONT_ENGINE_TYPE</td>
<td>Font engine/driver</td>
</tr>
</tbody>
</table>

The device types are defined in the header file target/h/ugl/ugltypes.h. A device or service is placed in the registry according to its type and its instance. Each device gets an instance number as it is added to the registry. The first type of a graphics display (UGL_DISPLAY_TYPE) is assigned instance 0, the second graphics device that is registered is assigned an instance of 1, and so on.

7.4.2 Registering Devices and Services

For each device and service that uglInitialize() creates, it puts a corresponding identifier in the registry. These identifiers include:

- graphics device
- keyboard device
- pointer device
- font engine device
- default event queue
- audio device

If an application creates other devices or services, it can add those identifiers to the registry using the uglDriverRegister() function. This code shows how an additional font engine is added to the registry:
Resource Management

#define NEW_FONTENG_INSTANCE 1
newFontDevID = xxxFontEngineCreate (graphicsDevID);
uglDriverRegister (UGL_FONT_ENGINE_TYPE, NEW_FONTENG_INSTANCE,
             (UGL_UINT32) newFontDevID);

This example adds the new font engine identifier to the registry with an instance of 1, indicating that it is the second font engine in the registry.

7.4.3 Obtaining a Device ID

The uglDriverFind() function is used to search the registry for a specific device or service identifier. This identifier must be obtained from the registry before calling any WindML function that requires a device or service identifier as one of its input arguments.

Getting a device and service ID is usually done during initialization using the following steps:

/* Initialize UGL */
uglInitialize();

/* Obtain the first instance of the display device identifier */
uglDriverFind (UGL_DISPLAY_TYPE, 0, (UGL_UINT32 *)&devId);

/* Obtain the first instance of the event service identifier */
uglDriverFind (UGL_EVENT_SERVICE_TYPE, 0, (UGL_UINT32 *)&eventServiceId);

/* Obtain the first instance of the font driver identifier */
uglDriverFind (UGL_FONT_ENGINE_TYPE, 0, (UGL_UINT32 *)&fontDrvId);

/* Now we have the identifiers for the display, event services, and */
/* the font driver, so we can create our event service queue, graphics */
/* contexts, etc. and start drawing operations. */

In this example, the uglDriverFind() function is used to find the first instance to each device and service. The uglInitialize() function places each of the devices and services that it initializes into the registry as the first instance (that is, instance equals 0). For an application to find the identifier for the font engine that was added in the example provided in section 7.4.2, the following code would be used:

uglDriverFind (UGL_FONT_ENGINE_TYPE, NEW_FONTENG_INSTANCE, &pNewFontDevID);
7.4.4 Removing a Device from the Registry

When a WindML application terminates, it must remove all entries from the registry as part of the termination process. All entries added by the `uglInitialize()` function are removed from the registry by the `uglDeinitialize()` function. Any entries explicitly added to the registry by the application must also be removed. In the example provided 7.4.2 Registering Devices and Services, p.130, when a new font engine was added to the registry, the following code removes the entry from the registry:

```c
uglDriverUnRegister (UGL_FONT_ENGINE_TYPE, NEW_FONTENG_INSTANCE);
```

7.5 Overlay Surfaces

The overlay API creates an overlay surface for a specified graphics device. An overlay surface allows you to access different graphics planes on the graphics device. An overlay surface is typically used to allow an application to mix pixel data in different formats. Common situations where overlays are used are:

- so that a single display device can simultaneously support two different pixel data formats (color models)

  For example, you can make a single display device simultaneously support RGB and YUV422 color models if you configure the primary frame buffer for a display device to use the RGB color model, and an overlay surface to support the YUV422 color model. This allows you to display video (which uses YUV422) and graphics (which uses RGB) simultaneously on the same display.

- to blend two images to provide special visual effects

  For example, if the frame buffer and the overlay use the same pixel data format, the overlay can be used to blend two images and provide a translucency effect.

**NOTE:** The hardware must be capable of supporting overlays.

Overlay support is provided by the following routines, see Table 7-5.
7.5.1 Creating an Overlay

An overlay surface and a corresponding overlay engine is created by the `uglOverlayCreate()` routine. When you create an overlay surface, you must specify several parameters:

- the pixel format of the overlay
- the overlay mode
- a color key, if the overlay mode is set to use a color key
- the location and size of the overlay surface

The following code creates an overlay surface for the graphics device specified by `devId`:

```c
ovlyId = uglOverlayCreate (devId, 1, UGL_YUV422,
            (UGL_OVERLAY_COLORKEY_MODE | UGL_OVERLAY_BOB_MODE),
            colorKey,50, 60, 520, 410, 1);
```

The parameters are:

- Overlay engine (1)
- YUV422 pixel format
- Color key display mode using the bob sampling algorithm
- Color key value
- Overlay window, positioned at (50, 60, 520, 410)
- Overlay position follows the primary frame buffer

**Pixel Format**

The *pixel format* specifies the color model and the layout of the pixel components. For a color model of YUV with a pixel layout of 422, the pixel format would be

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uglOverlayCreate()</code></td>
<td>Creates an overlay surface.</td>
</tr>
<tr>
<td><code>uglOverlayMove()</code></td>
<td>Moves an overlay surface.</td>
</tr>
<tr>
<td><code>uglOverlayDestroy()</code></td>
<td>Destroys an overlay surface.</td>
</tr>
</tbody>
</table>
specified as **UGL_YUV422**. For a color model of RGB with a pixel format of 565, the pixel format would be specified as **UGL_RGB565**.

The header file `target/h/ugl/uglclir.h` provides a list of all the pixel formats that are supported.

**Overlay Mode**

The *overlay mode* is a bit mask specifying the operating parameters for the overlay surface. The bit mask specifies the display mode and overlay sampling algorithm. The header file `target/h/ugl/ugltypes.h` provides the definitions for each of the overlay modes.

The display mode may be color key (**UGL_OVERLAY_COLORKEY_MODE**) or window (**UGL_OVERLAY_WINDOW_MODE**).

**ColorKey**

In this mode, the application sets an area in the primary frame buffer to a specific color which is not used for rendering graphics.

When the color in the specified region on the primary frame buffer matches the color as specified by the `colorKey` parameter, the data on the overlay region is presented on the display.

**Window**

In this mode, a rectangular area is defined to be the overlay area. The data on the overlay region is always presented to the display.

If the underlying graphics hardware supports alpha blending, then this mode can additionally specify that the window is to incorporate alpha blending. When alpha blending is used, the frame buffer’s alpha value determines how the overlay data is displayed. The alpha field of a pixel varies in size depending on the RGB pixel format.

For example, for a pixel format of ARGB8888, the range of values for an alpha value is 0 through 0xFF. For a pixel format of ARGB4444, the range of values for an alpha value is 0 through 0xF. Assuming a pixel format of ARGB8888, the interpretation of alpha values would be:

- **alpha value of 0**
  - The overlay data is displayed.

- **alpha value of 0xFF**
  - The data in the primary frame buffer is displayed.
alpha value between 0 and 0xFF

The data in the overlay and the data in the primary frame buffer is mixed to varying degrees (depending on the exact alpha value). This results in a translucent effect.

If alpha blending is being used, the overlay mode must also specify `UGL_OVERLAY_ALPHA_BLEND`, signifying that the window is to incorporate alpha blending:

```
(UGL_OVERLAY_WINDOW_MODE | UGL_OVERLAY_ALPHA_BLEND).
```

**Overlay Sampling Algorithm**

The sampling mode is important when the overlay data is from an interlaced video source, such as a camera, and can also be specified in the overlay mode bitmask. There are two possible sampling modes:

Bob (`UGL_OVERLAY_BOB_MODE`)
- Each odd and even field is displayed individually.

Weave (`UGL_OVERLAY_WEAVE_MODE`)
- This blends two interlaced images into a frame. This mode produces a crisper image than the bob mode.

If a sampling algorithm is not specified, the sampling mode defaults to the bob mode.

The overlay API does not provide a software implementation of the bob or weave algorithms. It relies on the hardware supplying a hardware implementation.

### 7.5.2 Accessing an Overlay

The object returned by `uglOverlayCreate()` is an overlay identifier, similar to a graphics device ID. The overlay ID may be used in the same manner as a graphics device ID. You can call all the regular 2D API routines with using the overlay ID. For example, the video image in an overlay surface may be read using the `uglBitmapRead()` routine. It may also be captured as a JPEG file, by attaching the JPEG image extension to the overlay.

⚠️ **CAUTION:** Overlays do not always implement all of the functionality provided by a graphics device. Different devices provide different levels of overlay support. Consult the driver-specific documentation for the pixel primitives and pixel formats supported. For WindML-supplied drivers, this documentation can be found in the appropriate `.txt` file for the specific driver.
7.5.3 Moving an Overlay

The uglOverlayMove() routine allows the overlay area to be moved with respect to the primary frame buffer and resized. If multiple overlays are supported, this routine can be used to change the stacking order of overlays. Note that you cannot resize an overlay to be greater than the size specified when the overlay surface was created.

7.6 Driver Information and Control

WindML provides an API to obtain information from a driver and to adjust driver parameters. The information and control API is provided in Table 7-6.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglInfo()</td>
<td>Obtain information or set control parameters for graphics devices.</td>
</tr>
<tr>
<td>uglFontDriverInfo()</td>
<td>Obtain information or set control parameters for font drivers</td>
</tr>
<tr>
<td>uglInputDeviceInfo()</td>
<td>Obtain information or set control parameters for input drivers</td>
</tr>
<tr>
<td>uglModeAvailGet()</td>
<td>Obtains the list of graphics modes.</td>
</tr>
<tr>
<td>uglModeSet()</td>
<td>Sets the graphics mode.</td>
</tr>
</tbody>
</table>

Table 7-6 Information and Control API

7.6.1 Querying a Driver with uglInfo()

The uglInfo() function allows you to obtain control information from the graphics device and to attach extensions to the graphics driver. The uglInfo() function closely parallels the standard ioctl() function. Its format is:

```c
uglInfo (deviceId, requestType, argument);
```
Getting Information

The header file `target/h/ugl/uglinfo.h` provides the definitions for each of the `uglInfo()` request types. Control information that can be obtained from a graphics device includes:

1. Frame buffer organization by specifying `UGL_FB_INFO_REQ` as the request type. A data structure, `UGL_FB_INFO`, is returned containing the following characteristics of the frame buffer:
   - frame buffer width and height
   - location of frame buffer
   - amount of display memory
   - miscellaneous control flags, such as whether double buffering is available

2. Pixel organization by specifying `UGL_COLOR_INFO_REQ` as the request type. A data structure, `UGL_COLOR_INFO`, is returned containing the following characteristics of the pixel format:
   - color model (such as direct, indexed, and so on)
   - color space (such as, RGB, YUV, and so on)
   - color depth, that is number of bits per pixel
   - color look up table (CLUT) size when indexed color model and whether the CLUT can be written

3. Current mode by specifying `UGL_MODE_INFO_REQ` as the request type. The currently selected mode is returned in a `UGL_MODE_INFO` data structure. The information returned within this data structure is:
   - display resolution (width and height)
   - color model (such as direct, indexed, and so on)
   - pixel format (such as RGB565, ARGB4444, and so on)
   - color depth, that is number of bits per pixel
   - CLUT size when indexed color model
   - location of frame buffer
   - amount of display memory
   - miscellaneous control flags, such as whether double buffering is available
Attaching an Extension

In addition, you can attach an extension to the driver by specifying the UGL_EXT_INFO_REQ as the request type. When an extension is attached to a graphics driver, it provides additional functionality that the 2D layer can use.

Each extension is identified by a unique name. Before the extension can be used, it must be attached to a driver. A call to the uglInfo(), with the UGL_EXT_INFO_REQ request type, queries the driver for the presence of a specific extension and, if it is present, attaches the extension to the driver. The argument to the uglInfo() is an extension control structure, UGL_EXT_INFO. The format of this data structure is as follows:

```c
typedef struct uglExtInfo
{
    char * name; /* name of the extension */
    int version; /* extension version */
    void * pExt; /* extension specific */
} uglExtInfo;
```

When the application passes this structure to uglInfo(), only the extension name field of this data structure is filled in. If the extension is available, the driver fills in the extension version and the extension handle, pExt. Typically, the value provided by pExt is used whenever any of the extension API routines are called.

The uglInfo() routine returns with a status of UGL_STATUS_OK if it successfully attaches the extension. If the extension is not present, the driver returns a UGL_STATUS_ERROR.

**NOTE:** Typically, an extension will provide a wrapper around the uglInfo call that attaches an extension. This is so that it can provide a simpler interface and do any additional 2D initialization that it may require. For example, the video extension uses the uglVideoInit() function to perform general video extension initialization in addition to attaching the video extension to the driver.

### 7.6.2 Controlling a Font Driver

The uglFontDriverInfo() routine provides control of a font driver. The control functions are specific to a particular font engine. This function has the following format:

```c
UglFontDriverInfo(fontDevId, requestType, argument);
```

The fontDevId is the identifier given to the font engine when it was created, usually by the uglInitialize() function, and obtained from the driver registry. Refer
to the font driver documentation for the control operations that can be specified for a specific font engine.

7.6.3 Controlling an Input Driver

The *uglInputDeviceInfo()* routine allows you to control an input driver. This function has the following format:

```
uglInputDeviceInfo(inputDevId, requestType, argument);
```

The `inputDevId` is the identifier given to the input device when the device was created, usually by the `uglInitialize()` function. The identifier must be obtained from the driver register before calling the `uglInputDeviceInfo()` function. This function allows you to control the following features in an input device:

1. Pointer devices
   - Obtain pointer type
     By specifying `UGL_DEVICE_GET_POINTER_TYPE` as the request type, the pointer type is returned as the argument, as follows:
     
     ```
     UGL_DEVICE_POINTER_TYPE_TOUCH_SCREEN
     touch screen pointer type
     UGL_DEVICE_POINTER_TYPE_MOUSE
     standard mouse providing relative (x, y) movement
     UGL_DEVICE_POINTER_TYPE_PEN_TABLET
     pen
     ```
   - Set pointer location
     By specifying `UGL_DEVICE_SET_POINTER_LOCATION` as the request type, the pointer is repositioned to the coordinates provided by the `argument`. This does move the pointer on the display, it only resets the reported coordinates to the new location.
   - Constrain pointer location
     By specifying `UGL_DEVICE_SET_SCREEN_CONSTRAINT` as the request type, the reported location is constrained to the rectangle specified by the argument.

For additional details concerning constraining the pointer location and the repositioning of the reported cursor position, refer to section 3.5 Cursor Management, p.62.
2. Keyboard devices
   – Allow application to control LEDs
     By specifying UGL_DEVICE_SET_LED_CONTROL as the request type, you can specify that the keyboard driver control the state of the LEDs or the application control the LEDs.

     When the argument is set to UGL_TRUE, the application controls the LEDs. When it is set to UGL_FALSE, the driver controls the state of the LEDs based upon the Num Lock, Caps Lock, and Scroll Lock keys.

   – Set the LED state
     When the application controls the LED states, the UGLDEVICE_SET_LED request causes the LEDs to change state according to the value in the argument. The argument is a bit mapped value representing each of the LEDs, as follows:
     
     Bit 0 = Scroll Lock
     Bit 1 = Caps Lock
     Bit 2 = Scroll Lock

     When the bit is set the corresponding LED is set, and when it is clear the LED is turned off.

   – Get the LED state
     By specifying the request as UGL_DEVICE_GET_LED, the application can obtain the current setting of the LEDs in the argument. The argument is formatted in the same way as for the Set LED request.

7.6.4 Setting the Display Mode

Most graphics devices are capable of operating in several modes. The mode describes the primary features of the graphics device, such as:

- resolution (width and height)
- color depth
- output device type (CRT monitor or flat panel)
- refresh rate

The modes that are available for the graphics drivers supplied with WindML are described in the driver specific documentation in A. Standard Drivers. The mode is
normally configured in the WindML configuration process and is set at run time by the `uglInitialize()` function.

Typically, an application does not use the `uglModeAvailGet()` and `uglModeSet()` functions. They are used if an application wants to switch output devices or resolutions.

⚠️ **CAUTION:** WindML graphics drivers must be explicitly designed to support mode switching, after the initial mode has been set. Switching the mode of a driver that does not support dynamic mode switching will result in undefined behavior.
8

Graphics Device Drivers

8.1 Introduction

The WindML graphics driver architecture provides assistance to the developer whether the priority is on rapid driver development or a high degree of customization. It allows:

- rapid bootstrapping of standard output drivers using a generic driver which provides a comprehensive set of common driver functionality. Several types of generic drivers are shipped with WindML.
- development of accelerated and optimized graphics drivers for unique and specialized graphics devices

The graphics driver architecture provides this flexibility without compromising the implementation of either task. The generic API functionality can be used as extensively as possible, or can be completely overridden or bypassed as required. The approach chosen by the developer will be dictated by end-product requirements and the hardware being used.

NOTE: In addition to generic drivers, WindML also supplies reference drivers for common hardware. You can use one of the reference drivers supplied with WindML if your target hardware exactly matches the target hardware for which the reference driver was written. See the WindML Release Notes for the specifications of the reference drivers. You do not have to do any further driver development unless you want to optimize or customize the reference driver.
There are three generic drivers provided with WindML. Choose the appropriate driver based on your target hardware:

- **The 16 bit generic linear frame buffer** driver. This is suitable if the driver is for a 16 bit direct color graphics device with a linear frame buffer.
- **The 8 bit generic linear frame buffer** driver. This is suitable if the driver is for an 8 bit indexed color graphics device with a linear frame buffer.
- **The generic pixel-based frame buffer** driver. This is suitable if the driver is for a device that does not have a linear frame buffer, or for a device that uses a color depth not supported by WindML. This driver provides the fastest way of getting a non-linear driver up and running. However, because it relies upon pixel operations, its performance is substantially slower.

For each of these generic drivers, you need to implement a minimal set of hardware specific routines.

The generic drivers were written to perform a trade-off between optimization and portability. After implementing your driver, you may also want to perform additional optimization for your specific hardware.

### 8.1.1 The Generic Drivers

The architecture of a generic driver is shown in Figure 8-1.

A graphics driver:

- Exposes a public hardware abstraction layer to the 2-D API, using the WindML Graphics Interface layer (UGI) layer. The driver routines and data are directly accessed by the WindML 2D layer, which does not need to know the specifics of the underlying hardware. The developer-written set of routines are accessed by setting function pointers in the public driver data structures.
- Implements a private set of routines and data which are internal to the driver. Most of these routines and data are provided by the generic driver.

Custom WindML driver development generally starts with the generic driver code as its foundation and migrates toward more specific and optimized implementations by selectively replacing generic driver routines. As WindML is shipped in source code, the developer can rework the driver API if the application or hardware requires it.
8.1.2 The UGL Graphics Interface (UGI) API

The public API, through which the 2-D layer accesses the graphics driver routines, is known as the UGL Graphics Interface (UGI) layer. The primary interface mechanism for the UGI is a data structure consisting of function pointers and various data items. The function pointers allow the 2D layer to access the graphics driver. This data structure, the `ugl_ugi_driver` structure, is shown in Example 8-1.

Example 8-1  The `ugl_ugi_driver` Structure

typedef struct ugl_ugi_driver
{
    /* Data Members */
    UGL_MODE * pMode; /* display mode */
    UGL_PAGE * pPageZero; /* First Page */
    void * extension; /* optional driver extensions */
    /* UGI function pointers */
    /* General */
    UGL_STATUS (* info) (struct ugl_ugi_driver * pDriver,
                           UGL_INFO_REQ infoRequest, void *info);
    UGL_STATUS (* destroy) (struct ugl_ugi_driver * pDriver);
}
An instance of the `ugl_ugi_driver` structure must provide all the global data and function pointers required for a graphics device. If the graphics device requires additional data items that are not present within this data structure, another data structure can be derived from this data structure that contains the additional data items. For example, the generic drivers create a new data structure derived from the `ugl_ugi_driver` structure. The function pointers can point to generic driver functions or functions supplied by a driver writer.

The `ugl_ugi_driver` structure instance is allocated and initialized by the driver’s `xxxDevCreate()` routine.

A programmer could write a complete graphics driver by writing a driver to the interface defined by `ugl_ugi_driver` structure. However, writing a driver directly to this API, from scratch, generally requires substantial effort and is not practical. The generic driver code provides a complete graphics driver framework which the programmer can use to develop a WindML graphics driver.
8.1.3 The Generic Driver API

Each `ugl_generic_driver` structure contains pointers to the generic driver functions, global data for a particular device, and an instance of the `ugi_driver`.

It also contains function pointers and variables used internally by the generic driver code which are not visible to the WindML 2D routines. Example 8-2 shows some of the `ugl_generic_driver` structure.

Example 8-2 A Portion of the `ugl_generic_driver` Structure

```c
typedef struct ugl_generic_driver {
    UGL_UGL_DRIVER ugi;  /* UGI structure (required) */

    /* Device Data */
    void * fbAddress;   /* Fixed Frame Buffer Address */
    UGL_MEM_POOL_ID videoMemPoolId; /* ID of video memory pool */
    UGL_PAGE * pDrawPage; /* page to which rendering occurs*/
    UGL_PAGE * pVisiblePage; /* page visible on display */

    /* Generic Driver Data */
    UGL_GC_ID gc;        /* Active graphics context */
    UGL_GEN_DDB * scratchBitmap; /* used for transparent Blts */
    UGL_ORD transBitmapCount; /* used for transparent Blts */
    UGL_CLUT_STRUCT * pClutStruct; /* color lookup table */
    void * pCursorData;   /* used for cursor support */
    void * extension;     /* optional driver extensions */
    UGL_BOOL gpBusy;      /* GP wait */

    /* Generic Driver Routines */
    UGL_STATUS (* fbPixelSet) (struct ugl_generic_driver * pDriver,
                                UGL_POINT * pPoint, UGL_COLOR color);
    UGL_STATUS (* fbPixelGet) (struct ugl_generic_driver * pDriver,
                                UGL_POINT * pPoint, UGL_COLOR *pColor);
    UGL_STATUS (* hLine) (struct ugl_generic_driver * pDriver, UGL_POS y,
                            UGL_POS x1, UGL_POS x2, UGL_COLOR color);

    ...}
```

The core set of hardware-specific routines that need to be written for all of the generic driver types is:

- `xxxDevCreate()`, `xxxDevDestroy()`, `xxxInfo()`  
- `xxxModeSet()` and `xxxModeAvailGet()`

In addition to this set, generic 8 bit linear frame buffer devices require:

- `xxxClutEntrySet()` and `xxxClutEntryGet()`
In addition to all of the above routines, generic pixel-based frame-buffer devices require:

- `xxxFbPixelGet()` and `xxxFbPixelSet()`

8.2 Writing a Graphics Driver

8.2.1 Initial Planning and Design

Before you start actual coding of a new graphics driver, you must make the following design decisions based on your target graphics device.

1. Determine the following information about the graphics device:
   - The default single screen resolution to be supported. As a general rule, the more chip models, resolutions, and color depths that a driver supports, the larger the code size and memory requirements of the driver.
   - The associated color depth, 8 bit, 16 bit, or some other color depth. WindML generic linear frame buffer drivers support only one color depth to conserve memory.
   - The monitor type.
   - Whether the frame buffer will be linear and reside in continuous memory.

2. Decide which type of WindML generic driver implementation to start with—the 8 or 16 bit generic linear frame buffer implementation, or the generic pixel-based frame buffer implementation.

3. Determine the initialization necessary for the desired resolution and color depth on the chosen monitor type.

   In some embedded applications a graphics driver may only support a particular graphics chip model, with a specific monitor type, on a particular board, with a single resolution and color depth.

4. Typically, an embedded system contains a single graphics output device such as a single CRT or an flat panel display. However, some embedded systems have multiple graphics devices or multiple graphical layers in a single graphics device. In these cases, you may need to:
• have the WindML graphics driver initialize more than one device if the devices belong to the same chip family
• implement a graphics device driver for each device

5. Programmers should document each newly created driver. The documentation can take the form of a simple xxx.txt file or equivalent and should contain the following types of information:
   – the resolutions and color depths supported by the driver
   – the hardware platforms the driver was tested on
   – the types of monitors the driver will or will not run on
   – the graphics chip models on which the driver was tested
   – special compilation or configuration issues

The following excerpts are from the uglVga.txt file:

=================================================================
VGA - 4/8 Bit Device Driver
=================================================================

This driver provides graphics support for UGL on standard VGA devices. Only VGA modes 12 and 13 are currently supported as described below.

-----
Modes
-----

The following resolutions are supported by this driver:

640x480 at refresh rate of 60 Hz (mode 12)
320x200 at refresh rate of 60 Hz (mode 13)
...

-------------
Scalability
-------------

This driver may be scaled using the following defines:

#define INCLUDE_UGL_JPEG /* include Generic (software) JPEG extension */
...

-------------
Configuration
-------------
...

...
Supported Processors

This driver has been tested on the following processors:

Intel X86

Tested in all supported resolutions and frame buffer formats.

8.2.2 Creating the Graphics Driver Files

You should become familiar with the WindML driver directory structure, determine the new files and directories that are required for the new driver, and how these files will fit into the existing WindML directory structure. For a complete breakdown of the WindML directory structure, see 15. WindML Source Conventions.

NOTE: It is strongly recommended that you follow the conventions described in 15. WindML Source Conventions when you name directories, source files, and routines.

Source Code Directory Structure

The base directory for the source code of all WindML graphics drivers is as follows:

target/src/ugl/driver/graphics

The subdirectories found in this directory contain the source code for generic drivers and the reference drivers. The user can add other subdirectories as needed.

Examples of subdirectories located in this directory are:

- generic (generic driver source code)
  - generic/16bit (generic 16 bit driver source code)
  - generic/8bit (generic 8 bit driver source code)
- vga (VGA driver source code).
- chips (Chips & Technologies driver source code)

Create the directory structure for your driver as follows:

1. Create a new subdirectory if you are creating a driver for a device that does not already have a subdirectory under graphics. Any subdirectory under the
graphics directory can have further subdirectories, to manage several models of a particular family of graphics chips.

2. Copy a Makefile from one of the existing subdirectories, such as the chips directory to your new subdirectory.

3. Copy and rename any existing driver source code to use as a template for your new driver.

4. Create the xxx.txt file containing information about anything that a user of the driver should know about.

See 2.7.2 Adding a Custom Driver, p. 39 for more information on adding a new driver to the WindML configuration.

**Header File Directory Structure**

Next you need to create header files for the driver. The base directory for the header files is as follows:

```
target/h/ugl/driver/graphics
```

This directory contains several subdirectories that each contain driver-specific header files. Examples of subdirectories located in this directory are:

- **generic** (generic driver definitions)
- **chips** (Chips and Technologies driver-specific definitions)
- **vga** (VGA mode-specific driver definitions)

A new subdirectory in the **graphics** directory must be created for each new device containing device-specific header files.

**8.2.3 Implementing the Device Creation Routine (xxxDevCreate())**

All drivers must implement a device creation routine. This function, typically called the `xxxDevCreate()` function, is the main configuration point for a driver and is the first function in the driver. This function is called when the application calling the `uglInitialize()` function to initialize WindML. The driver's `xxxDevCreate()` function is the only required driver function that does not correspond to a function pointer in the `ugl_ugi_driver` or `ugl_generic_driver` structures.
xxxDevCreate() must perform the following actions:

1. Allocate the UGL_GENERIC_DRIVER or equivalent structure.
2. Initialize the UGL_GENERIC_DRIVER structure.
3. Locate and verify the graphics device.
4. Place the device in a quiescent state.
5. Perform double buffering initialization, if necessary.
6. Allocate and initialize video memory pool, if necessary.
7. Return a pointer to the driver’s structure.

**NOTE:** The driver’s header file should also contain the xxxDevCreate() function prototype

The routine must return a pointer to an initialized ugl_ugi_driver data structure. If the xxxDevCreate() routine fails to create the data structure or to initialize the device, a NULL pointer must be returned.

**Input Parameters**

The input parameters to xxxDevCreate() are of type UGL_UINT32.

The instance variable identifies the location of the graphics device. For example, in the case where the graphics device resides on a PCI bus, the first graphics device of a particular type would be instance number zero. For devices that are accessed by a physical address, the instance would identify the actual address of the graphics device.

The intLevel and intVector variables are used if a driver sets up an ISR (e.g. vertical blanking interrupt). Otherwise, they are zero.

The initialization tasks performed by the xxxDevCreate() routine are detailed as follows.

**Step 1: Allocating the UGL_GENERIC_DRIVER or Equivalent Structure**

The first step in writing the xxxDevCreate() routine is to allocate the driver’s ugl_generic_driver structure, or a derived form if one has been defined. The following example shows how this can be done assuming that the driver is being created using the following generic driver code.
UGL_UGI_DRIVER * uglVgaDevCreate
{
    UGL_UINT32 notUsed0,
    UGL_UINT32 notUsed1,
    UGL_UINT32 notUsed2
}
{
    UGL_VGA_DRIVER * pDriver;
    UGL_GENERIC_DRIVER * pGenDriver;
    UGL_UGI_DRIVER * pUgiDriver;

    /* Allocate the VGA driver */
    pDriver = (UGL_VGA_DRIVER *) UGL_CALLOC(1, sizeof (UGL_VGA_DRIVER));

    /* Check for valid memory */
    if (UGL_NULL == pDriver)
    {
        return (UGL_NULL);
    }

    /* Initialize the UGI portion of the driver (mandatory) */
    pUgiDriver = &pDriver->generic.ugi;
    uglUgiDevInit (pUgiDriver);
    ...
}

The VGA reference driver supplied with WindML does not allocate the
UGL_GENERIC_DRIVER structure, but allocates its own specific VGA driver
structure, UGL_VGA_DRIVER. This structure is derived from the
UGL_GENERIC_DRIVER but has some additional fields also.

Using the UGL_GENERIC_DRIVER structure is the easiest option. If you require
additional fields, define your own driver structure to include the
UGL_GENERIC_DRIVER structure

For more information on the generic driver definition, refer to the reference API
documentation for the generic driver routines.

Allocating Memory for the Driver Structure

Many graphics systems are configured to use a memory pool separate from the
system memory pool or possibly even a memory management system separate
from the general memory management system. This helps to prevent the graphical
application from fragmenting the system memory pool.

All memory allocations in the driver should be done using the WindML memory
management routines UGL_MALLOC(), UGL_CALLOC(), and so on. If a
separate memory pool or memory manager is being used, all memory allocated by WindML will be allocated from the designated memory pool.

Since extensions may be added to the UGL_GENERIC_DRIVER structure in the future that may not be supported by this driver, it is important that all fields be initialized to null, or a valid value.

After you allocate the UGL_GENERIC_DRIVER structure, you must call the uglUgiDevInit() routine. This function allocates system resources such as the system semaphore that provides mutual exclusion for the driver. The resources created by this function are used internally by WindML.

**Step 2: Initializing the UGL_GENERIC_DRIVER Structure**

The next step is to initialize the function pointers and data members of the driver structure.

At this point, you must decide which generic driver routines to use and which routines you need to implement yourself or customize. Your choice of generic linear frame buffer device type, or generic pixel-based frame buffer device type has a direct effect on these mappings and routines.

The initialization is a two step process:

- Initialize the UGL_UGI_DRIVER structure’s function pointers and data. This structure is part of the UGL_GENERIC_DRIVER data structure.
- Initialize the UGL_GENERIC_DRIVER structure’s function pointers.

The following code is an example of driver initialization.

```c
/* Establish VGA specific functionality */

pUgiDriver->destroy = uglVgaDevDestroy;
pUgiDriver->modeAvailGet = uglVgaModeAvailGet;
pUgiDriver->modeSet = uglVgaModeSet;
pUgiDriver->info = uglVgaInfo;
pUgiDriver->clutGet = uglVgaClutGet;
pUgiDriver->clutSet = uglVgaClutSet;
...

/* Establish the generic UGI functionality */

pUgiDriver->line = uglGenericLine;
pUgiDriver->polygon = uglGenericPolygon;
pUgiDriver->rectangle = uglGenericRectangle;
...
```
Step 3: Locating and Verifying the Graphics Device

After the driver’s data structure has been fully initialized, the graphics device should be located and verified as being present in the system.

For PCI-based systems, the following section of code will locate the device and determine where the device has been mapped into system memory:

```c
/* Find the device */
if (pciFindDevice (deviceID, chipID, instance,
        &busno, &devno, &funcno) == OK)
{
    /* Device found, obtain the base address */
    pciConfigInLong (busno, devno, funcno, PCI_CFG_BASE_ADDRESS_0,
        &baseAdrs);

    /* Processing when device found */
    . . .
}
else
{
    /* Device not found, release resources and abort initialization */
}
```

In this example, `pciFindDevice()` will probe the PCI bus for the device identified by `deviceId` and manufactured by vendor specified by `vendorId`. Note the `instance` variable, this is the variable that was passed into the `xxxDevCreate()` function.

The method used to locate the device varies according to the type of bus and graphics device. For example, if the graphics device resides on a PCI bus, the bus should be queried for particular device, using the `instance` variable passed into the routine. If the device is not present, `xxxDevCreate()` should do the following:

- call the `uglOutputDevDeinit()` routine, passing it the previously allocated `UGL_UGI_DRIVER` pointer
- free the `UGL_GENERIC_DRIVER` with `UGL_FREE` before returning NULL

Step 4: Placing the Graphics Device in a Quiescent State

After the existence of a graphics device has been verified, the driver performs minimal initialization of the device to place it in a quiescent state. (The device graphics mode is not set until a later stage—by calling the routine `uglModeSet()`).

The following operations may be performed at this stage:

- Disable graphics related interrupts
- Initialize the color palette by calling `uglGenericClutCreate()`
• Initialize the memory controller on the graphics chip
• Add the device’s frame buffer and register memory footprint to the CPU’s MMU table
• Place in PCI burst write mode
• Turn off the display
• Place VCO in a default or safe setting

The steps required to place a graphics device in the quiescent state vary according to the graphics device type. They can be complex due to the number of registers requiring initialization.

Step 5: Returning a Pointer to the Driver’s Structure
If the xxxDevCreate() function is successful, it should return a pointer to the initialized UGL_UGI_DRIVER structure, as follows:

```c
return(pUgiDriver);
```

If device creation fails, NULL must be returned.

### 8.2.4 Implementing the Destroy Routine (xxxDevDestroy())

Each driver must supply a xxxDevDestroy() routine. This routine frees system resources and deinitializes graphics hardware. The xxxDevDestroy() routine should perform the following tasks:

• Free the allocated color table
• Free any system resources
• Free any other driver specific resources
• Restore the graphics hardware to a text mode if appropriate
• Free the driver structure
• Return the status of the operation

An example of an xxxDevDestroy() routine supplied with WindML, shown below, is uglVgaDevDestroy():

```c
UGL_STATUS uglVgaDevDestroy
(
    UGL_DEVICE_ID devId /* device identifier */
)
8.2.5 Implementing the Info Routine (xxxInfo())

Each driver must supply a xxxInfo() routine. This routine allows the application to query the driver about the support it provides and to control various options. Typical uses of the xxxInfo() routine are to:

1. Obtain frame buffer characteristics such as frame buffer address, amount of video memory, display width and height.
2. Obtain color based information, such as color model (indexed or direct), color space (RGB, YUV, and so on), color depth, and size of color lookup table (when indexed color system).
3. Obtain the current operating mode such as the display width, display width height, color depth, and so on.
4. Instantiate an extension to the graphics device

The Info function structure is similar to the standard ioctl() function. This is an example of the uglVgaInfo() function for the VGA device driver:
UGL_STATUS uglVgaInfo
{
    UGL_DEVICE_ID devId, /* Device identifier - points to the driver data */
    UGL_INFO_REQ infoRequest, /* Type of request */
    void info /* Request argument */
}
{
    UGL GENERIC DRIVER * pDriver = (UGL GENERIC DRIVER *)devId;

    /* Make sure a valid mode was set, if not abort */
    if (devId->pMode == UGL NULL)
        return(UGL_STATUS_ERROR);

    /* Process based on the request type */
    switch (infoRequest)
    {
        case UGL FB_INFO_REQ:
        {
            UGL_FB_INFO *fbInfo = (UGL_FB_INFO *)info;
            fbInfo->width = devId->pMode->width;
            . .
            fbInfo->flags = 0;
        }
        break;
        case UGL COLOR_INFO_REQ:
        {
            UGL_COLOR_INFO *colorInfo = (UGL_COLOR_INFO *)info;
            colorInfo->cmode = UGL CMODEL_INDEXED;
            . .
            colorInfo->flags = UGL CLUT_WRITE;
        }
        break;
        case UGL MODE_INFO_REQ:
        {
            UGL_MODE_INFO *modeInfo = (UGL_MODE_INFO *)info;
            modeInfo->width = devId->pMode->width;
            modeInfo->height = devId->pMode->height;
            modeInfo->colorDepth = devId->pMode->colorDepth;
            . .
            modeInfo->flags = 0;
        }
        break;
        case UGL_EXT_INFO_REQ:
        {
            #ifdef INCLUDE UGL JPE
            UGL_EXT_INFO * extInfo = (UGL_EXT_INFO *)info;
            int version;
            if (strcmp(extInfo->name, UGL EXT JPE NAME) == 0)
            {
                . .
            }
        {
        (null)
8.2.6 Implementing the Mode Get and Set Routines

A mode is defined as the following information:

- Color depth or bits per pixel used by the frame buffer
- Resolution of the screen—the width and height of the display in pixels
- Refresh rate
- Output device type

Each driver must supply xxxModeSet() and xxxModeAvailGet() routines. These routines allow the application to determine which modes are available and to set the mode. This information is encapsulated in a UGL_MODE structure which is defined as follows:

```c
typedef struct ugl_mode
{
    UGL_UINT32 width; /* width in pixels */
    UGL_UINT32 height; /* height in pixels */
    UGL_UINT32 colorDepth; /* color depth */
    UGL_UINT32 refreshRate; /* refresh rate */
    UGL_MONITOR_TYPE monitorType; /* monitor type */
    UGL_UINT32 flags; /* general flags */
} UGL_MODE;
```
Each driver should declare a static global array of mode structures. The array should be initialized to contain all of the modes supported by a particular driver, as follows:

```c
UGL_LOCAL UGL_MODE modes[] =
{
    /* 640, 480, 8Bit Color */
    {640, 480, UGL_COLOR_DEPTH_8, 72, UGL_GENERIC_CRT, UGL_PSEUDO_COLOR_MODE}
};
```

The application will query the driver with the `uglModeAvailGet()` function to determine which modes are supported by a particular driver.

```c
xxxModeAvailGet()
```

The `xxxModeAvailGet()` function will return the complete array of modes. Some drivers may have multiple entries. For instance, for the VGA driver supplied with WindML:

```c
UGL_LOCAL UGL_MODE modes[] =
{
    /* VGA mode 12 */
    {640, 480, 4, 60, UGL_MODE_CRT, UGL_MODE_INDEXED_COLOR},
    /* VGA mode 13 */
    {320, 200, 8, 60, UGL_MODE_CRT, UGL_MODE_INDEXED_COLOR}
};
```

```c
UGL_LOCAL UGL_MODE * pDevModes = modes;
UGL_STATUS uglVgaModeAvailGet
{
    UGL_DEVICE_ID     devId,      /* device identifier */
    UGL_UINT32  *     pNumModes,  /* number of supported modes */
    const UGL_MODE ** ppMode      /* graphics device mode */
}
{
    /* Point to the mode information */

    *ppMode = pDevModes;
    *pNumModes = NELEMENTS(modes);

    return (UGL_STATUS_OK);
}
```

```c
xxxModeSet()
```

Using the mode array returned by `xxxModeAvailGet()`, the application determines which mode to use and then passes the desired mode entry from the array to the driver by calling the `uglModeSet()` function. The
**xxxModeSet()** function is responsible for setting the driver to a particular mode.

The **uglVgaModeAvailGet()** routine is as follows:

```c
UGL_STATUS uglVgaModeSet
{
    UGL_DEVICE_ID devId, /* device identifier */
    UGL_MODE * pMode    /* graphics device mode */
}
{
    UGL_VGA_DRIVER * pDriver = (UGL_VGA_DRIVER *)devId;
    UGL_GENERIC_DRIVER * pGenDriver = &pDriver->generic;
    UGL_UGI_DRIVER * pUgiDriver = &pGenDriver->ugi;
    UGL_INT32 index;
    UGL_GEN_DDB * pDdb;

    /* Find the requested mode mode */
    index = uglGenericModeFind (modes, pMode, NELEMENTS(modes));

    switch (index)
    {
    case 0: /* VGA mode 12 */
        ...
        /* Establish the GENERIC functionality */
        pGenDriver->bresenhamLine = uglGenericBresenhamLine;
        pGenDriver->fbPixelGet = uglVgaFbPixelGet;
        pGenDriver->fbPixelSet = uglVgaFbPixelSet;
        pGenDriver->fill = uglGenericFill;
        pGenDriver->hLine = uglVgaHLine;
        pGenDriver->vLine = uglVgaVLine;
        pGenDriver->rectFill = uglGenericRectFill;
        ...
        break;
    case 1: /* VGA mode 13 */
        ...
        /* Establish the GENERIC functionality */
        pGenDriver->bresenhamLine = uglGeneric8BitBresenhamLine;
        pGenDriver->fbPixelGet = uglGeneric8BitFbPixelGet;
        pGenDriver->fbPixelSet = uglGeneric8BitFbPixelSet;
        pGenDriver->fill = uglGenericFill;
        pGenDriver->hLine = uglGeneric8BitHLine;
        pGenDriver->vLine = uglGeneric8BitVLine;
        ...
        break;
    
```
Many embedded systems are only used with a single type of monitor for the life of the device, and therefore support one mode only. This allows them to conserve memory.

Initialization of a graphics chip to a particular mode can be one of the most challenging and time consuming areas of graphics driver development. In general, programmers must consult hardware manuals, OEM supplied code, or rely on prior experience to determine the initialization code required for a given type of graphics device.

Another step involved with setting the mode is to create a page zero, possibly initializing double buffering, and to create a video memory pool. These steps must be performed during the mode set since the amount of memory required for the display frame buffer is not known until the mode set phase.

1. Create Page Zero and Initialize Double Buffering

   All drawing operations need working memory, either from video memory or system memory. WindML refers to the first page of this memory as Page Zero. The xxxDevCreate() function must create Page Zero regardless of whether the driver supports double buffering or not.

   To create Page Zero, create a bitmap header—a Device Dependent Bitmap (DDB)—for the visible frame buffer. This allows the frame buffer to take on many of the same characteristics of a bitmap.

   An un-accelerated driver with a linear frame buffer refers to the frame buffer through this bitmap header. The visible page and draw page pointers in the UGL_GENERIC_DRIVER must be initialized and set equal to the Page Zero.

   The following example shows how the VGA driver creates Page Zero:

   /* In VGA Mode 13, we need to allocate an additional generic ddb to * get proper resolution to the frame buffer. Note the image address * is still the frame buffer (fbAddress). */
   UGL_GENERIC_DDB *pDdb = (UGL_GENERIC_DDB *)UGL_CALLOC(1, sizeof(UGL_GENERIC_DDB));
   pDdb->header.type = UGL_DDB_TYPE;
   pDdb->header.height = pUgiDriver->pMode->height;
   pDdb->header.width = pUgiDriver->pMode->width;
   pDdb->stride = pUgiDriver->pMode->width;
   pDdb->colorDepth = pUgiDriver->pMode->colorDepth;
   pDdb->image = pGenDriver->fbAddress;
   /* Allocate page zero */
   pUgiDriver->pPageZero = (UGL_PAGE *)UGL_CALLOC(1, sizeof(UGL_PAGE));
   pUgiDriver->pPageZero->pDdb = (UGL_DDB *)pDdb;
   /* Establish the GENERIC functionality */
/* Set the draw and visible pages */
...
break;

2. Allocating and Initializing a Video Memory Pool

In some cases, you may want to allocate a memory pool in video memory (that is, a device memory pool). This memory pool can be used as a bitmap cache or font glyph cache, or for off-screen bitmaps, double buffering, or any combination of these uses, see 3.8 Memory Management, p.73.

If adequate video memory exists and the proper acceleration routines are used, a memory pool in video memory can significantly improve performance. The following example shows how to create a memory pool in video memory:

/* Find the beginning of unused portion of the frame buffer */
videoMemAddress = (void *) ((UGL_UINT32)pGenDriver->fbAddress +
    displayWidth * displayHeight);

/* Determine the entire size of video memory */
*(UGL_UINT32 *)pDriver->fbAddress = MAGIC_NUMBER;
address = pGenDriver->fbAddress;

/* Video memory wraps back to the beginning */
while(*(UGL_UINT32 *)pDriver->fbAddress == MAGIC_NUMBER)
{
    address += 0x100000;
    *(UGL_UINT32 *)address = DIFFERENT_MAGIC_NUMBER;
}

/* Calculate the size of the unused portion of video memory */
videoMemSize = (UGL_UINT32)address - (UGL_UINT32)pGenDriver->fbAddress;

/* Create the video memory pool ID */
pGenDriver->videoMemPoolId = uglMemPoolCreate(videoMemAddress,
    videoMemSize -
    (displayWidth *
    displayHeight));

If the driver does not create a video memory pool, the videoMemPoolId in the UGL_GENERIC_DRIVER structure should be left as NULL.
8.2.7 Implementing the CLUT Get and Set Routines

If you are implementing the generic 8 bit linear frame buffer device (with indexed color) you must provide the xxxClutSet() and xxxClutGet() routines. Otherwise, these function pointers can be set to NULL.

xxxClutGet()

Gets a specific Clut entry.

xxxClutSet()

Sets a specific Clut entry.

The uglVgaClutSet() and uglVgaClutGet() routines for the VGA driver are as follows:

```c
UGL_STATUS uglVgaClutSet
{
    UGLDEVICE_ID devId,       /* device identifier */
    int startIndex,           /* starting color index */
    UGLARGB * pColors,        /* RGB colors */
    UGLSIZE numColors          /* number of colors to set */
}
{
    UGLSTATUS status;
    UGLVGA_DRIVER * pDriver = (UGLVGA_DRIVER *)devId;

    /* Defer to the generic clut routine */
    status = uglGenericClutSet (&pDriver->generic.ugi, startIndex, pColors, numColors);

    /* Assign the colors to the hardware */
    if (status == UGL_STATUS_OK)
    {
        int i;

        /* Specify the starting offset in the color table */
        UGL_X86_OUT_BYTE(0x3C8, startIndex);

        /* Assign each color */
        for (i = 0; i < numColors; i++)
        {
            UGL_X86_OUT_BYTE(0x3C9, (UGL_RGB_RED(pColors[i]) >> 2));
            UGL_X86_OUT_BYTE(0x3C9, (UGL_RGB_GREEN(pColors[i]) >> 2));
            UGL_X86_OUT_BYTE(0x3C9, (UGL_RGB_BLUE(pColors[i]) >> 2));
        }
    }

    return (status);
}
```

8.2.8 Implementing the Frame Buffer Pixel Manipulation Routines

If you based your driver on the generic linear frame buffer device code (8 or 16 bit) supplied with WindML, then you do not need to re-implement the pixel manipulation routines, the generic routines are sufficient.

If your driver is based on the generic pixel-based frame buffer device code, either because you have a non-linear frame buffer, or an unsupported color depth, then you need to implement the routines xxxFbPixelSet() and xxxFbPixelGet(). The generic driver includes uglGenericPixelGet() and uglGenericPixelSet() routines which use these frame buffer pixel set and get routines.

xxxFbPixelSet()

Reads the color value of a specified pixel, and returns it as a UGL_COLOR.

xxxFbPixelGet()

Sets the color of a specified pixel.

The function pointers would be set in the xxxDevCreate() routine as follows:

```c
case 0: /* VGA mode 12 */
    /* Establish the GENERIC functionality */
    pGenDriver->fbPixelGet = uglVgaFbPixelGet;
    pGenDriver->fbPixelSet = uglVgaFbPixelSet;
    ...
    break;
```
case 1: /* VGA mode 13 */
    /* Establish the GENERIC functionality */
    pGenDriver->fbPixelGet = uglGeneric8BitFbPixelGet;
    pGenDriver->fbPixelSet = uglGeneric8BitFbPixelSet;
    ...
    break;

Once you have written optimized xxxPixelSet() and xxxPixelGet() routines, you no longer need to use the uglGenericPixelGet() and uglGenericPixelSet() routines.

A VGA mode 12h driver, for example, uses four color planes that are adjacent to each other and linearly organized internally. Taking a byte from each plane, we can assemble eight pixels. Each pixel contains four bits that are used to index into a CLUT. To draw to a single pixel requires writing 4 bytes to the frame buffer. If the other pixels in the byte are not to be overwritten, all four bytes must first be read. The new pixel location is cleared while retaining the other pixels' data. The new pixel data is then stored to the bytes which are then written to their respective frame buffer planes. VGA mode 12h is a complex driver, but is representative of situations you may need to resolve.

You can then set the fbPixelGet and fbPixelSet function pointers in the UGL_GENERIC_DRIVER structure to NULL.

These are the VGA routines uglVgaFbPixelSet() and uglVgaFbPixelGet():

UGL_STATUS uglVgaFbPixelSet
{
    UGL_GENERIC_DRIVER * pGenDriver,
    UGL_POINT * pPoint,
    UGL_COLOR color
}
{
    volatile UGL_UINT8 dummy;
    UGL_UINT8 * address;
    UGL_UINT8 mask;
    UGL_VGA_DRIVER * pDriver = (UGL_VGA_DRIVER *)pGenDriver;

    /* Compute the pixel address and mask */
    address = ((UGL_UINT8 *)pGenDriver->fbAddress) +
              pPoint->y * (640 / 8) + pPoint->x / 8;
    mask = 0x80 >> (pPoint->x & 0x07);

    /* Set color into the set/reset register (if it has changed) */
    if (pDriver->color != color)
    {
        UGL_X86_OUT_WORD(0x3CE, (unsigned short)(color << 8));
        pDriver->color = color;
    }
void uglVgaFbPixelPut(UGL_GENERIC_DRIVER * pGenDriver, UGL_POINT * pPoint, UGL_COLOR * pColor)
{
    int plane;
    UGL_UINT8 *address;
    UGL_UINT8 mask;

    /* Compute the pixel address and mask */
    address = ((UGL_UINT8 *)pGenDriver->fbAddress) +
               pPoint->y * (640 / 8) + pPoint->x / 8;
    mask = 0x80 >> (pPoint->x & 0x07);

    /* Select the read plane register*/
    *pColor = 0;
    UGL_X86_OUT_BYTE(0x3CE, 0x04);
    for (plane = 0; plane < 4; plane++)
    {
        /* Select, then read one plane */
        UGL_X86_OUT_BYTE(0x3CF, plane);
        if (*address & mask)
            *pColor |= 1 << plane;
    }
    return (UGL_STATUS_OK);
}

UGL_STATUS uglVgaFbPixelGet(UGL_GENERIC_DRIVER * pGenDriver, UGL_POINT * pPoint, UGL_COLOR * pColor)
{
    int plane;
    UGL_UINT8 *address;
    UGL_UINT8 mask;

    /* Compute the pixel address and mask */
    address = ((UGL_UINT8 *)pGenDriver->fbAddress) +
               pPoint->y * (640 / 8) + pPoint->x / 8;
    mask = 0x80 >> (pPoint->x & 0x07);

    /* Select the read plane register*/
    *pColor = 0;
    UGL_X86_OUT_BYTE(0x3CE, 0x04);
    for (plane = 0; plane < 4; plane++)
    {
        /* Select, then read one plane */
        UGL_X86_OUT_BYTE(0x3CF, plane);
        if (*address & mask)
            *pColor |= 1 << plane;
    }
    return (UGL_STATUS_OK);
}
8.2.9 Packaging a Graphics Driver - For Third Party Developers

When you are developing your driver, you should use conditional compiles to allow scaling of the driver functionality. Scaling may involve including or excluding specific modes, optional components, or the selection of the output device supported.

A graphics driver uses the `uglInit.h` header file to obtain the specified driver configuration. The driver must follow the following code sequence to retrieve the configuration:

```c
/* Get the Device Driver Configuration Definitions */
#ifndef INCLUDE_
#define INCLUDE_
#define BUILD_DRIVER
#include <uglInit.h>
#endif /* INCLUDE_ */
```

In this example, `xxxx` specifies the name of the graphics device.

For example, for the MediaGx graphics card, the definition would be `INCLUDE_MEDIAGX_GRAPHICS`. For a user defined graphics device, the definition should be `INCLUDE_CUSTOM_GRAPHICS`.

Support for the following set of definitions must be implemented so that features in an WindML graphics driver can be included or excluded (that is, scaled). The driver must implement the following macros for conditional compilation:

- **Selection of specific mode**
  A driver may be scaled to include specific color depths, frame buffer formats, and resolutions. The code to include support for a specific mode is included when the following are defined:

  **INCLUDE_UGL_format**
  Includes support for a frame buffer with the format defined by `format`. For example, `INCLUDE_UGL_INDEXED8` would include code for 8 bit indexed frame buffers and `INCLUDE_UGL_RGB565` would include code for frame buffers with a format of RGB565.

  **INCLUDE_UGL_resolution**
  Includes support for a specific `resolution`. The resolution is represented as `width_height`. For example, `INCLUDE_UGL_640_480` defines a resolution of 640 x 480.

- **Selection of optional components**
A driver may also be scaled to include or exclude optional components, as follows:

**INCLUDE_UGL_OVERLAY**
Includes frame buffer overlay support

**INCLUDE_UGL_VIDEO**
Includes the video extension. Defining this macro includes the overlay support also.

**INCLUDE_UGL_ALPHA**
Includes support for alpha blending. This support allows you to blend an overlay with the primary frame buffer.

**INCLUDE_UGL_JPEG**
Includes JPEG image support.

**INCLUDE_UGL_SW_CURSOR**
Includes software cursor support.

**INCLUDE_UGL_DOUBLE_BUFFERING**
Includes support for multiple frame buffers.

- **Selection of output device**

A driver may also be scaled to include or exclude support for the type of output device, as follows:

**INCLUDE_UGL_CRT_MONITOR**
Includes support for standard CRT type monitors

**INCLUDE_UGL_flatPanel**
Includes support for the specified flatpanel display.

For example, **INCLUDE_UGL_SHARP_LM9V385** will provide support for the Sharp LM9V385 flat panel.

When the graphics device header file includes the **uglInit.h** file, the settings of each of these compile time definitions are retrieved. The retrieval of the configuration information by including this header file takes place regardless of whether the configuration is specified by the command line or the configuration tool mechanism.
9.1 Introduction

This chapter looks briefly at some specific advanced driver topics:

- clipping
- double buffering
- cursors
- overlays
- acceleration

9.2 Clipping

The graphics driver is responsible for performing clipping and viewport translation before the graphics data is written to the frame buffer. This section provides some general guidelines on how to use the clipping routines and macros provided by WindML.

With the exception of bitmaps, all WindML rendering routines must render to the default bitmap of the graphics context. At the 2D layer, drawing is clipped to the viewport, clip rectangle, and clip region. The 2D layer also uses the viewport to constrain the clip rectangle, so the driver need only concern itself with the translation of coordinates to the viewport, and clipping to the clip rectangle and region.
The general method of performing clipping is as follows:

1. Move the destination point(s) or rectangle so that the resulting coordinates are relative to the view port.

2. Check the `clipRegionId` member of the graphics context to see if a clip region ID exists.

3. If a clip region ID exists, traverse the clip rects with the `uglClipListGet()` routine. This routine will return a single clip rectangle for all of the clipping rectangles in the clip region, intersected with the `clipRect` member of the graphics context. Clip the destination data according to the clip rectangle returned by this routine.

4. If a clip region ID does not exist, clip the destination data according to the `clipRect` in the graphics context and ignore the clip region ID.

The following code example from `uglPixelSet()` shows how this can be done:

```c
/* move the destination point */
UGL_POINT_MOVE (*pPoint, gc->viewPort.left, gc->viewPort.top);
address = ((UGL_UINT16 *)pBitmap->image) + pPoint->y * stride + pPoint->x;

/* check for a clip region Id */
if (gc->clipRegionId != UGL_NULL)
{
    UGL_RECT clipRect;
    const UGL_RECT * pRegionRect = UGL_NULL;
    /* traverse clip list */
    while (uglClipListGet (gc, &clipRect, &pRegionRect) == UGL_STATUS_OK)
    {
        if (UGL_TRUE == UGL_POINT_IN_RECT(*pPoint, clipRect))
        {
            /* draw the pixel */
            if (gc->rasterOp == UGL_RASTER_OP_COPY)
                *address = (UGL_UINT16)color;
            else if (gc->rasterOp == UGL_RASTER_OP_XOR)
                *address ^= (UGL_UINT16)color;
            else if (gc->rasterOp == UGL_RASTER_OP_AND)
                *address &= (UGL_UINT16)color;
            else if (gc->rasterOp == UGL_RASTER_OP_OR)
                *address |= (UGL_UINT16)color;
            return (UGL_STATUS_OK);
        }
    }
}
```
Bitmap Clipping

The steps in 2D Drawing Clipping, p.172 are also valid for processing bitmaps. However, in the case of bitmaps, the default bitmap is not always required to be the source and/or destination. The source and destination data is always clipped to the bounds of the source and destination bitmap, as follows:

- The default bitmap is set in the graphics context along with the viewport. If the default bitmap is the source, then the source rectangle must be translated to be relative to the viewport in the graphics context. The `clipRect` and `clipRegionId` are not applied to the source.

- If the default bitmap is the destination, then the destination rectangle must be translated to be relative to the viewport in the graphics context. The `clipRect` and `clipRegionId` must be applied to the destination, just as with other drawing operations.

WindML also provides several generic bitmap clipping routines that may be used to perform bitmap clipping. These are defined in the file `target/src/ugl/driver/graphics/udgenclp.c`. Note that these clipping routines may modify the parameters passed to them. If necessary, use temporary variables.

This is an example of bitmap clipping:

```c
UGL_RECT clipRect;
const UGL_RECT * pRegionRect = UGL_NULL;

if (srcDdbId == UGL_DEFAULT_ID)
    UGL_RECT_MOVE (*pSourceRect, gc->viewPort.left, gc->viewPort.top);

if (destDdbId == UGL_DEFAULT_ID)
{
    if (uglClipListGet (gc, &clipRect, &pRegionRect) != UGL_STATUS_OK)
        return (UGL_STATUS_OK);

    UGL_POINT_MOVE (*pDestPoint, gc->viewPort.left, gc->viewPort.top);
}

do
{
    UGL_GEN_DDB * pSrcDdb = (UGL_GEN_DDB *)srcDdbId;
    UGL_GEN_DDB * pDstDdb = (UGL_GEN_DDB *)destDdbId;
    UGL_RECT srcRect = *pSourceRect;
    UGL_POINT dstPoint = *pDestPoint;

    /* Clip the source/destination points */
    if (UGL_TRUE == uglGenericClipDdbToDdb (devId, &clipRect,
                                             (UGL_BMAP_ID *)&pSrcDdb, &srcRect,
                                             (UGL_BMAP_ID *)&pDstDdb, &dstPoint))
    {
```
/* Bitmap processing code */
...
}

if (destDdbId != UGL_DEFAULT_ID)
    break;
}
while (uglClipListGet (gc, &clipRect, &pRegionRect) == UGL_STATUS_OK);

### 9.3 Double Buffering

Double buffering uses excess video memory for an off-screen frame buffer. In addition to having additional video memory, the graphics adapter hardware must support relocation of the frame buffer within video memory.

Graphics drawing and blitting can be done to the off-screen video memory. When it is time to move the off-screen data to the screen, double buffering simply relocates the frame buffer to the on-screen video memory’s location. This is a very quick operation that is typically synchronized with the vertical blanking of the monitor and thereby allows a flicker-free transition.

Double-buffering requires enough video memory to hold at least two frame buffers. Although it is called double buffering, WindML can support multiple off-screen buffers to be switched onto the screen. This means that the driver must be able to allocate the video memory specifically (some hardware may implement UMA (Unified Memory Architecture)) which uses regular memory for video memory.

If you are writing your own driver, you must implement WindML’s page management routines so that the double buffering capability is available to the 2D layer. In addition to the allocation of memory from the video pool, the creation of a page must also conform to the hardware requirements for frame buffers. For instance, the buffer may need to be aligned on particular byte boundaries and have a stride different from the frame buffer width.
9.4 Cursor Management

Cursor functions can be written to implement a software or a hardware cursor. Both types of cursor operate over the entire screen area.

**Hardware Cursor**

Hardware cursors are usually easier to use and move smoothly on the display. They have two disadvantages:
- not all graphics adapters support hardware cursors
- hardware cursors are typically restricted in their size

Hardware cursor configuration depends upon the particular hardware being used. Since a hardware cursor does not require any sort of blitting to move, the most difficult part of its implementation is creation and initialization.

The cursor may have its own CLUT, if an indexed color mode is being used. The cursor bitmap can be stored in video memory, using the video memory pool ID. Typically, the configuration of the cursor bitmap is different to the configuration used for other DDBs. Therefore, it requires its own special CDIB to CDDB conversion routine. Once the hardware cursor is set up, turning it on and off and moving it is usually just a matter of writing to control registers on the graphics adapter.

**Software Cursor**

WindML’s generic software cursor uses a specialized transparent bitmap for the cursor. You can implement the bitmap in any way you want, but if you intend to use the generic code, then you must make sure that the format is the same as that used by the generic code.

In WindML, a standard bitmap and monochrome mask are used to hold the transparent bitmap. The mask dictates whether the associated pixel in the standard bitmap is transparent (that is, not blitted). Transparency can be implemented in two ways:

- you can test the mask pixels to see if they are transparent or a color and then blit the standard bitmap’s pixels only if they are not transparent.
- you can use boolean operations on the pixels of the source (both the standard bitmap and the monochrome mask) and destination to get a transparency
effect. This method is called black source masking or AND XOR masking. The mask and standard bitmap of the source are ANDed, the result of that operation is then XORed with the destination bitmap. This allows both transparency, opacity, and inversion effects to occur.

If you do not want to use the generic code, you will need to implement the transparency code yourself.

The cursor must also keep a copy of the screen data underneath it, so that it can restore the screen data when it is moved across the screen.

The cursor must be hidden when any other routine in WindML draws to the screen, so that its screen buffer does not become invalid. This ensures that the cursor is not painted over, and also that the cursor does not subsequently try to restore the screen to stale data. As a performance optimization, you can hide the cursor only when drawing happens at the same location as the cursor. You can pass a rectangle to the generic cursor hide routine to indicate where drawing will take place. This optimization also reduces cursor flicker.

### 9.5 Overlay Surfaces

To create the overlay surface, the device driver must:

1. Validate the input arguments as supported by the device. If the arguments are invalid, return UGL_NULL.
2. Allocate an overlay control structure, derived from the ugl_ugi_driver structure.
3. Initialize the overlay control structure appropriately. This includes initializing the 2D rendering functions supported by the overlay. An overlay does not have to support all the 2D rendering functions. For example, if the overlay will never be used for drawing and its sole purpose is to capture video images from a camera, then the 2D rendering functions need not be implemented.

Function pointers that must be initialized are:

- info
- moveOverlay
- destroyOverlay
4. Position the overlay section, set the overlay mode, and start operations according to the input arguments.

5. Return the pointer to the created overlay control structure.

Moving an overlay repositions the overlay with respect to the primary frame buffer according to the input arguments. The total size of the overlay when it is moved must not exceed its size when it was initially created, that is, the width and the height of the overlay can change, but the area can not change:

\[(\text{newWidth} \times \text{newHeight}) \leq (\text{initialWidth} \times \text{initialHeight})\]

The steps required to move an overlay are:

1. Validate the input arguments
2. Move the overlay to the new position

### 9.6 Acceleration

Many graphics chips provide hardware support for the acceleration of rendering operations. These hardware features provide significant performance improvements by performing rendering operations while the CPU continues with other processing. The acceleration allows the processor to off-load graphics tasks to hardware so that the processor can perform other tasks in parallel.

Hardware acceleration provides two main benefits.

- The accelerator performs the operation with greater bus efficiency.
- The accelerator and the main processor perform parallel processing. WindML signals the accelerator to begin its operation and then returns from the WindML function. This means that the graphics processor is performing the accelerated operation while other code continues executing. If another graphics operation is required, the new operation blocks (using `gpWait()`) until the first is completed. All of the generic graphics routines test the `gpBusy` flag in the generic driver structure before changing frame buffer memory. If another graphics operation is not required, the CPU continues to execute in parallel with the graphics accelerator.
Two commonly accelerated operations are:

- **Bit Block Transfer (\texttt{bitBlt}())**
  
  Accelerating the \texttt{bitBlt}() operation achieves the greatest performance impact. This is a feature that the majority of accelerated graphics chips support in some fashion. The \texttt{bitBlt}() operation can be viewed as the movement of a rectangular image from one location to another. The source and destination locations may be off-screen or on-screen. Some graphics devices have limitations on which types of \texttt{bitBlt}() may be supported. For example, a device might only support on-screen to on-screen transfers.

- **Line Drawing**
  
  Graphics processors that support line drawing use various algorithms to implement accelerated line drawing. The two most common algorithms are the Bresenham and the Two Point algorithms. Also, since vertical and horizontal lines can be viewed as a filled rectangle, lines can be accelerated on graphics processors that do not have accelerated lines but do have an accelerated rectangle fill function.

### 9.6.1 Implementing Acceleration

Before you try to implement acceleration, you should have a thorough understanding of the accelerated features of the graphics device. If you accelerate inappropriate operations, they can actually take more time to complete using the accelerated features of the graphics device than if the main processor had performed the operation. For example, if a line of less than 10 pixels is being accelerated, it can take more time to set up the acceleration registers than it would take the main processor to directly write the 10 pixels to the frame buffer.

In general, you initialize the graphics hardware during \texttt{uglInitialize()} and then for each individual graphical operations. For each accelerated operation:

1. Prepare for operation, by making sure that the data is in a valid format for the accelerator. Calculate any values that are needed by the accelerator for this particular operation.

2. Wait for the graphics processor to be idle (using the \texttt{gpWait()} function).

3. Set the device's registers for the accelerated operation.

4. Signal the device to start the accelerated operation.
10

Input Drivers

10.1 Introduction

WindML supports two general classes of input device

- **pointing devices**—typically mice or touchscreens
  
  These devices can be either:
  
  - relative
    
    A relative pointer device reports the pointer movement since the previous movement, that is, relative to the previous movement.
  
  - absolute
    
    An absolute pointer device reports an absolute location.

- **keyboard devices**—a keyboard

In WindML, an input driver is responsible for reading raw data from an input device through a low level I/O system driver. It processes and formats the raw data into a WindML event and passes it to the event service which will pass the event to the application. A diagram of how the input driver interacts with the event service and application is shown in the Figure 10-1.
10.2 The Input Driver API

The public API through which the input task in the event service accesses the input driver is defined by the `ugl_input_driver` structure.

```c
typedef struct ugl_input_driver
{
    /* formatter function */
    UGL_STATUS (* format) (struct ugl_input_driver * pDevice, UGL_EVENT * pEvent);

    /* information function */
    UGL_STATUS (* info) (struct ugl_input_driver * pDevice, UGL_DEVICE_REQ devRequest, void * arg);

    /* destroy function */
    UGL_STATUS (* destroy) (struct ugl_input_driver * pDevice);
} 
```

Figure 10-1  Input Drivers

![Diagram of Input Drivers](image)
struct ugl_event_service * pService; /* associated input service */
UGL_UINT32  fd; /* file descriptor */
int devNum; /* device number */
UGL_UINT32  deviceType; /* type of device */
void * data; /* local data used by device */
} UGL_INPUT_DRIVER;

With the exception of the driver initialization routine (xxxDevInit()), the ugl_input_driver structure defines the minimum set of routines that must be written for all input drivers. These routines are:

- **xxxDevInit()**
  
  This routine is not a member of the ugl_input_driver structure, but is a routine that is called directly by uglInitialize(). This routine is responsible for the following tasks:
  
  - opening the device
  - setting the device to a proper mode
  - initializing the file descriptor, device number and type members of the ugl_input_driver structure
  - adding the device to the event service
  - returning a pointer to the newly created input driver

- **xxxDevDestroy()**

  This routine is pointed to by the ugl_input_driver (* destroy) field. It terminates operation of the device by closing the file descriptor and releasing any handler resources. This routine is called indirectly by the uglDeinitialize() routine.

- **xxxInfo()**

  This routine is pointed to by the ugl_input_driver (* info) field. It obtains and sets driver information, such as turning on or off keyboard LEDs. This routine can be called indirectly by the application with the uglInputDeviceInfo() routine.

- **xxxFormat()**

  This routine is pointed to by the ugl_input_driver (* format) field. It is called directly by the input task of the event service when the I/O system driver wakes the task up using select(). This routine reads raw data from the I/O system driver, processes the data, and assembles the processed data into an WindML input event which is returned to the input task.
Note that the API documentation in C. Event API contains more information on these routines.

In addition to the public routines described above, the input driver can also contain private routines and data structures as necessary.

### 10.3 Writing an Input Driver

#### Source Code Directory Structure

The source code of the WindML input devices is in the following locations:

- `target/src/ugl/driver/keyboard`
- `target/src/ugl/driver/pointer`

These directories contain the source code for the reference drivers. Typically, programmers can simply add the associated input driver file to either the keyboard or pointer sub-directory. You can also add other subdirectories as needed.

#### Header File Directory Structure

The header file directories are:

- `target/h/ugl/driver/keyboard`
- `target/h/ugl/driver/pointer`

As with the source directories, you can add files or sub-directories as needed for your input driver.

#### Input Driver Routine Naming Conventions

The names of the input driver routines should be unique to each driver and should fit within the following guidelines:

```c
vendorDeviceInputTypeDevInit()
```

`vendor` is a name that indicates the creator of the driver. For instance, `ugl` is used for drivers created by driver writers in with the WindML development group.
Device indicates the type of device for which the driver is intended. Ps2 for example, is used for PS/2 devices.

InputType indicates the type of input device and should be ptr for pointer devices or kbd for keyboard devices.

For example, these are the initialization routines for the input drivers written by the WindML team for a PS/2 mouse and a PC keyboard:

```
uglPs2PtrDevInit()
uglPcKbdDevInit()
```

### 10.3.1 Implementing the Initialize Routine (xxxDevInit())

Each input driver must provide a device initialization routine. This routine is called directly by the `uglInitialize()` routine and performs the following tasks:

- Registers a new input driver with the event service by calling the `uglInputDeviceAdd()` routine. This routine returns an instance of the `ugl_input_driver` structure.

- Initializes the `pService`, `fd`, `deviceType`, and `data` members as well as the `ugl_input_driver (*format), (*destroy), and (*info)` function pointers of a `ugl_input_driver` structure.

- Opens an I/O system device.

- Initializes communications with the I/O system device driver.

All WindML input device initialization routines should follow the prototype shown below:

```c
UGL_INPUT_DEVICE_ID xxxDevInit
{
    char * pDevName, /* name of device */
    UGL_EVENT_SERVICE_ID eventServiceId /* event service */
};
```

`xxx` is the device name.

`pDevName` is the name of the I/O system device that is to be opened.

`eventServiceId` is the ID of the event service to which the input device will report events.

The return value is an ID that can be used to identify the input device in subsequent operations. It is a pointer to the UGL_INPUT_DRIVER structure provided by the `uglInputDeviceAdd()` routine.
The members of the `ulg_input_driver` structure that are not function pointers are described below:

**pService**
This is a pointer to the event service associated with an input device. This field is initialized by the return value from `uglInputDeviceAdd()`.

**fd**
This is used to store the file descriptor returned when `open()` is used to open the associated I/O system device. This file descriptor is also used for read operations performed by the (*format*) routine.

**devNum**
This is a device number assigned to the input driver by the event service. This field is initialized by the `uglInputDeviceAdd()` routine.

**deviceType**
This indicates the type of input device. It should be assigned one of the following values:

```c
#define UGL_DEVICE_KEYBOARD         1   /* keyboard */
#define UGL_DEVICE_POINTER_ABSOLUTE 2   /* Absolute position pointer */
#define UGL_DEVICE_POINTER_RELATIVE 3   /* Relative position pointer */
```

**data**
This field can point to any private data that the input driver requires.

The following code is the initialization routine for the Microsoft serial pointer device:

```c
UGL_INPUT_DEVICE_ID  uglMsPtrInit
{
    char * pDevName, /* name of device */
    UGL_EVENT_SERVICE_ID eventServiceId  /* input service that device is assigned */

    UGL_MS_POINTER * pMsData;
    UGL_INPUT_DEVICE * pDevice;

    /* Create the device */
    pDevice = uglInputDeviceAdd (eventServiceId);
    if (pDevice == UGL_NULL)
        return (UGL_NULL);

    /* open the device */
    pDevice->fd = open (pDevName, 0, 0);
    if (pDevice->fd >= 0)
    {
        UINT32 cflag = CREAD | CS7;
```
/* identify as relative pointer device type */
pDevice->deviceType = UGL_DEVICE_POINTER_RELATIVE;

/* device was opened, initialize */
/* set baud rate to 1200 and operating mode */
ioctl (pDevice->fd, SIO_BAUD_SET, 1200);
ioctl (pDevice->fd, SIO_HW_OPTS_SET, cflag);
ioctl (pDevice->fd, FIOCSETOPTIONS, OPT_RAW);
ioctl (pDevice->fd, FIOFLUSH, 0);

/* set up function pointers for device */
pDevice->format = uglMsPtrFormatter;
pDevice->destroy = uglMsPtrDestroy;
pDevice->info = uglMsPtrInfo;

/* set device local data */
pMsData = pDevice->data = UGL_MALLOC (sizeof(UGL_MS_POINTER));
pMsData->state = 0;
pMsData->oldButton = 0;
}
else
    return (UGL_NULL);

return ((UGL_INPUT_DEVICE_ID) pDevice);

10.3.2 Implementing the Destroy Routine (xxxDestroy())

The destroy routine closes the associated I/O system device and frees any private resources used by the input driver. This routine is called indirectly by the uglDeinitialize() routine when uglInputDeviceDestroy() is called for each input driver.

The following code segment shows the destroy routine of the Microsoft-compatible serial mouse input driver.

UGL_LOCAL UGL_STATUS uglMsPtrDestroy
    
    
    (UGL_INPUT_DEVICE * pDevice /* device control structure */)
    
    { /* close the device */
        close (pDevice->fd);
    }
    
    /* free local storage */
    UGL_FREE (pDevice->data);

    return UGL_STATUS_OK;
}
10.3.3 Implementing the Format Routine (**xxxFormatter**)  

The format routine reads data from the associated I/O system device and formats it into **UGL_INPUT_EVENT** structures. The routine is called automatically by the event service’s input task whenever **select()** indicates that data is available from the associated I/O system device driver. The data will be in the format specified by the protocol for that device.

The following code segment shows the format routine used by the Microsoft serial mouse input driver supplied with WindML.

```c
/* Typedefs */
typedef struct ugl_ms_pointer
{
    UGL_INT8 byte0;              /* first byte in a packet */
    UGL_INT8 byte1;              /* second byte in a packet */
    UGL_INT8 byte2;              /* third byte in a packet */
    UGL_UINT8 state;              /* protocol state */
    UGL_UINT16 oldButton;          /* old button state */
} UGL_MS_POINTER;

 дену:

/**************************************************************************
 * uglMsPtrFormatter - format pointer input data to input event
 * *
 * This routine handles formatting of input device data into an appropriate
 * event. The Microsoft pointer packet consists of three bytes. Upon receipt
 * of a byte, the routine verifies synchronization and if not within sync,
 * then packet collection is reset to collect the first byte. After all three
 * bytes are collected, then the packet is decoded into an UGL input event.
 *
 * RETURNS:  UGL_STATUS_OK, when a packet was received and an input event
 * is ready,  UGL_STATUS_NOT_READY when an input event is not
 * ready,  or UGL_STATUS_DROP when the event had been dropped
 *
 * ERRNO: N/A
 *
 * SEE ALSO: uglMsPtrInit()
 *
 */

UGL_LOCAL UGL_STATUS uglMsPtrFormatter
(
    UGL_INPUT_DEVICE * pDevice, /* device control structure */
    UGL_EVENT *  pEvent /* event to build */
)
{
    UGL_INPUT_EVENT * pInputEvent = (UGL_INPUT_EVENT *)pEvent;
    unsigned char value;
    int readCnt = 1;
    UGL_MS_POINTER * pMsData = (UGL_MS_POINTER *) pDevice->data;
```
/* Get data, figure out which byte in 3 byte packet, and handle */
while (readCnt != 0) {
  ioctl (pDevice->fd, FIONREAD, (int) &readCnt);
  if (readCnt > 0) 
    readCnt = read(pDevice->fd, &value, 1);
  if (readCnt == 1) 
    { /* byte read from device, process */
      /* sync byte received, set to state 0 */
      if (value & 0x40)
        pMsData->state = 0;
      else
        pMsData->state++;

      switch (pMsData->state) 
      { 
        case 0: 
          pMsData->byte0 = value;
          break;
        case 1: 
          pMsData->byte1 = value;
          break;
        case 2: 
          pMsData->byte2 = value;

          /* build input event */
          pInputEvent->header.type = UGL_EVENT_TYPE_POINTER;
          pInputEvent->header.category = UGL_EVENT_CATEGORY_INPUT;

          /* get button presses and update state */
          if (pMsData->byte0 & 0x20)
            pInputEvent->type.pointer.buttonState = 
                UGL_POINTER_BUTTON1;
          if (pMsData->byte0 & 0x10)
            pInputEvent->type.pointer.buttonState |= 
                UGL_POINTER_BUTTON2;
          pInputEvent->type.pointer.buttonChange = 
              (pInputEvent->type.pointer.buttonState ^ 
              pMsData->oldButton);
          pMsData->oldButton = 
              pInputEvent->type.pointer.buttonState;

          /* get delta x and delta y */
          pInputEvent->type.pointer.dx = (signed char) 
              (((pMsData->byte0 & 0x03) << 6) | pMsData->byte1);
          pInputEvent->type.pointer.dy = (signed char) 
              (((pMsData->byte0 & 0x0C) << 4) | pMsData->byte2);

          /* return to input service to complete event and post */
          return (UGL_STATUS_OK);
      }
    }
}
The format routine may be called more than once for each generated event. For example, a serial mouse may report input data in the form of three byte packets. Rather than pending until an entire packet is read, the format routine should only read the data that is immediately available. If a packet is incomplete, it should store the data and return UGL_STATUS_EVENT_NOT_READY. When a complete packet is retrieved, it should fill in the `inputEvent` structure and return UGL_STATUS_OK.

### 10.3.4 Implementing the Info Routine (xxxInfo())

The (*Info) routine allows applications to send or retrieve information to or from an input driver. A pointer driver could allow sensitivity adjustments to be made by defined special information requests that it handles in its (*Info) routine. WindML defines the following type of information requests for input drivers:

```
#define UGL_DEVICE_SET_POINTER_LOCATION 1 /* set pointer location */
#define UGL_DEVICE_SET_SCREEN_CONSTRAINT 2 /* set pointer constraint */
#define UGLDEVICE_SET_LED_CONTROL 3 /* UGL/App control LEDs*/
#define UGL_DEVICE_SET_LED 4 /* set LED state */
#define UGL_DEVICE_GET_LED 5 /* get LED state */
#define UGL_DEVICE_SET_CALIBRATION 6 /* set pointer calibration */
```

Information requests defined by a driver should be negative in value. Positive values are reserved for definition by WindML.

The following code segment shows the Info routine of the Microsoft serial mouse input driver supplied with WindML.

```c
UGL_LOCAL UGL_STATUS uglMsPtrInfo
(  UGL_INPUT_DEVICE * pDevice, /* device control structure */
```
UGL DEVICE REQ request, /* request to perform */
void * arg /* argument for request */
)
{
if(pDevice == UGL NULL)
return(UGL_STATUS_ERROR);
switch (request)
{
case UGL DEVICE GET POINTER TYPE:
    if(arg != UGL NULL)
    {
        *(int *)arg = UGL DEVICE POINTER TYPE MOUSE;
        return(UGL STATUS OK);
    }
    else
    return(UGL STATUS ERROR);
    break;
default:
    return (UGL_STATUS_ERROR);
}
return UGL_STATUS_OK;

10.3.5 I/O System Drivers

I/O system drivers are expected to be standard I/O system drivers that provide the following basic I/O routines:
- open()
- close()
- read()
- write()
- ioctl()

In addition, the driver must also provide support for the select() routine which is used by the input task to detect incoming data. While most drivers provide open(), close(), read(), write(), and ioctl(), select() is not supported on some drivers. If not, support for select() must be added.

WindML ships with some sample BSP extensions showing the implementation of these routines. If your driver does not support these routines, the BSP extension implementations are available for different drivers. The BSP Extensions are found in the directory target/src/ugl/bspExt.
10.3.6 Packaging an Input Driver - For Third Party Developers

To add an input driver to WindML, developers should perform the following steps:

1. Create the source and header files in the following locations:

   target/src/ugl/driver/keyboard
   target/src/ugl/driver/pointer
   target/h/ugl/driver/keyboard
   target/h/ugl/driver/pointer

2. The header files must provide #define statements for the initialization routine in addition to the name of the I/O system device.

   The following example is from the Microsoft compatible serial mouse:

   ```c
   #define UGL_POINTER_INIT    uglMsPtrInit
   /* Default to the first serial port */
   ifndef SYS_POINTER_NAME
   #define SYS_POINTER_NAME "/tyCo/0"
   endif /* SYS_POINTER_NAME */
   ```
11 Audio Drivers

11.1 Introduction

This chapter assumes that you are familiar with basic audio device terminology. Audio support is provided in WindML by an audio driver which is structured as a standard I/O device, supporting the I/O routines:

- `open()`
- `close()`
- `read()`
- `write()`
- `ioctl()`

An application may pend for the availability of a buffer using `select()`, therefore the DSP device must also implement the `ioctl()` control functions `FIO_SELECT` and `FIO_UNSELECT`.

An audio driver uses two sub-devices to support the various functions required of a sound system. The sub-devices are:

- Digital Signal Processor (DSP)
  
  The DSP provides the digital to analog conversion required to playback a digitized sound. It also provides an analog to digital conversion allowing audio to be recorded.

- Mixer
  
  The mixer device is used by the application to control playback and recording levels and to mix the sound with other audio sources.
The primary device name is assigned when the device driver is installed. Typically, the name that is used when a single audio device is present is /sound. If the audio device driver is named /sound, then the DSP and the mixer are referenced as /sound/dsp and /sound/mixer, respectively.

The header file target/h/ugl/audio/sound.h provides the definitions for the audio API.

Device Initialization

Each audio driver must have an initialization routine—the xxxSndDevCreate() routine—where xxx indicates the name of the device. Parameters passed to this routine are the name of the device (for instance, /sound), the hardware channel to be used, the instance number of the board, the port number, and the IRQ number. Not all parameters are required by all devices.

Device Control

The audio driver interfaces with the Mixer device and the DSP device through the use of ioctl() control functions.

The ioctl() calls to the Mixer device identify the capabilities of the mixer device—if it provides support for stereo, for volume control, bass and treble modifications, microphones, and so on.

The ioctl() calls to the DSP device control the operation of the device. In addition to ioctl() calls, you use read() and write() routines with the DSP device to read and write audio streams.

Double Buffering

The WindML audio API supports a double buffering scheme. When an audio stream is being played, the hardware plays from one buffer while the processor is filling a second buffer with data. Double buffering is supported by the DSP sub-device.
11.2 Writing an Audio Driver

11.2.1 Writing the xxxSndDevCreate() Routine

The installation of audio drivers is done using the following routine xxxSndDevCreate():

```c
STATUS xxxSndDevCreate
{
    char *devName,    /* name of device */
    int channel,      /* hardware channel to use */
    int instance,     /* instance of the board */
    int port,         /* h/w port number */
    int irq,          /* IRQ number */
    int dma8,         /* DMA channel for 16 bit */
    int dma16         /* DMA channel for 8 bit */
}
```

The name of the install routine is always xxxSndDevCreate(), where the xxx specifies the unique device. For example, the installation routine for the IGS Technologies sound card is igsSndDevCreate().

A driver installation routine for a given audio driver may not use all of these parameters but each of the parameters must be present.

The parameters are as follows:

`devname`

The name that is assigned to the primary device, such as /sound. The name must always be preceded by the slash.

`channel`

The channel of the audio hardware that is to be used for the driver. Some hardware has multiple audio channels while other hardware may only have a single audio channel.

> NOTE: The parameter `channel` does not refer to a channel in the sense of left and right channels.

`instance`

The instance is used to uniquely identify the audio hardware location. The interpretation of this parameter is variable and highly dependent on the architecture of the hardware.

For example, if the hardware uses a PCI bus, then this value identifies the relative location of the PCI audio device. The first device is 0, the second is 1,
and so on. However, for a VME based audio system, this parameter would identify a physical address on the VME bus. For an ISA type device, the instance would typically identify the physical I/O port.

intLevel
This identifies the interrupt level that the device uses. This parameter is used only when the audio hardware has a variable interrupt level that is usually jumper selectable. For devices that have a fixed interrupt level which is read from device registers, or defined within the PCI configuration header, this parameter is not used.

intVec
This parameter identifies the interrupt vector that the device is to use. This parameter is only used when the system architecture requires an interrupt vector separate from the interrupt level and the information can not be obtained dynamically, such as from a PCI configuration header.

dma8
This parameter specifies the DMA port or channel that is to be used for 8 bit transfer. This parameter is only used when the audio driver uses a separate DMA controller for 8 bit transfers. If it uses a single DMA controller for both 8 and 16 bit transfers or the device has an integral DMA controller, then this parameter is not used.

dma16
This parameter specifies the DMA port or channel that is to be used for 16 bit transfers. It also provides the DMA controller for 8 bit transfers when the device uses a single DMA controller for both 8 and 16 bit transfers. When the device has an integral DMA controller then this parameter is not used.

These parameters attempt to cover the all of the potential data needed for an audio device. Usually, a given audio device driver will not require all of these parameters.

For example, the driver for the IGS Technologies sound card only uses the devName, channel, and instance parameters. The IGS audio driver has an embedded DMA controller, uses memory mapped I/O, and obtains the irq from the PCI configuration header—therefore the intLevel, intVec, dma8, and dma16 parameters are not used.
The following code is an example of calling the IGS initialization routine, assigning the driver the name `/sound` and setting it to use audio channel 0:

```c
if (igsSndDevCreate ("/sound", 0, 0, 0, 0, 0, 0) != OK)
   /* device was not installed successfully */
else
   /* device was successfully installed */
```

The device initialization routine should perform the following steps:

1. Check if the device is already installed. If so, return `UGL_STATUS_OK`.
2. Verify that the audio hardware is present. For PCI type devices, this involves finding the device on the PCI bus and reading the configuration header to obtain its base address and interrupt level. For other device types, this may include performing a probe of specific device registers to verify that the device is present. If the device is not present, the initialization sequence should abort and return `ERROR`.
3. Install the device driver using `iosDrvInstall()`.
4. Initialize `select()` handling using `selWakeupListInit()`.
5. Initialize the DSP component by placing it in a quiescent state.
6. Initialize the mixer component to default values.
7. Add the device to the I/O system.
8. Connect the interrupt handler.

The following example shows creation of an ac97 audio device.

```c
STATUS igsSndDevCreate
{
    char *devName, /* name of device */
    int channel, /* hardware channel to use */
    int instance, /* instance of the board */
    int intLevel, /* interrupt level (not used) */
    int intVec, /* interrupt vector (not used - from PCI header) */
    int dma8, /* DMA channel for 16 bit (not used) */
    int dma16 /* DMA channel for 8 bit (not used) */

    SND_DEV *pdev;
    UINT32 baseAdrs;
    UINT8 intLine;
    int busno, devno, funcno;
    DEV_HDR *hdr;

    /* if device is already present, do not create again */
    if (deviceIsPresent (devName, &hdr))
        return (OK);
```
/* Locate the device on the PCI bus */
if (pciFindDevice (IGS_VENDOR_ID, IGS5000_DEVICE_ID, instance, &busno, &devno, &funcno) == OK) {
    /* Device found, obtain the base address and interrupt */
    pciConfigInLong (busno, devno, funcno, PCI_CFG_BASE_ADDRESS_0, &baseAdrs);
    pciConfigInByte (busno, devno, funcno, PCI_CFG_DEV_INT_LINE, &intLine);
} else
    return (ERROR);

/* Get audio control structure */
pDev = (SND_DEV *)malloc (sizeof(SND_DEV));
bzero ((char *)pDev, sizeof(SND_DEV));
pDev->pBaseIOAdrs = (UINT8 *)0xb0000;
pDev->irq = intLine;
pDev->devSem = semBCreate (SEM_Q_FIFO, SEM_FULL);
pDev->intSem = semBCreate (SEM_Q_FIFO, SEM_FULL);
pDev->mode = O_RDONLY;
pDev->channel = channel;

/* Install the driver */
DrvNum = iosDrvInstall (NULL, NULL, igsSndOpen, igsSndClose,
                         igsSndRead, igsSndWrite, igsSndIoctl);

/* hook in select processing */
selWakeupListInit (&pDev->selList);

/* Initialize the DSP */
igsDspInit (pDev);

/* Initialize the mixer */
igsMixerInit (pDev);

if (iosDevAdd (&pDev->devHdr, devName, DrvNum) == ERROR) {
    free ((char *)pDev);
    return (ERROR);
}

/* Attach interrupt handler and enable interrupts */
intConnect (INUM_TO_IVEC (INT_VEC_GET (pDev->irq)), audioInterrupt, (int)pDev);
return (OK);
11.2.2 DSP Device

The DSP device must support the following operations:

- Initialization
- Control
- Playback and/or Recording

There is no requirement that DSP device supports both the playing and recording of audio streams. A functioning DSP device must support at least one of these operations.

Initialization

The DSP device is initialized when the audio device is created. The steps that are required for the DSP device initialization are:

1. Enable the audio at the hardware level.
2. Allocate a cache safe audio buffer for the playing and/or recording of audio samples.

The cache safe buffer implements a double buffering scheme. Although hardware implementations of an audio buffer scheme vary, this buffer is typically implemented as a circular buffer divided into equally sized segments. The entire buffer is called an audio block and each of the segments is an audio fragment.

A audio block is the amount of audio samples that the audio device can handle at any time. An audio fragment is the smallest quantity that the processor can send to the audio device (play) or receive from the audio device (record). However, the final fragment in a session may be smaller than an entire fragment.

DSP Control

The DSP is controlled using the ioctl() mechanism. The list of ioctl() functions that control the audio are listed in 3.11 Audio API, p.82. All of those ioctl() functions must be supported by the DSP device. In addition, the ioctl() FIOSELECT and FIOUNSELECT control functions must be supported to implement the select() operation.

An ioctl() control function which will change the configuration of the driver should not be accepted if the audio driver is in the process of either playing or
recording audio samples. The `ioctl()` control functions that fall into this category are:

- `SNDCTL_DSP_SETFRAGMENT`
- `SNDCTL_DSP_SPEED`
- `SNDCTL_DSP_CHANNELS`
- `SNDCTL_DSP_STEREO`
- `SNDCTL_DSP_SAMPSIZE`
- `SNDCTL_DSP_SETFMT`

If an `ioctl()` call is received that is not one of the supported functions, the `errno` should be set to `S_iolib_UNKNOWN_REQUEST` and the `ioctl()` should return `ERROR`.

**Playback**

A play operation starts when the processor sends an audio sample to the driver using the `write()` operation. The steps involved in playing the sample are:

1. The audio format is verified. If no format is selected, then the driver will default to the following:
   - For a 8 bit sample - 8 bit unsigned PCM
   - For a 16 bit samples - 16 bit signed little endian PCM

2. If audio is being played already, then the next available fragment to be filled is found and the audio samples are transferred to the fragment. Filling a fragment always starts at the beginning of the fragment. If there is not a sufficient number of samples to entirely fill the fragment, the remainder of the fragment will be filled with no audio—that is, the absence of audio volume. For example, in the case of 8 bit unsigned PCM, the fragment is set to 0.

3. If the audio hardware is not in the process of playing an audio stream, then the audio hardware is initialized appropriately and audio playing initiated.

4. If there are more audio samples and more fragments to fill, then the next available fragment is filled.

5. If there are more audio samples to play, but there are no available fragments, then the driver must wait for an available fragment. However, if the driver was set up as non-blocking (`SNDCTL_DSP_NONBLOCK`), then the `write()` operation should return indicating the number of bytes that were transferred to audio fragments.
PlaybacK of the audio samples is tightly coupled to audio interrupts. Typically, the audio hardware generates an interrupt when an audio fragment has been played. The audio ISR must work in conjunction with the `write()` operation to fill the recently completed audio fragment. The audio interrupt must also wake up any tasks that were pending on the completion of the audio sample by using `select()`.

**Recording**

The recording process is similar to the playing process. The application calls `read()` with the DSP device file descriptor:

1. Upon receipt of the read operation—(the application calls `read()` with the DSP device file descriptor)—a recording sequence is initiated unless one exists already. The audio format should have been previously set by `ioctl()` with the control function `SNDCTL_DSP_SETFMT`. If none was specified, it will default to the following:
   - For a 8 bit sample - 8 bit unsigned PCM
   - For a 16 bit samples - 16 bit signed little endian PCM
2. The `read()` operation will pend until a complete audio fragment is received. Then the pending `read()` operation will complete and the `read()` operation will return with the number of bytes read. If there was no `read()` pending (the interface was set for non-blocking), then any tasks pending on `select()` will be unblocked.
3. The record operation will continue until either the `ioctl()` routine `SNDCTL_DSP_SYNC` is received or there is no pending read for a fragment.

As with the audio playing sequence, the audio interrupt works closely with the recording operation. Typically, the audio hardware generates an interrupt when an audio fragment as been filled with recorded audio. The audio ISR works in conjunction with the `read()` operation to transfer the recently completed audio fragment to the application. The audio interrupt must also wake up any tasks that were pending on a filled audio fragment.

### 11.2.3 Mixer Device

The mixer device must support the following operations:

- initialization
- control
Initialization

The mixer device is initialized when the audio device is created. Typically, the only initialization required for the mixer device is the assignment of each of the controls (that is, volume, treble, bass, and so on) to a default value.

Control

The control functions of the mixer device are accessed through the `ioctl` interface. The `ioctl()` routines that the mixer device can support are listed in 3.11 Audio API, p.82. Support for these functions varies between different hardware configurations, for example, some hardware configurations will have treble controls while others will not. If a control is not available for a specific device, then an attempt to set that control should return `ERROR`. Support for the following `ioctl()` control functions is mandatory:

- `SOUND_MIXER_RECSRC`
- `SOUND_MIXER_DEVMASK`
- `SOUND_MIXER_STEREODEVS`

They are used to convey the capabilities of the mixer to the application, so that the application can find out which of the other `ioctl()` routines are supported.

When a control is set or read, there is usually a difference between the values that the application will use and what the hardware expects. The application uses values in the range 0 through 100 for the controls. 0 is minimum and 100 is maximum. It is the responsibility of the mixer to normalize these values to the values that the hardware requires.

If an `ioctl()` control function is received that is not a valid command or a feature that is not supported by the mixer, the `errno` should be set to `S_ioLib_UNKNOWN_REQUEST` and the `ioctl()` should return `ERROR`. 
12.1 Introduction

A WindML font driver provides a common and portable abstraction layer on top of a font engine's proprietary API. This layer allows an application to use many different font engines through a common font API. This architecture is shown in Figure 12-1.

![Figure 12-1: Font Driver DDK](image)

In many respects, a font driver is like any other WindML driver, but unlike the other drivers, the font driver does not implement or use any routines beneath the UGI graphics driver interface. It also does not directly access the hardware. It
accesses the graphics driver routines and hardware indirectly through the `ugl_ugi_driver` structure. This architecture allows the font driver and font engine to be hardware independent and run on any graphics device.

## 12.2 Writing a Font Driver

The public API through which the 2D font API accesses the font driver is defined by the `ugl_font_driver` structure. This structure is defined in the file `target/h/ugl/uglfont.h`:

```c
typedef struct ugl_font_driver
{
    UGL_FONT * (*fontCreate) (struct ugl_font_driver * pFontDriver,
                               struct ugl_font_def * pFontDefinition);
    UGL_STATUS (*fontDestroy) (struct ugl_font * pFont);
    UGL_STATUS (*fontDriverDestroy) (struct ugl_font_driver * pFontDriver);
    UGL_SEARCH_ID (*fontFindFirst) (struct ugl_font_driver * pFontDriver,
                                     struct ugl_font_desc * pFontDesc);
    UGL_STATUS (*fontFindNext) (struct ugl_font_driver * pFontDriver,
                                 struct ugl_font_desc * pFontDesc,
                                 UGL_SEARCH_ID searchId);
    UGL_STATUS (*fontFindClose) (struct ugl_font_driver * pFontDriver,
                                  UGL_SEARCH_ID searchId);
    UGL_STATUS (*fontDriverInfo) (struct ugl_font_driver * pFontDriver,
                                   UGL_INFO_REQ infoRequest, void * pInfo);
    UGL_STATUS (*fontMetricsGet) (struct ugl_font * pFont,
                                   struct ugl_font_metrics * pFontMetrics);
    UGL_STATUS (*setTextDraw) (struct ugl_gc * pGc, UGL_POS x, UGL_POS y,
                                UGL_SIZE length, const UGL_CHAR * text);
    UGL_STATUS (*setTextDrawW) (struct ugl_gc * pGc, UGL_POS x, UGL_POS y,
                                UGL_SIZE length, const UGL_WCHAR * text);
    UGL_STATUS (*setTextSizeGet) (struct ugl_font * pFont, UGL_SIZE * width,
                                  UGL_SIZE * height, UGL_SIZE length,
                                  const UGL_CHAR * text);
    UGL_STATUS (*setTextSizeGetW) (struct ugl_font * pFont, UGL_SIZE * width,
                                   UGL_SIZE * height, UGL_SIZE length,
                                   const UGL_WCHAR * text);
    UGL_STATUS (*fontSizeSet) (struct ugl_font * pFont, UGL_SIZE pixelSize);
    UGL_STATUS (*fontWeightSet) (struct ugl_font * pFont, UGL_SIZE weight);
    UGL_STATUS (*fontRotationAngleSet) (struct ugl_font * pFont,
                                         UGL_ORD angle);
    UGL_UGI_DRIVER * pDriver;   /* A font driver is "connected" to a
                                 particular graphics driver */
    void * pFontDriverExtension; /* Normally NULL unless the font driver
                                    provides a driver extension */
}UGL_FONT_DRIVER;
```
In addition to the members of the `ugl_font_driver` structure, each font driver must have a driver creation routine.

All but two of the 2D Font API routines map down onto specific driver routines; `uglFontFind()` and `uglFontFindString()` are implemented at the 2D layer and are not part of the font driver. These two routines are use the routines available in the font driver.

**Source Code Directory Structure**

The source code for WindML font drivers is in the following location:

`target/src/ugl/driver/font`

This directory contains subdirectories for each font driver. Typically, programmers can simply add new subdirectories as needed when a new font driver is created.

The header file directory is:

`target/h/ugl/driver/font`

As with the source directory, you can add files or subdirectories as needed for your font driver.

### 12.2.1 The Font Driver Creation Routine

The font driver creation routine is responsible for performing any necessary initialization of the font engine and for allocating and initializing the members of the `ugl_font_driver` structure. The font driver creation routine is usually called during `uglInitialize()` but also may also be called by the application.

All WindML font driver creation routines should follow the prototype shown below:

```c
UGL_FONT_DRIVER * xxxFontDriverCreate ( UGL_UGI_DRIVER * pDriver )
```

`pDriver` is a pointer to the graphics driver that will be used by the font driver.

The return value of the creation routine is a pointer to an `UGL_FONT_DRIVER` structure. This structure must be allocated by the creation routine. You can define your own font driver structure to contain additional driver specific data if needed. The first member of the your font driver structure must be an `ugl_font_driver` structure.
The following code is an example of a creation routine from the WindML BMF font driver. This code shows the creation routine and the private driver structure used by the driver. Note that the first member of the private driver structure is an UGL_FONT_DRIVER.

typedef struct ugl_bmf_font_driver
{
    UGL_FONT_DRIVER header;
    UGL_BMF_FONT * pFirstFont;
    UGL_BMF_FONT * pLastFont;
    UGL_FONT_DESC * pFontList;
    UGL_SIZE numCachedGlyphs;
    UGL_SIZE glyphCacheSize;
    UGL_ORD textOrigin;
    UGL_MEM_POOL_ID glyphCachePoolId;
    UGL_LOCK_ID lockId;
    UGL_GLYPH_CACHE_ELEMENT *pFirstCacheElement;
    UGL_GLYPH_CACHE_ELEMENT *pLastCacheElement;
} UGL_BMF_FONT_DRIVER;

UGL_FONT_DRIVER * uglBMFFontDriverCreate ( UGL_UGI_DRIVER * pDriver)
{
    UGL_BMF_FONT_DRIVER * pBMFFontDriver = UGL_NULL;
    UGL_FONT_DRIVER * pFontDriver = UGL_NULL;
    UGL_SIZE numFonts;
    
    /* Verify that fonts are available */
    if (uglBMFFontData == UGL_NULL)
    {
        return(UGL_NULL);
    }

    /* Count the number of available fonts */
    for(numFonts = 0; UGL_NULL != uglBMFFontData[numFonts];
        numFonts++);

    /* If there are no fonts available, return */
    if (numFonts == 0)
    {
        return(UGL_NULL);
    }

    pBMFFontDriver = (UGL_BMF_FONT_DRIVER *)UGL_CALLOC(1,
        sizeof(UGL_BMF_FONT_DRIVER));

    if (UGL_NULL != pBMFFontDriver)
    {
        pFontDriver = (UGL_FONT_DRIVER *)pBMFFontDriver;
        pFontDriver->fontCreate = uglBMFFontCreate;
        pFontDriver->fontDestroy = uglBMFFontDestroy;
        pFontDriver->fontDriverDestroy = uglBMFFontDriverDestroy;
        pFontDriver->fontFindFirst = uglBMFFontFindFirst;
        pFontDriver->fontFindNext = uglBMFFontFindNext;
        pFontDriver->fontFindClose = uglBMFFontFindClose;
    }
pFontDriver->fontDriverInfo = uglBMFFontDriverInfo;
pFontDriver->fontMetricsGet = uglBMFFontMetricsGet;
pFontDriver->fontRotationAngleSet = uglBMFFontRotationAngleSet;
pFontDriver->fontSizeSet = uglBMFFontSizeSet;
pFontDriver->fontWeightSet = uglBMFFontWeightSet;
pFontDriver->textDraw = uglBMFTextDraw;
pFontDriver->textSizeGet = uglBMFTextSizeGet;
#endif /* INCLUDE_UGL_BMF_UNICODE */
pFontDriver->textDrawW = uglBMFTextDrawW;
pFontDriver->textSizeGetW = uglBMFTextSizeGetW;
#endif /* INCLUDE_UGL_BMF_UNICODE */

12.2.2 The Font Driver Destroy Routine

The font driver destroy routine is responsible for deinitializing the font engine and freeing any resources used by the engine or driver, including the ugl_font_driver structure. This routine is also responsible for freeing any fonts that are associated with the font engine and have not yet been freed by the application. This routine is typically called during uglDeinitialize().

12.2.3 Querying Font Driver Information

The xxxFontDriverInfo() routine is a multi-purpose routine that can be used as an ioctl()-style mechanism to extend font driver capability.

In the current release, this routine is only required to support the UGL_FONT_TEXT_ORIGIN command. A font driver may support additional commands that are specific to that particular font engine. The xxxFontDriverInfo() routine is the recommended mechanism for allowing the application to extend font driver configuration and capabilities.
This is the `uglBMFFontDriverInfo()` routine:

```c
UGL_LOCAL UGL_STATUS uglBMFFontDriverInfo
{
    UGL_FONT_DRIVER * pFontDriver,
    UGL_INFO_REQ infoRequest,
    void * pInfo
}

UGL_STATUS status = UGL_STATUS_ERROR;

switch (infoRequest)
...

case UGL_FONT_TEXT_ORIGIN:
    {
        UGL_BMF_FONT_DRIVER * pBMFFontDriver =
            (UGL_BMF_FONT_DRIVER *)pFontDriver;

        if(UGL_NULL != pInfo)
        {
            if(*(UGL_ORD *)pInfo == UGL_FONT_TEXT_UPPER_LEFT ||
                *(UGL_ORD *)pInfo == UGL_FONT_TEXT_BASELINE)
                {
                    pBMFFontDriver->textOrigin = *(UGL_ORD *)pInfo;
                    status = UGL_STATUS_OK;
                }
            else
                status = UGL_STATUS_ERROR;
        }
        else
            status = UGL_STATUS_ERROR;
        break;
    }
    default:
        return (UGL_STATUS_ERROR);
}

return(status);
```

12.2.4 Creating a Font

The `xxxFontCreate()` driver routine must ensure that a font can be used by the application for rendering text. This may involve font engine initialization, reading a font from ROM or a file system, and so on.

This routine is not responsible for performing nearest matching. This routine expects that the font attributes specified in the `UGL_FONT_DEF` structure match
those of a font that is available and can be rendered by the font engine. If the font is successfully created, \texttt{xxxFontCreate()} returns a pointer to a \texttt{UGL\_FONT}. If the information in the \texttt{UGL\_FONT\_DEF} structure does not match an available font, NULL should be returned.

You can define your own structure, based on \texttt{UGL\_FONT}, to contain any additional font specific data or information. The \texttt{UGL\_FONT} structure must be the first member in your structure.

The \texttt{xxxFontDestroy()} routine is responsible for freeing any resources allocated by \texttt{xxxFontCreate()} routine. However, note that any fonts not destroyed by the application must be destroyed when the font driver is destroyed.

12.2.5 Enumerating the Fonts on the System

The \texttt{xxxFontFindFirst()}, \texttt{xxxFontFindNext()}, and \texttt{xxxFontFindClose()} routines allow an application to enumerate all fonts that are available in the system. The application must call \texttt{uglFontFindFirst()} (\texttt{xxxFontFindFirst()}) before calling the other two routines.

\texttt{xxxFontFindFirst()} creates a \texttt{UGL\_SEARCH\_ID} which it uses to keep track of the enumeration process. \texttt{UGL\_SEARCH\_ID} is a void pointer that can point to any valid structure allocated by the driver. The \texttt{xxxFontFindNext()} routine is called repeatedly until all available fonts have been enumerated. When the last font is enumerated, \texttt{UGL\_STATUS\_FINISHED} is returned to indicate to the application that all fonts have been enumerated. \texttt{uglFontFindClose()} (\texttt{xxxFontFindClose()}) is then called to free any resources allocated in \texttt{UGL\_SEARCH\_ID}.

12.2.6 Text Rendering

The application renders text by calling the \texttt{uglTextDraw()} (\texttt{xxxTextDraw()}) and \texttt{uglTextDrawW()} (\texttt{xxxTextDrawW()}) routines. The \texttt{uglTextDrawW()} routine is used for double byte text.

The WindML Font API does not enforce any encoding limitations. The API simply passes the encoding in the text string and the font engine is responsible for any translation or interpretation of the encoding.

The application obtains the width and height of a text string using the \texttt{uglTextSizeGet()} and \texttt{uglTextSizeGetW()} routines. The width and height parameters are pointers that may optionally point to NULL if the information is not needed.

\id{12}
The font driver can use any of the routines within the WindML UGI interface to render fonts to the frame buffer. However, no part of the font driver or font engine should use any part of the graphics driver below the WindML UGI interface. This architecture allows font drivers to be graphics device independent so they can be used on any type of graphics device.

This is a code segment from the `uglBMFTextDraw()` routine:

```c
UGL_LOCAL UGL_STATUS uglBMFTextDraw
    (UGL_GC * pGc,
    UGL_POS x,
    UGL_POS y,
    UGL_SIZE length,
    const UGL_CHAR * text
    )
{
    UGL_UGI_DRIVER * pDriver = pGc->pDriver;
    UGL_BMF_FONT * pBMFFont = (UGL_BMF_FONT *)pGc->pFont;
    UGL_BMF_FONT_DRIVER * pBMFFontDriver =
        (UGL_BMF_FONT_DRIVER *)pBMFFont->header.pFontDriver;
    UGL_FONT_DRIVER * pFontDriver =
        (UGL_FONT_DRIVER *)pBMFFont->header.pFontDriver;
    ...

    /* Draw the text background */
    if (pGc->backgroundColor != UGL_COLOR_TRANSPARENT)
    {
        UGL_SIZE width, height;
        UGL_RECT rect;
        UGL_POS tmpY;
        ....

        (*pFontDriver->textSizeGet)(pGc->pFont, &width, &height, length, text);
        ....

    /* Blit glyph to the screen */
    if(pBMFFontDriver->textOrigin == UGL_FONT_TEXT_UPPER_LEFT)
    {
        point.x = x;
        point.y = y + maxAscent - (UGL_POS)pCacheElement->ascent;
        (*pDriver->monoBitmapBlt)(pDriver, pCacheElement->bitmapID,
                        &pCacheElement->bitmapRect,
                        UGL_DEFAULT_ID, &point);
        x += pCacheElement->width;
    }
    else if(pBMFFontDriver->textOrigin == UGL_FONT_TEXT_BASELINE)
    {
        point.x = x;
        point.y = y - (UGL_POS)pCacheElement->ascent;
        (*pDriver->monoBitmapBlt)(pDriver, pCacheElement->bitmapID,
                        &pCacheElement->bitmapRect,
                        UGL_DEFAULT_ID, &point);
    }
```

208
&pCacheElement->bitmapRect,
UGL_DEFAULT_ID, &point);  

x += pCacheElement->width;
}
else
{
    status = UGL_STATUS_ERROR;
    break;
}

...  

The xxxFontMetricsGet() routine is used to obtain detailed information about a specific font. For more information, refer to the UGL_FONT_METRICS structure defined in target/h/ugl/font.h.

The xxxFontRotationAngleSet(), xxxFontSizeSet(), and xxxFontWeightSet() are optional routines. If they are not supported by the font engine, they return UGL_STATUS_ERROR.

### 12.3 Packaging a Font Driver - For Third Party Developers

To add an font driver to WindML, perform the following steps:

1. Create the source and header subdirectories and files in the following locations:
   ```
   target/src/ugl/driver/font
   target/h/ugl/driver/font
   ```
2. The header files must provide #define statements for the initialization routine. The following example is from the WindML BMF font driver in target/h/ugl/driver/font/udbmffnt.h:

```c
/* Font definitions */
#define UGL_FONT_DRIVER_NAME "BMF Font"
#define UGL_FONT_DRIVER_CREATE uglBMFFontDriverCreate
```
13.1 Introduction

The Extension API allows you to add functionality to graphics drivers using extensions. The extension mechanism attaches itself to the graphics driver, providing extra functionality that the 2D API in the SDK can utilize.

The following set of routines are used to manage an extension at the driver level:

<table>
<thead>
<tr>
<th>Table 13-1 Driver Level Extension API</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Routine</strong></td>
</tr>
<tr>
<td>uglGenExtAdd()</td>
</tr>
<tr>
<td>uglGenExtDelete()</td>
</tr>
<tr>
<td>uglGenExtAllDelete()</td>
</tr>
</tbody>
</table>

These are general library routines that any graphics driver uses to manage extensions, and are not typically called by an application.

13.2 Adding an Extension

An extension is added at the request of the application, see 3.13.1 Querying and Attaching an Extension, p.91.
The application invokes `uglInfo()` with the UGL_EXT_INFO_REQ command and the UGL_EXT_INFO structure.

The name of the extension to be added is specified by the extension name passed to `uglInfo()`. In response, the driver does the following:

1. The driver allocates the extension control structure. This is the structure that is passed back to the routine that invoked `uglInfo()`, using the `pExt` field of the UGL_EXT_INFO data structure.

   The extension control structure differs from one extension to another, depending on the extension type. However, all extension control structures contain a common header—UGL_EXT_HEADER—which is the same for all extensions:

   ```c
   typedef struct ugl_ext_header
   {
     char * name; /* name of extension */
     int version; /* version of extension */
     struct ugl_ext_header * nextExt;/* next extension */
     UGL_STATUS (* extDelete) (struct ugl_ugi_driver * pDriver,
                                struct ugl_ext_header * extNode);
     struct ugl_ugi_driver * pDriver; /* associated driver */
   } UGL_EXT_HEADER;
   ```

2. The driver then calls `uglGenExtAdd()` to initialize the extension control data structure and add the extension. `uglGenExtAdd()` does not initialize the `extDelete` function pointer in the UGL_EXT_HEADER structure. Each extension is required to initialize this field explicitly.

3. The driver then initializes the `extDelete` function pointer, and each of the remaining functions of the extension API.

4. The driver sets the version field of the UGL_EXT_INFO structure, and sets the `pExt` field of the UGL_EXT_INFO structure to point to the extension control structure.

5. If the previous steps were successful, the driver returns UGL_STATUS_OK. If the extension was not present, or a problem was found, the driver returns UGL_STATUS_ERROR.
13.3 Deleting an Extension

A graphics driver may remove an extension from the system using the `uglGenExtDelete()` and `uglGenExtAllDelete()` routines.

`uglGenExtDelete()` removes a single extension. `uglGenExtAllDelete()` removes all extensions attached to a driver.

13.4 Example - Using the Video Extension

The video extension supplied with WindML allows live video from a camera or VCR to be displayed on a monitor along with graphics from a WindML application. A video extension is typically the implementation of video functions for a graphics device.

This chapter describes the process of implementing a video extension to a graphics driver. Since the majority of the video implementations involve the use of overlay surfaces, this example uses video hardware that makes use of overlay surfaces.

To provide system scaling, the portions of the graphics driver that deal with the video extension should provide conditional compilation. Then the video may be selectively removed if the WindML application does not want the extra code associated with the video extension.

The extension developer wrote a wrapper, `uglVideoInfo()` around the standard `uglInfo()` routine. The application calls this routine specifying the video extension name.

The following section of code from the `uglVideoInfo()` function for the IGS Technologies, Inc. graphics device illustrates the handling of this operation:

```c
case UGL_EXT_INFO_REQ:
    {
        #ifdef INCLUDE_UGL_VIDEO
            if (strcmp(extInfo->name, UGL_EXT_VIDEO_NAME) == 0)
            {
                extInfo->pExt = (void *)uglIGSVedioInit (pDriver,
                                                        &version,
                                                        fbFormat,
                                                        pixelSize);

                extInfo->version = version;
            }
        else
```
In this example implementation, the function `uglIGSVideoInit()` is called to handle the initialization of the video support. The frame buffer format and the size of a pixel, along with a pointer to the `ugl_ugi_driver` data structure (pDriver) are passed to the `uglIGSVideoInit()` function.

The steps required for adding the video extension are:

1. Allocate a video extension data structure `UGL_VIDEO_EXT`. Typically, this data structure will be augmented to contain private information as needed for the specific video hardware.
2. Add the extension to the list of extensions by calling `uglGenExtAdd()`.
3. Initialize the `UGL_VIDEO_EXT` data structure by filling in the extension `extDelete()` function, the video signature, and the video extension implementation functions. The video signature field is used by the video extension at the users API level to verify that the user has passed a valid video identifier (`UGL_VIDEO_ID`).
4. Initialize any private data which should include verification that the video hardware is present.

### 13.5 Implementing the Video Extension Driver API

The device driver that supports the video extension must implement the entire video extension API. This API is defined by the following structure:

```c
typedef struct ugl_video_ext
{
    UGL_EXT_HEADER header; /* must be first field */
    int signature; /* must be set */

    /* implementation functions */
    UGL_STATUS (* adapterInfo) (struct ugl_video_ext * pVideoCtrl,
                                UGL_ORD port,
                                int videoReq,
                                void * pInfo);
    UGL_STATUS (* portLock) (struct ugl_video_ext * pVideoCtrl,
                              UGL_ORD port);
    UGL_STATUS (* streamPut) (struct ugl_video_ext * pVideoCtrl,
                              UGL_ORD port);
    UGL_STATUS (* streamGet) (struct ugl_video_ext * pVideoCtrl,
                              UGL_ORD port);
    UGL_STATUS (* streamQuery) (struct ugl_video_ext * pVideoCtrl,
                                UGL_ORD port);
    UGL_STATUS (* streamCB) (struct ugl_video_ext * pVideoCtrl,
                              UGL_ORD port);
    UGL_STATUS (* destroy) (struct ugl_video_ext * pVideoCtrl);

    /* private data */
    /* private structures */
}
```

214
UGL_GC_ID gc,
UGL_ORD port,
UGL_RECT * pSrcRect);
UGL_STATUS (* stillPut) (struct ugl_video_ext * pVideoCtrl,
UGL_GC_ID gc,
UGL ORD port,
UGL_RECT * pSrcRect);
UGL_STATUS (* stop) (struct ugl_video_ext * pVideoCtrl,
UGL_ORD port);
UGL_STATUS (* portUnlock) (struct ugl_video_ext * pVideoCtrl,
UGL_ORD port);
}
UGL_VIDEO_EXT;

/* adapterInfo */

This function implements a calling sequence that is similar to the standard
ioctl() operation. The control functions that must be supported are:

UGL_VIDEO_CONFIG_SET
Sets the video capture mode.

The mode is specified by the UGL_VIDEO_CONFIG data structure. The
mode defines the type of input signal, as either PAL or NTSC, and whether
the signal is interlaced. Also specified by the configuration is the overlay
surface to be used to display the video image. When this control function
is invoked, the extension should set the mode. It should also set the video
processor to a known default state including setting the color attributes to
known values.

UGL_VIDEO_CONFIG_GET
Returns the configuration that was established with the
UGL_VIDEO_CONFIG_SET function.

UGL_VIDEO_ATTRIB_SET
Modifies the attributes of the video—hue, brightness, saturation, and so
on.

UGL_VIDEO_ATTRIB_GET
Obtains the current setting for the video attributes.

UGL_VIDEO_AVAIL_GET
Obtains the number of video ports that are available.

UGL_VIDEO_PORT_INFO
Obtains the capabilities of a video port. The capabilities that are returned
by this command are provided in the UGL_VIDEO_INFO data structure
and contain the video modes available, the current mode, and the current
size of the video image and the maximum size possible.
UGL_VIDEO_PRESENT
  Returns identification if there is video present.

(* portLock)
  When an application wants to use the video extension, it will typically lock the
  video port that it uses, to prevent any other use of the video port. Any attempt
  to use the video extension for a port that was locked results in the rejection of
  the request, if the requestor was not the application that locked the video port.

(* portUnlock)
  Removes a lock that was previously established.

(* streamPut)
  Places live video onto the overlay surface that was designated as the recipient
  of the video when the video port was configured. The function takes as a
  parameter, a source rectangle that defines the location within the video image
  that should be placed within the video image. This source rectangle,
  depending on the hardware, should fill the entire area of the overlay surface.
  Therefore, by changing the size of the source rectangle in relationship to the
  overlay size, effects such as zooming in and out may be achieved.

(* stillPut)
  This function is the same as the (* streamPut) function except that it will
  capture a single frame of video.
14.1 OS Abstraction Layer

WindML provides an abstraction layer for fundamental OS functionality so that it can be used on different operating systems with minimal modifications.

Both the SDK and the DDK layers of WindML use the OS abstraction layer for all OS system calls. If you are developing a new feature for WindML—such as writing a new device driver—you should use the OS abstraction layer to make all OS system calls.

If WindML is ported to a new operating system, the OS abstraction layer needs to be re-implemented to map WindML’s OS requirements to the corresponding OS functionality.

This OS API is provided to make WindML portable and is not provided as an application service. An WindML application can use the OS API, but there is no requirement that it does.

The following tables show the OS abstraction layer routines in different functional areas.
Memory Management Services, see Table 14-1.

**Table 14-1 Memory Management**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uglOSMemPoolCreate()</code></td>
<td>Creates a private memory pool from which memory can be allocated and freed.</td>
</tr>
<tr>
<td><code>uglOSMemPoolDestroy()</code></td>
<td>Destroys a memory pool that was created with <code>uglOSMemPoolCreate()</code>.</td>
</tr>
<tr>
<td><code>uglOSMemAlloc()</code></td>
<td>Allocates a block of memory from a memory pool.</td>
</tr>
<tr>
<td><code>uglOSMemCalloc()</code></td>
<td>Allocates a block of memory from a memory pool and clear its contents to zero.</td>
</tr>
<tr>
<td><code>uglOSMemRealloc()</code></td>
<td>Reallocates a block of memory.</td>
</tr>
<tr>
<td><code>uglOSMemFree()</code></td>
<td>Frees a block of memory in a memory pool.</td>
</tr>
</tbody>
</table>

Task/Process Services, see Table 14-2.

**Table 14-2 Task/Process Services**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uglOSTaskCreate()</code></td>
<td>Creates a task.</td>
</tr>
<tr>
<td><code>uglOSTaskDelete()</code></td>
<td>Deletes a task.</td>
</tr>
<tr>
<td><code>uglOSTaskSuspend()</code></td>
<td>Suspends a task.</td>
</tr>
<tr>
<td><code>uglOSTaskActivate()</code></td>
<td>Activates a task.</td>
</tr>
<tr>
<td><code>uglOSGetTaskIdSelf()</code></td>
<td>Gets the task ID of running task.</td>
</tr>
</tbody>
</table>

Timer Services, see Table 14-3.

**Table 14-3 Timer Services**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uglOSTaskDelay()</code></td>
<td>Delays a task.</td>
</tr>
<tr>
<td><code>uglOSTimeStamp()</code></td>
<td>Obtains the current time stamp.</td>
</tr>
</tbody>
</table>
Message Queue Services, see Table 14-4.

**Table 14-4  Message Queue Services**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglOSMsgQCreate()</td>
<td>Creates a message queue.</td>
</tr>
<tr>
<td>uglOSMsgQDelete()</td>
<td>Deletes a message queue.</td>
</tr>
<tr>
<td>uglOSMsgQPost()</td>
<td>Posts a message to a message queue.</td>
</tr>
<tr>
<td>uglOSMsgQGet()</td>
<td>Gets a message to a message queue.</td>
</tr>
</tbody>
</table>

Error Service, see Table 14-5.

**Table 14-5  Error Service**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglLog()</td>
<td>Logs WindML errors and aborts when fatal.</td>
</tr>
</tbody>
</table>

Semaphore/Lock Services, see Table 14-6.

**Table 14-6  Semaphore/Lock Services**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglOSLockCreate()</td>
<td>Creates a WindML lock.</td>
</tr>
<tr>
<td>uglOSLockDelete()</td>
<td>Deletes a WindML lock.</td>
</tr>
<tr>
<td>uglOSLock()</td>
<td>Takes a WindML lock.</td>
</tr>
<tr>
<td>uglOSUnlock()</td>
<td>Gives a WindML lock.</td>
</tr>
</tbody>
</table>

Interrupt Services, see Table 14-7.

**Table 14-7  Interrupt Services**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglOSIntLock()</td>
<td>Locks interrupts.</td>
</tr>
<tr>
<td>uglOSIntUnlock()</td>
<td>Unlocks interrupts.</td>
</tr>
<tr>
<td>uglOSIntAttach()</td>
<td>Attaches an ISR to an interrupt.</td>
</tr>
</tbody>
</table>
UglOSIntDetach() Detaches an ISR from an interrupt.
UglOSIntOff() Disables interrupt from an interrupt source.
UglOSIntOn() Enables interrupt from an interrupt source.

List Handling, see Table 14-8.

Table 14-8

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uglListCreate()</td>
<td>Creates a linked list.</td>
</tr>
<tr>
<td>uglListDestroy()</td>
<td>Destroys a linked list.</td>
</tr>
<tr>
<td>uglListAdd()</td>
<td>Adds a node to a linked list.</td>
</tr>
<tr>
<td>uglListInsert()</td>
<td>Inserts a node into a linked list.</td>
</tr>
<tr>
<td>uglListRemove()</td>
<td>Removes a node from a linked list.</td>
</tr>
<tr>
<td>uglListFirst()</td>
<td>Gets the first node in a linked list.</td>
</tr>
<tr>
<td>uglListLast()</td>
<td>Gets the last node in a linked list.</td>
</tr>
<tr>
<td>uglListNth()</td>
<td>Gets the Nth node in a linked list.</td>
</tr>
<tr>
<td>uglListNext()</td>
<td>Gets the next node in a linked list.</td>
</tr>
<tr>
<td>uglListPrev()</td>
<td>Gets the previous node in a linked list.</td>
</tr>
<tr>
<td>uglListCount()</td>
<td>Gets the number of nodes in a linked list.</td>
</tr>
</tbody>
</table>
15.1 Introduction

This chapter describes the UGL source tree layout and suggests naming conventions, for functions, macros, and so on.

If you are an ISV extending WindML with new output and input drivers, font packages, or developer tools, you should install new components into specific areas of the WindML source tree. This will provide developer’s with a uniform experience when working with WindML technology, as well as preventing additions or enhancements to the WindML source tree interfering with one another.

15.2 Directory Contents

This section describes the contents of the directories contained in the WindML source tree.

WindML has been structured to isolate generic code from any architecture-specific or operating-system-specific branches. This maximizes portability and also provides a uniform directory structure for all developers to follow. If you have access to cross-compilers of various architectures, they can each be used to compile WindML for different architectures.
This directory is the top-level source directory for WindML. The WindML source code has been modularized into multiple directories to better organize the various source components. The src directory includes the following five subdirectories:

**src/ugl/2d Directory**

The files in this directory comprise the WindML 2-D API layer. These libraries and subroutines support application programs and validation suites that run on WindML. For detailed information about the entire 2-D API, see 3. The SDK.

⚠️ **CAUTION:** As part of the developer license for WindML, you are not allowed to modify any of the WindML 2-D API routines. Any problem reports or enhancement requests related to the WindML 2-D source code can be submitted to the Wind River Customer Services Department at support@windriver.com.

You can, of course, create and/or modify driver code to suit your needs, and may add your own APIs.

**src/ugl/config**

This directory contains configuration files for WindML.

**src/ugl/demo**

This directory contains some example programs using WindML.

**src/ugl/driver**

This directory contains driver code for the hardware platforms upon which WindML runs. This includes hardware independent generic source code for WindML drivers as well as hardware dependent drivers.

The *graphics* directory contains the source code for drivers that output data to a display.

The *keyboard* and *pointer* directories contain source code for drivers that read data from input devices for WindML.

The *font* directory contains the source code for font drivers which implement the 2-D layer font related calls.
The files in this directory contain implementations of the WindML OS API for various platforms. WindML takes advantage of optimized synchronization primitives provided by the underlying OS. If you are implementing a new OS port, those aspects of the multi-threading functionality that are OS-dependent are abstracted as part of an API defined in the file `target/h/ugl/uglos.h`.

The following implementations are provided with this release:

- `ugl/src/os/vxworks` (implementation for Wind Rivers VxWorks RTOS)
- `ugl/src/os/simnt` (implementation for the VxWorks simulator running under Microsoft Windows NT OS)
- `ugl/src/os/simsolaris` (implementation for VxWorks simulator running under Suns Solaris OS)

If you want to provide support for a new OS, the generic directory provides a good starting point for your implementation.

`src/ugl/win` contains the code for the windowing API layer of WindML.

`src/ugl/fonts` contains the code for the fonts.

`src/ugl/event` contains the code for WindML’s event API.

`src/ugl/bspExt` contains the code for extensions to the BSP required by WindML.

`src/ugl/audio` contains the code for the audio extension to WindML.

`src/ugl/media` contains the code for WindML’s Media layer API.
General WindML definitions are located in header files in this directory.

- *ugl.h* (all generic WindML definitions)
- *uglclr.h* (definitions related to the WindML color management API)
- *uglfont.h* (definitions of the WindML font API)
- *uglinput.h* (definitions of the WindML input driver API)
- *uglos.h* (any OS-specific definitions)
- *ugltypes.h* (definitions of all WindML data types)
- *uglucode.h* (Unicode definitions for WindML)
- *uglgeom.h* (macros and definitions for clipping routines)
- *uglmem.h* (definitions for memory management)
- *uglpage.h* (definitions for double buffering)
- *uglugi.h* (definitions the output driver API — Universal Graphics Interface)
- *ugllog.h* (definitions for UGLs error logging)
- *uglevent.h* (WindML Event, Event Handler, and Event Service definitions)
- *uglmode.h* (definitions for the graphics mode settings)
- *ugildib.h* (definitions for device independent bitmaps (DIBs))
- *uglinfo.h* (definitions for WindML driver information retrieval API)
- *uglmedia.h* (definitions for UGLs Media layer)
- *ugllist.h* (definitions for WindML linked list support)
- *uglwin.h* (definitions for WindML windowing support)

The following subdirectories are also part of the *h/ugl* directory:

**h/ugl/config**

This directory contains header files which configure the WindML library for particular hardware, OS, and so on.

**h/ugl/driver**

WindML driver-specific definitions are located in the following sub-directory:
h/ugl/driver/keyboard

h/ugl/driver/pointer

The files in this directory comprise the input driver-specific definitions. If you are developing a new WindML input driver, you should place all necessary definitions in a header file in the appropriate directory.

Examples of files located in these directories are:

uglms.h (Microsoft mouse driver definitions)
uglpckbd.h (PC 8042 keyboard driver definitions)
uglps2.h (PS/2 mouse driver definitions)
uglasbt.h (StrongARM Assabet touchscreen definitions)

h/ugl/driver/graphics

This directory contains the graphics driver specific header files. The files are organized by hardware platform in separate sub-directories.

Examples of sub-directories located in this directory are:

chips (Chips and Technologies graphics driver definitions)
arm (StrongARM Assabet graphics driver definitions)
vga (VGA mode-specific graphics driver definitions)

If you are developing a new WindML graphics driver, you should create a new directory (if the driver is for a new manufacturer) and place all your driver header files in there.

h/ugl/driver/ext

This directory is where the driver specific header files for any extensions to WindML should be placed. JPEG and video extension definitions are provided here.

h/ugl/driver/font

This directory is where particular font or font engine header files should be placed. A header for the BMF font format is provided.

h/ugl/driver/audio

This directory is where particular audio driver header files should be placed. Header files for Intel Audio Codec 97 and IGS audio hardware are provided.
h/ugl/os

Any WindML definitions for a specific OS environment are located in this directory. To port WindML to another OS environment, developers must implement the routines documented in the h/ugl/uglos.h file. Currently, the VxWorks OS portability layer software is provided as reference code which can easily be extended to other environments: udvxw.h (OS-specific definitions for the VxWorks RTOS platform)

NOTE: It is not necessary to include these header files in your driver. The proper OS-specific header file will be automatically mapped in by including h/ugl/ugltypes.h in your WindML driver source code.

h/ugl/private

The WindML source code provides utility functions to avoid duplication of code. The header files in this directory document the prototypes for these functions.

uglutil.h (function prototype definitions for WindML utility functions)

NOTE: The routines documented in the private directory are considered internal to WindML only. The routines (and their parameters) described in this directory are subject to change without notice.

h/ugl/ext

This directory contains header files for the WindML extension capability and for particular extensions (such as JPEG and video).

h/ugl/bspExt

This directory contains header files for board specific functions (extensions to the Board Support Package (BSP)) that are required by WindML.

h/ugl/audio

This directory contains header files that define the sound API and for processing wave form audio.
15.3 WindML Software Conventions

The WindML distribution contains generic reference drivers, as well as drivers for specific graphics chipsets. These files have been organized hierarchically to facilitate finding and extending existing driver source code. This section describes the file and function management conventions used for driver source code.

⚠️ CAUTION: Follow WindML naming conventions to avoid problems with integration, compatibility, and portability.

15.3.1 Directory Naming Conventions

This section describes the naming conventions used for WindML directories.

If you have a driver that does not fit into the existing source tree, see 4. Creating a New Driver, on how to integrate a new driver into the WindML source tree.

Output Driver Subdirectories

Output driver directory names are based on the chip vendor. Table 15-1 provides examples of output driver directory names.

<table>
<thead>
<tr>
<th>Semiconductor Vendor Name</th>
<th>Directory Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chips &amp; Technology</td>
<td>chips</td>
</tr>
<tr>
<td>Cirrus Logic</td>
<td>cirrus</td>
</tr>
<tr>
<td>Cyrix</td>
<td>cyrix</td>
</tr>
<tr>
<td>Generic implementation</td>
<td>generic</td>
</tr>
<tr>
<td>Motorola</td>
<td>motorola</td>
</tr>
<tr>
<td>S3</td>
<td>s3</td>
</tr>
<tr>
<td>VGA chip driver</td>
<td>vga</td>
</tr>
</tbody>
</table>

For example, if you are developing a chip driver for an S3 graphics processor, create and name the directory s3 (if not already created). Currently, WindML does not define any further directory subdivisions.
Input Driver Subdirectories

All input drivers provided with WindML are located in the directory `ugl/src/driver/input`. WindML does not define any further directory subdivisions for input drivers.

15.3.2 File Naming Conventions

Files in the WindML source tree are named using a lower-case file naming convention. This allows the developer maximum flexibility when installing the developer’s kit on a wide range of computers running Windows and Solaris.

NOTE: When creating new files within the WindML source tree, you should follow the lower-case naming convention.

Table 15-2 summarizes the WindML file naming conventions and provides examples for input and output drivers.

<table>
<thead>
<tr>
<th>WindML Component</th>
<th>Naming Convention</th>
<th>Example(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output Driver</td>
<td><code>udDriver_name</code></td>
<td>Chips and Technologies 65550 driver: <code>ugl/src/driver/output/chips/udct8.c</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>VGA driver: <code>ugl/src/driver/output/vga</code>. Each of the files written for the VGA driver uses the <code>udvga</code> prefix.</td>
</tr>
<tr>
<td>Input Driver</td>
<td><code>udDriver_name</code></td>
<td>WindML PS/2 mouse driver: <code>ugl/src/driver/input/udps2.c</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WindML 8042 keyboard driver: <code>ugl/src/driver/input/ud8042.c</code>.</td>
</tr>
</tbody>
</table>

15.3.3 Routine Naming Conventions

When creating new driver routines to implement or extend graphics devices, use the appropriate prefix guidelines described in Table 15-3.
WindML Source Conventions

It is important that you give the function names for your driver a unique prefix, otherwise they might conflict with other drivers distributed in the WindML source code and they might cause problems during code integration and debugging.

Where possible, it is recommended you encapsulate the entire driver source code with a file name that reflects the chip vendor and driver type (at a minimum). For example, the code for the WindML Chips and Technologies 65555 8-bit output driver is contained in the file ugl/src/driver/output/chips/udct8.c, while the source code for the 16-bit implementation of the Chips and Technologies 65555 driver is contained in the file ugl/src/driver/output/chips/udct16.c. This approach provides unified access to the driver source code and minimizes the potential for conflicts with other files in the source kit.

Table 15-3  Routine Naming Conventions for WindML Source Code

<table>
<thead>
<tr>
<th>WindML Component</th>
<th>Naming Convention</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphics or Output Driver</td>
<td>uglVendorDriverRoutine()</td>
<td>VGA driver uglVgaScreenPixelSet() Chips &amp; Technologies 65555 driver uglCT8BitDevDestroy()</td>
</tr>
<tr>
<td>Input Driver</td>
<td>uglVendorDriverRoutine()</td>
<td>WindML PS/2 mouse driver uglPs2MouseDevCreate() WindML StrongARM 1100 keypad driver uglSaKbdDevCreate()</td>
</tr>
</tbody>
</table>

It is important that you give the function names for your driver a unique prefix, otherwise they might conflict with other drivers distributed in the WindML source code and they might cause problems during code integration and debugging.

Where possible, it is recommended you encapsulate the entire driver source code with a file name that reflects the chip vendor and driver type (at a minimum). For example, the code for the WindML Chips and Technologies 65555 8-bit output driver is contained in the file ugl/src/driver/output/chips/udct8.c, while the source code for the 16-bit implementation of the Chips and Technologies 65555 driver is contained in the file ugl/src/driver/output/chips/udct16.c. This approach provides unified access to the driver source code and minimizes the potential for conflicts with other files in the source kit.

WindML Extension Functions

As part of the license agreement for using WindML, you are not allowed to make any changes to the WindML 2-D software or make changes to the driver layer. However, if you are an independent software vendor (ISV) looking to provide “value added” packages with WindML, use the prefix “uglX” for any custom routines you implement. The “uglX” prefix is designed to isolate custom routines from core WindML source code, and eliminate confusion. Table 15-4 summarizes WindML routine extension conventions and provides examples.
Data and Return Types

To accommodate abstraction and portability from any OS or CPU dependencies, the WindML source code kit provides a set of pre-defined types for use in your code. The following are examples of available WindML types:

**UGL_SIZE**
Use when manipulating data that describes size.

**UGL_POS**
Use when manipulating data that describes the concept of a coordinate or position on the display.

**UGL_LOCAL**
Use when declaring functions with limited scope. **UGL_LOCAL** is equivalent to C **static** functions.

To explore the full set of WindML data and return types, consult the file `h/ugl/ugltypes.h`.

Driver Documentation and Standards

Graphics drivers tend to be complex, having direct references to many portions of the underlying hardware and operating system. You should provide adequate documentation of your implementation to allow other developers to reuse or extend your driver.

In the source code, provide some form of documentation regarding the graphics chip, along with the relevant features or operating modes on which your driver depends. Code maintainers need to determine the basis of the chip, without resorting to a complete library search. Although most manufacturers have their data sheets on the Web, a good one-paragraph summary of the chip can save developers a lot of time. Where possible, include a URL to the appropriate Web location for a particular manufacturer’s data sheet.
15.4 Working with WindML Source Code

WindML source code is written to be portable across a wide variety of architectures and hardware reference platforms. If you want to be able to customize WindML, you must modify two of its major components:

- Recompile the necessary driver source code for your target architecture.
- Utilize supported WindML driver code for your application.

For detailed information about configuring and building WindML, see 5. Configuring UGL.

WindML Drivers And Portability

The WindML output drivers provided with WindML were developed and tested on a specific set of CPU architectures. Along with the driver source code required for rendering operations, each WindML driver must provide all the definitions necessary to properly initialize the graphics chip for the hardware reference board. For example, the PowerPC 821 on-chip LCD controller requires a set of memory-mapped registers to be initialized before you can perform any rendering operations. Similar constraints—such as display (LCD, monitor, touchscreen), data-bus (PCI, VESA), and endian-ness (big or little)—affect the portability of all drivers.

It is recommended that you address portability in your WindML driver by using preprocessing statements. The following example is an excerpt from the IGS5050 WindML output driver, where the PCI initialization code is been properly guarded through the use of architecture-specific defines:

```c
#if CPU == I80486
#elif INCLUD_PCI

if (pciFindDevice(
    PCI_IGS_ID, PCI_IGS_CHIP_ID, 0, &busno, &devno, &funcno) != ERROR)
{
    pciDevConfig(busno, devno, funcno, 0, FB_PHYS_ADDR,
        (PCI_CMD_MASTER_ENABLE|PCI_CMD_IO_ENABLE|PCI_CMD_MEM_ENABLE));
};
#else
# warning "User need to map phys address to virt"
#endif
#endif
```
When the above code segment is compiled for the Intel 80486 CPU architecture, the proper PCI initialization code is included and executed at run-time. However, if you are compiling this driver for an architecture other than the 80486, the following message appears on the display:

```
warning: #warning "User need to map phys address to virt"
```

This means that you need to tweak the driver to compile on the desired architecture. The build output from compiling the IGS 5050 output driver for the StrongARM CPU architecture follows:

```bash
make CPU=ARMSA110
Creating depend.ARM5110gnu
udigs.c:222: warning: #warning "User need to map phys address to virt"
carm -mcpu=strongarm110 -mapcs-32 -ansi -nostdinc -O2 -mno-sched-prolog
-fno-builtins -Wall -I/view/dmm.graphics/vobs/wpwr/target/h
-DCPU=ARMSA110   -c -o
/view/dmm.graphics/vobs/wpwr/target/lib/objARM5110gnuvx/udigs.o udigs.c
udigs.c:222: warning: #warning "User need to map phys address to virt"
udigsmo.inc: In function `igsGetModeInfo':
In file included from udigs.c:70:
udigsmo.inc:893: warning: comparison between signed and unsigned
udigscop.inc: In function `igsTtxt2ScrBlt':
In file included from udigs.c:71:
udigscop.inc:202: warning: comparison between signed and unsigned
udigs.c: In function `uglIG5050DevCreate':
udigs.c:222: warning: unused variable 'p'
udigs.c:118: warning: unused variable 'l'
udigs.c:117: warning: unused variable 'funcno'
udigs.c:116: warning: unused variable 'devno'
udigs.c:115: warning: unused variable 'busno'
```

When porting WindML to a new architecture, you must define and implement a set of macros required for the WindML drivers to perform input and output functions. You need to define the following set of macros for your CPU architecture, see Table 15-5.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uglInByte(address)</code></td>
<td>Read a byte from the specified address.</td>
</tr>
<tr>
<td><code>uglInWord(address)</code></td>
<td>Read a word value from the specified address.</td>
</tr>
<tr>
<td><code>uglOutByte(address, value)</code></td>
<td>Write a byte value to the specified address.</td>
</tr>
<tr>
<td><code>uglOutWord(address, value)</code></td>
<td>Write a word value to the specified address.</td>
</tr>
</tbody>
</table>
Each of these macros is defined on a per-CPU architecture basis in the file h/ugl/os/udvxw.h. The contents of this file for the PowerPC, ARM, and MIPS CPU architectures follows:

```c
/* PPC Support */
#define uglInByte(address) sysInByte(0x80000000 + address)
#define uglInWord(address) (sysInByte(0x80000000 + address) |
                         (sysInByte(0x80000000 + address+1) << 8))
#define uglOutByte(address, value) sysOutByte(0x80000000 + address, \ 
                          (UGL_UINT8)value)
#define uglOutWord(address, value)    
                do { 
                    sysOutByte(0x80000000 + address, (UGL_UINT8)value);\ 
                    sysOutByte(0x80000000 + address+1, (UGL_UINT8)(value>>8)); \ 
                } while (0)
/* Sync memory accesses; */
#undef UGL_MEM_SYNC
#define UGL_MEM_SYNC __asm__(" eieio; sync");
#elif CPU_FAMILY == ARM /* These are dummy placeholders for now. */
#define uglInByte(address) (0)
#define uglInWord(address) (0)
#define uglOutByte(address, value)
#define uglOutWord(address, value)
/* MIPS SUPPORT */
#elif CPU_FAMILY == MIPS /* These are dummy placeholders for now. */
#define uglInByte(address) (0)
#define uglInWord(address) (0)
#define uglOutByte(address, value)
#define uglOutWord(address, value)
#else
    #warning "No architecture support."
#endif
```

As shown above, if you are using WindML on a CPU architecture for which there are existing macros, you do not have to define the input and output routines. However, if you are planning to use WindML on a CPU architecture not currently supported, you must add the required definitions to the list shown above. If you forget to do this, the following warning message is displayed when compiling WindML:

```c
In file included from
/view/dmm.graphics/vobs/wpwr/target/h/ugl/ugltypes.h:50,
    from /view/dmm.graphics/vobs/wpwr/target/h/ugl/uglos.h:31,
    from /view/dmm.graphics/vobs/wpwr/target/h/ugl/ugl.h:26,
    from uglbatch.c:21:
/view/dmm.graphics/vobs/wpwr/target/h/ugl/os/udvxw.h:125: warning: #warning "No architecture support."
```

In file included from
/view/dmm.graphics/vobs/wpwr/target/h/ugl/ugltypes.h:50,
    from /view/dmm.graphics/vobs/wpwr/target/h/ugl/uglos.h:31,
    from /view/dmm.graphics/vobs/wpwr/target/h/ugl/ugl.h:26,
    from uglbitmap.c:32:
"No architecture support."

In file included from
/view/dmm.graphics/vobs/wpwr/target/h/uglify وطني:警告: #warning

"No architecture support."

In file included from
/view/dmm.graphics/vobs/wpwr/target/h/uglify وطني:50,
  from /view/dmm.graphics/vobs/wpwr/target/h/uglify الوطني:31,
  from /view/dmm.graphics/vobs/wpwr/target/h/uglify الوطني:26,
  from uglcm.c:24:
A

Standard Drivers

A.1 Introduction

This appendix lists important information about the drivers that ship with WindML. This information is also available, as a text file, in the same directory as the driver header files for each driver. The information in the text file may contain updated information for the specific driver.

NOTE: Wind River releases new drivers for WindML on an ongoing basis. For a complete list of available drivers, refer to the WindSurf website http://www.windriver.com/csdocs/product/ugl/. This chapter only lists those drivers that were available at release time.

A.2 VGA Driver

Refer to the file target/h/ugl/driver/graphics/vga/uglVga.txt for possible updates to this driver.

This driver provides graphics support for WindML on standard VGA devices using color depths of 1, 4, and 8 bits per pixel. These color depths correspond to VGA modes 11, 12, and 13, respectively.
Modes

This driver supports the following resolutions:

- 640x480 at refresh rate of 60 Hz (VGA mode 11 and 12)
- 320x200 at refresh rate of 60 Hz (VGA mode 13)

VGA mode 11 is a generic 1 bit with a linear frame buffer.

VGA mode 12 is a non-generic 4-bit indexed mode where the frame buffer is represented in 4 separate planes. In this mode, each byte addresses 8 pixels and hardware registers are used to select the affected color planes.

VGA mode 13 is a generic 8-bit indexed mode with a linear frame buffer.

Optional Components

This driver supports the optional JPEG extension. This extension can be added either using the configuration tool or by defining `INCLUDE_UGL_JPEG` within the `uglInit.h` file.

VxWorks BSP Modifications

This driver does not require any modifications to the VxWorks BSP.

Supported Processors

This driver has been tested on the following processors in all supported resolutions and frame buffer formats:

- Intel X86 (80486, Pentium, and PentiumPro)

A.3 ARM Driver

Refer to the file `target/h/ugl/driver/graphics/vga/udsa.txt` for possible updates for this driver.

This driver provides graphics support for WindML on Intel Strong Arm Sa111x compatible devices.
Modes

The following frame buffer formats are supported by this driver:

- 8 bit indexed color
- 16 bit direct (RGB565) color

The driver supports the following resolutions:

- 320x240 (8 and 16 bit modes)

Optional Components

This driver supports the optional JPEG extension. This extension can be added either using the configuration tool or by defining `INCLUDE_UGL_JPEG` within the `uglInit.h` file.

VxWorks BSP Modifications

The Assabet is the only VXWorks platform that this driver currently supports. The following code must be added to the bottom of the `sysPhysMemDesc[]` table in `sysLib.c` in the BSP:

```c
{
    (void *) SA1110_LCD_BASE,
    (void *) SA1110_LCD_BASE,
    PAGE_SIZE,
    VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE |
    VM_STATE_MASK_CACHEABLE,
    VM_STATE_VALID | VM_STATE_WRITABLE | VM_STATE_CACHEABLE_NOT
},
{
    (void *) SA1110_MCP_BASE,
    (void *) SA1110_MCP_BASE,
    PAGE_SIZE,
    VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE |
    VM_STATE_MASK_CACHEABLE,
    VM_STATE_VALID | VM_STATE_WRITABLE | VM_STATE_CACHEABLE_NOT
}
```

After these modifications are made then VxWorks must be rebuilt.

Supported Processors

This driver has been tested on the Intel Strong Arm Sa111x (Assabet) processor in all supported resolutions and frame buffer formats.
A.4 Chips and Technologies Driver

Refer to the target/h/ugl/driver/graphics/chips/udchips.txt file for possible updates for this driver.

This driver provides graphics support for WindML on Chips and Technologies graphics processors. It supports the 65550, 65554, 65555 and 69000 devices within the Chips and Technology family of graphics devices.

Modes

The following frame buffer formats are supported by this driver:

- 8 bit indexed color
- 16 bit direct (RGB565) color

The following resolutions are supported by this driver:

- 640x480 at 60 Hz
- 800x600 at 80 Hz
- 1024x768 at 60 Hz

Additional refresh rates may be added by modifying the pixel clock settings in the files udct8mod.c or udct16md.c.

Optional Components

This driver supports the optional JPEG extension and double buffering. The JPEG extension can be added either using the configuration tool or by defining INCLUDE_UGL_JPEG within the uglInit.h file. The double buffering may be enabled using either the configuration tool or by defining INCLUDE_UGL_DOUBLE_BUFFERING within the uglInit.h file.

VxWorks BSP Modifications

This driver does not require any modifications to the VxWorks BSP other than to set up the mapping of the PCI memory. Refer to the configuration section on procedures to set up the PCI mapping.

Supported Processors

This driver has been tested on the following processors in all supported resolutions and frame buffer formats:

- X86 (Pentium)
A.5 Tvia IGS5050 Driver

Refer to the file target/h/ugl/driver/graphics/igs/uglIgs5050.txt for possible updates for this driver.

This driver provides graphics support for WindML on IGS 5050 compatible devices.

Modes

The IGS5050 support available within UGL are the following frame buffers configurations:

- 8 bit pseudo color
- 16 bit (RGB565 and ARGB4444 pixel configurations)

Supported monitor resolutions are:

- 640x480 at refresh rate of 50 Hz
- 640x480 at refresh rate of 60 Hz
- 800x600 at refresh rate of 50 Hz
- 800x600 at refresh rate of 75 Hz
- 1024x768 at refresh rate of 60 Hz

Optional Components

This driver has several optional components that may be defined either by use the configuration tool or within the uglInit.h file.

The 16 bit driver can be configured for either the RGB565 or the ARGB4444 format. When using the configuration tool as the configuration mechanism RGB565 will be used is a 16 bit color depth is selected without alpha support. When alpha support is selected, the ARGB4444 format will be selected. When editing the uglInit.h file, the frame buffer is explicitly selected.

The driver provides support for the JPEG extension that may be enabled through the configuration tool or by defining the INCLUDE_UGL_JPEG macro within the uglInit.h file when the command line is used for building WindML.

The driver provides support for the video extension that may be enabled through the configuration tool or by defining the INCLUDE_UGL_VIDEO macro within the uglInit.h file when the command line is used for building WindML. When the video extension is selected, the overlay optional component is automatically included.
This driver has support for both software and hardware cursors. The hardware cursor has a limitation that it can be a maximum of 64x64 and can only have two colors. If the cursor fits within these parameters, then the preferred cursor to use is the hardware cursor. However, if the cursor exceeds these restrictions then a hardware cursor may not be used and the attempt to create a cursor will fail. By default the hardware cursor is used. If it is desired to use the software cursor, then the software cursor must be enabled either through the use of the configuration tool or by defining `INCLUDE_UGL_SW_CURSOR` within the `uglInit.h` file.

This driver provides support for two frame buffer overlays which may be alpha blended with primary frame buffer (i.e., the graphics) when the frame buffer is set up as ARGB4444. An overlay may be configured such it used to capture video from a video source (such as, a camera, VCR, etc.). Using the JPEG extension, the video image may be captured as a JPEG image. The driver currently limits the overlay pixel formats to YUV422. The graphics operations that may be performed on the overlay are limited to bitmap reading. Therefore, an image from a video source may be read as a bitmap or saved as a JPEG image. To enable the use of overlays, either the configuration tool may be used or the `INCLUDE_UGL_OVERLAY` may be defined within `uglInit.h`. If alpha is to be associated with the overlays, the alpha component must be included either using the configuration tool or by defining `INCLUDE_UGL_ALPHA` macro within `uglInit.h`.

**VxWorks BSP Modifications**

This driver does not require any modifications to the VxWorks BSP other than to set up the mapping of the PCI memory. Refer to the configuration section on procedures to set up the PCI mapping. This driver is designed to obtain the frame buffer address from the PCI configuration header. Hence, the PCI information must be correctly configured prior to the initialization of the driver. The memory mapping must be set up such that there exists 16 Megabytes of cache inhibited memory at the address of the frame buffer.

Signal M0D26 on the CyberPro5050 controls whether the I/O mode is supported. When this signal has a pull-up resistor, the I/O mode is disabled and all operations must be performed by memory mapped I/O. The manner in which this is accomplished varies among CyberPro5050 implementations. The Tvia demo card uses a jumper at JP3 to enable/disable the I/O mode. When the jumper is installed, the I/O mode is disabled. If the device is used on a processor that does not support an ISA I/O cycle, then the graphics device must be forced into the linear frame buffer mode by asserting M0D26 (JP3 installed on the Tvia demo card).
Supported Processors

This driver has been tested on the following processors:

Intel X86 and PowerPC 603

Tested for both 8 and 16 bit frame buffers at all supported resolutions and frame buffer formats. Video extension has been tested in all modes. The platform used for testing the powerPC was the SandPoint evaluation platform using the 8240 processor module.

MIPs (R4000)

Tested in 8 bit frame buffers at all supported resolutions. Video extension has been tested in the 8 bit frame buffer mode.

A.6 MediaGX Driver

Refer to the target/h/ugl/driver/graphics/mediagx/udmgx.txt for possible updates for this driver.

This driver provides graphics support for WindML on Cyrix MediaGX processors WindML.

Modes

The following frame buffer formats are supported by this driver:

- 8 bit indexed color
- 16 bit direct (RGB565) color

The following resolutions are supported by this driver:

- 640x480 at 60 Hz
- 640x480 at 72 Hz
- 640x480 at 75 Hz
- 800x600 at 60 Hz
- 800x600 at 72 Hz
- 800x600 at 75 Hz
- 1024x768 at 60 Hz
- 1024x768 at 70 Hz
- 1024x768 at 75 Hz
Optional Components

This driver supports the optional JPEG extension and double buffering. The JPEG extension can be added either using the configuration tool or by defining INCLUDE_UGL_JPEG within the uglInit.h file. The double buffering may be enabled using either the configuration tool or by defining INCLUDE_UGL_DOUBLE_BUFFERING within the uglInit.h file.

VxWorks BSP Modifications

This driver does not require any modifications to the VxWorks BSP other than to set up the mapping of the PCI memory. When this driver starts up it will print out the memory space that will be used by the graphics device. This memory space must be reflected within the memory space setup by sysLib.c. Refer to the configuration section on procedures to set up the PCI mapping.

Supported Processors

This driver has been tested on the Cyrix MediaGX processor in all supported resolutions and frame buffer formats:

A.7 Motorola M821 Driver

Refer to the target/h/ugl/driver/graphics/motorola/udm821.txt file for possible updates for this driver.

This driver provides graphics support for WindML on the Motorola MPC821 processors using the embedded LCD controller.

Modes

The following resolution is supported by this driver:

640x480 at refresh rate of 60 Hz

The only a generic 8-bit indexed mode with a linear frame buffer is available.
Optional Components

This driver supports the optional JPEG extension which can be added either using the configuration tool or by defining INCLUDE_UGL_JPEG within the uglInit.h file.

VxWorks BSP Modifications

This driver does not require any modifications to the VxWorks BSP.

Supported Processors

This driver has been tested using the ACT/Technico EMBoX using the EMVuE LCD display (640 x 480 8 bit color).

A.8 PC BIOS Driver

Refer to the target/h/ugl/driver/graphics/pcbios/udpcbios.txt file for possible updates for this driver.

This driver provides graphics support for WindML on IBM PC compatible devices with VESA Video Bios Extension (VBE) version 2.0 or later. This driver uses the Video BIOS Extentions (VBE) defined by the Video Electronics Standards Association (VESA) to initialize the display, to find available modes, and to implement double buffering.

Modes

The following frame buffer formats are supported by this driver:

- 8 bit indexed color
- 16 bit direct (RGB565) color

All resolutions supported in BIOS with the above frame buffer formats are supported. The following resolutions can be configured in Tornado:

- 640x480
- 800x600
- 1024x768
Optional Components

This driver supports the optional JPEG extension and double buffering. The JPEG extension can be added either using the configuration tool or by defining `INCLUDE_UGL_JPEG` within the `uglInit.h` file. The double buffering may be enabled using either the configuration tool or by defining `INCLUDE_UGL_DOUBLE_BUFFERING` within the `uglInit.h` file.

VxWorks BSP Modifications

VBE compliant video BIOS can be found in most PC compatible systems. The BIOS routines, with the exception of double buffering routines, are executed in real-mode. During driver creation, and mode setting, the task is locked and all interrupts are disabled. This must be taken into consideration for real-time applications.

A BSP modification is required for this driver to work. The modifications are performed as follows:

1. Add the following line to the BSP file romInit.s:
   ```
   #include <ugl/driver/graphics/pcbios/romInit.h>
   ```
2. Rebuild the boot ROM.

Supported Processors

This driver has been tested on Intel x86 processor using all supported resolutions and frame buffer formats.

A.9 Q2SD Driver

Refer to the file `target/h/ugl/driver/graphics/q2sd/udQ2SD.txt` for possible updates for this driver.

This driver provides graphics support for WindML on the Q2SD (HD64413A) Quick 2D renderer for the SH-4 processor. This driver has been specifically developed to function with the SH-4 SolutionsEngine using the Q2-SD daughter board.
Modes

This driver only supports the 8 bit indexed color mode with a resolution of 640x480 at 60 Hz. It has only been tested using the RGB output mode.

Optional Components

This driver supports the optional JPEG extension. The JPEG extension can be added either using the configuration tool or by defining `INCLUDE_UGL_JPEG` within the `uglInit.h` file.

VxWorks BSP Modifications

No modifications are required to the BSP to use this driver.

Supported Processors

This driver has been tested on the SH-4 SolutionsEngine processor. It has only been tested using area 2 of the address map.

A.10 SimNT Driver

Refer to the file `target/h/ugl/driver/graphics/simnt/udsimnt.txt` for possible updates for this driver.

This driver is used with the VxWorks PC simulator to display WindML graphics in a window on a Microsoft Windows host.

Modes

This driver simulates any resolution, and the following frame buffer formats:

- 1 bit monochrome
- 2 bit gray scale
- 4 bit gray scale
- 4 bit indexed color
- 8 bit gray scale
- 8 bit indexed color
- 16 bit direct (RGB565) color
- 32 bit direct (RGB888) color
Only a subset of these modes can be configured in Tornado. These include the following:

- 640x480 4-bit indexed color
- 320x200 8-bit indexed color
- 640x480 8-bit indexed color
- 800x600 8-bit indexed color
- 1024x768 8-bit indexed color
- 640x480 16-bit direct (RGB565) color
- 800x600 16-bit direct (RGB565) color
- 1024x768 16-bit direct (RGB565) color

Optional Components

This driver supports the optional JPEG extension and double buffering. The JPEG extension can be added either using the configuration tool or by defining INCLUDE_UGL_JPEG within the uglInit.h file. The double buffering may be enabled using either the configuration tool or by defining INCLUDE_UGL_DOUBLE_BUFFERING within the uglInit.h file.

VxWorks BSP Modifications

A modification to the VxWorks image is required to use this driver. For more information, refer to 2.6.1 Configuring SimNT, p.36.

Supported Processors

This driver has been tested on Windows NT for Intel x86 in the 8 and 16 bit frame buffer formats.

A.11 SimSolaris Driver

Refer to the file target/h/ugl/driver/graphics/simsolaris/udx11.txt for possible updates for this driver.

This driver is used with the VxWorks Solaris simulator to display WindML graphics in a window on a X11 server. To use this driver, WindML must be compiled with CPU defined as SIMSOLARIS.
Standard Drivers

Modes

This driver simulates any resolution, and the following frame buffer formats:

- 640x480 4-bit indexed color
- 320x200 8-bit indexed color
- 640x480 8-bit indexed color
- 800x600 8-bit indexed color
- 1024x768 8-bit indexed color
- 640x480 16-bit direct (RGB565) color
- 800x600 16-bit direct (RGB565) color
- 1024x768 16-bit direct (RGB565) color

Optional Components

This driver supports the optional JPEG extension and double buffering. The JPEG extension can be added either using the configuration tool or by defining `INCLUDE_UGL_JPEG` within the `uglInit.h` file. The double buffering may be enabled using either the configuration tool or by defining `INCLUDE_UGL_DOUBLE_BUFFERING` within the `uglInit.h` file.

VxWorks BSP Modifications

There are a number of operations that must be performed to configure the Solaris vxSim for use in a WindML environment. For more information, see 2.6.2 Configuring SimSolaris, p.36.

Supported Processors

This driver has been tested on the Solaris processors using 8 and 16 bit frame buffer formats.
uglLib

NAME

uglLib - WindML 2D API

ROUTINES

uglBackgroundColorGet() - Gets a graphics context’s background color setting
uglBackgroundColorSet() - Sets a graphics context’s background color
uglBatchEnd() - Complete a rendering batch operation
uglBatchStart() - Starts a rendering batch
uglBitmapBlit() - Perform a block image transfer of a DDB
uglBitmapCreate() - Create a standard DDB
uglBitmapDestroy() - Destroys a DDB
uglBitmapRead() - Read image data from a standard DDB
uglBitmapSizeGet() - Get the size of a DDB
uglBitmapStretchBlit() - Performs a scaled block image transfer of a DDB
uglBitmapWrite() - Write image information to a standard DDB
uglClipListGet() - Get the clip list for a graphics context
uglClipRectSet() - Sets a graphics context’s clipping rectangle
uglClipRectGet() - Gets a graphics context’s clipping rectangle
uglClipRegionGet() - Get the clipping region of a graphics context
uglClipRegionSet() - Set the clipping region of a graphics context
uglClutGet() - Get entries from a color lookup table
uglClutSet() - Set entries in a color lookup table
uglColorAlloc() - Allocate colors to be used on a device
uglColorConvert() - Convert an array of colors from one color format to another
uglColorFree() - Frees colors that are currently in use on a device
uglCursorBitmapCreate() - Creates a cursor bitmap
uglCursorBitmapDestroy() - Destroy a cursor bitmap
uglCursorDeinit() - Deinitializes the cursor
uglCursorImageGet() - Gets the image displayed by a cursor
uglCursorImageSet() - Sets the image displayed by a cursor
uglCursorInit() - Initialize the cursor
uglCursorMove() - Sets the position of a cursor
uglCursorOff() - Turns a cursor off
uglCursorOn() - Turns a cursor on
uglCursorPositionGet() - Gets the position of a cursor
uglDefaultBitmapGet() - Gets a graphics context’s default bitmap
uglDefaultBitmapSet() - Sets a graphics context’s default bitmap
uglDriverFind() - Find a driver within the driver registry
uglDriverRegister() - Register a driver to driver directory services
uglDriverUnRegister() - Unregister a driver
uglEllipse() - Draws an ellipse, arc, or pie slice
uglFillPatternGet() - Get a graphics context’s fill pattern setting
uglFillPatternSet() - Sets a graphics context’s fill pattern
uglFontCreate() - Create a font
uglFontDestroy() - Destroy a font
uglFontDriverDestroy() - Destroy a font driver
uglFontDriverInfo() - Retrieve or send information to a font driver
uglFontFind() - Find a specific font
uglFontFindClose() - Free memory used by first/next font routines
uglFontFindFirst() - Obtain the first available font
uglFontFindNext() - Obtain the next available font.
uglFontFindString() - Find a specific font with a search string
uglFontGet() - Get a graphics context’s current font setting
uglFontMetricsGet() - Retrieve font metrics data
uglFontRotationAngleSet() - Set the rotation angle of a font
uglFontSet() - Sets a graphics context’s font
uglFontSizeSet() - Set the pixel size of a font
uglFontWeightSet() - Set the weight or bold of a font
uglForegroundColorGet() - Gets a graphics context’s foreground color setting
uglForegroundColorSet() - Sets a graphics context’s foreground color
uglGeCopy() - Copies a graphics context
uglGeCreate() - Creates a graphics context
uglGeDestroy() - Destroys a graphics context
uglInfo() - Retrieves information from the driver
uglLine() - Draws a line
uglLineStyleGet() - Gets a graphics context’s line style
uglLineStyleSet() - Sets a graphics context’s line style
uglLineWidthGet() - Get a graphics context’s line width setting
uglLineWidthSet() - Sets a graphics context’s line width
uglMemDefaultPoolGet() - Get the default memory pool
uglMemDefaultPoolSet() - Set the default memory pool
uglMemDevicePoolCreate() - Create a device memory pool
uglMemDevicePoolDestroy() - Destroy a device memory pool.
uglModeAvailGet() - Gets the available graphics device modes
uglModeAvailPrint() - Prints the available graphics device modes
uglModeNearestGet() - Gets the closest mode that graphics device supports
uglModeSet() - Sets the graphics device mode
uglMonoBitmapCreate() - Creates a monochrome bitmap
uglMonoBitmapDestroy() - Destroys a monochrome bitmap
uglMonoBitmapRead() - Read image data from a monochrome bitmap
uglMonoBitmapWrite() - Write image information to a monochrome bitmap
uglPageCopy() - Copy the contents of one page to another.
uglPageCreate() - Create a page.
uglPageDestroy() - Destroy a page.
uglPageDrawGet() - Get the active drawing page.
uglPageDrawSet() - Set active drawing page.
uglPageVisibleGet() - Get the active display page.
uglPageVisibleSet() - Set the visible display page.
uglPixelGet() - Gets the color of a pixel
uglPixelSet() - Draws a pixel
uglPolygon() - Draws a polygon
uglRasterModeGet() - Gets a graphics context's raster mode setting
uglRasterModeSet() - Sets a graphics context's raster mode
uglRectangle() - Draws a rectangle
uglRegionClipToRect() - Clip a region to a rectangle
uglRegionCopy() - Copy the contents of one region to another
uglRegionCreate() - Create a region
uglRegionDestroy() - Destroy a region
uglRegionEmpty() - Empty contents of a region
uglRegionIntersect() - Find the intersection of two regions
uglRegionIsEmpty() - Determine if a region is empty
uglRegionMove() - Move a region
uglRegionRectExclude() - Exclude a rectangle from a region
uglRegionRectGet() - Get the rectangles comprising a region
uglRegionRectInclude() - Include a rectangle in a region
uglRegionRegionExclude() - Exclude one region from another region
uglRegionUnion() - Find the union of two regions
uglTextDraw() - Draw text
uglTextDrawW() - Draw double-wide text
uglTextSizeGet() - Get text size
uglTextSizeGetW() - Get text size for double byte text.
uglTransBitmapCreate() - Creates a transparent bitmap
uglTransBitmapCreateFromDDB() - Creates a transparent bitmap from standard bitmap
uglTransBitmapDestroy() - Destroys a transparent bitmap
uglTransBitmapRead() - Read image data from a transparent bitmap
uglTransBitmapWrite() - Write image information to a transparent bitmap
uglViewPortGet() - Gets a graphics context's view port
uglViewPortSet() - Sets a graphics context's view port
This library provides the 2D functionality for WindML.

**uglBackgroundColorGet()**

**NAME**

`uglBackgroundColorGet()` - gets a graphics context’s background color setting

**SYNOPSIS**

```c
UGL_STATUS uglBackgroundColorGet
(  UGL_GC_ID gc,              /* graphics context */
  UGL_COLOR * pBackgroundColor /* background color setting */
)
```

**DESCRIPTION**

This routine returns the background color of graphics context `gc`, and places it in the location specified by `pBackgroundColor`.

**RETURNS**

- `UGL_STATUS_OK`
- `UGL_STATUS_ERROR` if the `gc` is null.

**ERRNO**

N/A

**SEE ALSO**

`uglLib`, `uglBackgroundColorSet()`

---

**uglBackgroundColorSet()**

**NAME**

`uglBackgroundColorSet()` - sets a graphics context’s background color

**SYNOPSIS**

```c
UGL_STATUS uglBackgroundColorSet
(  UGL_GC_ID gc,             /* graphics context */
  UGL_COLOR color           /* new background color */
)
```

**DESCRIPTION**

This routine sets the background color of graphics context `gc` to the color specified by `color`. The background color is used for filling the interior of shapes, such as rectangles, polygons, and ellipses. Additionally, the background color is used for 0 bits in monochrome bitmaps.

**RETURNS**

- `UGL_STATUS_OK`
- `UGL_STATUS_ERROR` if the `gc` is null.

**ERRNO**

N/A
uglBatchEnd()

NAME

uglBatchEnd() - complete a rendering batch operation

SYNOPSIS

UGL_STATUS uglBatchEnd

   (UGL_GC_ID gc /* Graphics Context */)

DESCRIPTION

This routine ends a rendering batch operation. Batching is a mechanism for arbitrating access to shared display resources, such as the frame buffer and graphics device. This routine releases a device lock set by uglBatchStart() and restores any cursors hidden during the uglBatchStart() call. Because batching calls can be nested, uglBatchEnd() must be called the same number of times as uglBatchStart() to end a batch.

RETURNS

UGL_STATUS_OK, or UGL_STATUS_ERROR if gc is UGL_NULL

ERRNO

N/A

SEE ALSO

uglLib, uglBackgroundColorGet()

uglBatchStart()

NAME

uglBatchStart() - starts a rendering batch

SYNOPSIS

UGL_STATUS uglBatchStart

   (UGL_GC_ID gc /* Graphics Context */)

DESCRIPTION

This routine begins a graphics rendering batching operation. Batching is a mechanism for arbitrating access to shared display resources. This function gives the calling thread exclusive access to the display resource. If another thread has control of the display, a call to uglBatchStart() pends (with no timeout) until the display becomes available. All software cursors are hidden during a batching operation.
Batching is performed internally by several UGL 2-D rendering operations. For this reason, batching calls are optional for applications.

The use of batching is illustrated by the following code:

```
uglBatchStart (gc);    /* Start protected rendering */
.....               /* Sequence of rendering statements */
uglBatchEnd (gc);      /* End of batched rendering statements */
```

Batching calls may be nested. `uglBatchEnd()` must be called the same number of times as `uglBatchStart()` for a lock to be released.

**RETURNS**

UGL_STATUS_OK, or UGL_STATUS_ERROR if gc is UGL_NULL

**ERRNO**

N/A

**SEE ALSO**

uglLib, `uglBatchEnd()`

---

### `uglBitmapBlt()`

**NAME**

`uglBitmapBlt()` - perform a block image transfer of a Device Dependant

**SYNOPSIS**

```c
UGL_STATUS uglBitmapBlt(
    UGL_GC_ID     gc,           /* graphics context */
    UGL_BITMAP_ID srcBitmapId,  /* source bitmap */
    UGL_POS       sourceLeft,   /* left boundary of the source rect */
    UGL_POS       sourceTop,    /* top boundary of the source rect */
    UGL_POS       sourceRight,  /* right boundary of the source rect */
    UGL_POS       sourceBottom, /* bottom boundary of the source rect */
    UGL_DDB_ID    dstBitmapId,  /* destination bitmap */
    UGL_POS       destX,        /* left boundary of destination */
    UGL_POS       destY         /* top boundary of destination */
);
```

**DESCRIPTION**

Bitmap (DDB)

This routine transfers image data from a Device Dependent Bitmap (DDB) to a standard DDB. The source and destination DDB may be the same DDB.

The graphics context `gc` provides raster operation, viewport, clipping, and driver information for the bitblt operation.
The source bitmap is specified by `srcBitmapId` and the destination bitmap is specified by `dstBitmapId`. The `srcBitmapId` and `dstBitmapId` parameters specify either a DDB or one of the following:

- **UGL_DEFAULT_ID** (0)
  Specifies the default bitmap in the gc.
- **UGL_DISPLAY_ID** (-1)
  Specifies the display.

A standard DDB is defined as either an **UGL_DDB_ID**, **UGL_DEFAULT_ID**, or **UGL_DISPLAY_ID**. The other possible types of a DDB are a monochrome bitmap (**UGL_MDBB_ID**) or a transparent bitmap (**UGL_TDDB_ID**). The `srcBitmapId` parameter may specify any bitmap type including monochrome and transparent bitmaps. However, the `dstBitmapId` parameter must always specify a standard bitmap.

When **UGL_DEFAULT_ID** is specified as the destination bitmap, the blt operation is performed similar to other drawing operations. In particular, the viewport, clipping, and raster op information contained in the gc are applied to the destination. When **UGL_DEFAULT_ID** is specified as the source bitmap, only the viewport and raster op are applied to the source. Clipping information is not applied to the source in this case. This information is true regardless if the default bitmap is the display or an offscreen bitmap.

When **UGL_DISPLAY_ID** or a valid DDB are specified as the source and/or destination bitmap, the viewport and clipping information in the gc are not applied. The raster op specified in the gc is applied.

Blt operations are always clipped to the bounds of both the source and destination bitmaps. The raster operation specified in the gc applies to all blt operations.

The `sourceLeft`, `sourceTop`, `sourceRight`, and `sourceBottom` parameters specify the rectangular region of image data that is to be copied from the source. The `destX`, `destY` parameters specify the top left corner of the rectangular region to which the source image data is to be copied. The width and height of the destination rectangle are the same as the width and height of the source rectangle.

**RETURNS**

- **UGL_STATUS_OK** (0), or a non zero value if the operation fails.

**SEE ALSO**

- `uglLib`, `uglBitmapCreate()`, `uglMonoBitmapCreate()`, `uglTransBitmapCreate()`
uglBitmapCreate()

NAME

uglBitmapCreate() - create a standard Device Dependant Bitmap (DDB)

SYNOPSIS

UGL_DDB_ID uglBitmapCreate

{ 
    UGL_DEVICE_ID    devId,    /* device identifier */
    UGL_DIB *        pDib,     /* DIB to use for creation of a */
                       /* bitmap */
    UGL_DIB_CREATE_MODE createMode, /* control bitmap initialization */
    UGL_UINT32       initValue, /* value to use to initialize bitmap */
                       /* */
    UGL_DEVICE_MEM_POOL_ID devicePoolId /* memory pool to contain bitmap */
}

DESCRIPTION

This routine creates a standard DDB for the graphics device devId. The format of the image data stored in a DDB is determined by the graphics driver and is typically the same format used by the graphics device. Thus the bitmap is “device dependent.” The bitmap is created by converting the image information containing the Device Independent Bitmap (DIB) specified by the pDib parameter. The DIB structure is designed in a portable manner and is thus “device independent”. The DIB structure is illustrated below:

```
+---------------------------------+    +-------------------+
|               |          |         |
| Header        |--------> |  CLUT   |
| Information   |          |_________|
|               |          |         |
| CLUT Pointer  |----       _________
|_______________|    |     |         |
|               |    |  Image  |
| Image Pointer |----       _________
|_______________|    |     |         |
|               |    |  Data   |
| Data          |_________|
```

The DIB consists of three parts: header information, a color lookup table (CLUT), and image data. The header is mandatory for all bitmaps. The CLUT and image data are optional depending upon the needs of the application.

There are two types of DIB image data formats: indexed color, and direct color. In indexed color format, all colors used by an image are found in the CLUT, and the image data is composed of indexes into the CLUT. In direct color format, a CLUT is not present and the image data contains all of the color information for the image.
The presence of a CLUT in the DIB does not necessarily indicate that the graphics hardware uses a CLUT. The color lookup table within the DIB is a means of compressing the image data. Conversely, specifying a direct image format does not necessarily indicate that the graphics hardware is direct color. The format of a DIB is independent of the graphics hardware.

The DIB provides a flexible manner in which to organize bitmap images. A DIB may be organized so as to minimize memory requirements or it may be organized to minimize processing time when a bitmap image is to be created. When a bitmap image is organized for minimal processing time, the image is coded in the same format as that used by the frame buffer. For example, if the frame buffer uses an RGB565 format, then the image data should also use the RGB565 format. When memory requirements are a concern, the data can be organized such that the data is compressed using an embedded color lookup table.

Any colors specified in the DIB (whether direct or indexed) not available on the graphics device will be converted to the nearest matching available colors. On indexed color devices, the application is responsible for calling `uglColorAlloc()` to make the required colors available (only colors previously allocated by the application are available for nearest matching). On direct color devices, `uglColorAlloc()` is optional.

When color conversion is necessary, it may be faster to supply a DIB with a CLUT so that only the CLUT is converted rather than each pixel in the image.

The DIB structure is shown below:

```c
typedef struct ugl_dib
{
    UGL_SIZE width;        /* width of image in pixels */
    UGL_SIZE height;       /* height of image in pixels */
    UGL_SIZE stride;       /* distance in pixels between rows of image data */
    UGL_COLOR_FORMAT imageFormat;  /* format of the index in bitmap image */
    UGL_COLOR_FORMAT colorFormat;  /* color format of the image */
    UGL_SIZE clutSize;     /* size of clut in number of elements */
    void * pClut;         /* pointer to CLUT */
    void * pImage;         /* pointer to image data */
} UGL_DIB;
```

`width` and `height` are the width and height of the image data, in pixels.

`stride` is the distance, in number of pixels, from the start of one row of image data to the start of the next row. (`stride` is often equal to `width`, but may be greater than `width` to align rows on word boundaries or to define a DIB that is a subset of a larger DIB.)

`imageFormat` specifies the format of the image data in the DIB. It may be any of the following values:

**UGL_DIRECT**

The image data is in direct color format. This means that no CLUT is present, and the image data is in a format...
specified by `colorFormat`. While this option means that no CLUT is present in the DIB, it does not mean that no CLUT is present in the graphics hardware.

**UGL_INDEXED_1**
The image data is in indexed color format, and the size of each index is 1 bit. Each 1 bit index may select from up to 2 colors within the clut. `clutSize` determines the number of colors in the image.

**UGL_INDEXED_2**
The image data is in indexed color format, and the size of each index is 2 bits. Each 2 bit index may select from up to 4 colors within the clut. `clutSize` determines the number of colors in the image.

**UGL_INDEXED_4**
The image data is in indexed color format, and the size of each index is 4 bits. Each 4 bit index may select from up to 16 colors within the clut. `clutSize` determines the number of colors in the image.

**UGL_INDEXED_8**
The image data is in indexed color format, and the size of each index is 8 bits. Each 8 bit index may select from up to 256 colors within the clut. `clutSize` determines the number of colors in the image.

colorFormat specifies the format of the color data within the CLUT for indexed color formats, or the format of the image data for direct color format. The following formats are defined by UGL:

**UGL_ARGB8888**
The data is organized in 32 bit pixels where there are four components, alpha channel, red, green, and blue. Each component is 1 byte, with alpha channel being the most significant byte, and blue being the least significant byte.

**UGL_RGB565**
The data is organized in 16 bit pixels where there are three components, red, green, and blue. The red and blue components are 5 bits and the green component is 6 bits. Red is located in the most significant bits, and blue is located in the least significant bits.

**UGL_DEVICE_COLOR_32**
The data is organized as UGL_COLOR values where each pixel occupies 32 bits and contains a value generated by `uglColorAlloc()`. The data interpretation is device driver specific. This format
2D Graphics API

is useful for increasing performance on indexed or “pseudo” color devices.

**UGL DEVICE COLOR**

The data format of the DIB matches the data format of the driver's DDB and the interpretation is device driver specific. This format is useful for maximizing data conversion speeds.

Additional format types may be defined by extended device drivers. UGL reserves color format values 0-127, so driver extended color formats must have a value greater than or equal to 128.

*pClut* is a pointer to a Color Look-Up Table (CLUT). The CLUT is an array of color values where each entry is the size and format specified by *colorFormat*. This can be initialized to NULL if *imageFormat* is **UGL_DIRECT**. Otherwise it must be initialized to point to a valid CLUT.

*pImage* is a pointer to the image data. When *imageFormat* is **UGL_DIRECT**, the image data is an array of color values where each value is the size and format specified by *colorFormat*. When *imageFormat* is an indexed format, the image data is array of index values. The size, in bits, of each index value can be obtained by ANDing *imageFormat* with **UGL_INDEX_MASK**.

The **createMode** parameter controls the manner in which the bitmap is initialized, as follows:

**UGL_DIB_INIT_DATA**

Initialize the bitmap with the data present within the DIB

**UGL_DIB_INIT_VALUE**

Initialize the bitmap to color value *initValue*

**UGL_DIB_INIT_NONE**

Perform no bitmap initialization.

When the **createMode** is set to **UGL_DIB_INIT_DATA**, the data pointed to by *pImage* in *pDib* is converted to the format that is used by the graphics device and its frame buffer.

*deviceMemPoolId* specifies the memory pool in which the bitmap is allocated. It must be the device ID returned from a call to *uglMemDevicePoolCreate()* or one of the following values:

**UGL_DEFAULT_MEM**

The bitmap is created in the devices default memory pool.

**UGL_VIDEO_MEM**

The bitmap is created in video memory not used by the visible display. In order to use this option, the application must ensure that video memory is available on the graphics device via *uglInfo()*.

**UGL_NULL**

Same as **UGL_DEFAULT_MEM**.
uglBitmapDestroy()

NAME
uglBitmapDestroy() - destroys a Device Dependant Bitmap (DDB)

SYNOPSIS
UGL_STATUS uglBitmapDestroy
{
    UGL_DEVICE_ID devId, /* device identifier */
    UGL_DDB_ID    ddbId   /* bitmap ID */
}

DESCRIPTION
pDdb specifies the bitmap to be destroyed. This bitmap may be a DDB returned by a prior
    call to uglBitmapCreate(), uglMonoBitmapCreate(), or uglTransBitmapCreate().

RETURNS
UGL_STATUS_OK (0), or a non zero value if the operation fails.

SEE ALSO
uglLib, uglBitmapCreate(), uglMonoBitmapCreate(), uglTransBitmapCreate()

uglBitmapRead()

NAME
uglBitmapRead() - read image data from a standard DDB

SYNOPSIS
UGL_STATUS uglBitmapRead
{
    UGL_DEVICE_ID devId, /* device identifier */
    UGL_DDB_ID    srcBitmapId, /* source bitmap */
    UGL_POS       sourceLeft,  /* left boundary of data */
    UGL_POS       sourceTop,   /* top boundary of data */
    UGL_POS       sourceRight, /* right boundary of data to read */
    UGL_POS       sourceBottom, /* bottom boundary of data to read */
    UGL_DIB *     pDib,        /* DIB that receives source information */
    UGL_POS       dstX,        /* X coordinate in DIB */
    UGL_POS       dstY /* Y coordinate in DIB */
}
DESCRIPTION

This routine reads image data from a standard Device Dependent Bitmap (DDB), and places it in a standard Device Independent Bitmap (DIB). pDdb specifies the source bitmap, and pDib specifies the destination bitmap. (See uglBitmapCreate() for a description of the standard DIB structure.)

pDdb specifies either a standard bitmap created by uglBitmapCreate() or the following:

UGL_DISPLAY_ID (-1)

Specifies the display.

Read operations are always clipped to the bounds of both the source and destination bitmaps. The raster operation, viewport, and clipping information specified in the graphics context will not be applied to the read operation.

The sourceLeft, sourceTop, sourceRight, and sourceBottom parameters specify the rectangular region of image data that is to be copied from the source. The destX, destY parameters specify the top left corner of the rectangular region to which the source image data is to be copied. The width and height of the destination rectangle are the same as the width and height of the source rectangle.

pDib must specify a DIB with an image format of UGL_DIRECT, and a color format of UGL_DEVICE_COLOR_32. The image data read from the DDB will be converted to this format. Other DIB formats are not supported.

The application must allocate the destination UGL_DIB pDib and associated image space needed to receive the UGL_DEVICE_COLOR_32 array. In addition to the pImage component, the application must specify the DIB stride, which is typically the source image width.

The following code shows an example of how uglBitmapRead() could be used read the contents of the current screen:

```c
UGL_DIB * pDib = UGL_CALLOC (1, sizeof (UGL_DIB));
pDib->imageFormat = UGL_DIRECT;
pDib->colorFormat = UGL_DEVICE_COLOR_32;
pDib->width = screenWidth;
pDib->height = screenHeight;
pDib->stride = screenWidth;
pDib->pImage = UGL_MALLOC (screenWidth * screenHeight * sizeof (UGL_COLOR));
uglBitmapRead(devId, UGL_DISPLAY_ID, 0, 0, screenWidth - 1, screenHeight - 1, pDib, 0, 0);
```

RETURNS

UGL_STATUS_OK (0), or a non zero value if the operation fails.
uglBitmapSizeGet()

NAME

uglBitmapSizeGet() - get the size of a Device Dependent Bitmap (DDB)

SYNOPSIS

UGL_STATUS uglBitmapSizeGet

(  
    UGL_DDB_ID ddbId,
    UGL_SIZE * pWidth,
    UGL_SIZE * pHeight
)

DESCRIPTION

This routine allows the application to retrieve the width and height values of a DDB specified by ddbId. This routine is appropriate for all bitmap types.

If pWidth and pHeight are not NULL, the width and height of the DDB will be returned in each respectively.

RETURNS

UGL_STATUS_OK (0), or a non zero value if the operation fails.

SEE ALSO

uglLib, uglBitmapCreate()

uglBitmapStretchBlt()

NAME

uglBitmapStretchBlt() - performs a scaled block image transfer of a

SYNOPSIS

UGL_STATUS uglBitmapStretchBlt

(  
    UGL_GC_ID     gc,           /* graphics context */
    UGL_BITMAP_ID srcBitmapId,  /* source bitmap */
    UGL_POS       sourceLeft,   /* left boundary of the source rect */
    UGL_POS       sourceTop,    /* top boundary of the source rect */
    UGL_POS       sourceRight,  /* right boundary of the source rect */
    UGL_POS       sourceBottom, /* bottom boundary of the source rect */
    UGL_DDB_ID    dstBitmapId,  /* destination bitmap */
    UGL_POS       destLeft,     /* left boundary of destination */
    UGL_POS       destTop,      /* top boundary of destination */
)
DESCRIPTION

Device Dependant Bitmap (DDB)

This routine transfers image data from a Device Dependent Bitmap (DDB) to another standard DDB, scaling if necessary. The source and destination DDB may be the same DDB.

The graphics context gc provides raster op, viewport, clipping, and driver information for the blt operation.

The source bitmap is specified by srcBitmapId and the destination bitmap is specified by dstBitmapId. The srcBitmapId and dstBitmapId parameters specify either a DDB or one of the following:

- **UGL_DEFAULT_ID** (0)
  - Specifies the default bitmap in the gc.
- **UGL_DISPLAY_ID** (-1)
  - Specifies the display.

A standard DDB is defined as either an **UGL_DDB_ID**, **UGL_DEFAULT_ID**, or **UGL_DISPLAY_ID**. The other possible types of a DDB are a monochrome bitmap (**UGL_MDDB_ID**) or a transparent bitmap (**UGL_TDDB_ID**). The srcBitmapId parameter may specify any bitmap type including monochrome and transparent bitmaps. However, the dstBitmapId parameter must always specify a standard bitmap.

When **UGL_DEFAULT_ID** is specified as the destination bitmap, the blt operation is performed similar to other drawing operations. In particular, the viewport, clipping, and raster op information contained in the gc are applied to the destination. When **UGL_DEFAULT_ID** is specified as the source bitmap, only the viewport and raster op are applied to the source. Clipping information is not applied to the source in this case. This information is true regardless if the default bitmap is the display or an offscreen bitmap.

When **UGL_DISPLAY_ID** or a valid DDB are specified as the source and/or destination bitmap, the viewport and clipping information in the gc are not applied. The raster op specified in the gc is applied.

Blt operations are always clipped to the bounds of both the source and destination bitmaps. The raster operation specified in the gc applies to all blt operations.

The sourceLeft, sourceTop, sourceRight, and sourceBottom parameters specify the rectangular region of image data that is to be copied from the source.

The destLeft, destTop, destRight, and destBottom specify the rectangular region to which the image data is to be copied. If the destination rectangle is not the same size as the source rectangle, then the image data is scaled (compressed, expanded) as necessary. Compression and expansion operations do not perform dithering on the destination image.
Returns

UGL_STATUS_OK (0), or a non zero value if the operation fails.

See Also

uglLib, uglBitmapBlt(), uglBitmapCreate(), uglMonoBitmapCreate(), uglTransBitmapCreate()

uglBitmapWrite()

NAME

uglBitmapWrite() - write image information to a standard DDB

SYNOPSIS

UGL_STATUS uglBitmapWrite

(  
UGL_DEVICE_ID devId,        /* device identifier */
UGL_DIB *     pDib,         /* DIB to write to bitmap */
UGL_POS       sourceLeft,   /* left boundary of data */
UGL_POS       sourceTop,    /* top boundary of data */
UGL_POS       sourceRight,  /* right boundary of data to read */
UGL_POS       sourceBottom, /* bottom boundary of data to read */
UGL_DDB_ID    dstBitmapId,  /* destination bitmap */
UGL_POS       dstX,         /* X coordinate in DIB */
UGL_POS       dstY,         /* Y coordinate in DIB */
)

DESCRIPTION

Bitmap (DDB)

This routine writes image data from a Device Independent Bitmap (DIB) to a standard
Device Dependent Bitmap (DDB). $pDib$ specifies the source bitmap, and $pDdb$ specifies the
destination bitmap. (see uglBitmapCreate() for a description of the DIB structure.)

$pDdb$ specifies either a standard bitmap created by uglBitmapCreate() or the following:

UGL_DISPLAY_ID (1)

Specifies the display.

Write operations are always clipped to the bounds of both the source and destination
bitmaps. The raster operation, viewport, and clipping information specified in the
graphics context will not be applied to the write operation.

The $sourceLeft$, $sourceTop$, $sourceRight$, and $sourceBottom$ parameters specify the rectangular
region of image data that is to be copied from the source. The $destX$, $destY$ parameters
specify the top left corner of the rectangular region to which the source image data is to be
copied. The width and height of the destination rectangle are the same as the width and
height of the source rectangle.

The image data copied from the DIB is converted to the format required by the graphics
device.
uglClipListGet()

NAME

uglClipListGet() - get the clip list for a graphics context

SYNOPSIS

UGL_STATUS uglClipListGet
{
    UGL_GC_ID gc,          /* graphics context */
    UGL_RECT * pClipRect,   /* clip list rectangle */
    const UGL_RECT * * ppRegionRect /* region rectangle */
}

DESCRIPTION

This routine gets the clip list for the specified gc. The clip list is composed by intersecting the graphics context’s clip rect with the rectangles comprising the graphics context’s clip region. The result is transposed to compensate for the graphics context’s view port. This routine is generally used by graphics drivers to perform clipping on graphics primitives.

dc identifies the graphics contexts for which a clip list is to be obtained.
pRect points to a rectangle that gets an element of the clip list. This rectangle is the intersection of the graphics context’s clip rect with an element of the graphics context’s clip region, transposed to compensate for the graphics context’s view port. (If the graphics context’s clipping region is NULL, then the result is simply the graphics context’s clip rect.) One rectangle is returned per call to this routine.

ppRegionRect is used for enumerating the rectangles in the graphics context’s clip region. This routine calls uglRegionRectGet() for this purpose and ppRegionRect is passed straight through. It must point to UGL_NULL the first time this routine is called, and can be UGL_NULL to terminate the traversal before retrieving the last rectangle.

The code sample below demonstrates the traversal of a clip list:

UGL_RECT clipRect;
UGL_RECT * pRegionRect = UGL_NULL;
/* Traverse the clip list. */
while (uglClipListGet (gc, &clipRect, &pRegionRect) == UGL_STATUS_OK)
{
    /* clipRect holds a clip list element */
    ...
    if (error)
/* End the traversal prematurely. */

uglClipListGet(regionId, UGL_NULL, UGL_NULL);
}

RETURNS
UGL_STATUS_OK if a rectangle is retrieved, UGL_STATUS_FINISHED if no rectangles
are left, or a negative value upon error.

SEE ALSO
uglLib, uglRegionRectGet()

uglClipRectGet()

NAME
uglClipRectGet() - gets a graphics context’s clipping rectangle

SYNOPSIS
UGL_STATUS uglClipRectGet
(
    UGL_GC_ID gc,             /* graphics context */
    UGL_POS * pLeft,          /* left boundary of clipping rectangle */
    UGL_POS * pTop,           /* top boundary of clipping rectangle */
    UGL_POS * pRight,         /* right boundary of clipping rectangle */
    UGL_POS * pBottom         /* bottom boundary of clipping rectangle */
)

DESCRIPTION
This routine gets the current clipping rectangle of graphics context gc. The arguments
pLeft, pTop, pRight, and pBottom are output parameters representing the boundaries of the
clipping rectangle. Any portion of a drawing operation that extends beyond the bounds of
the clipping rectangle is not rendered.

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null.

ERRNO
N/A

SEE ALSO
uglLib, uglClipRectSet()
**uglClipRectSet()**

**NAME**

`uglClipRectSet()` - sets a graphics context’s clipping rectangle

**SYNOPSIS**

```c
UGL_STATUS uglClipRectSet
    (  
        UGL_GC_ID gc,   /* graphics context */
        UGL_POS left,  /* left boundary of clipping rectangle */
        UGL_POS top,   /* top boundary of clipping rectangle */
        UGL_POS right, /* right boundary of clipping rectangle */
        UGL_POS bottom /* bottom boundary of clipping rectangle */
    )
```

**DESCRIPTION**

This routine sets the current clipping rectangle of graphics context `gc`. The arguments `left`, `top`, `right`, and `bottom` specify the boundaries of the clipping rectangle.

The clipping rectangle is set to the intersection of the specified rectangle, the graphics context’s view port, and the bounds of the default bitmap. Any portion of a drawing operation that extends beyond the bounds of the clipping rectangle is not rendered.

The clipping rectangle is inclusive, such that portions of drawing operations falling on the right and bottom boundaries are rendered.

**RETURNS**

`UGL_STATUS_OK`, or `UGL_STATUS_ERROR` if the `gc` is null.

**ERRNO**

N/A

**SEE ALSO**

`uglLib`, `uglClipRectGet()`

---

**uglClipRegionGet()**

**NAME**

`uglClipRegionGet()` - get the clipping region of a graphics context

**SYNOPSIS**

```c
UGL_STATUS uglClipRegionGet
    (  
        UGL_GC_ID gc,   /* graphics context */
        UGL_REGION_ID * pClipRegionId /* clip region */
    )
```

**DESCRIPTION**

This routine gets the clipping region of a specified `gc` and places the contents in `pClipRegionId`. 
uglClipRegionSet()

NAME

uglClipRegionSet() - set the clipping region of a graphics context

SYNOPSIS

UGL_STATUS uglClipRegionSet

(  
   UGL_GC_ID gc,          /* graphics context */
   UGL_REGION_ID clipRegionId /* clip region */
)

DESCRIPTION

This routine sets the clipping region of gc to clipRegionId. If clipRegionId is NULL, the clipping region of the graphics context will be empty.

RETURNS

UGL_STATUS_OK, or UGL_STATUS_ERROR if the operation fails.

SEE ALSO

uglLib

uglClutGet()

NAME

uglClutGet() - get entries from a color lookup table

SYNOPSIS

UGL_STATUS uglClutGet

(  
   UGL_DEVICE_ID devId,      /* driver identifier */
   UGL_ORD startIndex, /* start clut index */
   UGL_ARGB * pColors,    /* array of color values */
   UGL_SIZE numColors   /* number of entries to get */
)

DESCRIPTION

This routine gets an array of color entries from the color lookup table (CLUT) of an indexed color device. If the graphics chip is running in direct (true) color mode, the operation fails (see return values below).
A total of `numColors` entries, starting at the index specified by `startIndex`, are copied from the CLUT and stored in the array specified by `pColors`. The colors copied from the CLUT are converted to an 8-bit `argb` format.

The `use count` information used by `uglColorAlloc()` for CLUT entries is not modified by this routine.

**RETURNS**

`UGL_STATUS_OK` (0), or `UGL_STATUS_ERROR` if the operation fails.

**SEE ALSO**

`uglLib`, `uglClutSet()`, `uglColorAlloc()`

---

**uglClutSet()**

**NAME**

`uglClutSet()` - sets entries in a color lookup table

**SYNOPSIS**

```c
UGL_STATUS uglClutSet
(
    UGL_DEVICE_ID devId,      /* driver identifier */
    UGL_ORD       startIndex, /* start clut index */
    UGL_ARGB *    pColors,    /* array of color entries */
    UGL_SIZE      numColors   /* number of entries to set */
);```

**DESCRIPTION**

This routine sets the color entries in the color lookup table (CLUT) of an indexed pseudo color device. If the graphics chip is running in direct (true) color mode, the operation fails (see return values below).

A total of `numColors` entries are copied into the CLUT, starting at the index specified by `startIndex`, from the color array specified by `pColors`. The colors copied to the CLUT are converted from an 8-bit `argb` format to the format used by the underlying hardware.

The `use count` information used by `uglColorAlloc()` for CLUT entries is not modified by this routine.

**RETURNS**

`UGL_STATUS_OK` (0), or `UGL_STATUS_ERROR` if the operation fails.

**SEE ALSO**

`uglLib`, `uglClutGet()`, `uglColorAlloc()`
uglColorAlloc()

NAME

uglColorAlloc() - allocate colors to be used on a device

SYNOPSIS

UGL_STATUS uglColorAlloc

(  
  UGL_DEVICE_ID devId,  /* driver identifier */
  UGL_ARGB * pAllocColors,  /* list of RGB colors to allocate */
  UGL_ORD * pIndex,  /* clut indexes for allocations */
  UGL_COLOR * pUglColors,  /* device specific color representation */
  UGL_SIZE numColors  /* number of colors to allocate */
)

DESCRIPTION

This routine allocates a set of colors, making them available for use on a device. On indexed color (also called “pseudo color”) devices, the colors are added to the device’s Color Look-Up Table (CLUT). On direct color (also called “true color”) devices no allocation is necessary, and the colors are simply converted to the format used by the device. Both true color and indexed applications should call uglColorAlloc() and uglColorFree() to ensure the code runs properly on both direct color and indexed color devices.

pAllocColors is an array of colors to be allocated. On indexed devices, each entry is reserved in the device’s CLUT in the following manner:

1. The CLUT is searched for a matching color entry that has already been allocated. If a matching entry is found, a use count for that entry is incremented.

2. If no matching entry is found, and the CLUT is not full of allocated colors, an new entry is reserved, and a use count for that entry is set to 1.

3. If no matching entry is found, and the CLUT is full of allocated colors, the CLUT is searched for the color entry that most closely matches the color to be allocated. The used count of this entry is incremented.

pIndex is an array of indexes specifying the location in the CLUT where each color element should be allocated. pIndex is normally NULL, indicating that colors should be allocated in the first available locations. pIndex is unused on direct color devices.

pUglColors points to an array that gets the UGL_COLOR representation of the colors that are allocated. These values are typically indexes into a CLUT for indexed color devices, or the device’s internal color format for direct color devices.

numColors specifies the number of elements in the rgb and ugl color arrays.

RETURNS

UGL_STATUS_OK (0), or a non zero value if the operation fails.

SEE ALSO

uglLib, colorFree(), uglClutGet(), uglClutSet()
uglifyColorConvert() - convert an array of colors from one color format to another

**NAME**

uglifyColorConvert() - convert an array of colors from one color format to another

**SYNOPSIS**

```c
UGL_STATUS uglColorConvert
    (UGL_DEVICE_ID devId,    /* driver identifier */
     void * pSrcArray,
     UGL_COLOR_FORMAT srcFormat,
     void * pDstArray,
     UGL_COLOR_FORMAT dstFormat,
     UGL_SIZE arraySize)
```

**DESCRIPTION**

This routine converts an array of colors, `pSrcArray`, from the color format specified by `srcFormat` to the color format specified by `dstFormat`. The converted colors are stored at `pDstArray`. The parameter `numColors` specifies the number of colors to be converted.

The following `srcFormat` and `dstFormat` specifications are guaranteed to be supported:

**UGL_RGB888** (5)  
Specifies a 24 bit format with 8 bits for red, 8 bits for green, and 8 bits for blue. The colors are stored in a byte array such that 3 bytes represent each color. Red is represented by the first byte, green by the second byte, and blue by the third byte.

**UGL_ARGB8888** (2)  
Specifies a 32 bit format with 8 bits for alpha channel, 8 bits for red, 8 bits for green, and 8 bits for blue. The colors are stored in an array of 32 bit values. Alpha channel is represented in the most significant bits, and blue is represented in the least significant bits.

**UGL_DEVICE_COLOR** (1)  
Specifies the internal format used by the driver. This is typically the same format used by a device’s frame buffer. An application must have information provided by the device driver to properly use this color format.

**UGL_DEVICE_COLOR_32** (0)  
Specifies the UGL_COLOR format returned by `uglColorAlloc()`. This format is the format used internally by the driver. The device specific colors are stored in an array of 32 bit values.

Specific drivers may support additional color conversion.

**RETURNS**

`UGL_STATUS_OK` (0), or a non zero value if the operation fails.

**SEE ALSO**

uglLib
**uglColorFree()**

**NAME**

`uglColorFree()` - frees colors that are currently in use on a device

**SYNOPSIS**

```c
UGL_STATUS uglColorFree
(
    UGL_DEVICE_ID devId,      /* driver identifier */
    UGL_COLOR  *   pColors,    /* colors to free */
    UGL_SIZE      numColors   /* number of colors to free */
)
```

**DESCRIPTION**

This routine frees a set of colors that have been previously allocated by `uglColorAlloc()`.

This routine has no effect on direct color (also called "true color") devices, but its use allows applications to be written to run properly on both direct color and indexed color devices.

`pColors` is an array of colors to be freed, and `numColors` is the number of entries in the array. On indexed color devices, each color is freed from the device’s color lookup table (CLUT) in the following manner:

1. The use count associated with the specified CLUT entry is decremented.
2. If the use count is equal to zero, the CLUT entry is freed by added it to the CLUT’s free list. This makes it available in subsequent calls to `uglColorAlloc()`.

**RETURNS**

`UGL_STATUS_OK` (0), or a non zero value if the operation fails.

**ERRNO**

N/A

**SEE ALSO**

`uglLib`, `uglColorAlloc()`

---

**uglCursorBitmapCreate()**

**NAME**

`uglCursorBitmapCreate()` - creates a cursor bitmap

**SYNOPSIS**

```c
UGL_CDDB_ID uglCursorBitmapCreate
(
    UGL_DEVICE_ID devId,      /* device context */
    UGL_CDIB  *    pCdib       /* DIB for creation of a cursor bitmap */
)
```
This routine creates a cursor device dependent bitmap (CDDB) for use with cursor operations on the device specified by `devId`. The CDDB is constructed using the color information provided by the `pDib` parameter and the transparency mask provided by `pMdib`.

The format of the CDIB is similar to the DIB as described in `uglBitmapCreate()`. The CDIB structure is shown below:

```c
typedef struct ugl_cdib
{
    UGL_SIZE width;
    UGL_SIZE height;
    UGL_SIZE stride;
    UGL_POINT hotSpot;
    UGL_SIZE clutSize; /* size of clut in number of elements */
    UGL_ARGB * pClut; /* color lookup table */
    UGL_UINT8 * pImage; /* image data */
} UGL_CDIB;
```

The arguments are as follows:
- `width` and `height` are the width and height of the image data, in pixels.
- `stride` is the distance, in number of pixels, from the start of one row of image data to the start of the next row. (`stride` is often equal to width, but may be greater than width to align rows on word boundaries or to define a CDIB that is a subset of a larger CDIB.)
- `hotSpot` is the location of the cursor's hot spot relative to the top left corner of the cursor image.
- `clutSize` is the size of the CLUT in number of elements.
- `pClut` is a pointer to the CLUT.
- `pImage` is a pointer to the cursor image data.

The CDIB differs from a DIB in that the image data must be 8 bit indexed data accompanied with a CLUT. This limits the total number of colors available in a cursor to 256. Two of these colors are reserved by WindML as follows:

- `UGL_CURSOR_COLOR_TRANSPARENT` (255)
  The color in the background behind the cursor image will show through.
- `UGL_CURSOR_COLOR_INVERT` (254)
  The color in the background behind the cursor image will be inverted. Note that software cursor images do not support `UGL_CURSOR_COLOR_INVERT`.

Many hardware cursors will only support a few colors. Consult the driver documentation for more details.
uglCursorBitmapDestroy()

NAME

uglCursorBitmapDestroy() - destroy a cursor bitmap

SYNOPSIS

UGL_STATUS uglCursorBitmapDestroy
  
  ( 
    UGL_DEVICE_ID devId,      /* device identifier */
    UGL_CDDB_ID csrBitmapId /* cursor bitmap to destroy */
  )

DESCRIPTION

This routine destroys a bitmap used within cursor operations. All resources used for the
cursor bitmap are freed. Prior to destroying the cursor bitmap the cursor ID must be
destroyed by calling uglCursorDestroy().

RETURNS

UGL_STATUS_OK if success, or UGL_STATUS_ERROR if devId or
csrBitmapId is null or if the driver can not perform the
operation.

ERRNO

N/A

SEE ALSO

uglLib, uglCursorCreate(), uglBitmapCreate()

uglCursorDeinit()

NAME

uglCursorDeinit() - deinitializes the cursor

SYNOPSIS

UGL_STATUS uglCursorDeinit
  
  ( 
    UGL_DEVICE_ID devId       /* driver context */
  )
DESCRIPTION
This routine: 1) removes the cursor image from the display and 2) frees the resources used by a cursor on a specified display device devId.

To obscure a cursor image temporarily, applications should call uglCursorOff().

RETURNS
UGL_STATUS_OK if success, or UGL_STATUS_ERROR if devId is null or if the driver can not perform the operation.

ERRNO
N/A

SEE ALSO
uglLib, uglCursorCreate(), uglCursorBitmapDestroy(), uglCursorOff

---

**uglCursorImageGet()**

NAME
uglCursorImageGet() - gets the image displayed by a cursor

SYNOPSIS
UGL_STATUS uglCursorImageGet(
    UGL_DEVICE_ID devId,       /* device context */
    UGL_CDDB_ID * pImageBitmap /* cursor ID */
)

DESCRIPTION
This routine gets the current cursor image on the specified display device devId. The routine places the cursor's bitmap image in the cursor device dependent bitmap (UGL_CDDB_ID) argument specified by pImageBitmap.

RETURNS
UGL_STATUS_OK if success, or UGL_STATUS_ERROR if devId is null or if the driver can not perform the operation.

ERRNO
N/A

SEE ALSO
uglLib, uglCursorImageSet()

---

**uglCursorImageSet()**

NAME
uglCursorImageSet() - sets the image displayed by a cursor

SYNOPSIS
UGL_STATUS uglCursorImageSet(
    
)
uglCursorInit()

NAME

uglCursorInit() - Initialize the cursor

SYNOPSIS

UGL_STATUS uglCursorInit

(  
  UGL_DEVICE_ID devId,      /* device context */
  UGL_SIZE      maxWidth,   /* maximum allowed size of the cursor */
  UGL_SIZE      maxHeight,
  UGL_POS       xPosition,  /* location of the cursor */
  UGL_POS       yPosition
)

DESCRIPTION

This function initializes the system cursor. Each system may have only one cursor although its associated bitmap can be changed at any time.

The maxWidth and maxHeight arguments specify the maximum size a cursor could obtain on subsequent uglCursorImageSet() calls. Care must be taken to ensure transparent bitmaps that are used in cursor operations do not exceed this maximum size.

The xPosition and yPosition parameters specify the desired initial location of the cursor on the screen. The actual position of the cursor will be offset by the cursor’s specified hotspot.

RETURNS

UGL_STATUS_OK if success, or UGL_STATUS_ERROR if devId is null
or if the driver can not perform the operation.

ERRNO

N/A
SEE ALSO uglLib, uglCursorDestroy(), uglTransBitmapCreate()

uglCursorMove()

NAME uglCursorMove() - sets the position of a cursor

SYNOPSIS UGL_STATUS uglCursorMove
    (UGL_DEVICE_ID devId,       /* device context */
     UGL_POS x,               /* x position of cursor */
     UGL_POS y               /* y position of cursor */
    )

DESCRIPTION This routine moves the cursor cursorID to a new location on the display device devId. The x and y specify the new location of the cursor on the display as the number of pixels for top left of the display. The actual location of the cursor will be offset by the cursor’s hotspot.

RETURNS UGL_STATUS_OK if success, or UGL_STATUS_ERROR if devId is null
or if the driver can not perform the operation.

ERRNO N/A

SEE ALSO uglLib, uglCursorPositionGet()

uglCursorOff()

NAME uglCursorOff() - turns a cursor off

SYNOPSIS UGL_STATUS uglCursorOff
    (UGL_DEVICE_ID devId      /* device context */
    )

DESCRIPTION This routine turns a cursor image off. The specified device’s devId cursor is removed from the display.

RETURNS UGL_STATUS_OK if success, or UGL_STATUS_ERROR if devId is null
or the driver is unable to perform the operation.
uglCursorOn()

NAME uglCursorOn() - turns a cursor on

SYNOPSIS UGL_STATUS uglCursorOn

    {  
        UGL_DEVICE_ID devId       /* device context */  
    }

DESCRIPTION This routine turns on the specified device’s devId cursor, making it visible on the display.

RETURNS UGL_STATUS_OK if success, or UGL_STATUS_ERROR if devId is null

ERRNO N/A

SEE ALSO uglLib, uglCursorOn()

uglCursorPositionGet()

NAME uglCursorPositionGet() - gets the position of a cursor

SYNOPSIS UGL_STATUS uglCursorPositionGet

    {  
        UGL_DEVICE_ID devId,       /* device context */  
        UGL_POS * pX,            /* x position of cursor */  
        UGL_POS * pY             /* y position of cursor */  
    }

DESCRIPTION This routine gets the hotspot position of the specified device’s devId cursor. The position is returned via the pX and pY parameters. This position is relative to the left-top coordinate of the screen.

RETURNS UGL_STATUS_OK if success, or UGL_STATUS_ERROR if devId is null
uglDefaultBitmapGet()

NAME
    uglDefaultBitmapGet() - gets a graphics context's default bitmap

SYNOPSIS
    UGL_STATUS uglDefaultBitmapGet
    (    
        UGL_GC_ID gc,              /* graphics context */
        UGL_DDB_ID * pDefaultBitmapId /* default bitmap ID */
    )

DESCRIPTION
    This routine obtains the current default bitmap for the graphics context, specified by gc
    and places it in the parameter pDefaultBitmap. A value of UGL_DISPLAY_ID for
    pDefaultBitmap specifies the display as the target of drawing operations rendered for the
    graphics context.

RETURNS
    UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null.

ERRNO
    N/A

SEE ALSO
    uglLib, uglDefaultBitmapSet()

uglDefaultBitmapSet()

NAME
    uglDefaultBitmapSet() - sets a graphics context's default bitmap

SYNOPSIS
    UGL_STATUS uglDefaultBitmapSet
    (    
        UGL_GC_ID gc,            /* graphics context */
        UGL_DDB_ID defaultBitmap  /* new default bitmap setting */
    )

DESCRIPTION
    This routine sets the default bitmap for the graphics context gc to the bitmap identifier
    defaultBitmap. This bitmap becomes the target of drawing operations rendered using the
    graphics context specified by gc. A value of UGL_DISPLAY_ID for defaultBitmap specifies
    the display as the target of drawing operations rendered with gc.

SEE ALSO
    uglLib, uglDefaultBitmapGet()
uglDriverFind() - find a driver in the driver registry

NAME
uglDriverFind() - find a driver in the driver registry

SYNOPSIS
UGL_STATUS uglDriverFind
{
    UGL_UINT32   drvType,     /* driver type */
    UGL_UINT32   instance,    /* driver instance */
    UGL_UINT32 * pDriverId    /* driver identifier */
}

DESCRIPTION
This routine finds a driver in the driver registry. The instance instantiation of the driver
type, specified by drvType, is located in the driver registry.

The following values for drvType are supported:

UGL_EVENT_SERVICE_TYPE (1)
UGL_KEYBOARD_TYPE (2)
UGL_POINTER_TYPE (3)
UGL_DISPLAY_TYPE (4)
UGL_PRINTER_TYPE (5)
UGL_SOUND_TYPE (6)
UGL_FRAME_GRABBER_TYPE (7)
UGL_FONT_ENGINE_TYPE (8)
UGL_EVENT_ROUTER_TYPE (9)

The driver identifier is returned in pDriverId and its driver number is returned in pDriverNum.

RETURNS
UGL_STATUS_OK when a driver is found; otherwise UGL_STATUS_ERROR

ERRNO
N/A

SEE ALSO
uglLib
uglDriverRegister() 

NAME 

uglDriverRegister() - register a driver to the device driver registry

SYNOPSIS 

UGL_STATUS uglDriverRegister 

( 
     UGL_UINT32 drvType,      /* driver type */
     UGL_UINT32 instance,     /* driver instance */
     UGL_UINT32 driverId      /* driver identifier */
 )

DESCRIPTION 

This routine adds a driver identifier driverId to the device driver registry. The following drvType are supported:

UGL_EVENT_SERVICE_TYPE
UGL_KEYBOARD_TYPE
UGL_POINTER_TYPE
UGL_DISPLAY_TYPE
UGL_PRINTER_TYPE
UGL_SOUND_TYPE
UGL_FRAME_GRABBER_TYPE
UGL_FONT_ENGINE_TYPE
UGL_EVENT_ROUTER_TYPE

The instance argument specifies a previously created driver. The driverId argument identifies the driver control structure, which has a variable format based upon the driver type.

NOTE 

There are no safeguards against adding the same driver/instance multiple times. If this is done, then the first entry will be the returned entry on a find.

RETURNS 

UGL_STATUS_OK when successfully added; otherwise UGL_STATUS_ERROR.

ERRNO 

N/A

SEE ALSO 

ugLib
**uglDriverUnRegister()**

**NAME**

`uglDriverUnRegister()` - unregister a driver

**SYNOPSIS**

```c
UGL_STATUS uglDriverUnRegister(
    UGL_UINT32 drvType,       /* driver type */
    UGL_UINT32 instance       /* driver instance */
);
```

**DESCRIPTION**

This routine removes a driver from the driver registry. The specified instance of the driver, identified by `drvType`, is removed from the driver registry.

**RETURNS**

- `UGL_STATUS_OK` when a driver is found and removed;
- `UGL_STATUS_ERROR` otherwise.

**ERRNO**

N/A

**SEE ALSO**

`uglLib`, `uglDriverRegister()`

---

**uglEllipse()**

**NAME**

`uglEllipse()` - draws an ellipse, arc, or pie slice

**SYNOPSIS**

```c
UGL_STATUS uglEllipse(
    UGL_GC_ID gc,             /* graphics context */
    UGL_POS left,            /* left boundary of bounding rectangle */
    UGL_POS top,             /* top boundary of bounding rectangle */
    UGL_POS right,           /* right boundary of bounding rectangle */
    UGL_POS bottom,          /* bottom boundary of bounding rectangle */
    UGL_POS startX,          /* x location of point defining starting angle */
    UGL_POS startY,          /* y location of point defining starting angle */
    UGL_POS endX,            /* x location of point defining ending angle */
    UGL_POS endY             /* y location of point defining ending angle */
);
```

**DESCRIPTION**

This function draws an ellipse, arc, or pie slice. The ellipse is outlined with the foreground color of graphics context gc. If a fill pattern has been set in gc, the ellipse is filled with the pattern bitmap; otherwise it is filled with the background color.

The parameters left, top, right, and bottom specify the bounding rectangle of the ellipse. A radial line drawn from the center of the ellipse to the coordinate (startX, startY) defines the starting point of an arc. A radial line drawn from the center of the ellipse to the coordinate (endX, endY) defines the ending point of an arc.

If (startX,startY) and (endX, endY) specify the same point, the entire ellipse is drawn. Otherwise, an arc or pie slice is drawn. (A pie slice is defined as a filled arc.)

The graphics context gc specifies the details of how the ellipse, arc, or pie slice is to be drawn. The graphics context is set to define:

- **defaultBitmap**
  Specifies the bitmap to which rendering will occur. 
  **UGL_DISPLAY_ID** specifies the display. Set by the `uglDefaultBitmapSet()` function.

- **clipRect**
  Specifies a clipping rectangle. No drawing may take place outside of this rectangle. Set by the `uglClipRectSet()` function.

- **rasterOp**
  Specifies the raster operation (COPY, XOR, and so on) that is used. Set by the `uglRasterModeSet()` function.

- **foregroundColor**
  Specifies the color of the ellipse outline. Set by the `uglForegroundColorSet()` function.

- **lineStyle**
  Specifies the line style (solid, dashed, and so on) of the ellipse outline. Set by the `uglLineStyleSet()` function.

- **lineWidth**
  Specifies the width, in pixels, of the ellipse outline. Set by the `uglLineWidthSet()` function.

- **backgroundColor**
  Specifies the fill color for the ellipse. Set by the `uglBackgroundColorSet()` function.

- **patternBitmap**
  Specifies the fill pattern for the ellipse. Set by the `uglFillPatternSet()` function.

If an outlined ellipse is not desired, the foreground color should be set to **UGL_COLOR_TRANSPARENT** or the line width should be set to zero. The ellipse is not filled if the background color is **UGL_COLOR_TRANSPARENT**.
**uglFillPatternGet()**

**NAME**

*uglFillPatternGet()* - get a gc’s fill pattern setting

**SYNOPSIS**

```c
UGL_STATUS uglFillPatternGet
{
    UGL_GC_ID     gc,            /* graphics context */
    UGL_MDDB_ID * pPatternBitmap /* pattern bitmap ID */
}
```

**DESCRIPTION**

This routine gets the current fill pattern setting of the graphics context gc. The fill pattern bitmap identifier is written to *pPatternBitmap*. A NULL value indicates a solid pattern.

**RETURNS**

UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null.

**ERRNO**

N/A

**SEE ALSO**

uglLib, *uglFillPatternSet()*

---

**uglFillPatternSet()**

**NAME**

*uglFillPatternSet()* - sets a gc’s fill pattern

**SYNOPSIS**

```c
UGL_STATUS uglFillPatternSet
{
    UGL_GC_ID     gc,            /* graphics context */
    UGL_MDDB_ID  patternBitmap /* new pattern bitmap */
}
```
DESCRIPTION
This routine sets the fill pattern for the graphics context gc to the pattern bitmap specified by the monochrome bitmap patternBitmap. A NULL value for patternBitmap indicates a solid fill.

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null.

ERRNO
N/A

SEE ALSO
uglLib, uglFillPatternGet()

uglFontCreate()

NAME
uglFontCreate() - Create a font

SYNOPSIS
UGL_FONT_ID uglFontCreate
{
    UGL_FONT_DRIVER_ID fontDriverId, /* font driver */
    UGL_FONT_DEF * pFontDefinition /* definition used to create the font */
}

DESCRIPTION
This routine creates a font to be used by the application. This routine does not provide nearest matching. Thus, all parameters passed in the UGL_FONT_DEF structure must match an available font. See uglFontFind(), uglFontFindString(), and uglFontList() for nearest matching.

The UGL_FONT_DEF structure is defined as follows:

typedef struct ugl_font_def
{
    UGL_SIZE structSize;
    UGL_SIZE pixelSize;
    UGL_ORD weight;
    UGL_ORD italic;
    UGL_ORD charSet;
    char faceName[UGL_FONT_FACE_NAME_MAX_LENGTH];
    char familyName[UGL_FONT_FAMILY_NAME_MAX_LENGTH];
} UGL_FONT_DEF;

The fields in this structure are:

structSize is the size of the font definition structure in bytes. This is required in cases where the application and driver use an extended font definition structure. Normally this can be assigned a value of sizeof(UGL_FONT_DEF);
**pixelSize** is the average pixel size of the font. Scaling of fonts may also be performed after font creation by **uglFontSizeSet()**. However, this routine may not always be supported by a particular font driver.

**weight** determines the amount of emboldening for the font. UGL defines a bold range that has a maximum of 100 and a minimum of 0, where 0 indicates text with no emboldening, or “book weight”. UGL defines the following constants:

- **UGL_FONT_BOLD_OFF** = 0
- **UGL_FONT_BOLD_MIN** = 1
- **UGL_FONT_BOLD_LIGHT** = 25
- **UGL_FONT_BOLD** = 50
- **UGL_FONT_BOLD_HEAVY** = 75
- **UGL_FONT_BOLD_MAX** = 100

Scaling of weight may also be performed after font creation by calling **uglFontWeightSet()**. However, this routine may not always be supported by a particular font.

**italic** determines if the created font is italic. Supported values include:

- **UGL_FONT_ITALIC**
- **UGL_FONT_UPRIGHT**

**charSet** can be any one of the following choices:

- **UGL_FONT_ISO_8859_1**
- **UGL_FONT_UNICODE**

Individual font drivers may support additional character sets. UGL does not provide translation or mapping support between different character sets. The application must therefore insure that the encoding matches the selected font.

**RETURNS**

- **UGL_STATUS_OK** (0), or a non zero value if the operation fails.

**SEE ALSO**

- **uglLib**, **uglFontDestroy()**, **uglFontFind()**, **uglFontFindString()**, **uglFontList()**
uglFontDestroy()

NAME    uglFontDestroy() - Destroy a font

SYNOPSIS
UGL_STATUS uglFontDestroy
    (    
        UGL_FONT_ID fontId        /* allocated font */
    )

DESCRIPTION
This routine destroys a previously created font.

RETURNS
UGL_STATUS_OK (0), or a non zero value if the operation fails.

SEE ALSO
uglLib, uglFontCreate()

uglFontDriverDestroy()

NAME    uglFontDriverDestroy() - Destroy a font driver

SYNOPSIS
UGL_STATUS uglFontDriverDestroy
    (    
        UGL_FONT_DRIVER_ID fontDriverId /* font driver */
    )

DESCRIPTION
This routine destroys all the internal data structures associated with a particular font driver. Any fonts (fontIds) previously created with the font driver should be destroyed before this routine is called.

RETURNS
UGL_STATUS_OK (0), or a non zero value if the operation fails.

SEE ALSO
uglLib, N/A
uglFontDriverInfo()

NAME

uglFontDriverInfo() - Retrieve or send information to a font driver

SYNOPSIS

UGL_STATUS uglFontDriverInfo

(  UGL_FONT_DRIVER_ID fontDriverId, /* font driver */
   UGL_INFO_REQ infoRequest, /* information request */
   void * pInfo         /* pointer to receiving data */
)

DESCRIPTION

This routine retrieves or sends information to the font driver. The information and options may be driver specific. The font driver contains information that may be needed by an application or the UGL 2D font layer. The parameter infoRequest indicates which category of information is requested. The font driver must collect the requested information and return it via a predetermined data structure specified by the pInfo parameter.

The calling routine must ensure that the location specified by pInfo has sufficient space to contain the entire data structure that is associated with the information request.

RETURNS

UGL_STATUS_OK (0), or a non zero value if the operation fails.

SEE ALSO

uglLib, N/A

uglFontFind()

NAME

uglFontFind() - Find a specific font

SYNOPSIS

UGL_STATUS uglFontFind

(  UGL_FONT_DRIVER_ID fontDriverId, /* font driver */
   UGL_FONT_DESC * pFontDescriptor, /* description of font to find */
   UGL_FONT_DESC_PRIORITY * pFontDescPriority, /* search field priority */
   UGL_FONT_DEF * pFontDefinition /* resulting font match */
)

DESCRIPTION

This routine returns an UGL_FONT_DEF structure that matches the descriptor and priority information passed by pFontDescriptor and pFontDescPriority. The resulting font
definition is contained in pFontDefinition and can be used in subsequent calls to uglFontCreate().

The following information can be specified in pFontDescriptor (fields in this structure do not need to be specified if the associated priority for the field is UGL_FONT_DONT_CARE):

```c
typedef struct ugl_font_desc
{
    UGL_RANGE pixelSize;
    UGL_RANGE weight;
    UGL_ORD italic;
    UGL_ORD spacing;
    UGL_ORD charSet;
    char faceName[UGL_FONT_FACE_NAME_MAX_LENGTH];
    char familyName[UGL_FONT_FAMILY_NAME_MAX_LENGTH];
} UGL_FONT_DESC;
```

*pixelSize* contains the minimum and maximum (min and max), pixel sizes that are possible for the font. If min and max are equal, only a single pixel size is accepted for the specified font.

*weight* contains the range of emboldening possible for the font. UGL defines a bold range that has a minimum value of 0 and maximum value of 100, where 0 indicates text with no emboldening, or “book weight”. UGL defines the following constants:

- UGL_FONT_BOLD_OFF = 0
- UGL_FONT_BOLD_MIN = 1
- UGL_FONT_BOLD_LIGHT = 25
- UGL_FONT_BOLD = 50
- UGL_FONT_BOLD_HEAVY = 75
- UGL_FONT_BOLD_MAX = 100

If the minimum and maximum pixel sizes for *weight* are equal, only a single weight is accepted for the specified font. The application may also choose values in between the defined constants when creating the font, if the underlying font driver allows such specification.

*italic* may be set to one of the following values:

- UGL_ITALIC
- UGL_UPRIGHT

*spacing* may be set to one of the following values:
UGL_MONO_SPACED

UGL_PROPORTIONAL

charSet can be set to one of the following values:

UGL_ISO_8859_1

UGL_UNICODE

The UGL_FONT_DESC_PRIORITY structure is defined as follows:

```c
typedef struct ugl_font_desc_priority
{
    UGL_ORD pixelSize;
    UGL_ORD weight;
    UGL_ORD italic;
    UGL_ORD spacing;
    UGL_ORD charSet;
    UGL_ORD faceName;
    UGL_ORD familyName;
} UGL_FONT_DESC_PRIORITY;
```

Each field in this structure may either contain a value of UGL_FONT_DONT_CARE (0) or an incremental priority of 1..N, where N is the highest priority match. The uglFontFind() routine evaluates the priority placed on font information (e.g., pixel size, weight) and attempts to find the closest matching font that satisfies the description constraints with the specified priority.

RETURNS UGL_STATUS_OK if a match is found, otherwise UGL_STATUS_ERROR.

SEE ALSO uglLib, uglFontCreate(), uglFontFindString()

---

**uglFontFindClose()**

NAME uglFontFindClose() - Free memory used by first/next font routines

SYNOPSIS

```c
UGL_STATUS uglFontFindClose
(
    UGL_FONT_DRIVER_ID fontDriverId, /* font driver */
    UGL_SEARCH_ID      searchId      /* search identifier */
);
```

DESCRIPTION This routine frees any memory that has been allocated by performing prior calls to the uglFontFindFirst() and uglFontFindNext() routines.
uglFontFindFirst()

NAME

uglFontFindFirst() - Obtain the first available font

SYNOPSIS

UGL_SEARCH_ID uglFontFindFirst

{
    UGL_FONT_DRIVER_ID fontDriverId, /* font driver */
    UGL_FONT_DESC * pFontDescriptor /* description of font to find */
}

DESCRIPTION

This routine begins the process of enumerating fonts that are supported by the specified font driver. This routine returns the first available font via the pFontDescriptor parameter and returns an UGL_SEARCH_ID value. This value is typically a non-null value, if the routine is successful. The return value should be passed as the searchId argument on subsequent calls to uglFontFindNext().

If the UGL_SEARCH_ID is NULL, the call to this routine was not successful, indicating no additional fonts can be processed.

After the user enumerates all of the fonts, it must call uglFontFindClose() to free any memory allocated by the driver to perform the font enumeration process.

The UGL_FONT_DESC structure is defined as follows:

    typedef struct ugl_font_desc
    {
        UGL_RANGE pixelSize;
        UGL_RANGE weight;
        UGL_ORD italic;
        UGL_ORD spacing;
        UGL_ORD charSet;
        char faceName[UGL_FONT_FACE_NAME_MAX_LENGTH];
        char familyName[UGL_FONT_FAMILY_NAME_MAX_LENGTH];
    } UGL_FONT_DESC;

The UGL_RANGE structure (used for pixelSize and weight) is defined as follows:

    typedef struct ugl_range
    {
        UGL_SIZE max;
        UGL_SIZE min;
    } UGL_RANGE;
pixelSize contains the minimum and maximum pixel sizes that are possible for the font. It is possible for min and max to be equal. In this case, only a single pixel size is permitted for the specified font. Fractional pixel sizes are not allowed.

weight contains the range of emboldening possible for the font. UGL defines a bold range that has a minimum value of 0 and maximum value of 100, where 0 indicates text with no emboldening, or “book weight”. UGL defines the following constants:

- `UGL_FONT_BOLD_OFF = 0`
- `UGL_FONT_BOLD_MIN = 1`
- `UGL_FONT_BOLD_LIGHT = 25`
- `UGL_FONT_BOLD = 50`
- `UGL_FONT_BOLD_HEAVY = 75`
- `UGL_FONT_BOLD_MAX = 100`

The minimum and maximum pixel sizes for weight may be equal. The application may also choose values in between the defined constants when creating the font, if the underlying font driver allows such specification.

italic may be set to one of the following values:

- `UGL_ITALIC`
- `UGL_UPRIGHT`

spacing may be set to one of the following values:

- `UGL_MONO_SPACED`
- `UGL_PROPORTIONAL`

charSet can be set to one of the following values:

- `UGL_ISO_8859_1`
- `UGL_UNICODE`

Individual font drivers may support additional character sets. However, UGL does not provide translation or mapping support between different character sets. The application must ensure that the encoding matches the selected font.

scalable allows the application to determine if the font is scalable. This is a binary (UGL_TRUE/UGL_FALSE) variable.

The face name of the font is returned in `faceName`. Note that `faceName` may imply certain information about a font such as spacing, serifed or not, etc. The family name of the font is returned in `faceName`.

**RETURNS**

A non zero value if the operation is successful or `UGL_NULL` if the operation fails.
uglFontFindNext()

NAME
uglFontFindNext() - Obtain the next available font.

SYNOPSIS
UGL_STATUS uglFontFindNext

    
    ( 
        UGL_FONT_DRIVER_ID fontDriverId,    /* font driver */
        UGL_FONT_DESC *    pFontDescriptor, /* descriptor of font to find */
        UGL_SEARCH_ID      searchId         /* search result from last find */
    
)

DESCRIPTION
This routine enumerates the next font available in the specified font driver fontDriverId. It should be invoked after first calling uglFontFindFirst(). This routine is normally called in a loop until it returns UGL_STATUS_FINISHED.

The application must call uglFontFindClose() after all fonts have been enumerated to free any memory allocated by the driver.

RETURNS
UGL_STATUS_OK (0), UGL_STATUS_FINISHED (1) if last UGL_FONT_DESC was returned, or UGL_STATUS_ERROR if the operation fails.

SEE ALSO
uglLib, uglFontFindFirst(), uglFontFindClose()

uglFontFindString()

NAME
uglFontFindString() - find a specific font with a search string

SYNOPSIS
UGL_STATUS uglFontFindString

    
    ( 
        UGL_FONT_DRIVER_ID fontDriverId,    /* font driver */
        const char *       pFontPriorityString, /* prioritized search string */
        UGL_FONT_DEF *     pFontDefinition      /* resulting font match */
    
)

DESCRIPTION
This routine returns an UGL_FONT_DEF structure that matches the priority string specified in pFontPriorityString. The resulting font definition is contained in pFontDefinition and can be used in subsequent calls to uglFontCreate().
The format of the search string is:
"pixelSize=<pixelSize>;faceName=<faceName>;familyName=<familyName>;
italic;[mono|proportional];bold"

Where the order of the attributes in the string determines the search priority. White space in the string is ignored. See uglFontFind() for more information on the arguments that can be specified in the search string.

RETURNS UGL_STATUS_OK if a match is found, otherwise UGL_STATUS_ERROR.

SEE ALSO uglLib, uglFontCreate(), uglFontFind()
DESCRIPTION

This routine returns the metrics associated with a particular font. The information provided in the font metrics structure is:

```c
typedef struct ugl_font_metrics
{
    UGL_SIZE pixelSize;
    UGL_SIZE weight;
    UGL_ORD italic;
    UGL_SIZE height;
    UGL_SIZE maxAscent;
    UGL_SIZE maxDescent;
    UGL_SIZE maxAdvance;
    UGL_SIZE leading;
    UGL_ORD spacing;
    UGL_ORD fontType;
    UGL_ORD charSet;
    UGL_BOOL scalable;
    char faceName[UGL_FONT_FACE_NAME_MAX_LENGTH];
    char familyName[UGL_FONT_FAMILY_NAME_MAX_LENGTH];
} UGL_FONT_METRICS;
```

- `pixelSize` is the average pixel size of the font.
- `weight` is the amount of emboldening for the font, which has a minimum-maximum range of 0-100 (UGL_FONT_BOLD_OFF..UGL_FONT_BOLD_MAX).
- `italic` indicates if the created font is italic (UGL_FONT_ITALIC) or upright (UGL_FONT_UPRIGHT).
- `height` is the total of `maxAscent` and `maxDescent`.
- `maxAscent` specifies the maximum number of pixels (units above the baseline) required to represent any character in the font.
- `maxDescent` specifies the maximum number of pixels (units below the baseline) required to represent any character in the font.
- `maxAdvance` specifies the maximum space needed to render a character on the display. This includes the space needed between characters.
- `leading` specifies the amount of extra leading (space) that the application adds between rows. This area is outside the font, so it contains no marks and is not altered by text output calls.
- `spacing` specifies if the font is mono (UGL_MONO_SPACED) or proportionally (UGL_PROPORTIONAL) spaced.
- `fontType` specifies whether the font is in a bitmapped form (UGL_FONT_BITMAPPED) or a true type font (UGL_FONT_TRUE_TYPE).
- `charSet` specifies either an ISO (UGL_ISO_8859_1) or unicode (UGL_UNICODE) encoding.
scaleable specifies whether the font can be scaled to a number of different pixel sizes such as
12pt, 14pt, and 24pt in addition to the size indicated by pixelSize.

faceName identifies a face name

familyName identifies a commonly understood font name (e.g., Helvetica, Courier, Times).

RETURNS  
UGL_STATUS_OK (0), or a non zero value if the operation fails.

SEE ALSO  
uglLib, N/A

uglFontRotationAngleSet()

NAME  
uglFontRotationAngleSet() - set the rotation angle of a font

SYNOPSIS  
UGL_STATUS uglFontRotationAngleSet
  (  
    UGL_FONT_ID fontId,
    UGL_ORD     angle
  )

DESCRIPTION  
This routine allows the user to set the rotation angle for a particular font. Note that this
routine will only work if the font driver supports the rotation operation. Consult to font
driver documentation for further details.

RETURNS  
UGL_STATUS_OK (0), or a non zero value if the operation fails.

SEE ALSO  
uglLib, N/A

uglFontSet()

NAME  
uglFontSet() - sets a gc's font

SYNOPSIS  
UGL_STATUS uglFontSet
  (  
    UGL_GC_ID   gc,           /* graphics context */
    UGL_FONT_ID fontId        /* new font identifier */
  )
DESCRIPTION
This routine sets the font for the specified gc to the font id specified by fontId. The fontId argument must contain a font identifier that was received by calling uglFontCreate().

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc or fontId is null.

SEE ALSO
uglLib, uglFontGet(), uglFontCreate()

---

uglFontSizeSet()

NAME
uglFontSizeSet() - Set the pixel size of a font

SYNOPSIS
UGL_STATUS uglFontSizeSet
{
    UGL_FONT_ID fontId,
    UGL_SIZE pixelSize
}

DESCRIPTION
This routine allows the user to set the pixel size of a particular font. Note that this routine will only work if the font is scalable. Normally bitmapped fonts are not scalable.

Consult the font driver documentation to determine the actual range of pixel sizes that are supported for individual fonts. Normally the range will be greater than zero and less than 100.

RETURNS
UGL_STATUS_OK (0), or a non zero value if the operation fails.

SEE ALSO
uglLib, uglFontMetricsGet()

---

uglFontWeightSet()

NAME
uglFontWeightSet() - Set the weight or bold of a font

SYNOPSIS
UGL_STATUS uglFontWeightSet
{
    UGL_FONT_ID fontId,
    UGL_SIZE weight
}
This routine allows the user to set the weight of a particular font. Note that this routine will only work if the font is scalable. Normally bitmapped fonts are not scalable. Consult the font driver documentation to determine the actual range of weights that are supported for individual fonts.

**Returns**

UGL_STATUS_OK (0), or a non zero value if the operation fails.

**See Also**

uglLib, uglFontMetricsGet()

---

### uglForegroundColorGet()

**Name**

uglForegroundColorGet() - gets a gc’s foreground color setting

**Synopsis**

UGL_STATUS uglForegroundColorGet

```
( UGL_GC_ID   gc,              /* graphics context */
   UGL_COLOR * pForegroundColor /* forground color setting */
)
```

**Description**

This routine gets the foreground color of graphics context gc. The foreground color is written to the location specified by the parameter pForegroundColor.

**Returns**

UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null.

**Errno**

N/A

**See Also**

uglLib, uglForegroundColorSet()

---

### uglForegroundColorSet()

**Name**

uglForegroundColorSet() - sets a gc’s foreground color

**Synopsis**

UGL_STATUS uglForegroundColorSet

```
( UGL_GC_ID gc,             /* graphics context */
   UGL_COLOR color           /* new foreground color */
)
```
### uglGcCopy()

**NAME**  
`uglGcCopy()` - copies a graphics context

**SYNOPSIS**  
```c
UGL_STATUS uglGcCopy
(  
UGL_GC_ID srcGc, /* source graphics context */  
UGL_GC_ID dstGc /* destination graphics context */  
)
```

**DESCRIPTION**  
This routine copies the contents of a graphics context `srcGc` to another graphics context `dstGc`. Both the source and destination graphics contexts must previously exist. After the copy operation the `dstGc` is identical to the `srcGc` with the exception that each graphics context will have its own lock identifier.

**RETURNS**  
`UGL_STATUS_OK`, or `UGL_STATUS_ERROR` if the `srcGc` or `dstGc` are null.

**ERRNO**  
N/A

**SEE ALSO**  
`uglLib`, `uglForegroundColorGet()`

### uglGcCreate()

**NAME**  
`uglGcCreate()` - creates a graphics context

**SYNOPSIS**  
```c
UGL_GC_ID uglGcCreate
(  
UGL_DEVICE_ID devId /* Output device ID */  
)
```

**DESCRIPTION**  
This routine sets the foreground color of the graphics context specified by the parameter `gc` to the color specified by the parameter `color`. The foreground color is used during UGL rendering operations such as drawing lines, outlines for shapes, and text.

**RETURNS**  
`UGL_STATUS_OK`, or `UGL_STATUS_ERROR` if the `gc` is null.

**ERRNO**  
N/A

**SEE ALSO**  
`uglLib`, `uglForegroundColorGet()`
DESCRIPTION
This routine creates a graphics context for use in subsequent rendering operations. The graphics context (GC) defines the manner in which a graphics rendering operation is performed. For instance, the `uglLine()` operation uses the defaultBitmap, clipRect, rasterOp, foregroundColor, lineStyle, and lineWidth portions of the gc to render a line.

The created graphics context is directly associated with the specified graphics device. The graphics context provides definitions on:
- clipping region and rectangle
- view port rectangle
- background and foreground color
- line style and width
- fill patterns
- default bitmap
- font for text rendering
- current raster operation

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc or devId are null.

ERRNO
N/A

SEE ALSO
uglLib

---

**uglGcDestroy()**

NAME
`uglGcDestroy()` - destroys a graphics context

SYNOPSIS
```c
UGL_STATUS uglGcDestroy(UGL_GC_ID gc); /* graphics context */
```

DESCRIPTION
This routine destroys the graphics context gc. All memory resources and GC locks associated with the graphics context are freed. Once the gc has been destroyed, it cannot be used in any future rendering operations.

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null.

ERRNO
N/A
NAME
uglInfo() - retrieves information from the driver

SYNOPSIS
UGL_STATUS uglInfo
{
    UGL_DEVICE_ID   devId,       /* device identifier */
    UGL_INFO_REQ    infoRequest, /* request type */
    void *          pInfo        /* location to store requested info */
}

DESCRIPTION
This routine retrieves information from the graphics device driver. The device driver contains information that may be needed by an application or the UGL 2D layer. Information maintained by the device driver may be divided into categories. The parameter infoRequest indicates which category of information is required. The device driver collects that information and returns it using an appropriate data structure at the location specified by the pInfo parameter.

The infoRequest value UGL_MODE_INFO is supported by all graphics device drivers:

UGL_MODE_INFO
Obtains general information concerning the organization of the frame buffer. The information is returned within the uglModeInfo data structure. Elements that are returned with this request are:

width
height
frame buffer address
amount of display memory
format of the frame buffer
size of the color lookup table
various flags

It is up to the routine that calls this function to ensure that the location specified by pInfo has sufficient space to contain the entire data structure that is associated with the request infoRequest.

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if the operation fails.

ERRNO
N/A

SEE ALSO
uglLib
uglLine()

NAME

uglLine() - draws a line

SYNOPSIS

UGL_STATUS uglLine

(  
  UGL_GC_ID gc,             /* the graphics context */
  UGL_POS   x1,             /* the x-coordinate for the starting point */
  UGL_POS   y1,             /* the y-coordinate for the starting point */
  UGL_POS   x2,             /* the x-coordinate for the ending point */
  UGL_POS   y2              /* the y-coordinate for the ending point */
  )

DESCRIPTION

This routine draws a line from the point (x1,y1) to the point (x2,y2). Both points are included within the line.

The graphics context gc specifies the details of how the line is to be drawn. The graphics context is set to define:

defaultBitmap
  Specifies the bitmap to which rendering will occur. UGL_DISPLAY_ID specifies the display. Set by the uglDefaultBitmapSet() function.

clipRect
  Specifies a clipping rectangle. No drawing may take place outside of this rectangle. Set by the uglClipRectSet() function.

rasterOp
  Specifies the raster operation (COPY, XOR, and so on) that is used. Set by the uglRasterModeSet() function.

glColor
  Specifies the color of the rectangle outline. Set by the uglForeColorSet() function.

lineStyle
  Specifies the line style (solid, dashed, and so on) of the rectangle outline. Set by the uglLineStyleSet() function.

lineWidth
  Specifies the width, in pixels, of the rectangle outline. Set by the uglLineWidthSet() function.

RETURNS

UGL_STATUS_OK, or UGL_STATUS_ERROR if gc is null or the driver is unable to perform the rendering operation.

ERRNO

N/A
**uglLineStyleGet()**

**NAME**  
uglLineStyleGet() - gets a gc’s line style

**SYNOPSIS**  
UGL_STATUS uglLineStyleGet
  (  
    UGL_GC_ID        gc,        /* graphics context */
    UGL_LINE_STYLE * pLineStyle /* line style setting */
  )

**DESCRIPTION**  
This routine returns the current settings of the line style in the graphics context gc to the calling routine. The information is stored in the parameter pLineStyle.

**RETURNS**  
UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null.

**ERRNO**  
N/A

**SEE ALSO**  
uglLib, uglLineStyleSet()

**uglLineStyleSet()**

**NAME**  
uglLineStyleSet() - sets a gc’s line style

**SYNOPSIS**  
UGL_STATUS uglLineStyleSet
  (  
    UGL_GC_ID        gc,        /* graphics context */
    UGL_LINE_STYLE lineStyle  /* new line style */
  )

**DESCRIPTION**  
This routine sets the line style of the graphics context specified by gc to the line style specified by lineStyle. The parameter lineStyle identifies a 32 bit pattern that is repeated when a line is drawn. A bit set to a one causes the line to be displayed and a value of 0 causes the line not to be displayed. Thus, a value of 0xffffffff will cause a solid line to be drawn. The following predefined values are available:

**RETURNS**  
UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null.
uglLineWidthGet()  

NAME

uglLineWidthGet() - get a gc’s line width setting

SYNOPSIS

UGL_STATUS uglLineWidthGet

(  
  UGL_GC_ID gc,            /* graphics context */  
  UGL_SIZE * pLineWidth     /* line width setting */  
)

DESCRIPTION

This routine returns the current settings of the line width (in pixels) in the graphics context gc to the calling routine. The information is stored in the parameter pLineWidth.

RETURNS

UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null.

ERRNO

N/A

SEE ALSO

uglLib, uglLineWidthSet()

uglLineWidthSet()

NAME

uglLineWidthSet() - sets a gc’s line width

SYNOPSIS

UGL_STATUS uglLineWidthSet

(  
  UGL_GC_ID gc,            /* graphics context */  
  int lineWidth       /* new line width */  
)
DESCRIPTION
This routine sets the line width (in pixels) of graphics context gc to the value specified by the parameter lineWidth.

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null.

ERRNO
N/A

SEE ALSO
ugLib, uglLineWidthGet()

---

 uglMemDefaultPoolGet()

NAME
uglMemDefaultPoolGet() - get the default memory pool

SYNOPSIS
UGL_STATUS uglMemDefaultPoolGet
(  
  UGL_MEM_POOL_ID * pPoolId
)

DESCRIPTION
This gets the ID of default memory pool and places it in pPoolId. The default * memory pool is the pool from which UGL allocates memory, and the pool used by UGL_MALLOC(), or uglMemAlloc() when poolId is NULL.

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if <pPoolId is NULL.

SEE ALSO
ugLib, uglMemDefaultPoolSet()

---

 uglMemDefaultPoolSet()

NAME
uglMemDefaultPoolSet() - set the default memory pool

SYNOPSIS
UGL_STATUS uglMemDefaultPoolSet
(  
  UGL_MEM_POOL_ID poolId
)

DESCRIPTION
This routing sets the default memory pool to \texttt{poolId}. The default memory pool is the pool from which UGL allocates memory, and the pool used by \texttt{UGL_MALLOC()}, or \texttt{uglMemAlloc()} when \texttt{poolId} is NULL.

RETURNS
\texttt{UGL_STATUS_OK}, or \texttt{UGL_STATUS_ERROR} if \texttt{poolId} is NULL.

SEE ALSO
\texttt{uglLib}, \texttt{uglMemAlloc()}, \texttt{UGL_MALLOC()}

\textbf{uglMemDevicePoolCreate()}

NAME
\texttt{uglMemDevicePoolCreate()} - create a device memory pool

SYNOPSIS
\begin{verbatim}
UGL_DEVICE_MEM_POOL_ID uglMemDevicePoolCreate
{
    UGL_DEVICE_ID devId,
    UGL_SIZE      poolSize,
    UGL_BOOL      displayMem
}
\end{verbatim}

DESCRIPTION
This routine creates a memory pool for use by the display device. If \texttt{displayMem} is true, the memory pool is allocated from display memory not used by the visible frame buffer. If \texttt{displayMem} is false then the memory pool is allocated from the default memory pool for local devices.

RETURNS
A device memory pool ID, or \texttt{UGL_NULL} if memory is insufficient.

SEE ALSO
\texttt{uglLib}, \texttt{uglMemDevicePoolDestroy()}

\textbf{uglMemDevicePoolDestroy()}

NAME
\texttt{uglMemDevicePoolDestroy()} - destroy a device memory pool.

SYNOPSIS
\begin{verbatim}
UGL_STATUS uglMemDevicePoolDestroy
{
    UGL_DEVICE_ID          devId,
    UGL_DEVICE_MEM_POOL_ID devPoolId
}
\end{verbatim}

306
DESCRIPTION
This routine destroys a device memory pool previously created by 
`uglMemDevicePoolCreate()`. Any bitmaps or other allocations made in a pool should be 
freed before the pool is destroyed.

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR upon failure.

SEE ALSO
`uglLib`, `uglMemDevicePoolCreate()`;

---

`uglModeAvailGet()`

NAME
`uglModeAvailGet()` - gets the available graphics device modes

SYNOPSIS
```c
UGL_STATUS uglModeAvailGet(
    UGL_DEVICE_ID      devId,     /* device identifier */
    UGL_UINT32 *       pNumModes, /* number of supported modes */
    const UGL_MODE * * ppMode     /* graphics device mode */
);
```

DESCRIPTION
This routine identifies the operating modes that may be used for the graphics output 
device. The operating mode includes the resolution, refresh rate, color depth, and 
graphics device type.

The type and characteristics that the graphics device may be initialized are defined by the 
UGL_MODE data structure provide by the `ppMode` parameter. This parameter is an array 
of modes where the number of modes (that is, size of the array), is defined by the 
`pNumModes` parameter. This UGL_MODE data structure identifies the following:

- **width**
  Identifies the width of the display in pixels

- **height**
  Identifies the height of the display in pixels.

- **refreshRate**
  Identifies the refresh rate that the device uses.

- **monitorType**
  Identifies the type of graphics device, such as TFT flat panel, 
  EL flat panel, or generic CRT

- **colorDepth**
  Identifies the color depths that the mode supports.

- **flags**
  Identifies additional information about the operating mode
uglModeAvailPrint()

NAME
uglModeAvailPrint() - prints the available graphics device modes

SYNOPSIS
UGL_STATUS uglModeAvailPrint
{
    UGL_UINT32 numModes, /* number of supported modes */
    const UGL_MODE * pModes /* graphics device mode */
}

DESCRIPTION
This routine prints the available output modes to standard out.

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if the operation fails

ERRNO
N/A

SEE ALSO
uglLib, uglModeSet()

uglModeNearestGet()

NAME
uglModeNearestGet() - gets the closest mode that graphics device supports

SYNOPSIS
UGL_STATUS uglModeNearestGet
{
    UGL_MODE * pOutMode, /* desired graphics mode */
    UGL_UINT32 * pColorDepth, /* desired color depth */
    UGL_UINT32 colorDepths, /* available color depths */
    UGL_UINT32 numModes, /* number of available modes */
    UGL_MODE * pModes /* list of available modes */
}

DESCRIPTION
This routine determines the closest graphics mode to the modes the device supports.
uglModeSet()

NAME

uglModeSet() - sets the graphics device mode

SYNOPSIS

UGL_STATUS uglModeSet
{
    UGL_DEVICE_ID devId, /* device identifier */
    UGL_MODE * pMode /* display mode */
}

DESCRIPTION

This routine sets the operating mode for the graphics output device. The operating mode includes display resolution, refresh rate, color depth, and graphics device type.

The type and characteristics of the output device are defined by the UGL_MODE data structure provide by the pMode parameter. This data structure identifies the resolution of the display (width and height), the refresh rate, color depth, and the graphics device type (generic CRT monitor, TFT flat panel, etc.).

The device driver will validate that the operating mode is a legal mode and when the mode is legal, then the device driver will implement the mode. If it is not legal, then the device driver will reject the mode set operation. The resolution and color depth must be explicitly supported. The refresh rate that is closest to the requested value is used.

RETURNS

UGL_STATUS_OK, or UGL_STATUS_ERROR if the operation fails

ERRNO

N/A

SEE ALSO

uglLib, uglModeAvailGet()
**uglMonoBitmapCreate()**

**NAME**

*uglMonoBitmapCreate()* - creates a Monochrome bitmap

**SYNOPSIS**

```c
UGL_MDDB_ID uglMonoBitmapCreate

( UGL_DEVICE_ID          devId,       /* device identifier */
  UGL_MDIB *             pMdib,       /* MDIB to use for creation of a */
                          /* bitmap */
  UGL_DIB_CREATE_MODE    createMode,  /* control bitmap initialization */
  UGL_UINT32             initValue, /* value to use to initialize bitmap */
  UGL_DEVICE_MEM_POOL_ID devicePoolId /* memory pool to contain bitmap */
)
```

**DESCRIPTION**

This routine creates a monochrome bitmap. The process of creating a bitmap builds a monochrome Device Dependent Bitmap (MDDB) for the device `devId`. The MDDB is constructed using the information contained within the monochrome Device Independent Bitmap (DIB) specified by the `pMdib` parameter. The `createMode` parameter controls the manner in which the bitmap is initialized, as follows:

- **UGL_DIB_INIT_DATA**
  Initialize the bitmap with the data present within the DIB

- **UGL_DIB_INIT_VALUE**
  Initialize the bitmap to value `initValue`

- **UGL_DIB_INIT_NONE**
  Perform no initialization

When the `createMode` is set to **UGL_DIB_INIT_DATA**, the data within the `pMdib` is converted to the format that is used by the graphics device and its frame buffer.

The `deviceMemPoolId` specifies the memory pool in which the bitmap is allocated. It must be the device ID returned from a call to *uglMemDevicePoolCreate()* or one of the following values:

- **UGL_DEFAULT_MEM**
  The bitmap is created in the devices default memory pool.

- **UGL_VIDEO_MEM**
  The bitmap is created in video memory not used by the visible display.

- **UGL_NULL**
  Same as **UGL_DEFAULT_MEM**.
The structure of the MDIB, represented by pMdib, is shown in the following figure.

```
|            |
|  Header    |
|            |
|_________|
|_________|

```

The MDIB consists of two sections; header and image data. The header provides organization information concerning the MDDB which includes size and stride parameters. Fields within the header are:

- **width**
  Indicates the width of the image in pixels

- **height**
  Indicates the height of the image in pixels

- **stride**
  Indicates the stride of the image in pixels. The stride is the number of pixels between pixels in adjacent rows. This is usually the same as the width of the image.

- **image**
  Pointer to the image

**RETURNS**
Identifier for the created bitmap, otherwise UGL_NULL when the gc or pMdib is null or the graphics device is unable to create the bitmap.

**SEE ALSO**
uglLib, uglMonoBitmapDestroy(), uglMonoBitmapBlt()

---

### uglMonoBitmapDestroy()

**NAME**
uglMonoBitmapDestroy() - destroy a monochrome bitmap

**SYNOPSIS**

```
UGL_STATUS uglMonoBitmapDestroy
    (  
        UGL_DEVICE_ID devId,     /* device identifier */
        UGL_MDDB_ID   mDbId     /* bitmap ID */
    )
```
DESCRIPTION This routine destroys a bitmap created by `uglMonoBitmapCreate()`. The bitmap specified by `mDdbId` is destroyed for the device `devId`. All resources that the bitmap used are freed as part of the destroy process.

RETURNS UGL_STATUS_OK, or UGL_STATUS_ERROR if the `devId` or `mDdbId` is null.

ERRNO N/A

SEE ALSO `uglLib`, `uglMonoBitmapCreate()`

---

**uglMonoBitmapRead()**

NAME `uglMonoBitmapRead()` - read image data from a monochrome bitmap

SYNOPSIS

```c
UGL_STATUS uglMonoBitmapRead
(
    UGL_GC_ID   gc,           /* graphics context */
    UGL_MDDB_ID mDdbId,       /* source bitmap */
    UGL_POS     srcLeft,      /* left boundary of data */
    UGL_POS     srcTop,       /* top boundary of data */
    UGL_POS     srcRight,     /* right boundary of data to read */
    UGL_POS     srcBottom,    /* bottom boundary of data to read */
    UGL_MDIB *  pMdib,        /* DIB to write */
    UGL_POS     dstX,         /* X coordinate in DIB */
    UGL_POS     dstY          /* Y coordinate in DIB */
)
```

DESCRIPTION This routine reads image data from a monochrome bitmap, specified by `mDdbId`. The data is read from an area of bitmap `mDdbId` and converted to the MDIB format and stored in the MDIB `pMdib`. The area in the source to be read is defined by the rectangle `(srcLeft, srcTop, srcBottom, srcRight)`, and the destination starting at point is `(dstX, dstY)`.

RETURNS UGL_STATUS_OK, or UGL_STATUS_ERROR if the operation fails.

ERRNO N/A

SEE ALSO `uglLib`, `uglMonoBitmapBlt()`, `uglMonoBitmapWrite()`
uglMonoBitmapWrite()

NAME

uglMonoBitmapWrite() - write image information to a monochrome bitmap

SYNOPSIS

UGL_STATUS uglMonoBitmapWrite

(  
    UGL_GC_ID   gc,           /* graphics context */
    UGL_MDIB *  pMdib,        /* DIB to write to bitmap */
    UGL_POS     srcLeft,      /* left boundary of data */
    UGL_POS     srcTop,       /* top boundary of data */
    UGL_POS     srcRight,     /* right boundary of data to read */
    UGL_POS     srcBottom,    /* bottom boundary of data to read */
    UGL_MDDB_ID mDdbId,       /* destination bitmap */
    UGL_POS     dstX,         /* X coordinate in DIB */
    UGL_POS     dstY          /* Y coordinate in DIB */
)

DESCRIPTION

This routine writes image information to an existing monochrome bitmap specified by the mDdbId parameter. It must be a bitmap that was created by uglMonoBitmapCreate(). The rectangular area defined by (srcLeft, srcTop, srcBottom, srcRight) of the monochrome DIB pMdib is converted to the format required of the device attached to the graphics context gc, and stored in the bitmap mDdbId starting at point (dstX, dstY). This coordinate is viewed as relative to the top left corner of the bitmap.

RETURNS

UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null.

ERRNO

N/A

SEE ALSO

uglLib, uglMonoBitmapBlt(), uglMonoBitmapRead()

uglPageCopy()

NAME

uglPageCopy() - Copy the contents of one page to another.

SYNOPSIS

UGL_STATUS uglPageCopy

(  
    UGL_DEVICE_ID devId,      /* Output Device */
    UGL_PAGE_ID srcPage,     /* Source Page */
    UGL_PAGE_ID destPage     /* Destination Page */
)

DESCRIPTION
This copies the entire contents of the srcPage to the destPage. This is a destructive operation to the existing contents of destPage.

RETURNS
UGL_STATUS_OK or UGL_STATUS_ERROR

ERRNO
N/A

SEE ALSO
uglLib, N/A

uglPageCreate()

NAME
uglPageCreate() - Create a page.

SYNOPSIS
UGL_PAGE_ID uglPageCreate
{
    UGL_DEVICE_ID devId       /* Output Device */
}

DESCRIPTION
Multiple pages may be created. The actual location of the pages is driver specific. Except for the first page, pageZero, the application is responsible for creating each page and also deleting each page when it no longer is using it.

This function does not automatically set the newly created page to either the active drawing or display page.

RETURNS
UGL_PAGE_ID or UGL_NULL on error.

ERRNO
N/A

SEE ALSO
uglLib, uglPageDestroy()

uglPageDestroy()

NAME
uglPageDestroy() - Destroy a page.

SYNOPSIS
UGL_STATUS uglPageDestroy
{
    UGL_DEVICE_ID devId,    /* Graphics Context */
    UGL_PAGE_ID   page      /* Page */
}
DESCRIPTION
This routine deletes a page created previously by uglPageCreate. The application is responsible for creating each page and also deleting each page when it no longer is using it.

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if page could not be deleted.

ERRNO
N/A

SEE ALSO
ugLib, uglPageCreate()

uglPageDrawGet()

NAME
uglPageDrawGet() - Get the active drawing page.

SYNOPSIS
UGL_STATUS uglPageDrawGet
    (UGL_DEVICE_ID devId, /* Output Device */
    UGL_PAGE_ID * page)

DESCRIPTION
This routine gets the active drawing page.

RETURNS
UGL_PAGE_ID

ERRNO
N/A

SEE ALSO
ugLib, uglDrawPageSet()

uglPageDrawSet()

NAME
uglPageDrawSet() - Set active drawing page.

SYNOPSIS
UGL_STATUS uglPageDrawSet
    (UGL_DEVICE_ID devId, /* Output Device */
     UGL_PAGE_ID page /* Page */)

}
DESCRIPTION
The Active Drawing page is where all rendering operations intended for the
UGL_DISPLAY_ID will be drawn. The Active Drawing page is set independent of the
Active Display page.

If the Active Display page and the Active Drawing page are equal, rendering operations
sent to the UGL_DISPLAY_ID will be drawn to the visible display. If the Active Display
page and the Active Drawing page are not equal, rendering operations sent to the
UGL_DISPLAY_ID will not be drawn to the currently visible page. Instead they will be
drawn to the Active Display page.

RETURNS
UGL_STATUS_OK or UGL_STATUS_ERROR

ERRNO
N/A

SEE ALSO
uglLib, uglPageDrawGet()

uglPageVisibleGet()

NAME
uglPageVisibleGet() - Get the active display page.

SYNOPSIS
UGL_STATUS uglPageVisibleGet

    (UGL_DEVICE_ID devId,      /* Output Device */
     UGL_PAGE_ID * page
    )

DESCRIPTION
This routine gets the active display page.

RETURNS
UGL_PAGE_ID

ERRNO
N/A

SEE ALSO
uglLib, uglPageVisibleSet()
 uglPageVisibleSet()

NAME

uglPageVisibleSet() - Set the visible display page.

SYNOPSIS

UGL_STATUS uglPageVisibleSet
{
    UGL_DEVICE_ID devId,    /* Output Device */
    UGL_PAGE_ID page       /* Page */
}

DESCRIPTION

The visible display page is the page that is currently visible on-screen. If the Active Display page and the Active Drawing page are equal, rendering operations sent to the UGL_DISPLAY_ID will be drawn to the visible display. If the Active Display page and the Active Drawing page are not equal, rendering operations sent to the UGL_DISPLAY_ID will not be drawn to the currently visible page. Instead they will be drawn to the Active Display page.

RETURNS

UGL_STATUS_OK or UGL_STATUS_ERROR

ERRNO

N/A

SEE ALSO

uglLib, uglPageVisibleGet()

 uglPixelGet()

NAME

uglPixelGet() - gets the color of a pixel

SYNOPSIS

UGL_STATUS uglPixelGet
{
    UGL_GC_ID gc,          /* graphics context */
    UGL_POS x,            /* X position to get pixel */
    UGL_POS y,            /* Y position to get pixel */
    UGL_COLOR * pColor    /* location to store pixel color */
}

DESCRIPTION

This routine get the color of pixel located at the point (x, y), returning the value in pColor.

RETURNS

UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null,

ERRNO

N/A
**uglPixelSet()**

**NAME**

`uglPixelSet()` - draws a pixel

**SYNOPSIS**

```c
UGL_STATUS uglPixelSet
    (UGL_GC_ID gc,             /* graphics context */
     UGL_POS x,              /* X location to set pixel */
     UGL_POS y,              /* Y position to set pixel */
     UGL_COLOR color          /* color to set pixel */
    )
```

**DESCRIPTION**

This routine sets the pixel at point \((x, y)\) to the color specified by `color`. The graphics context `gc` specifies the details of how the pixel is to be drawn. The graphics context is set to define:

- **defaultBitmap**
  Specifies the bitmap to which rendering will occur. `UGL_DISPLAY_ID` specifies the display. Set by the `uglDefaultBitmapSet()` function.

- **clipRect**
  Specifies a clipping rectangle. No drawing may take place outside of this rectangle. Set by the `uglClipRectSet()` function.

- **rasterOp**
  Specifies the raster operation (COPY, XOR, and so on) that is used. Set by the `uglRasterModeSet()` function.

**RETURNS**

- `UGL_STATUS_OK`, or `UGL_STATUS_ERROR` if the `gc` is null,

**ERRNO**

N/A

**SEE ALSO**

`uglLib`, `uglDefaultBitmapSet()`, `uglClipRectSet()`, `uglRasterModeSet()`, `uglPixelGet()`
uglPolygon()

NAME

uglPolygon() - draws a polygon

SYNOPSIS

UGL_STATUS uglPolygon
(
    UGL_GC_ID gc,             /* graphics context */
    UGL_SIZE  numPoints,      /* number of data points in polygon */
    UGL_POS * pData           /* list of data points */
)

DESCRIPTION

This routine draws a polygon. The polygon is outlined with the current foreground color. If a fill pattern bitmap has been specified, then the polygon is filled the fill pattern bitmap. Otherwise, the polygon is filled with the current background color. The argument numPoints specifies the number of points in the polygon. The argument pData is an array of numPoints * 2 values. Each pair of values represents the coordinate of a polygon point. The polygon is drawn by connecting the points. It is not automatically closed; a closed polygon must have the same starting and ending points. The fill behavior for an open polygon is undefined. This routine draws a polygon by connecting the data points contained in the array pointed to by pData.

The graphics context gc specifies the details of how the polygon is to be drawn. The graphics context is set to define:

defaultBitmap
    Specifies the bitmap to which rendering will occur. UGL_DISPLAY_ID specifies the display. Set by the uglDefaultBitmapSet() function.

clipRect
    Specifies a clipping rectangle. No drawing may take place outside of this rectangle. Set by the uglClipRectSet() function.

rasterOp
    Specifies the raster operation (COPY, XOR, and so on) that is used. Set by the uglRasterModeSet() function.

foregroundColor
    Specifies the color of the polygon outline. Set by the uglForegroundColorSet() function.

lineStyle
    Specifies the line style (solid, dashed, and so on) of the polygon outline. Set by the uglLineStyleSet() function.

lineWidth
    Specifies the width, in pixels, of the polygon outline. Set by the uglLineWidthSet() function.
**backgroundColor**

Specifies the fill color for the polygon. Set by the `uglBackgroundColorSet()` function.

**patternBitmap**

Specifies the fill pattern for the polygon. Set by the `uglFillPatternSet()` function.

If an outlined polygon is not desired, the foreground color should be set to `UGL_COLOR_TRANSPARENT` or the line width should be set to zero. The polygon is not filled if the background color is `UGL_COLOR_TRANSPARENT`.

**RETURNS**

`UGL_STATUS_OK`, or `UGL_STATUS_ERROR` if `gc` is null, the number of data points is zero, or the driver is unable to perform the rendering operation.

**ERRNO**

N/A

**SEE ALSO**

`uglLib`, `uglDefaultBitmapSet()`, `uglClipRectSet()`, `uglRasterModeSet()`, `uglForegroundColorSet()`, `uglLineStyleSet()`, `uglLineWidthSet()`, `uglBackgroundColorSet()`, `uglFillPatternSet()`

---

### `uglRasterModeGet()`

**NAME**

`uglRasterModeGet()` - gets a gc's raster mode setting

**SYNOPSIS**

```c
UGL_STATUS uglRasterModeGet(
    UGL_GC_ID       gc,       /* graphics context */
    UGL_RASTER_OP * pRasterOp /* raster mode setting */
)
```

**DESCRIPTION**

This routine obtains the current raster operation that is being used performing graphic drawing operations. The raster operation is written to `pRasterOp` and may be any one of the following:

- `UGL_RASTER_OP_COPY`
- `UGL_RASTER_OP_AND`
- `UGL_RASTER_OP_OR`
- `UGL_RASTER_OP_XOR`

In addition, the following raster operations are defined on alpha capable systems:
UGL_RASTER_OP_COPY_RGB_ONLY
UGL_RASTER_OP_COPY_ALPHA_ONLY
UGL_RASTER_OP_ALPHA_KEY_FROM_SOURCE

For additional information on these operations, see uglRasterModeSet().

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null.

ERRNO
N/A

SEE ALSO
uglLib, uglRasterOpSet()

---

**uglRasterModeSet()**

**NAME**

uglRasterModeSet() - sets a gc’s raster mode

**SYNOPSIS**

`UGL_STATUS uglRasterModeSet(`

`UGL_GC_ID     gc,         /* graphics context */`

`UGL_RASTER_OP rasterOp    /* new raster mode setting */`

`);`

**DESCRIPTION**

This routine sets the raster operation for the specified graphics context gc. rasterOp may be one of the following, see Table 15-7:

**Table 15-7 Raster Modes**

<table>
<thead>
<tr>
<th>Value</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>UGL_RASTER_OP_COPY</td>
<td>Replace affected portions of the destination or target bitmap with new data.</td>
</tr>
<tr>
<td>UGL_RASTER_OP_AND</td>
<td>Combine affected portions of the destination or target bitmap by ANDing the current contents with new data.</td>
</tr>
<tr>
<td>UGL_RASTER_OP_OR</td>
<td>Combine affected portions of the destination or target bitmap by ORing the current contents with new data.</td>
</tr>
</tbody>
</table>
uglRectangle()

NAME

uglRectangle() - draws a rectangle

SYNOPSIS

UGL_STATUS uglRectangle

    (    
        UGL_GC_ID gc,     /* graphics context */
        UGL_POS   left,   /* location of left edge of rectangle */
        UGL_POS   top,    /* location of top edge of rectangle */
        UGL_POS   right,  /* location of right edge of rectangle */
        UGL_POS   bottom  /* location of bottom edge of rectangle */
    )

DESCRIPTION

This routine draws a rectangle on the display.

The rectangle is outlined with the current foreground color. If a fill pattern bitmap has
been specified, the rectangle is filled with the fill pattern bitmap. Otherwise, the rectangle
is filled with the current background color. The parameters left, top, right, and bottom
specify the bounding coordinates of the rectangle. The rectangle is inclusive, thus the right
and bottom boundaries are drawn.

Table 15-7  Raster Modes

<table>
<thead>
<tr>
<th>Value</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>UGL_RASTER_OP_XOR</td>
<td>Combine affected portions of the destination or target bitmap by performing exclusive-OR between the current contents and new data.</td>
</tr>
<tr>
<td>UGL_RASTER_OP_COPY_RGB_ONLY</td>
<td>Copy only color information.</td>
</tr>
<tr>
<td>UGL_RASTER_OP_COPY_ALPHA_ONLY</td>
<td>Copy only alpha channel information</td>
</tr>
<tr>
<td>UGL_RASTER_OP_ALPHA_KEY_FROM_SOURCE</td>
<td>Mix source and destination colors</td>
</tr>
</tbody>
</table>

RETURNS

UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null.

ERRNO

N/A

SEE ALSO

uglLib, uglRasterModeGet()
The graphics context gc specifies the details of how the rectangle is to be drawn. The graphics context is set to define:

**defaultBitmap**
- Specifies the bitmap to which rendering will occur. UGL_DISPLAY_ID specifies the display. Set by the `uglDefaultBitmapSet()` function.

**clipRect**
- Specifies a clipping rectangle. No drawing may take place outside of this rectangle. Set by the `uglClipRectSet()` function.

**rasterOp**
- Specifies the raster operation (COPY, XOR, and so on) that is used. Set by the `uglRasterModeSet()` function.

**foregroundColor**
- Specifies the color of the rectangle outline. Set by the `uglForegroundColorSet()` function.

**lineStyle**
- Specifies the line style (solid, dashed, and so on) of the rectangle outline. Set by the `uglLineStyleSet()` function.

**LineWidth**
- Specifies the width, in pixels, of the rectangle outline. Set by the `uglLineWidthSet()` function.

**backgroundColor**
- Specifies the fill color for the rectangle. Set by the `uglBackgroundColorSet()` function.

**patternBitmap**
- Specifies the fill pattern for the rectangle. Set by the `uglFillPatternSet()` function.

If an outlined rectangle is not desired, the foreground color should be set to UGL_COLOR_TRANSPARENT or the line width should be set to zero. The rectangle is not filled if the background color is UGL_COLOR_TRANSPARENT.

**RETURNS**
- `UGL_STATUS_OK`, or `UGL_STATUS_ERROR` if gc is null or the driver is unable to perform the rendering operation.

**ERRNO**
- N/A

**SEE ALSO**
- `uglLib`, `uglDefaultBitmapSet()`, `uglClipRectSet()`, `uglRasterModeSet()`, `uglForegroundColorSet()`, `uglLineStyleSet()`, `uglLineWidthSet()`, `uglBackgroundColorSet()`, `uglFillPatternSet()`
uglRegionClipToRect()

NAME

uglRegionClipToRect() - clip a region to a rectangle

SYNOPSIS

UGL_STATUS uglRegionClipToRect

    (UGL_REGION_ID regionId, /* region ID */
     const UGL_RECT * pClipRect /* clip rect */)

DESCRIPTION

This routine clips a region to a rectangle. Any portion of the region that lies outside of the rectangle is discarded.

RETURNS

UGL_STATUS_OK, or a non zero value on error

SEE ALSO

uglLib

uglRegionCopy()

NAME

uglRegionCopy() - copy the contents of one region to another

SYNOPSIS

UGL_STATUS uglRegionCopy

    (UGL_REGION_ID srcRegionId, /* source region */
     UGL_REGION_ID destRegionId /* destination region */)

DESCRIPTION

This routine copies the contents of the region identified by srcRegionId to the region identified by destRegionId. After the copy, the destination region encloses the same space as the source region.

RETURNS

UGL_STATUS_OK, or UGL_STATUS_ERROR if srcRegionId or destRegionId is NULL, or if memory is insufficient.

SEE ALSO

uglLib, uglRegionUnion(), uglRegionIntersect()
uglRegionCreate( )

NAME uglRegionCreate() - create a region

SYNOPSIS UGL_REGION_ID uglRegionCreate (void)

DESCRIPTION This routine creates a region. A region is an arbitrary space defined by the union of a set of rectangles. The created region is initially empty.

RETURNS An UGL_REGION_ID value, or UGL_NULL if memory is insufficient.

SEE ALSO uglLib, uglClipRegionSet( )

uglRegionDestroy( )

NAME uglRegionDestroy() - destroy a region

SYNOPSIS UGL_STATUS uglRegionDestroy

   ( UGL_REGION_ID regionId    /* region to be destroyed */
   )

DESCRIPTION This routine destroys the region identified by regionId. This region must not be selected as the clip region of any allocated graphics contexts.

RETURNS UGL_STATUS_OK, or UGL_STATUS_ERROR if regionId is NULL.

SEE ALSO uglLib

uglRegionEmpty( )

NAME uglRegionEmpty( ) - empty contents of a region

SYNOPSIS UGL_STATUS uglRegionEmpty

   ( UGL_REGION_ID regionId    /* region identifier */
   )
DESCRIPTION
This routine removes all contents of a region so that the region encloses no space. The set of rectangles defining the region is made empty.

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if regionId is NULL.

ERRNOSE ALSO
uglLib

uglRegionIntersect()

NAME
uglRegionIntersect() - find the intersection of two regions

SYNOPSIS
UGL_STATUS uglRegionIntersect
{
    UGL_REGION_ID region1Id,       /* region 1 */
    UGL_REGION_ID region2Id,       /* region 2 */
    UGL_REGION_ID intersectRegionId /* gets intersection region */
}

DESCRIPTION
This routine determines the intersection of the region identified by region1Id and the region identified by region2Id and places the result in the region identified by intersectRegionId. intersectRegionId may identify the same region as either region1Id or region2Id.

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if region1Id, region2Id or intersectRegionId is NULL, or if memory is insufficient.

ERRNOSE ALSO
uglLib, uglRegionUnion(), uglRegionCopy()

uglRegionIsEmpty()

NAME
uglRegionIsEmpty() - determine if a region is empty

SYNOPSIS
UGL_BOOL uglRegionIsEmpty
{
    UGL_REGION_ID regionId
}

DESCRIPTION
This routine determines if a region is empty. An empty region encloses no space.
uglRegionMove()

NAME uglRegionMove() - move a region

SYNOPSIS UGL_STATUS uglRegionMove
                   />
                UGL_ORD deltaX, /* x offset */
                UGL_ORD deltaY /* y offset */
            }

DESCRIPTION This routine moves a region by moving each rectangle comprising the region. deltaX and
deltaY specify the offset by which the region is to be moved.

RETURNS UGL_STATUS_OK, or UGL_STATUS_ERROR if regionId is NULL.

SEE ALSO uglLib

uglRegionRectExclude()

NAME uglRegionRectExclude() - exclude a rectangle from a region

SYNOPSIS UGL_STATUS uglRegionRectExclude
                    */
                const UGL_RECT *pExcludeRect /* rectangle to be excluded from region */
            }

DESCRIPTION This routine excludes a rectangle from a region. The set of rectangles defining the region is
modified to exclude the specified rectangle. Optimizations are performed to minimize
the number of rectangles in the region.

RETURNS UGL_STATUS_OK, or UGL_STATUS_ERROR if regionId or rect is NULL, or if memory
is insufficient.
uraLib, uglRegionRectInclude()
uglRegionRectInclude()

NAME

uglRegionRectInclude() - include a rectangle in a region

SYNOPSIS

UGL_STATUS uglRegionRectInclude

(  
    UGL_REGION_ID    regionId,    /* region identifier */
    const UGL_RECT * pIncludeRect /* rectangle to be included in region */
)

DESCRIPTION

This routine includes a rectangle in a region. The set of rectangles defining the region is modified to include the specified rectangle. Rectangles are combined when possible to minimize the number of rectangles in the region.

RETURNS

UGL_STATUS_OK, or UGL_STATUS_ERROR if regionId or rect is NULL, or if memory is insufficient.

ERRNOSEE ALSO

uglLib, uglRegionRectExclude()

uglRegionRegionExclude()

NAME

uglRegionRegionExclude() - exclude one region from another region

SYNOPSIS

UGL_STATUS uglRegionRegionExclude

(  
    UGL_REGION_ID regionId,       /* region ID */
    UGL_REGION_ID excludeRegionId /* region to be excluded */
)

DESCRIPTION

This routine excludes one removes one region from another. The set of rectangles defining the region is modified to exclude the specified region.

RETURNS

UGL_STATUS_OK, or a non zero value on error.

ERRNOSEE ALSO

uglLib, uglRegionUnion()
**uglRegionUnion()**

**NAME**

*uglRegionUnion()* - find the union of two regions

**SYNOPSIS**

```c
UGL_STATUS uglRegionUnion
    (    
    UGL_REGION_ID region1Id,    /* region 1 */
    UGL_REGION_ID region2Id,    /* region 2 */
    UGL_REGION_ID unionRegionId /* gets union region */
    )
```

**DESCRIPTION**

This routine determines the union of the region identified by `region1Id` and the region identified by `region2Id` and places the result in the region identified by `unionRegionId`. `unionRegionId` may identify the same region as either `region1Id` or `region2Id`.

**RETURNS**

`UGL_STATUS_OK`, or `UGL_STATUS_ERROR` if `region1Id`, `region2Id` or `unionRegionId` is NULL, or if memory is insufficient.

**SEE ALSO**

`uglLib`, `uglRegionIntersect()`, `uglRegionCopy()`, `uglTextDraw()`

---

**uglTextDraw()**

**NAME**

*uglTextDraw()* - draw text

**SYNOPSIS**

```c
UGL_STATUS uglTextDraw
    (    
    UGL_GC_ID        gc,      /* graphics context */
    UGL_POS          x,       /* left position of the text */
    UGL_POS          y,       /* top position of the text */
    UGL_SIZE         length,  /* number of characters in text */
    const UGL_CHAR * text     /* single-byte text array */
    )
```

**DESCRIPTION**

This routine draws single byte text to the bitmap specified within `gc`. A `length` value of -1 causes the routine to continue drawing until a null value is encountered in the text.

If any of the font’s attributes are changed during a text draw operation, the remaining characters are rendered using the new font.

**RETURNS**

`UGL_STATUS_OK` (0), or a non zero value if the operation fails.
uglTextDrawW()

NAME
uglTextDrawW() - draw double-wide text

SYNOPSIS
UGL_STATUS uglTextDrawW

    ( 
        UGL_GC_ID         gc,     /* graphics context */
        UGL_POS           x,      /* left position of the text */
        UGL_POS           y,      /* top position of the text */
        UGL_SIZE          length, /* number of characters in text */
        const UGL_WCHAR * text    /* double-byte text array */
    )

DESCRIPTION
This routine draws double byte text to the bitmap specified within gc. A length value of -1
causes the routine to continue drawing until a null value is encountered in the text.
If any of the font's attributes are changed during a text draw operation, the remaining
characters are rendered using the new font.

RETURNS
UGL_STATUS_OK (0), or a non zero value if the operation fails.

SEE ALSO
uglLib, uglText()

uglTextSizeGet()

NAME
uglTextSizeGet() - Get text size

SYNOPSIS
UGL_STATUS uglTextSizeGet

    ( 
        UGL_FONT_ID      fontId,  /* font identifier */
        UGL_SIZE *       width,   /* value to receive the text width */
        UGL_SIZE *       height,  /* value to receive the text height */
        UGL_SIZE         length,  /* number of characters in text */
        const UGL_CHAR * text     /* single-byte text array */
    )
This routine computes the run-time width and height of the specified single-byte text (in pixel values). If either width or height is NULL, the respective size is not computed. In addition, if length is set to -1, the routine will continue computing a size until a null value is encountered.

UGL_STATUS_OK (0), or a non zero value if the operation fails.

uglLib, uglTextSizeGetW()

---

uglTransBitmapCreate() - creates a transparent bitmap

UGL_TDDB_ID uglTransBitmapCreate

( 
    UGL_DEVICE_ID devId, /* device context */
    UGL_DIB * pDib, /* DIB to use for creation of a */
)
DESCRIPTION

This routine creates a transparent bitmap for use in rendering images with transparency. The process of creating a transparent bitmap builds a transparent Device Dependent Bitmap (TDDB) for the device specified by `devId`. The TDDB is constructed using the information contained within the DIB specified by the `pDib` parameter and the transparency mask provided by the MDIB `pMdib`. The `createMode` parameter controls the manner in which the bitmap is initialized, as follows:

- **UGL_DIB_INIT_DATA**
  - Initialize the bitmap with the data present within the DIB

- **UGL_DIB_INIT_VALUE**
  - Initialize the bitmap to value `initValue`

- **UGL_DIB_INIT_NONE**
  - Perform no initialization

When the TDDB is to be initialized with a value (UGL_DIB_INIT_VALUE), the value provided by `initValue` only applies to the DIB portion of the TDDB. The mask is set to all ones.

`deviceMemPoolId` specifies the memory pool in which the bitmap is allocated. It must be the device ID returned from a call to `uglMemDevicePoolCreate()`, or one of the following values:

- **UGL_DEFAULT_MEM**
  - The bitmap is created in the device’s default memory pool.

- **UGL_VIDEO_MEM**
  - The bitmap is created in video memory not used by the visible display.

- **UGL_NULL**
  - Same as UGL_DEFAULT_MEM.

The format of the DIB and the MDDB is provided in the descriptions for the `uglBitmapCreate()` and `uglMonoBitmapCreate()` functions.

RETURNS

Identifier for the created transparent bitmap, otherwise `UGL_NULL`

if `devId`, `pDib`, or `pMdib` are null, or the bitmap creation by the driver fails.

ERRNO

N/A

SEE ALSO

`uglLib`, `uglTransBitmapDestroy()`, `uglBitmapCreate()`, `uglMonoBitmapCreate()`
**uglTransBitmapCreateFromDDB()**

**NAME**

`uglTransBitmapCreateFromDDB()` - creates a transparent bitmap from standard bitmap

**SYNOPSIS**

```c
UGL_TDBB_ID uglTransBitmapCreateFromDDB
    ( 
        UGL_DEVICE_ID devId,       /* device context */
        UGL_DDB_ID ddbId,       /* standard bitmap */
        UGL_MDDB_ID mDdbId,      /* mask for the bitmap */
        UGL_DEVICE_MEM_POOL_ID devicePoolId /* memory pool to contain bitmap */
    )
```

**DESCRIPTION**

This routine creates a transparent bitmap from a standard (non-transparent) bitmap and a monochrome bitmap. An existing standard bitmap specified by `ddbdId` is converted to a transparent bitmap using the monochrome bitmap specified by `mDdbId` to provide the transparency mask. The standard and monochrome bitmaps are not modified by this function. Any subsequent modifications to `ddbdId` or `mDdbId`, will be reflected in the transparent bitmap.

`deviceMemPoolId` specifies the memory pool in which the bitmap is allocated. It must be the device ID returned from a call to `uglMemDevicePoolCreate()`, or one of the following values:

- **UGL_DEFAULT_MEM**
  - The bitmap is created in the device’s default memory pool.

- **UGL_VIDEO_MEM**
  - The bitmap is created in video memory not used by the visible display.

- **UGL_NULL**
  - Same as **UGL_DEFAULT_MEM**.

**RETURNS**

Identifier for the created transparent bitmap, otherwise **UGL_NULL** if `devld` or `mDdbld`, `ddbdId` are null, or the bitmap creation by the driver fails.

**ERRNO**

N/A

**SEE ALSO**

`uglLib`, `uglTransBitmapCreate()`, `uglBitmapCreate()`, `uglMonoBitmapCreate()`
uglTransBitmapDestroy()

NAME

uglTransBitmapDestroy() - destroys a transparent bitmap

SYNOPSIS

UGL_STATUS uglTransBitmapDestroy

(  
  UGL_DEVICE_ID devId,     /* graphics context */
  UGL_TDDB_ID tDdbId      /* transparent bitmap ID */
)

DESCRIPTION

This routine destroys a transparent bitmap that was created by either the
uglTransBitmapCreate() or uglTransBitmapCreateFromDDB() function. The bitmap
specified by tDdbId is destroyed for the device devId. All resources that the bitmap used
are freed as part of the destroy process.

RETURNS

UGL_STATUS_OK, or UGL_STATUS_ERROR if the devId or tDdbId are null

ERRNO

N/A

SEE ALSO

uglLib, uglTransBitmapCreate()

uglTransBitmapRead()

NAME

uglTransBitmapRead() - read image data from a transparent bitmap

SYNOPSIS

UGL_STATUS uglTransBitmapRead

(  
  UGL_GC_ID   gc,           /* graphics context */
  UGL_TDDB_ID tDdbId,       /* source bitmap */
  UGL_POS     srcLeft,      /* left boundary of data */
  UGL_POS     srcTop,       /* top boundary of data */
  UGL_POS     srcRight,     /* right boundary of data to read */
  UGL_POS     srcBottom,    /* bottom boundary of data to read */
  UGL_DIB *   pDib,         /* DIB to write */
  UGL_MDIB *  pMdib,        /* mask data */
  UGL_POS     dstX,         /* X coordinate in DIB */
  UGL_POS     dstY          /* Y coordinate in DIB */
)


DESCRIPTION
This routine reads image data from a bitmap, specified by tDbId. The data is read from the area of transparent bitmap tDbId defined by the rectangle (srcLeft, srcTop, srcBottom, srcRight), stored in the pMdib DIB starting at point (dstX, dstY). The mask data associated with the transparent bitmap is stored in the monochrome DIB provided by pMdib using the same rectangular source and destination point. The data is stored in the DIB using the native format for the graphics device, that is, it will be stored using UGL_COLOR.

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if gc, dib, or mDib are null.

ERRNO
N/A

SEE ALSO
uglLib, uglTransBitmapBlt(), uglTransBitmapWrite()

uglTransBitmapWrite()

NAME
uglTransBitmapWrite() - write image information to a transparent bitmap

SYNOPSIS
UGL_STATUS uglTransBitmapWrite
{
    UGL_GC_ID   gc,      /* graphics context */
    UGL_DIB *   pDib,    /* DIB to write to bitmap */
    UGL_MDIB *  pMdib,   /* monochrome DIB for transparency mask */
    UGL_POS    srcLeft,  /* left boundary of data */
    UGL_POS    srcTop,   /* top boundary of data */
    UGL_POS    srcRight, /* right boundary of data to read */
    UGL_POS    srcBottom,/* bottom boundary of data to read */
    UGL_TDDB_ID tDbId,   /* destination bitmap */
    UGL_POS    dstX,     /* X coordinate in DIB */
    UGL_POS    dstY      /* Y coordinate in DIB */
}

DESCRIPTION
This routine writes image information along with transparency mask data to an existing transparent bitmap specified by the tDbId parameter. The data is read from the rectangular area defined by (srcLeft, srcTop, srcBottom, srcRight) of DIB pDib, converted to the format required of the device attached to the graphics context gc, and stored in the transparent bitmap tDbId starting at point (dstX, dstY). The mask data provided by the monochrome DIB pMdib that provides transparency control for the DIB is transferred to the transparent bitmap tDbId using the same rectangular source and destination point as used for the DIB.

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc, pDib, or pMdib is null.
uglViewPortGet()

NAME
uglViewPortGet() - gets a gc's view port

SYNOPSIS
UGL_STATUS uglViewPortGet
{
    UGL_GC_ID gc, /* graphics context */
    UGL_POS * pLeft, /* left boundary of gc's view port */
    UGL_POS * pTop, /* top boundary of gc's view port */
    UGL_POS * pRight, /* right boundary of gc's view port */
    UGL_POS * pBottom /* bottom boundary of gc's view port */
}

DESCRIPTION
This routine returns the current view port settings of the view port for the specified
graphics context gc. The setting for the view port is left-top relative and is written to the
parameters pLeft, pTop, pRight, and pBottom.

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null.

ERRNO
N/A

SEE ALSO
uglLib, uglViewPortSet()

uglViewPortSet()

NAME
uglViewPortSet() - sets a gc's view port

SYNOPSIS
UGL_STATUS uglViewPortSet
{
    UGL_GC_ID gc, /* graphics context */
    UGL_POS left, /* left boundary of view port */
    UGL_POS top, /* top boundary of view port */
    UGL_POS right, /* right boundary of view port */
    UGL_POS bottom /* bottom boundary of view port */
}

ERRNO
N/A

SEE ALSO
uglLib, uglViewPortSet()
DESCRIPTION

This routine sets the viewport of graphics context gc to the rectangle bounded by left, top, right, and bottom. All rendering operations are clipped to the viewport and performed relative to the top-left coordinate of the viewport.

RETURNS

UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null.

ERRNO

N/A

SEE ALSO

uglLib, uglViewPortGet()
Event API

**uglEvt**

**NAME**

`uglEvt` - WindML Event API

**ROUTINES**

- `uglEventCallbackAdd()` - add a callback to an event handler
- `uglEventCallbackChain()` - chain a callback to an event handler
- `uglEventCallbackRemove()` - remove a callback from an event handler
- `uglEventDefaultQGet()` - get an event service's default event queue
- `uglEventDefaultQSet()` - set an event service's default event queue
- `uglEventHandler()` - cause an event handler to handle an event
- `uglEventHandlerCreate()` - create an event handler
- `uglEventHandlerDestroy()` - destroy an event handler
- `uglEventHandlerGet()` - get an event service's event handler
- `uglEventHandlerSet()` - set an event service's event handler
- `uglEventGet()` - get an event from an event queue
- `uglEventPost()` - post an event to an event queue
- `uglEventQCreate()` - create an event queue
- `uglEventQDestroy()` - destroy an event queue
- `uglEventRouterGet()` - get the event router associated with an event service
- `uglEventRouterSet()` - set the event router associated with an event service
- `uglEventServiceCreate()` - create an event service
- `uglEventServiceDestroy()` - terminate an event service
- `uglEventServiceIdle()` - idle an event service
- `uglEventServiceResume()` - resume an event service
- `uglInputDeviceAdd()` - adds an input device to an event service
- `uglInputDeviceDestroy()` - destroy an input device
- `uglInputDeviceInfo()` - control/obtain information
- `uglInputEventGet()` - retrieve an event from an event service
DESCRIPTION
This library provides the event APIs for WindML.

uglEventCallbackAdd()

NAME
uglEventCallbackAdd() - add a callback to an event handler

SYNOPSIS
UGL_STATUS uglEventCallbackAdd
{
    UGL_EVENT_HANDLER_ID eventHandlerId, /* event handler */
    UGL_UINT32 eventType, /* type of event to trap */
    UGL_UINT32 eventCategory, /* category of event to trap */
    UGL_CALLBACK * pCallback /* callback routine to call */
}

DESCRIPTION
This routine adds a callback to an event handler. If a callback of the same type and
category already exists in the event handler, it is replaced.

eventHandlerId identifies the event handler to which the callback is added.

eventType identifies the type of event to be sent to the callback routine. If zero, all events in
category are sent to the callback routine.

category identifies the category of events to be sent to the callback routine. If type is
non-zero, then this field is ignored. If both type and category are zero, all events are sent to the callback routine.

callback specifies the routine to be called for the type or category of event.

RETURNS
UGL_STATUS_OK (0) or a non zero value on error.

SEE ALSO
uglEvt

uglEventCallbackChain()

NAME
uglEventCallbackChain() - chain a callback to an event handler

SYNOPSIS
UGL_STATUS uglEventCallbackChain
{
    UGL_EVENT_HANDLER_ID eventHandlerId, /* event handler */
    UGL_UINT32 eventType, /* type of event to trap */
    UGL_UINT32 eventCategory, /* category of event to trap */
}
UGL_CALLBACK * pCallback,        /* callback routine to call */
UGL_BOOL             callFirst       /* chaining order */
)

DESCRIPTION
This routine adds a callback to an event handler, and chains it to any existing callbacks for
the type or category of event. The new callback is called in addition to any existing
callbacks that already exist for the event.

eventHandlerId identifies the event handler to which the callback is to be added.

eventType identifies the type of event to be sent to the callback routine. If zero, all events in
category are sent to the callback routine.

category identifies the category of events to be sent to the callback routine. If type is
non-zero, then this field is ignored. If both type and category are zero, all events are sent to
the callback routine.

callback specifies the routine to be called for the type or category of event.

callFirst determines if the new callback is call before or after the callback(s) to which it is
chained. If UGL_TRUE, the new callback is called first; if UGL_FALSE, it is called last.

RETURNS
UGL_STATUS_OK (0) or a non zero value on error.

SEE ALSO
uglEvt

uglEventCallbackRemove()

NAME
uglEventCallbackRemove() - remove a callback from an event handler

SYNOPSIS
UGL_STATUS uglEventCallbackRemove

{
    UGL_EVENT_HANDLER_ID eventHandlerId, /* event handler */
    UGL_UINT32           eventType,      /* type of event to trap */
    UGL_UINT32           eventCategory,  /* category of event to trap */
    UGL_CALLBACK *       pCallback       /* callback routine to call */
}

DESCRIPTION
This routine removes a callback item from an event handler’s callback array.

eventHandlerId identifies the event handler from which the callback is to be removed.

eventType identifies the type of event that is sent to the callback routine that is to be
removed.

341
eventCategory identifies the category of events that are sent to the callback routine that is to be removed.

callback specifies the callback routine that is to be removed.

eventType, eventCategory, and callback must all match a callback item for the item to be removed.

RETURNS UGL_STATUS_OK (0) or a non zero value on error.

SEE ALSO uglEvt

uglEventDefaultQGet()

NAME uglEventDefaultQGet() - get an event service’s default event queue

SYNOPSIS UGL_STATUS uglEventDefaultQGet

( UGL_EVENT_SERVICE_ID eventServiceId, /* event service */
  UGL_EVENT_Q_ID * pEventQId /* default event queue */
)

DESCRIPTION This routine gets the default event queue for an event service.

RETURNS UGL_STATUS_OK (0) or a non zero value on error.

SEE ALSO uglEvt, uglEventDefaultQSet

uglEventDefaultQSet()

NAME uglEventDefaultQSet() - set an event service’s default event queue

SYNOPSIS UGL_STATUS uglEventDefaultQSet

( UGL_EVENT_SERVICE_ID eventServiceId, /* event service */
  UGL_EVENT_Q_ID eventQId /* default event queue */
)

DESCRIPTION This routine sets the default event queue for an event service.
uglEventGet()

NAME

uglEventGet() - get an event from an event queue

SYNOPSIS

UGL_STATUS uglEventGet
{
    UGL_EVENT_Q_ID qId,       /* event queue */
    UGL_EVENT *    pEvent,    /* gets event */
    UGL_SIZE       eventSize, /* size of * pEvent */
    UGL_TIMEOUT    timeout    /* time to wait for event */
}

DESCRIPTION

This routine gets an event from an event queue.

RETURNS

UGL_STATUS_OK (0) or a non zero value on error.

SEE ALSO

uglEvt, uglEventDefaultQSet()

uglEventHandle()

NAME

uglEventHandle() - cause an event handler to handle an event

SYNOPSIS

UGL_STATUS uglEventHandle
{
    UGL_EVENT_HANDLER_ID eventHandlerId, /* event handler */
    UGL_ID               objectId,       /* object receiving event */
    UGL_EVENT *          pEvent,         /* event to be handled */
    UGL_UINT32 *         pResult         /* result of handling */
}

DESCRIPTION

This routine passes and event to an event handler to be handled.

eventHandlerId identifies the event handler to handle the event.

objectId identifies the instance of an object receiving an event. This may be an event service ID, a window ID, or some other identifier as defined by the owner of the event handler.
pEvent is a pointer to the event to be handled.
pResult specifies a location to place the result of the event handling. This result may be initialized with a default value.

RETURNS UGL_STATUS_OK (0) or a non zero value on error.

SEE ALSO uglEvt

**uglEventHandlerCreate()**

NAME uglEventHandlerCreate() - create an event handler

SYNOPSIS UGL_EVENT_HANDLER_ID uglEventHandlerCreate

( const UGL_CALLBACK_ITEM * pCallbackArray /* initialization array */
)

DESCRIPTION This routine creates an event handler. An event handler is a list of callback routines used to handle events.

pCallbackArray points to an array of UGL_CALLBACK_ITEM structures used to initialize the event handler. If UGL_NULL, the event handler will initially contain no callbacks. If provided, the array must be terminated with an item that has a UGL_NULL entry for its callback field.

The UGL_CALLBACK_ITEM structure is shown below:

```
typedef struct ugl_callback_item
{
    UGL_UINT32        eventType;      /* type of events to trap */
    UGL_UINT32        eventCategory;  /* category of events to trap */
    UGL_CALLBACK      callback;       /* callback routine to call */
    UGL_ORD           chainIndex;     /* index of chained item */
    UGL_BOOL          callFirst;      /* chaining order */
} UGL_CALLBACK_ITEM;
```

eventType identifies the type of event to be sent to the callback routine. If zero, all events in eventCategory are sent to the callback routine.

eventCategory identifies the category of events to be sent to the callback routine. If type is non-zero, then this field is ignored. If both type and category are zero, all events are sent to the callback routine.

callback specifies the routine to be called for a type or category of event.
chainIndex is an index in the array to an additional callback item that is chained to the first item. The callback of this item will be called in addition to the callback of the item chaining it. Any number of callbacks can be chained for handling a single type or category of event. If chainIndex is -1, the callback is not chained.

callFirst determines if a callback is called before or after a callback to which it is chained. If UGL_TRUE, the callback is called before the chained callback; if UGL_FALSE, the callback is called after the chained callback. If chainIndex is -1, this field is ignored.

The UGL_CALLBACK type definition is shown below:

```c
typedef UGL_STATUS (* UGL_CALLBACK) (UGL_UINT32 objectId, UGL_EVENT * pEvent,
                                         UGL_UINT32 * pResult);
```

objectId identifies the instance of an object receiving an event. This may be an event service ID, a window ID, or some other identifier as defined by the owner of the event handler.

pEvent is a pointer to the event to be handled.

pResult is a pointer to a 32-bit value used to return a result from the event.

The value returned from a callback should be UGL_STATUS_OK, or a non zero value upon error.

**RETURNS**

An event handler ID, or UGL_NULL on error.

**NOTE**

Only the pointer pCallbackArray is copied by this routine. The array is not copied, and therefore should be static. The array may be copied at a later time if the event handler is dynamically modified by a call to uglEventCallbackAdd(), uglEventCallbackRemove, or uglEventCallbackChain().

**NOTE**

Only the first item in a callback array that matches an event type or category is called (along with any callbacks that are chained to that item).

**SEE ALSO** uglEvt

---

**uglEventHandlerDestroy()**

**NAME**

uglEventHandlerDestroy() - destroy an event handler

**SYNOPSIS**

```c
UGL_STATUS uglEventHandlerDestroy
    (   UGL_EVENT_HANDLER_ID eventHandlerId /* event handler */
    )
```

**DESCRIPTION**

This routine destroys an event handler.
**eventHandlerId** identifies the event handler to be destroyed.

**RETURNS**

UGL_STATUS_OK (0) or a non zero value on error.

**SEE ALSO**

uglEvt

---

### `uglEventHandlerGet()`

**NAME**

`uglEventHandlerGet()` - get an event service’s event handler

**SYNOPSIS**

```c
UGL_STATUS uglEventHandlerGet
(
    UGL_EVENT_SERVICE_ID eventServiceId, /* event service */
    UGL_EVENT_HANDLER_ID * pEventHandlerId /* gets event handler */
)
```

**DESCRIPTION**

This routine gets the event handler used by an event service to route events.

**RETURNS**

UGL_STATUS_OK, or a non zero value upon error.

**SEE ALSO**

uglEvt, `uglEventHandlerSet()`

---

### `uglEventHandlerSet()`

**NAME**

`uglEventHandlerSet()` - set an event service’s event handler

**SYNOPSIS**

```c
UGL_STATUS uglEventHandlerSet
(
    UGL_EVENT_SERVICE_ID eventServiceId, /* event service */
    UGL_EVENT_HANDLER_ID eventHandlerId /* event handler */
)
```

**DESCRIPTION**

This routine sets the event handler used by an event service to route events.

**RETURNS**

UGL_STATUS_OK, or a non zero value upon error.

**SEE ALSO**

uglEvt, `uglEventHandlerGet()`
** uglEventPost() **

** NAME **

uglEventPost() - post an event to an event queue

** SYNOPSIS **

```
UGL_STATUS uglEventPost
    (  
        UGL_EVENT_SERVICE_ID eventServiceId, /* event service */
        UGL_EVENT_Q_ID qId,            /* event queue to place event */
        UGL_EVENT * pEvent,         /* pointer to event */
        UGL_SIZE             eventSize       /* size of event */
    )
```

** DESCRIPTION **

This routine handles an event, and places it in an event queue. The event service's event handler is responsible for routing the event to the proper event queue.

*eventServiceId* identifies the event service that will handle the event.

*eventQId* suggests a queue into which the event is to be placed. It may be *UGL_NULL*, indicating the default queue. The destination of the event is ultimately determined by the event service's event handler.

** RETURNS **

UGL_STATUS_OK (0) or a non zero value on error.

** SEE ALSO **

uglEvt

---

** uglEventQCreate() **

** NAME **

uglEventQCreate() - create an event queue

** SYNOPSIS **

```
UGL_EVENT_Q_ID uglEventQCreate
    (  
        UGL_EVENT_SERVICE_ID eventServiceId, /* event service */
        UGL_SIZE             queueSize       /* number of events in queue */
    )
```

** DESCRIPTION **

This routine creates an event queue. The first queue created for an event service becomes the default queue for that event service.

** RETURNS **

An event queue ID, or *UGL_NULL* on error.

** SEE ALSO **

uglEvt, uglEventPost, uglEventGet(), uglEventQDestroy()
**uglEventQDestroy()**

**NAME**

`uglEventQDestroy()` - destroy an event queue

**SYNOPSIS**

```c
UGL_STATUS uglEventQDestroy
    (    
    UGL_EVENT_SERVICE_ID eventServiceId, /* event service */
    UGL_EVENT_Q_ID       qId             /* event queue to be destroyed */
    )
```

**DESCRIPTION**

This routine destroys an event queue.

**RETURNS**

`UGL_STATUS_OK`, or a non zero value upon error.

**SEE ALSO**

`uglEvt`, `uglEventQCreate()`

---

**uglEventRouterGet()**

**NAME**

`uglEventRouterGet()` - get the event router associated with an event service

**SYNOPSIS**

```c
UGL_STATUS uglEventRouterGet
    (    
    UGL_EVENT_SERVICE_ID eventServiceId, /* event service */
    UGL_ID *             pEventRouterId  /* gets event router */
    )
```

**DESCRIPTION**

This routine gets the event router (e.g. window manager) associated with an event service.

**RETURNS**

`UGL_STATUS_OK`, or a non zero value upon error.

**SEE ALSO**

`uglEvt`, `uglEventRouterSet()`
**uglEventRouterSet()**

**NAME**

`uglEventRouterSet()` - set the event router associated with an event service

**SYNOPSIS**

```c
UGL_STATUS uglEventRouterSet
    (  
        UGL_EVENT_SERVICE_ID eventServiceId, /* event service */
        UGL_ID               eventRouterId   /* event router */
    )
```

**DESCRIPTION**

This routine sets the event router (e.g. window manager) associated with an event service.

**RETURNS**

`UGL_STATUS_OK`, or a non zero value upon error.

**SEE ALSO**

`uglEvt`, `uglEventRouterGet()`

---

**uglEventServiceCreate()**

**NAME**

`uglEventServiceCreate()` - create an event service

**SYNOPSIS**

```c
UGL_EVENT_SERVICE_ID uglEventServiceCreate
    (  
        UGL_RECT *          pInputRect,    /* input event bounding rect */
        UGL_CALLBACK_ITEM * pCallbackArray /* event handler initializer */
    )
```

**DESCRIPTION**

This routine creates an event service. An event service retrieves input events from input devices, provides an event handler for servicing events, and manages event queues for event storage and retrieval by applications.

`pInputRect` specifies a rectangle to which all input events must be bounded. It is normally a rectangle that represents the entire display.

`pCallbackArray` points to an array of `UGL_CALLBACK_ITEM` structures used to initialize the event service’s event handler. This event handler is responsible for routing events to the proper event queues, and for performing any other actions, such as cursor movement or front end processing (FEP) that may be required by the events. (See `uglEventHandlerCreate()` for more information.)

The result returned from the event service’s event handler must be the ID of the event queue into which the event is to be placed.
RETURNS

An event service ID, or UGL_NULL on error.

SEE ALSO

uglEvt, uglEventHandlerCreate()

----------

uglEventServiceDestroy()

NAME

uglEventServiceDestroy() - terminate an event service

SYNOPSIS

UGL_STATUS uglEventServiceDestroy

{
    UGL_EVENT_SERVICE_ID eventServiceId
}

DESCRIPTION

This routine causes an event service's input retrieval thread to terminate, and frees any
resources used by the event service.

RETURNS

UGL_STATUS_OK (0) or a non zero value on error.

SEE ALSO

uglEvt, uglEventServiceResume(), uglEventServiceIdle()

----------

uglEventServiceIdle()

NAME

uglEventServiceIdle() - idle an event service

SYNOPSIS

UGL_STATUS uglEventServiceIdle

{
    UGL_EVENT_SERVICE_ID eventServiceId
}

DESCRIPTION

This routine idles the event service eventServiceId.

RETURNS

UGL_STATUS_OK when the service is idle; otherwise UGL_STATUS_ERROR
when the service identifier is not legal

SEE ALSO

uglEvt, uglEventServiceResume(), uglEventServiceDestroy()
**uglEventServiceResume()**

**NAME**

`uglEventServiceResume()` - resume an event service

**SYNOPSIS**

```
UGL_STATUS uglEventServiceResume
    (    
        UGL_EVENT_SERVICE_ID eventServiceId
    )
```

**DESCRIPTION**

This routine resumes the event service `eventServiceId`.

**RETURNS**

`UGL_STATUS_OK` when the service is resumed; otherwise `UGL_STATUS_ERROR` when the service identifier is not legal.

**SEE ALSO**

`uglEvt`, `uglEventServiceIdle()`, `uglEventServiceDestroy()`

---

**uglInputDeviceAdd()**

**NAME**

`uglInputDeviceAdd()` - adds an input device to an event service

**SYNOPSIS**

```
UGL_INPUT_DEVICE * uglInputDeviceAdd
    (    
        UGL_EVENT_SERVICE_ID eventServiceId /* event service */
    )
```

**DESCRIPTION**

This routine adds a device for the specified event service `eventServiceId`.

**RETURNS**

Pointer to the device control structure when success; otherwise `UGL_NULL`.

**ERRNO**

N/A

**SEE ALSO**

`uglEvt`
**uglInputDeviceDestroy()**

**NAME**

`uglInputDeviceDestroy() - destroy an input device`

**SYNOPSIS**

```c
UGL_STATUS uglInputDeviceDestroy
    (UGL_INPUT_DEVICE_ID inDevId /* input device identifier */);
```

**DESCRIPTION**

This routine destroys a device, that is remove from the event service.

**RETURNS**

`UGL_STATUS_OK`, or `UGL_STATUS_ERROR` if the operation fails.

**SEE ALSO**

`uglEvt`

---

**uglInputDeviceInfo()**

**NAME**

`uglInputDeviceInfo() - control/obtain information`

**SYNOPSIS**

```c
UGL_STATUS uglInputDeviceInfo
    (UGL_INPUTDEVICE_ID inDevId, /* input device identifier */
     UGL_DEVICE_REQ devRequest, /* device control request */
     void * arg /* argument for control request */);
```

**DESCRIPTION**

This routine obtains information from an input device and also sets new information to the device. It is used to modify the operation of a device. Support operations are:

**UGL_DEVICE_SET_POINTER_LOCATION**

Modify the location of the absolute position of the pointer within pointers constraints. This does NOT modify the location of the pointer as viewed on the screen. It only modifies the location of the pointer as reported in the input event. To modify the pointer’s location, an `UGL_POINT` structure is set to identify the location. The following is an example of setting the pointer location:

```c
UGL_POINT pt;

pt.x = newPtrX;
```
pt.y = newPtrY;
uglInputDeviceInfo (inSvcId, ptrDevNum,
                      UGL_DEVICE_SET_POINTER_LOCATION, &pt);

UGL_DEVICE_SET_SCREEN_CONTRAINT
Modify the constraints of the cursor. This modifies the minimum and maximum
values that the pointer can reside. Movement of the pointer beyond these constraints
is not possible. The following is an example of setting the screen constraints:

    UGL_RECT rect;

    rect.left = newMinX;
    rect.top = newMinY;
    rect.right = newMaxX;
    rect.bottom = newMaxY;
    uglInputDeviceInfo (inSvcId, ptrDevNum,
                        UGL_DEVICE_SET_SCREEN_CONTRAINT, &rect);

UGL_DEVICE_SET_LED_CONTROL
Modify the state of the keyboard LED processing. By default, the LEDs are handled
by the keyboard driver. This may be changed to allow an application to change the
state of the LEDs. The following example illustrates the manner in which to switch
the operating mode:

    uglInputDeviceInfo (inSvcId, kbdDevNum, UGL_DEVICE_SET_LED_CONTROL,
                        UGL_TRUE);

UGL_DEVICE_SET_LED
Modify the state of the LEDs when application control is enabled (see
UGL_DEVICE_SET_LED_CONTROL). This command allows an application to
change each of the LEDs on the keyboard. The command changes all the LEDs, when
the specific LED is set then the LED is turned on and when an LED is clear, then the
LED is turned off. The following example illustrates turning on the scroll lock and
caps lock and turning off the num lock LEDs:

    ledValue = UGL_KEYBOARD_CAPS_LOCK | UGL_KEYBOARD_SCROLL_LOCK;
    uglInputDeviceInfo (inSvcId, kbdDevNum, UGL_DEVICE_SET_LED, ledValue);

UGL_DEVICE_GET_LED
Obtain the current state of the keyboard LEDs. The following example illustrates the
query of the keyboard to obtain the current state of the keyboard LEDs:

    uglInputDeviceInfo (inSvcId, kbdDevNum, UGL_DEVICE_SET_LED, &ledValue);

UGL DEVICE_SET_CALIBRATION
Set the calibration data for an absolute pointer.

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if the operation fails.

ERRNO
N/A
uglInputEventGet() - retrieve an event from an event service

NAME
uglInputEventGet() - retrieve an event from an event service

SYNOPSIS
#if CPU != SIMNT UGL_STATUS uglInputEventGet
{(number)
    UGL_EVENT_SERVICE_ID eventServiceId, /* event service */
    UGL_TIMEOUT timeout, /* time to wait for an event */
    UGL_EVENT * pEvent, /* pointer to event buffer */
    UGL_SIZE eventSize, /* size of event buffer */
}

DESCRIPTION
This routine retrieves an event from an event service. The next event available from the
input service inSvcId is returned in pEvent.

The timeout specifies how long the function should pend awaiting an event when the
queue is empty. In addition to specifying a timeout, the following values may be specified:

UGL_NO_WAIT
    an immediate return when no event is available

UGL_WAIT_FOREVER
    waits until an event becomes available

Other values of timeout specifies the milliseconds that the routine will wait for an event to
become present.

When reading events from the queue, pointer input events are compressed to return the
latest pointer movement with no button changes.

RETURNS
UGL_STATUS_OK when event is present and is returned within pEvent,
UGL_STATUS_EMPTY when no event is present, or UGL_STATUS_ERROR when
event queue has problems.

ERRNO
N/A

SEE ALSO ugEvt
uglWin

NAME

uglWin - WindML Windowing API

ROUTINES

winAppCreate() - create a windowing application context
winAppDestroy() - destroy a windowing application context
winAttach() - attach a child window to a parent window
winAppGet() - get the application context associated with a window
winCallbackCursorMove() - event service callback for moving cursor
winCallbackEventRoute() - event service callback for event routing with windowing
winClipModeGet() - get the clipping mode used by a window
winClipModeSet() - set clipping mode to be used by a window
winCount() - get the number of child windows
winCreate() - create a window
winDataGet() - get data associated with a window
winDataSet() - set data associated with a window
winDestroy() - destroy a window
winDetach() - detach a child window from a parent window
winDisplayGet() - get the display associated with a window
winDirtyRegionGet() - get a window's dirty region
winDrawEnd() - stop drawing to a window
winDrawStart() - start drawing to a window
winEventGet() - get an event for a windowing application
winEventQGet() - get the event queue associated with a window
winEventRouterCreate() - create an event router
winEventRouterDestroy() - destroy an event router
winEventRouterGet() - get the event router associated with a window
winFirst() - get a window's first child

355
This library provides the windowing interface for WindML.

**DESCRIPTION**

This library provides the windowing interface for WindML.

---

**winAppCreate()**

**NAME**

winAppCreate() - create a windowing application context

**SYNOPSIS**

UGL_APP_ID winAppCreate

    (  
    UGL_EVENT_ROUTER_ID eventRouterId, /* event router (eg. window manager) */
    UGL_SIZE qSize /* application event queue */
    )

356
DESCRIPTION
This routine creates an application context for a windowing application.

RETURNS
An application ID, or UGL_NULL_ID on error.

SEE ALSO
uglWin, winAppDestroy()

---

**winAppDestroy()**

NAME

*winAppDestroy()* - destroy a windowing application context

SYNOPSIS

```c
UGL_STATUS winAppDestroy
{
    UGL_APP_ID appId
}
```

DESCRIPTION
This routine destroys an application context create by a previous call to *winAppCreate()*

RETURNS
UGL_STATUS_OK, or a non zero value on error.

SEE ALSO
uglWin, winAppCreate()

---

**winAppGet()**

NAME

*winAppGet()* - get the application context associated with a window

SYNOPSIS

```c
UGL_APP_ID winAppGet
{
    UGL_WINDOW_ID windowId
}
```

DESCRIPTION
This routine gets the application context associated with a window.

RETURNS
An application ID, or UGL_NULL_ID on error.

SEE ALSO
uglWin
**winAttach()**

**NAME**

`winAttach()` - attach a child window to a parent window

**SYNOPSIS**

```c
UGL_STATUS winAttach
    (    
        UGL_WINDOW_ID parentId,   /* parent window */
        UGL_WINDOW_ID childId,    /* child window */
        UGL_WINDOW_ID nextId      /* child window */
    )
```

**DESCRIPTION**

This routine attaches one window to another, creating a parent-child relationship. The process of attaching a window generates a containment hierarchy and causes visible and dirty regions to be created as necessary for the affected windows. The window is attached directly before (or in other words below) the window specified by `nextId`. If `nextId` is `UGL_NULL_ID`, then the window is placed at the end of the window list, which is the top of the z-order.

**RETURNS**

`UGL_STATUS_OK`, or a non zero value on error.

**SEE ALSO**

`uglWin`

---

**winCallbackCursorMove()**

**NAME**

`winCallbackCursorMove()` - event service callback for moving cursor

**SYNOPSIS**

```c
UGL_STATUS winCallbackCursorMove
    (    
        UGL_ID       objectId,
        UGL_EVENT *  pEvent,
        UGL_UINT32 * pResult
    )
```

**DESCRIPTION**

This callback routine is called by an event service to cause a cursor to be moved whenever a pointer device is moved.

**RETURNS**

`UGL_STATUS_OK`, or a non zero value on error.

**SEE ALSO**

`uglWin`
**winCallbackEventRoute()**

**NAME**

*winCallbackEventRoute() - event service callback for event routing with windowing*

**SYNOPSIS**

```c
UGL_STATUS winCallbackEventRoute
    (    
        UGL_ID       objectId,
        UGL_EVENT *  pEvent,
        UGL_UINT32 * pResult
    )
```

**DESCRIPTION**

This callback routine is called by an event service to route input events with windowing.

**RETURNS**

*UGL_STATUS_OK*, or a non zero value on error.

**SEE ALSO**

uglWin

---

**winClipModeGet()**

**NAME**

*winClipModeGet() - get the clipping mode used by a window*

**SYNOPSIS**

```c
UGL_STATUS winClipModeGet
    (    
        UGL_WINDOW_ID windowId, /* ID of window to get clip mode from */
        UGL_CLIP_MODE * pClipMode /* clip mode */
    )
```

**DESCRIPTION**

This routine gets the clipping mode used by a window.

**RETURNS**

*UGL_STATUS_OK* or a non zero value on error.

**SEE ALSO**

window, *winClipModeSet()*
**winClipModeSet()**

**NAME**

`winClipModeSet()` - set clipping mode to be used by a window

**SYNOPSIS**

```c
UGL_STATUS winClipModeSet
    (    
        UGL_WINDOW_ID windowId, /* ID of window to set clip mode to */
        UGL_CLIP_MODE clipMode /* clip mode */
    )
```

**DESCRIPTION**

This routine sets the clipping mode to be used by a window. The clipping mode is set on a per window basis. The following clipping modes are available:

*WIN_CLIP_MODE_CLIP_CHILDREN* (default)

With this clipping mode, a window is never permitted to draw over the top of its children. All output to the window is clipped around the area occupied by its children. This mode generally produces the best performance, but requires more memory for region data.

*WIN_CLIP_MODE_PAINTERS*

With this clipping mode, windows are drawn according to the painter's algorithm. This algorithm describes how a painter might apply paint. First, the topmost window is drawn, including the area occupied by its children. Then, its children are drawn on top to produce the proper appearance. This mode requires the least memory, but may produce flickering effects in some circumstances.

**RETURNS**

*UGL_STATUS_OK* or a non zero value on error.

**SEE ALSO**

`window`, `winClipModeGet()`

---

**winCount()**

**NAME**

`winCount()` - get the number of child windows

**SYNOPSIS**

```c
UGL_SIZE winCount
    (    
        UGL_WINDOW_ID windowId /* window ID */
    )
```

**DESCRIPTION**

This routine returns the number of child windows contained within the window identified by *windowId*. 
**winCreate()**

**NAME**  
`winCreate()` - create a window

**SYNOPSIS**

```c
UGL_WINDOW_ID winCreate
    (  
        UGL_APP_ID appId,        /* application context */  
        UGL_POS  x,             /* left coordinate of window */  
        UGL_POS  y,             /* top coordinate of window */  
        UGL_SIZE width,         /* width of window */  
        UGL_SIZE height         /* height of window */  
    )
```

**DESCRIPTION**
This routine creates a window. A window is simply a rectangular object that can be moved, sized, placed in a containment hierarchy, etc. `x` and `y` specify the coordinates of the top left corner of the window. `width` and `height` specify the size of the window.

**RETURNS**
A `UGL_LIST_ID`, or `UGL_NULL_ID` on error.

**SEE ALSO**
`uglWin`

**winDataGet()**

**NAME**  
`winDataGet()` - get data associated with a window

**SYNOPSIS**

```c
UGL_STATUS winDataGet
    (  
        UGL_WINDOW_ID windowId,  
        void ** ppData        
    )
```

**DESCRIPTION**
This routine is used by applications to retrieve data stored with a window by a previous call to `winDataSet()`. This data may be used to keep track of derived structures or other such purposes.
RETURNS

UGL_STATUS_OK or a non zero value on error.

SEE ALSO

uglWin

---

winDataSet()

NAME

winDataSet() - set data associated with a window

SYNOPSIS

UGL_STATUS winDataSet

{
    UGL_WINDOW_ID windowId, /* window Id */
    void * pData       /* data to store with window */
}

DESCRIPTION

This routine is used by applications to store data with a window. This data may be used to keep track of derived structures or other such purposes.

RETURNS

UGL_STATUS_OK or a non zero value on error.

SEE ALSO

uglWin

---

winDestroy()

NAME

winDestroy() - destroy a window

SYNOPSIS

UGL_STATUS winDestroy

{
    UGL_WINDOW_ID windowId    /* ID of window to be destroyed */
}

DESCRIPTION

This routine destroys the window identified by windowId and frees all associated resources.

RETURNS

UGL_STATUS_OK, or a non zero value on error.

SEE ALSO

uglWin
**winDetach()**

**NAME**

`winDetach()` - detach a child window from a parent window

**SYNOPSIS**

```c
UGL_STATUS winDetach

(  UGL_WINDOW_ID parentId,  /* parent window */
   UGL_WINDOW_ID childId  /* child window */
)
```

**DESCRIPTION**

This routine detaches a child window from a parent window. The child window is not destroyed.

**RETURNS**

`UGL_STATUS_OK`, or a non zero value on error.

**SEE ALSO**

`uglWin`

---

**winDirtyRegionGet()**

**NAME**

`winDirtyRegionGet()` - get a window’s dirty region

**SYNOPSIS**

```c
UGL_REGION_ID winDirtyRegionGet

(  UGL_WINDOW_ID windowId    /* window ID */
)
```

**DESCRIPTION**

This routine returns the dirty region of the window identified by `windowId`. The dirty region is the portion of the window that has been obscured and exposed, or otherwise requires a refresh. If `UGL_NULL_ID` is returned, the entire window is dirty. By definition, a window’s dirty region is a sub-set of its visible region.

**RETURNS**

A window’s dirty region

**SEE ALSO**

`uglWin`
**winDisplayGet()**

**NAME**

`winDisplayGet()` - get the display associated with a window

**SYNOPSIS**

```c
UGL_DEVICE_ID winDisplayGet
    (  
        UGL_WINDOW_ID windowId  
    )
```

**DESCRIPTION**

This routine gets the display associated with window.

**RETURNS**

A display ID, or `UGL_NULL_ID` on error.

**SEE ALSO**

`uglWin`

---

**winDrawEnd()**

**NAME**

`winDrawEnd()` - stop drawing to a window

**SYNOPSIS**

```c
UGL_STATUS winDrawEnd
    (  
        UGL_WINDOW_ID windowId, /* ID of window */
        UGL_GC_ID gc, /* GC used for drawing */
        UGL_BOOL clearDirty /* clear dirty region ? */
    )
```

**DESCRIPTION**

This routine is used to end a batch of drawing operations to a window. All drawing performed to a window should take place between calls to `winDrawStart()` and `winDrawEnd()`. The following general operations are performed by `winDrawEnd()`:

- `uglBatchEnd()` is used to end a drawing batch.
- The window’s dirty region is cleared if `clearDirty` is TRUE.
- The window identified by `windowId` is unlocked so that it can be moved, sized, and so on.

`clearDirty` should be set to `UGL_TRUE` when drawing in response to events of type `WIN_EVENT_TYPE_DRAW`. This prevents additional draw events from being sent to the window until it becomes dirty again.

**RETURNS**

`UGL_STATUS_OK` or a non zero value on error.
**DESCRIPTION**

This routine is used to start a batch of drawing operations to a window. All drawing performed to a window should take place between calls to `winDrawStart()` and `winDrawEnd()`. The following general operations are performed by `winDrawStart()`:

- The window identified by `windowId` is locked so that it can't be moved, sized, etc. during the drawing batch.
- The GC identified by `gc` is given the clipping region and view port needed when drawing to the window.
- `uglBatchStart()` is used to start a drawing batch.

If `clipToDirty` is true, drawing is clipped to the dirty portion of the window. Otherwise, drawing is only clipped to the visible portion of the window. The dirty region is the portion of a window that has been invalidated and needs to be redrawn. This can happen when the window is moved or sized, when it is exposed by the closure of a window in front of it, or for various other reasons.

`clipToDirty` should be set to `UGL_TRUE` when drawing in response to events of type `WIN_EVENT_TYPE_DRAW`. This prevents unnecessary updates, which improves performance and reduces flicker.

**RETURNS**

`UGL_STATUS_OK` or a non zero value on error.

**SEE ALSO**

`window`, `winDrawEnd()`, `uglBatchEnd()`, `winClipModeSet()`
**winEventGet()**

**NAME**

*winEventGet()* - get an event for a windowing application

**SYNOPSIS**

```c
UGL_STATUS winEventGet
(
    UGL_APP_ID appId,
    UGL_EVENT * pEvent,
    UGL_TIMEOUT timeout
)
```

**DESCRIPTION**

This routine gets an event for the windowing application identified by `appId`. The event information is copied to the location specified by `pEvent`.

**RETURNS**

- `UGL_STATUS_OK`, or
- a non-zero value on error

**SEE ALSO**

`uglWin`

**winEventQGet()**

**NAME**

*winEventQGet()* - get the event queue associated with a window

**SYNOPSIS**

```c
UGL_EVENT_Q_ID winEventQGet
(
    UGL_WINDOW_ID windowId
)
```

**DESCRIPTION**

This routine gets the event queue associated with a window.

**RETURNS**

- An event queue ID, or `UGL_NULL_ID` on error.

**SEE ALSO**

`uglWin`
**winEventRouterCreate()**

**NAME**

`winEventRouterCreate()` - create an event router

**SYNOPSIS**

```c
UGL_EVENT_ROUTER_ID winEventRouterCreate
    (  
        UGL_DEVICE_ID displayId,
        UGL_EVENT_SERVICE_ID eventServiceId
    )
```

**DESCRIPTION**

This routine creates an event router.

**RETURNS**

An event router ID, or `UGL_NULL` on error.

**SEE ALSO**

`uglWin`, `winAppCreate()`

---

**winEventRouterDestroy()**

**NAME**

`winEventRouterDestroy()` - destroy an event router

**SYNOPSIS**

```c
UGL_STATUS winEventRouterDestroy
    (  
        UGL_EVENT_ROUTER_ID eventRouterId
    )
```

**DESCRIPTION**

This routine destroys an event router.

**RETURNS**

`UGL_STATUS_OK`, or a non zero value on error.

**SEE ALSO**

`uglWin`, `winEventROuterCreate()`
**winEventRouterGet()**

**NAME**

`winEventRouterGet()` - get the event router associated with a window

**SYNOPSIS**

```c
UGL_EVENT_ROUTER_ID winEventRouterGet
(    
    UGL_WINDOW_ID windowId
)
```

**DESCRIPTION**

This routine gets the event router associated with a window.

**RETURNS**

An event router ID, or `UGL_NULL_ID` on error.

**SEE ALSO**

`uglWin`

---

**winFirst()**

**NAME**

`winFirst()` - get a window’s first child

**SYNOPSIS**

```c
UGL_WINDOW_ID winFirst
(    
    UGL_WINDOW_ID windowId /* parent window */
)
```

**DESCRIPTION**

This routine returns the first child window of a parent window. The first window is at the bottom of the z-order.

**RETURNS**

A `UGL_WINDOW_ID`, or `UGL_NULL_ID` if window has no children.

**SEE ALSO**

`uglWin`
**winKeyboardGrab()**

**NAME**

`winKeyboardGrab()` - route all keyboard input to a window

**SYNOPSIS**

```c
UGL_STATUS winKeyboardGrab
    (    
    UGL_WINDOW_ID windowId
    )
```

**DESCRIPTION**

This routine causes all keyboard input to be routed to the window specified by `windowId`.

**RETURNS**

`UGL_STATUS_OK`, or a non zero value on error.

**SEE ALSO**

`uglWin`, `winKeyboardUngrab()`

---

**winKeyboardGrabGet()**

**NAME**

`winKeyboardGrabGet()` - get the window to which the keyboard is currently grabbed

**SYNOPSIS**

```c
UGL_WINDOW_ID winKeyboardGrabGet
    (    
    UGL_EVENT_ROUTER_ID eventRouterId
    )
```

**DESCRIPTION**

This routine gets the window to which the keyboard is currently grabbed.

**RETURNS**

A `UGL_WINDOW_ID`, or `UGL_NULL_ID` if no window has grabbed the keyboard

**SEE ALSO**

`uglWin`

---

**winKeyboardUngrab()**

**NAME**

`winKeyboardUngrab()` - release keyboard grab for a window

**SYNOPSIS**

```c
UGL_STATUS winKeyboardUngrab
    (    
    UGL_WINDOW_ID windowId
    )
```
### DESCRIPTION
This routine releases keyboard grab performed by a previous call to `winKeyboardGrab()`. This causes keyboard events to be routed to the default event queue.

### RETURNS
`UGL_STATUS_OK`, or a non zero value on error.

### SEE ALSO
`uglWin`, `winKeyboardGrab()`

---

**winLast()**

### NAME
`winLast()` - get a window’s last child

### SYNOPSIS
```c
UGL_WINDOW_ID winLast
{
    UGL_WINDOW_ID windowId /* parent window */
}
```

### DESCRIPTION
This routine returns the last child window of a parent window. The last window is at the top of the z-order.

### RETURNS
A `UGL_WINDOW_ID`, or `UGL_NULL_ID` if window has no children.

### SEE ALSO
`uglWin`

---

**winLock()**

### NAME
`winLock()` - lock a window for exclusive use

### SYNOPSIS
```c
void winLock
{
    UGL_WINDOW_ID windowId /* window ID */
}
```

### DESCRIPTION
This routine locks the window for exclusive use by the calling task. Calls to `uglWindowLock()` may be nested, and `winUnlock()` must be called the same number of times as `winLock()` to unlock the window.
A window should be locked before using its visible and dirty regions for drawing operations to prevent another task to modifying these regions during the draw.

**RETURNS**

UGL_STATUS_OK or a non zero value on error.

**SEE ALSO**

uglWin

---

**winMove()**

**NAME**

`winMove()` - move a window by a delta value

**SYNOPSIS**

```c
UGL_STATUS winMove
(
    UGL_WINDOW_ID windowId,   /* window ID */
    int           dx,         /* delta on x-axis */
    int           dy          /* delta on y-axis */
)
```

**DESCRIPTION**

This routine moves the window identified by a delta of `dx` and `dy`. The actual move is performed by a call to `winRectSet()`.

**RETURNS**

UGL_STATUS_OK or a non zero value on error.

**SEE ALSO**

uglWin

---

**winMoveTo()**

**NAME**

`winMoveTo()` - move a window to an absolute position

**SYNOPSIS**

```c
UGL_STATUS winMoveTo
(
    UGL_WINDOW_ID windowId,   /* window ID */
    UGL_POS       x,          /* new location for left side of window */
    UGL_POS       y           /* new location for top side of window */
)
```

**DESCRIPTION**

This routine moves the window identified by `window` so the its top left corner is located at the coordinate `(x, y)`. The actual move is performed by a call to `winRectSet()`.
RETURNS UGL_STATUS_OK or a non zero value on error.

SEE ALSO uglWin

---

**winNext()**

**NAME**

winNext() - get the next window

**SYNOPSIS**

UGL_WINDOW_ID winNext

(    
    UGL_WINDOW_ID windowId    /* current window */
)

**DESCRIPTION**

This routine returns the window immediately following windowId in a window list.

**RETURNS**

The ID of the next window in the list, or UGL_NULL if at the end of the list.

**SEE ALSO**

uglWin

---

**winNth()**

**NAME**

winNth() - get a window’s Nth child

**SYNOPSIS**

UGL_WINDOW_ID winNth

(    
    UGL_WINDOW_ID windowId,   /* parent window */
    UGL_ORD       windowNum   /* number of window to get */
)

**DESCRIPTION**

This routine returns the Nth child window of a parent window. windowNum specifies the number of the window to get. If windowNum is less than zero, it specifies the window number counting backwards from the last window. Thus, window number -1 is the last window or, in other words, the window at the top of the z-order.

**RETURNS**

A UGL_WINDOW_ID, or UGL_NULL_ID if there is no Nth child.

**SEE ALSO**

uglWin
### winParent()

**NAME**

`winParent()` - get a window’s parent

**SYNOPSIS**

```c
UGL_WINDOW_ID winParent
{
    UGL_WINDOW_ID windowId /* window ID */
}
```

**DESCRIPTION**

This routine returns the parent window of the window identified by `windowId`.

**RETURNS**

A `UGL_WINDOW_ID` or `UGL_NULL_ID` if the window has no parent.

**SEE ALSO**

`uglWin`

### winPointerGrab()

**NAME**

`winPointerGrab()` - route all pointer input to a window

**SYNOPSIS**

```c
UGL_STATUS winPointerGrab
{
    UGL_WINDOW_ID windowId
}
```

**DESCRIPTION**

This routine causes all pointer input to be routed to the window specified by `windowId`.

**RETURNS**

`UGL_STATUS_OK`, or a non zero value on error.

**SEE ALSO**

`uglWin`, `winPointerUngrab()`

### winPointerGrabGet()

**NAME**

`winPointerGrabGet()` - get the window to which the pointer is currently grabbed

**SYNOPSIS**

```c
UGL_WINDOW_ID winPointerGrabGet
{
    UGL_EVENT_ROUTER_ID eventRouterId
}
```
DESCRIPTION
This routine gets the window to which the pointer is currently grabbed.

RETURNS
A UGL_WINDOW_ID, or UGL_NULL_ID if no window has grabbed the pointer

SEE ALSO
uglWin

---

**winPointerUngrab()**

**NAME**
*winPointerUngrab()* - release pointer grab for a window

**SYNOPSIS**

```
UGL_STATUS winPointerUngrab
{
    UGL_WINDOW_ID windowId
}
```

**DESCRIPTION**
This routine releases pointer grab performed by a previous call to *winKeyboardGrab()*.
This causes pointer events to be routed to the window directly under the cursor.

**RETURNS**
UGL_STATUS_OK, or a non zero value on error.

**SEE ALSO**
uglWin, *winPointerGrab()*

---

**winPrev()**

**NAME**
*winPrev()* - get the previous window

**SYNOPSIS**

```
UGL_WINDOW_ID winPrev
{
    UGL_WINDOW_ID windowId   /* current window */
}
```

**DESCRIPTION**
This routine returns the window before *windowId* in a window list.

**RETURNS**
The ID of the previous window in the list, or UGL_NULL if at the start of the list.

**SEE ALSO**
uglWin
**winRaise()**

**NAME**

`winRaise()` - raise a window to the top of the z-order

**SYNOPSIS**

```c
UGL_STATUS winRaise

    (    
        UGL_WINDOW_ID windowId    /* window ID */
    )
```

**DESCRIPTION**

This routine raises a window to the top of the z-order within its parent.

**RETURNS**

`UGL_STATUS_OK` or a non zero value on error.

**SEE ALSO**

`uglWin`

---

**winRectExpose()**

**NAME**

`winRectExpose()` - expose a rectangular portion of a window

**SYNOPSIS**

```c
UGL_STATUS winRectExpose

    (    
        UGL_WINDOW_ID windowId, /* window ID */
        const UGL_RECT * pRect    /* expose rect */
    )
```

**DESCRIPTION**

This routine causes a rectangular portion of a window to become visible by adding the rectangle to the windows visible region. The revealed portion of the window is also added to the window’s dirty region.

**WARNING**

This routine is not normally required by applications and should be used only with care.

**RETURNS**

`UGL_STATUS_OK` or a non zero value on error.

**SEE ALSO**

`uglWin`
**winRectGet()**

**NAME**  
*winRectGet()* - get the dimensions of a window

**SYNOPSIS**  
```c
UGL_STATUS winRectGet
    (  
        UGL_WINDOW_ID windowId, /* window ID */
        UGL_RECT * pRect       /* gets window rect */
    )
```

**DESCRIPTION**  
This routine gets the rectangle that encloses the window identified by *windowId* and places it in the location specified by *pRect*.

**RETURNS**  
*UGL_STATUS_OK* or a non zero value on error.

**SEE ALSO**  
*uglWin*

---

**winRectHide()**

**NAME**  
*winRectHide()* - hide a rectangular portion of a window

**SYNOPSIS**  
```c
UGL_STATUS winRectHide
    (  
        UGL_WINDOW_ID windowId, /* window ID */
        const UGL_RECT * pRect     /* hide rect */
    )
```

**DESCRIPTION**  
This routine hides a rectangular portion of a window by removing the rectangle from the window’s visible region. Portions of child windows overlapping the hide rectangle are also hidden.

**WARNING**  
This routine is not normally required by applications and should be used only with care.

**RETURNS**  
*UGL_STATUS_OK* or a non zero value on error.

**SEE ALSO**  
*uglWin*
**winRectInvalidate()**

**NAME**

*winRectInvalidate()* - add a rectangle to a window’s dirty region

**SYNOPSIS**

```c
UGL_STATUS winRectInvalidate
    (        
        UGL_WINDOW_ID    windowId, /* window ID */
        const UGL_RECT * pRect     /* invalid rect */
    )
```

**DESCRIPTION**

This routine adds a rectangle to a window’s dirty region, indicating that it needs to be refreshed.

**RETURNS**

UGL_STATUS_OK or a non zero value on error.

**SEE ALSO**

uglWin

---

**winRectSet()**

**NAME**

*winRectSet()* - set the dimensions of a window;

**SYNOPSIS**

```c
UGL_STATUS winRectSet
    (        
        UGL_WINDOW_ID    windowId, /* window ID */
        const UGL_RECT * pRect     /* window rect */
    )
```

**DESCRIPTION**

This routine sets the size and position of a window to the rectangle specified by pRect.

**RETURNS**

UGL_STATUS_OK or a non zero value on error.

**SEE ALSO**

uglWin
**winRectValidate()**

NAME

*winRectValidate()* - remove a rectangle from a window’s dirty region

SYNOPSIS

```
UGL_STATUS winRectValidate
    (    
        UGL_WINDOW_ID    windowId, /* window ID */
        const UGL_RECT * pRect     /* valid rect */
    )
```

DESCRIPTION

This routine removes a rectangle from a window’s dirty region.

RETURNS

*UGL_STATUS_OK* or a non zero value on error.

SEE ALSO

*uglWin*

**winRegionExpose()**

NAME

*winRegionExpose()* - expose a region of a window

SYNOPSIS

```
UGL_STATUS winRegionExpose
    (    
        UGL_WINDOW_ID windowId,   /* window ID */
        UGL_REGION_ID regionId    /* expose region */
    )
```

DESCRIPTION

This routine causes a portion of a window to become visible by adding a region to its visible region. The revealed portion of the window is also added to the window’s dirty region. The visible and dirty regions of the window’s children are also updated as appropriate.

`regionId` identifies the region to be added to the visible region of the window identified by `windowId`.

WARNING

This routine is not normally required by applications and should be used only with care.

RETURNS

*UGL_STATUS_OK* or a non zero value on error.

SEE ALSO

*uglWin*
**winRegionHide()**

**NAME**

`winRegionHide()` - hide a region of a window

**SYNOPSIS**

```c
UGL_STATUS winRegionHide
    (  
        UGL_WINDOW_ID windowId, /* window ID */  
        UGL_REGION_ID regionId    /* hide region */  
    )
```

**DESCRIPTION**

This routine hides a region of a window by removing the region from the window’s visible region. Portions of child windows overlapping the hide region are also hidden.

**WARNING**

This routine is not normally required by applications and should be used only with care.

**RETURNS**

`UGL_STATUS_OK` or a non zero value on error.

**SEE ALSO**

`uglWin`

---

**winRootGet()**

**NAME**

`winRootGet()` - get the root window

**SYNOPSIS**

```c
UGL_WINDOW_ID winRootGet
    (  
        UGL_EVENT_ROUTER_ID eventRouterId  
    )
```

**DESCRIPTION**

This routine gets the root window associated with an event router. The root window generally represents the display, and is an ancestor to all other windows.

**RETURNS**

`UGL_STATUS_OK`, or a non zero value on error.

**SEE ALSO**

`uglWin`, `rootWindowSet()`
**winRootSet()**

**NAME**

`winRootSet()` - set the root window

**SYNOPSIS**

```c
UGL_STATUS winRootSet
(
    UGL_EVENT_ROUTER_ID eventRouterId,
    UGL_WINDOW_ID rootWindowId
)
```

**DESCRIPTION**

This routine sets the root window associated with an event router. The root window generally represents the display, and is an ancestor to all other windows.

**RETURNS**

`UGL_STATUS_OK`, or a non zero value on error.

**SEE ALSO**

`uglWin`, `rootWindowGet()`

---

**winSizeSet()**

**NAME**

`winSizeSet()` - set the size of a window

**SYNOPSIS**

```c
UGL_STATUS winSizeSet
(
    UGL_WINDOW_ID windowId,   /* window ID */
    int           width,      /* new width for window */
    int           height      /* new height for window */
)
```

**DESCRIPTION**

This routine sets the size of a window to `width` pixels wide, and `height` pixels high. The actual sizing is performed by a call to `winRectSet()`.

**RETURNS**

`UGL_STATUS_OK` or a non zero value on error.

**SEE ALSO**

`uglWin`
**winUnlock()**

**NAME**

`winUnlock()` - unlock a window

**SYNOPSIS**

```c
void winUnlock
    (   UGL_WINDOW_ID windowId    /* window ID */
    )
```

**DESCRIPTION**

This routine unlocks a window that has been locked by a previous call to `winLock` so that it can be used by other tasks. `winUnlock()` must be called the same number of times as `winLock()` for the window to be unlocked.

**RETURNS**

UGL_STATUS_OK or a non zero value on error.

**SEE ALSO**

uglWin

---

**winVisibleRegionGet()**

**NAME**

`winVisibleRegionGet()` - get a window’s visible region

**SYNOPSIS**

```c
UGL_REGION_ID winVisibleRegionGet
    (   UGL_WINDOW_ID windowId    /* window ID */
    )
```

**DESCRIPTION**

This routine returns the visible region of the window identified by `windowId`. The visible region is the portion of the window that is not obscured by any other window. If UGL_NULL_ID is returned, the entire window is visible.

**RETURNS**

A window’s visible region

**SEE ALSO**

uglWin
**winZPosGet()**

**NAME**

winZPosGet() - get the z-order position of a window

**SYNOPSIS**

```c
UGL_ORD winZPosGet
(    
    UGL_WINDOW_ID windowId  /* window ID */
)
```

**DESCRIPTION**

This routine gets the z-order position of the window identified by `windowId`. The z-order position of a window is its position relative to its siblings in the stacking order. In other words, it is the window’s position along an imaginary z-axis that extends into the display. A value of one indicates that the window is at the top of the stacking order.

**RETURNS**

The depth of a window along the z-axis, or zero on error.

**SEE ALSO**

uglWin

---

**winZPosSet()**

**NAME**

winZPosSet() - set the z-order position of a window

**SYNOPSIS**

```c
UGL_STATUS winZPosSet
(    
    UGL_WINDOW_ID windowId,    /* window Id */
    UGL_ORD       zPos        /* z position */
)
```

**DESCRIPTION**

This routine sets the z-order position of the window identified by `windowId`. The z-order position of a window is its position relative to its siblings in the stacking order. In other words, it is the window’s position along an imaginary z-axis that extends into the display. A value of one indicates that the window is at the top of the stacking order.

**RETURNS**

UGL_STATUS_OK or a non-zero value on error.

**SEE ALSO**

uglWin

---

382
uglExt

NAME

uglExt - Universal Graphics Library Extension API

ROUTINES

uglCallbackCursorMove() - event service callback for moving cursor
uglJpegFromDDB() - create a JPEG image from a bitmap
uglJpegInit() - initializes the UGL JPEG support
uglJpegModeGet() - get JPEG image compress/decompress parameters
uglJpegModeSet() - set JPEG image compress/decompress parameters
uglJpegToDDBFromData() - decompress JPEG image and write to bitmap
uglJpegToDDBFromFile() - decompress JPEG image and write to bitmap
uglMediaDeinit() - deinitialize UGL’s Media layer
uglMediaInit() - initialize UGL’s Media layer
uglOverlayCreate() - creates an overlay surface
uglOverlayDestroy() - destroys an overlay surface
uglOverlayMove() - moves an overlay surface
uglVideoAdapterInfo() - obtain/set video adapter information
uglVideoInit() - initializes the UGL Video support
uglVideoPortLock() - lock a video port for exclusive access
uglVideoPortUnlock() - unlock a video port
uglVideoStillPut() - puts a single video frame to a region on the display
uglVideoStop() - stop video on a video port
uglVideoStreamPut() - puts a video stream to a region on the display
wavChunkFind() - locate a chunk within a WAVE audio file
wavHeaderRead() - read a WAVE header from file

DESCRIPTION

This library provides the extension APIs for WindML.
uglCallbackCursorMove()

NAME

uglCallbackCursorMove() - event service callback for moving cursor

SYNOPSIS

UGL_STATUS uglCallbackCursorMove

(       
UGL_ID objectId,
UGL_EVENT * pEvent,
UGL_UINT32 * pResult
)       

DESCRIPTION

This callback routine is called by an event service to cause a cursor to be moved whenever a pointer device is moved.

SEE ALSO

uglExt

uglJpegFromDDB()

NAME

uglJpegFromDDB() - create a JPEG image from a bitmap

SYNOPSIS

UGL_STATUS uglJpegFromDDB

(       
UGL_JPEG_ID jpegId,       /* JPEG identifier */
UGL_DDB_ID ddbId,        /* source bitmap */
UGL_POS srcLeft,      /* left boundary of data */
UGL_POS srcTop,       /* top boundary of data */
UGL_POS srcRight,     /* right boundary of data to read */
UGL_POS srcBottom,    /* bottom boundary of data to read */
FILE * fp            /* file to write JPEG info */
)       

DESCRIPTION

This routine creates a JPEG image from a standard bitmap (UGL_DDB_ID). The JPEG image is created from the area of bitmap ddbId defined by the rectangle (srcLeft, srcTop, srcBottom, srcRight). The image is written to file specified by fp.

The image buffer is allocated by this routine.

RETURNS

UGL_STATUS_OK, or UGL_STATUS_ERROR if the operation fails.

ERRNO

N/A
SEE ALSO uglExt, uglJpegToDDB()

---

**uglJpegInit()**

**NAME**

*uglJpegInit*() - initializes the UGL JPEG support

**SYNOPSIS**

```c
UGL_JPEG_ID uglJpegInit
(           
    UGL_DEVICE_ID devId,    /* device identifier */
    int *         version   /* version of JPEG */
)
```

**DESCRIPTION**

This routine initializes the JPEG support for UGL on the specified device `devId`. This function must be called prior to calling any other JPEG functions. It returns an identifier for the JPEG support for the indicated device identifier. The JPEG identifier must be used for all subsequent JPEG function calls.

**RETURNS**

- `UGL_JPEG_ID` if successful, otherwise `UGL_NULL`.

**ERRNO**

N/A

**SEE ALSO**

uglExt, uglJpegModeSet()

---

**uglJpegModeGet()**

**NAME**

*uglJpegModeGet*() - get JPEG image compress/decompress parameters

**SYNOPSIS**

```c
UGL_STATUS uglJpegModeGet
(           
    UGL_JPEG_ID     jpegId,   /* JPEG identifier */
    UGL_JPEG_MODE * pJpegMode /* JPEG processing mode */
)
```

**DESCRIPTION**

This routine gets the current JPEG image compression/decompression parameters.

**RETURNS**

- `UGL_STATUS_OK` if successful, otherwise `UGL_STATUS_ERROR` if the operation fails.

**ERRNO**

N/A
**uglJpegModeSet()**

**NAME**

`uglJpegModeSet()` - set JPEG image compress/decompress parameters

**SYNOPSIS**

```c
UGL_STATUS uglJpegModeSet
    (            
    UGL_JPEG_ID jpegId,   /* JPEG identifier */
    UGL_JPEG_MODE * pJpegMode /* JPEG processing mode */
    )
```

**DESCRIPTION**

This routine sets JPEG image compression/decompression parameters. For the most part these parameters provide a trade-off between image quality and image processing time. Also provided are controls for setting image scaling.

**RETURNS**

`UGL_STATUS_OK`, or `UGL_STATUS_ERROR` if the operation fails.

**ERRNO**

N/A

**SEE ALSO**

`uglExt`, `uglJpegModeGet()`

---

**uglJpegToDDBFromData()**

**NAME**

`uglJpegToDDBFromData()` - decompress JPEG image and write to bitmap

**SYNOPSIS**

```c
UGL_STATUS uglJpegToDDBFromData
    (            
    UGL_JPEG_ID jpegId,   /* JPEG identifier */
    char * buf,
    int len,
    UGL_DDB_ID * pDdbId,
    UGL_RECT * pSrcRect,
    UGL_POS dstX,
    UGL_POS dstY
    )
```

**DESCRIPTION**

This routine decompresses a JPEG image and writes the image data to the bitmap pointed to by `pDdbId`. If `pDdbId` points to NULL, a bitmap will be created. Otherwise, the bitmap

---

SEE ALSO `uglExt, uglJpegModeSet()`
pointed to by pDdbId must be either a bitmap that was created by the uglBitmapCreate() function or the following:

UGL_DISPLAY_ID
   Specifies the frame buffer (the display itself). The rectangular area is relative to the top-left corner of the display. The image transfer is clipped only to the bounds of the display.

pSrcRect may be NULL if the information is not known ahead of time. If it is NULL, the source rectangle passed to the blit function will be the full size of the JPEG image.

The buffer pointed at by buf must contain a complete JPEG image before calling this function.

RETURNS
   UGL_STATUS_OK, or UGL_STATUS_ERROR if the gc is null.

ERRNO
   N/A

SEE ALSO
   uglExt, uglJpegToDDBFromFile()

---

uglJpegToDDBFromFile()

NAME
   uglJpegToDDBFromFile() - decompress JPEG image and write to bitmap

SYNOPSIS
   UGL_STATUS uglJpegToDDBFromFile
   (   
       UGL_JPEG_ID jpegId,      /* JPEG identifier */
       FILE * fp,          /* file to read JPEG image */
       UGL_DDB_ID * pDdbId,      /* destination bitmap */
       UGL_RECT * pSrcRect,    /* source rectangle */
       UGL_POS dstX,        /* destination X in DDB */
       UGL_POS dstY         /* destination Y in DDB */
   )

DESCRIPTION
   This routine decompresses a JPEG image and writes the image data to the bitmap pointed to by pDdbId. If pDdbId points to NULL, a bitmap will be created. Otherwise, the bitmap pointed to by pDdbId must be a bitmap that was created either by the uglBitmapCreate() function or the following:

UGL_DISPLAY_ID
   Specifies the frame buffer (the display itself). The rectangular area is relative to the top-left corner of the display. The image transfer is clipped only to the bounds of the display.
$pSrcRect$ may be NULL if the information is not known ahead of time. If it is NULL, the source rectangle passed to the blit function will be the full size of the JPEG image.

**RETURNS**

UGL\_STATUS\_OK, or UGL\_STATUS\_ERROR if the gc is null.

**ERRNO**

N/A

**SEE ALSO**

uglExt, uglJpegFromDDB()

---

### uglMediaDeinit()

**NAME**

uglMediaDeinit() - deinitialize UGL’s Media layer

**SYNOPSIS**

UGL\_STATUS uglMediaDeinit (void)

**DESCRIPTION**

This routine is used to deinitialize UGL’s media layer and free up any associated resources.

**SEE ALSO**

uglExt

---

### uglMediaInit()

**NAME**

uglMediaInit() - initialize UGL’s Media layer

**SYNOPSIS**

UGL\_STATUS uglMediaInit (void)

**DESCRIPTION**

uglMediaInit() is used to configure and initialize UGL’s media layer and is similar in function to uglInitialize(), which is used to configure and initialize the driver level components. The prime function of uglMediaInit() is to associate a display device with an event service and with windowing components.

**SEE ALSO**

uglExt
uglOverlayCreate()

NAME

uglOverlayCreate() - creates an overlay surface

SYNOPSIS

UGL_DEVICE_ID uglOverlayCreate

( UGL_DEVICE_ID devId,       /* graphics device ID */
  UGL_UINT32 overlayNum,  /* overlay level/number */
  UGL_COLOR_FORMAT format,  /* overlay format */
  UGL_UINT32 overlayMode, /* overlay mode */
  UGL_COLOR colorKey,    /* color key to use */
  UGL_POS left,        /* left pixel position */
  UGL_POS top,         /* top pixel position */
  UGL_POS right,       /* right pixel position */
  UGL_POS bottom,      /* bottom pixel position */
  UGL_UINT32 zOrder    /* position in overlay stack */
)

DESCRIPTION

This routine creates an overlay surface for the specified graphics device devId. The
hardware must be capable of supporting an overlay for an overlay device to be created.
The capabilities of the overlay support available with a graphics device varies
considerably. For example, number of overlay buffers, priority of the overlay buffers,
available pixel formats, stacking order of the overlays, and the method to merge overlay
buffer data with the primary frame buffer data.

The overlay surface is created the using the following parameters:

overlayInstance
  Identifies the overlay instance. This parameter can also be considered the overlay
  level. The first overlay instances start at 1. The number of instances or levels is
  specific to the graphics hardware.

format
  Identifies the format of each pixel within the overlay buffer.

overlayMode
  Provides a set of flags to identify that manner in which the overlay is to be created.
  This parameter is a set of bit-mapped flags to control the overlay creation.

  The mode may be to use color key UGL_OVERLAY_COLORKEY_MODE or to use a
  window UGL_OVERLAY_WINDOW_MODE.

  In the color key mode the data on the overlay region is presented on the display
  when the color in the specified region on the primary frame buffer matches the color
  as specified by colorKey.

  In the window mode the data on the overlay region is always presented to the
display. The window mode may be augmented by specifying that the overlay surface and the primary frame buffer should be alpha blended (requires that the hardware supports alpha blending). To enable alpha blending the display mode should be defined as (UGL_OVERLAY_WINDOW_MODE | UGL_OVERLAY_ALPHA_BLEND).

bob/weave mode identifies the manner in which the overlay image is sampled. The setting of this field usually has an effect when the overlay region is being used to display data from a video source (such as, a camera, VCR, etc.)

The Bob method UGL_OVERLAY_BOB_MODE of displaying video data is to display each odd and even field individually.

The Weave mode UGL_OVERLAY_WEAVE_MODE blends two interlaced images into a frame and produces a crisper image than one that is produced with the Bob method. The bob mode is the mode that is selected by default.

Selection of the bob/weave sampling method requires hardware support.

colorKey
When the overlayMode is set to UGL_OVERLAY_COLORKEY_MODE then this parameter specifies the color key.

deviceIdentifier
The left pixel position of the overlay surface on the primary frame buffer.

top
The top pixel position of the overlay surface on the primary frame buffer.

right
The right pixel position of the overlay surface on the primary frame buffer.

bottom
The bottom pixel position of the overlay surface on the primary frame buffer.

zOrder
Specifies the relationship of the overlay buffer with other overlay surfaces and the primary frame buffer. The primary frame buffer is always has a zOrder of 0 which means it has the highest priority. The next highest overlay surface would have a zOrder of 1.

The return value for the overlay surface is a UGL DEVICE ID. The created overlay surface may be used for all drawing functions. For example, a bitmap may be read using the uglBitmapRead(). In addition, an extension may be attached to the overlay; such as the JPEG extension. However, the an overlay will usually not support all drawing primitives.

returns
Device identifier assigned to the overlay buffer; otherwise UGL NULL when the overlay device was not created.

errno
N/A
uglOverlayDestroy()

NAME

uglOverlayDestroy() - destroys an overlay surface

SYNOPSIS

UGL_STATUS uglOverlayDestroy
{
    UGL_DEVICE_ID devId       /* graphics device ID */
}

DESCRIPTION

This routine destroys the overlay surface overlay for the graphics device devId thus removing it from the system.

RETURNS

UGL_STATUS_OK when the overlay surface was successfully removed; otherwise UGL_STATUS_ERROR

ERRNO

N/A

SEE ALSO

uglExt, uglOverlayCreate(), uglOverlayMove()

uglOverlayMove()

NAME

uglOverlayMove() - moves an overlay surface

SYNOPSIS

UGL_STATUS uglOverlayMove
{
    UGL_DEVICE_ID devId,       /* graphics device ID */
    UGL_POS       left,       /* X position */
    UGL_POS       top,        /* Y position */
    UGL_POS       right,      /* right pixel position */
    UGL_POS       bottom,     /* bottom pixel position */
    UGL_UINT32    zOrder      /* position in overlay stack */
}

DESCRIPTION

This routine moves an existing overlay surface, resizes the overlay surface, and changes its stacking order. The new position of the overlay surface is at (left, top, right, bottom). The stacking order is provided by zOrder where a zOrder of 1 places the overlay surface at a
level below the primary frame buffer and a zOrder of 2 places the overlay surface below the overlay surface with a zOrder of 1.

When an overlay is moved which results in an increase in the size of the overlay, the new size can not exceed the size of the overlay when the overlay was initially created.

RETURNS

UGL_STATUS_OK when the overlay surface was successfully moved;
otherwise UGL_STATUS_ERROR

ERRNO

N/A

SEE ALSO

uglExt, uglOverlayDevDestroy(), uglOverlayCreate()

uglVideoAdapterInfo()

NAME

uglVideoAdapterInfo() - obtain/set video adapter information

SYNOPSIS

UGL_STATUS uglVideoAdapterInfo
{
    UGL_VIDEO_ID videoId, /* device identifier */
    UGL_ORD     port,     /* video port */
    int        videoReq, /* request type */
    void *     pInfo      /* requested info */
}

DESCRIPTION

This routine obtains miscellaneous information from the video adapter. Information that the video adapter can be queried for or set are:

Obtain the number of video ports supported by video adapter
(UGL_VIDEO_AVAIL_GET). The available video ports are returned as a bitmapped 32 bit word where each bit set corresponds to a video port being available. For example, a value of 0x00000005 would indicate that there are two video ports available numbered 0 and 2.

Set the video port basic configuration (UGL_VIDEO_CONFIG_SET)
The basic configuration can only be defined once per video port, subsequent attempts to define the basic configuration is an error. The basic configuration defines the essential hardware configuration, which includes the mapping of video ports to overlay memory, definition of the input type as NTSC, PAL, SECAM; whether the video port is interlaced video, and the overlay surface that is to be used for the video. The configuration is defined by filling in the UGL_VIDEO_CONFIG data structure.

Obtain the video port configuration (UGL_VIDEO_CONFIG_GET)
This which was previously set using the UGL_VIDEO_CONFIG_SET operation. The
configuration is returned in a `UGL_VIDEO_CONFIG` data structure.

Obtain the video adapter capabilities (`UGL_VIDEO_PORT_INFO`)
This includes:
- Video encoding
- Size of incoming video (width and height)

This information is returned within the `UGL_VIDEO_INFO` data structure.

Set the video port attributes (`UGL_VIDEO_ATTRIB_SET`)
This includes:
- Degree of luminance brightness [0..256] where 0 is minimal brightness and 256 is maximum brightness
- Degree of luminance contrast [-128..128] where 0 is when luminance is off and negative values represent inverse contrast and positive values represent positive contrast.
- Degree of chrominance hue [-128..128] where -128 is minimal hue and 128 is maximum hue
- Degree of chrominance saturation [-128..128] where 0 is when chrominance is off and negative values represent inverse chrominance and positive values represent positive chrominance.

This information is provided in the `UGL_VIDEO_ATTRIB` data structure.

Obtain the video port attributes (`UGL_VIDEO_ATTRIB_GET`)
The same attributes that were set with `UGL_VIDEO_ATTRIB_SET` are obtained. The current settings for the video attributes is obtained within the `UGL_VIDEO_ATTRIB` data structure.

Determine if video is present (`UGL_VIDEO_PRESENT`)

**RETURNS**
`UGL_STATUS_OK` if successful, otherwise `UGL_STATUS_ERROR`.

**ERRNO**
N/A

**SEE ALSO**
`ulgExt`
uglVideoInit()

NAME
uglVideoInit() - initializes the UGL Video support

SYNOPSIS
UGL_VIDEO_ID uglVideoInit
{
    UGL_DEVICE_ID devId, /* device identifier */
    int * version /* version of extension */
}

DESCRIPTION
This routine initializes the video extension for the specified device devId. This function
must be called prior to calling any other video functions. It returns an identifier for the
video subsystem for the indicated device identifier. The video identifier must be used for
all subsequent video function calls. The current version of the video extension is returned
via the version parameter.

RETURNS
UGL_VIDEO_ID is successful, otherwise UGL_NULL.

ERRNO
N/A

SEE ALSO
uglExt

uglVideoPortLock()

NAME
uglVideoPortLock() - lock a video port for exclusive access

SYNOPSIS
UGL_STATUS uglVideoPortLock
{
    UGL_VIDEO_ID videoId, /* video device identifier */
    UGL_ORD port /* port to grab */
}

DESCRIPTION
This routine locks a video port; thus reserving for exclusive use.

RETURNS
UGL_STATUS_OK for success; otherwise UGL_STATUS_ERROR

ERRNO
N/A

SEE ALSO
uglExt
**uglVideoPortUnlock()**

**NAME**

`uglVideoPortUnlock()` - unlock a video port

**SYNOPSIS**

```c
UGL_STATUS uglVideoPortUnlock
    (    
    UGL_VIDEO_ID videoId, /* video device identifier */
    UGL_ORD port /* port to remove lock */
    )
```

**DESCRIPTION**

This routine unlocks a video port; thus freeing the port for use by other applications.

**RETURNS**

- `UGL_STATUS_OK` for success; otherwise `UGL_STATUS_ERROR`

**ERRNO**

N/A

**SEE ALSO**

`uglExt`, `uglVideoStillPut()`

---

**uglVideoStillPut()**

**NAME**

`uglVideoStillPut()` - puts a single video frame to a region on the display

**SYNOPSIS**

```c
UGL_STATUS uglVideoStillPut
    (    
    UGL_VIDEO_ID videoId, /* video device identifier */
    UGL_GC_ID gc, /* graphics context */
    UGL_ORD port, /* video port number */
    UGL_POS srcX, /* video source X position */
    UGL_POS srcY, /* video source Y position */
    UGL_SIZE srcWidth, /* width of video source */
    UGL_SIZE srcHeight /* height of video source */
    )
```

**DESCRIPTION**

This routine puts a single video frame into a region on a screen. The position and size of the video source rectangle is specified by `srcX, srcY, srcWidth, and srcHeight`. Typically the video image is stretched or shrunk to fill the entire overlay surface, although this can be dependent on hardware.

**RETURNS**

- `UGL_STATUS_OK` for success; otherwise `UGL_STATUS_ERROR`
### uglVideoStop()

**NAME**

`uglVideoStop()` - stop video on a video port

**SYNOPSIS**

```c
UGL_STATUS uglVideoStop(     
    UGL_VIDEO_ID videoId,     /* video device identifier */
    UGL_ORD      port         /* video port number */
)
```

**DESCRIPTION**

This routine stops active video for the specified video port `port` for the video device identified by `videoId`.

**RETURNS**

`UGL_STATUS_OK` for success; otherwise `UGL_STATUS_ERROR`

**ERRNO**

N/A

**SEE ALSO**

`uglExt`, `uglVideoPut()`, `uglVideoPutStill()`, `uglVideoStreamPut()`

### uglVideoStreamPut()

**NAME**

`uglVideoStreamPut()` - puts a video stream to a region on the display

**SYNOPSIS**

```c
UGL_STATUS uglVideoStreamPut(     
    UGL_VIDEO_ID videoId,         /* video device identifier */
    UGL_GC_ID    gc,              /* graphics context */
    UGL_ORD      port,            /* video port number */
    UGL_POS      srcX,            /* video source X position */
    UGL_POS      srcY,            /* video source Y position */
    UGL_SIZE     srcWidth,        /* width of video source */
    UGL_SIZE     srcHeight        /* height of video source */
)
```
DESCRIPTION
This routine puts video into a region on a screen. The position and size of the video source rectangle is specified by srcX, srcY, srcWidth, and srcHeight. Typically the video image is stretched or shrunk to fill the entire overlay surface, although this can be dependent on hardware.

RETURNS
UGL_STATUS_OK for success; otherwise UGL_STATUS_ERROR

ERRNO
N/A

SEE ALSO
uglExt, uglVideoStillPut()

---

**wavChunkFind()**

**NAME**
wavChunkFind() - locate a chunk within a WAVE form audio file

**SYNOPSIS**
```c
char *wavChunkFind
{
    char * pstart,            /* start of the search */
    char * fourcc,            /* chunk type */
    size_t n                  /* length of the search */
}
```

**DESCRIPTION**
This routine locates a chunk within an audio WAVE form file. The starting location to locate the chunk is specified by pStart and the chunk to locate is specified by fourcc. The length of the data section to search is provided in n.

**RETURNS**
Pointer to the start of the chunk when found; otherwise NULL

**ERRNO**
N/A

**SEE ALSO**
uglExt
wavHeaderRead()

NAME

wavHeaderRead() - read a WAVE form header from file

SYNOPSIS

STATUS wavHeaderRead

(int      fd,              /* file descriptor of wave form file */
 int *    pChannels,       /* number of channels */
 UINT32 * pSampleRate,     /* sample rate */
 int *    pSampleBits,     /* size of sample */
 UINT32 * pSamples,        /* number of samples */
 UINT32 * pDataStart       /* start of data section */
 )

DESCRIPTION

This routine reads the WAVE form file fd to retrieve its header information. The number of
channels, sample rate, number of bits per sample, number of samples, and the starting
location of the data are returned as parameters.

RETURNS

OK when header successfully read; otherwise ERROR

ERRNO

N/A

NOTES

This function is not reentrant

SEE ALSO

uglExt
uglUgi

NAME
uglUgi - WindML Generic Interface

ROUTINES
- uglGraphicsDevDestroy() - Destroys a graphics driver
- uglScratchBufferAlloc() - Allocate a scratch buffer for optimizing draw
- uglScratchBufferFree() - Free a scratch buffer
- uglUgiDevInit() - Performs graphics driver initialization
- uglUgiDevDeinit() - Frees resources for graphics device

DESCRIPTION
This library provides the generic interface for WindML.

uglGraphicsDevDestroy()

NAME
uglGraphicsDevDestroy() - destroys a graphics driver

SYNOPSIS
UGL_STATUS uglGraphicsDevDestroy

DESCRIPTION
This routine destroys the graphics driver devId. The act of destroying a graphics device,
frees all associated resources.

RETURNS
UGL_STATUS_OK, or UGL_STATUS_ERROR if the device can not be destroyed.
**uglScratchBufferAlloc()**

**NAME**

*uglScratchBufferAlloc* - allocate a scratch buffer for optimizing draw operations

**SYNOPSIS**

```c
void * uglScratchBufferAlloc
    (UGL_UGI_DRIVER * pDriver,
     UGL_SIZE allocSize)
```

**DESCRIPTION**

This routine allocates a scratch buffer and flags it as being in use. If a scratch buffer already exists and is too small, it is reallocated.

**RETURNS**

Returns a UGL_NULL if the buffer is in use, or if it is unable to allocate the memory. Otherwise, a pointer to the buffer is returned.

**SEE ALSO**

uglUgi

---

**uglScratchBufferFree()**

**NAME**

*uglScratchBufferFree* - flags the buffer as being available

**SYNOPSIS**

```c
UGL_STATUS uglScratchBufferFree
    (UGL_UGI_DRIVER * pDriver,
     void * pMem)
```

**DESCRIPTION**

This routine flags the buffer as being available.

**RETURNS**

Returns a UGL_STATUS_OK if the buffer exists, otherwise it returns UGL_STATUS_ERROR.

**SEE ALSO**

uglUgi
**uglUgiDevDeinit()**

**NAME**

`uglUgiDevDeinit` - frees resources for graphics device

**SYNOPSIS**

```c
UGL_STATUS uglUgiDevDeinit
    (UGL_DEVICE_ID devId       /* graphics device ID */)
```

**DESCRIPTION**

This routine frees resources for the device `devId`. This routine must be called by a driver’s destroy function before driver specific deinitialization is performed.

**RETURNS**

`UGL_STATUS_OK`, or `UGL_STATUS_ERROR` if the de-initialization can not be accomplished.

**ERRNO**

N/A

**SEE ALSO**

`uglUgi`, `uglOutputDevInit()`, `uglOutputDevDestroy()`

**uglUgiDevInit()**

**NAME**

`uglUgiDevInit` - performs graphics driver initialization

**SYNOPSIS**

```c
UGL_STATUS uglUgiDevInit
    (UGL_DEVICE_ID devId       /* UGI device ID */)
```

**DESCRIPTION**

This routine performs graphics driver initialization that is common to all graphics drivers. This routine must be called by a driver’s create function after driver specific initialization is performed. The device `devId` is initialized which includes creation of the device lock.

**RETURNS**

`UGL_STATUS_OK`, or `UGL_STATUS_ERROR` if initialization can not be accomplished.

**ERRNO**

N/A

**SEE ALSO**

`uglUgi`, `uglUgiDevDestroy()`, `uglUgiDevDeinit()`
G.1 Bitmap Fonts Supplied with WindML

Courier Pitch 10
Courier Pitch 12
Courier Pitch 14
Courier Pitch 18
Courier Pitch 24
Courier Pitch 8
Courier Bold Pitch 10
Courier Bold Pitch 12
Courier Bold Pitch 14
Courier Bold Pitch 18
Courier Bold Pitch 24
Courier Bold Pitch 8
Courier Bold Oblique Pitch 10
Courier Bold Oblique Pitch 12
Courier Bold Oblique Pitch 14
Courier Bold Oblique Pitch 18
Courier Bold Oblique Pitch 24
Courier Bold Oblique Pitch 8

Courier Oblique Pitch 10
Courier Oblique Pitch 12
Courier Oblique Pitch 14
Courier Oblique Pitch 18
Courier Oblique Pitch 24
Courier Oblique Pitch 8
Lucida Bright Demibold Pitch 24
Lucida Bright Demibold Pitch 8

Lucida Bright Demibold Italic Pitch 10
Lucida Bright Demibold Italic Pitch 12
Lucida Bright Demibold Italic Pitch 14
Lucida Bright Demibold Italic Pitch 18
Lucida Bright Demibold Italic Pitch 19
Lucida Bright Demibold Italic Pitch 24
Lucida Bright Demibold Italic Pitch 8

Lucida Bright Italic Pitch 10
Lucida Bright Italic Pitch 12
Lucida Bright Italic Pitch 14
Lucida Bright Italic Pitch 18
Lucida Bright Italic Pitch 19
Lucida Bright Italic Pitch 24
Lucida Bright Italic Pitch 8

Lucida Sans Pitch 10
Lucida Sans Pitch 12
Lucida Sans Pitch 14
Lucida Sans Pitch 18
Lucida Sans Pitch 19
Lucida Sans Pitch 24
Lucida Sans Pitch 8

Lucida Sans Bold Pitch 10
Lucida Sans Bold Pitch 12
Lucida Sans Bold Pitch 14
Lucida Sans Bold Pitch 18
Lucida Sans Bold Pitch 19
Lucida Sans Bold Pitch 24
Lucida Sans Bold Pitch 8

Lucida Sans Bold Italic Pitch 10
Lucida Sans Bold Italic Pitch 12
Lucida Sans Bold Italic Pitch 14
Lucida Sans Bold Italic Pitch 18
Lucida Sans Bold Italic Pitch 19
Lucida Sans Bold Italic Pitch 24
Lucida Sans Bold Italic Pitch 8

Lucida Sans Italic Pitch 10
Lucida Sans Italic Pitch 12