

**QNX[®] Neutrino[®] Realtime
Operating System**

**Graphics Framework and 3D[®]
*Developer's Guide***

For QNX[®] Neutrino[®] 6.3.x

QNX Software Systems

175 Terence Matthews Crescent
Kanata, Ontario
K2M 1W8
Canada
Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: <http://www.qnx.com/>

© 2005, QNX Software Systems Ltd. All rights reserved.

Technical support options

To obtain technical support for any QNX product, visit the **Technical Support** section in the **Services** area on our website (www.qnx.com). You'll find a wide range of support options, including our free web-based **Developer Support Center**.

QNX, Momentics, Neutrino, and Photon microGUI are registered trademarks of QNX Software Systems in certain jurisdictions. All other trademarks and trade names belong to their respective owners.

Printed in Canada.

Part Number: @ @ @ ??? @ @ @

Contents

	Beta Documentation Review	xi
	We need your feedback...	xiii
	About This Guide	xv
	Typographical conventions	xvii
	What you'll find in this guide	xix
1	Introduction	1
	Overview of the GF architecture	3
	GF components	5
	GF compared to Photon	7
2	Basic Drawing	9
	Steps to use GF	11
	Drawing rectangles	15
	Drawing lines and polygons	16
	Bitmaps	17
	Blitting	17
	Multi-threaded applications	18
3	Working With Images	19
4	Working with Layers, Surfaces and Contexts	25
	Using Layers	27
	Using Surfaces	30

	Using Contexts	31
5	Handling User Input	37
6	Using Fonts	41
7	Using OpenGL ES	45
	Using OpenGL ES	47
	Creating surfaces	49
8	Embedding GF	53
A	Hardware Capabilities	57
	Coral	59
B	QNX Graphics Framework Library Reference	61
C	Summary of Entries	65
	3D rendering	69
	Contexts	69
	Cursors	70
	Devices and displays	71
	Drawing	71
	Fonts	72
	Layers	72
	Surfaces	73
	<i>gf_3d_target_create()</i>	75
	<i>gf_3d_target_free()</i>	77
	<i>gf_3d_query_config()</i>	79
	gf_alpha_t	80
	gf_chroma_t	86
	<i>gf_context_create()</i>	88
	<i>gf_context_disable_alpha()</i>	90

<i>gf_context_disable_antialias()</i>	91
<i>gf_context_disable_chroma()</i>	93
<i>gf_context_disable_clipping()</i>	94
<i>gf_context_disable_linedash()</i>	95
<i>gf_context_disable_pattern()</i>	96
<i>gf_context_disable_rop()</i>	97
<i>gf_context_disable_transform()</i>	98
<i>gf_context_disable_translation()</i>	99
<i>gf_context_free()</i>	100
<i>gf_context_set_alpha()</i>	101
<i>gf_context_set_antialias()</i>	103
<i>gf_context_set_bgcolor()</i>	105
<i>gf_context_set_chroma()</i>	107
<i>gf_context_set_clipping()</i>	109
<i>gf_context_set_fgcolor()</i>	111
<i>gf_context_set_linedash()</i>	112
<i>gf_context_set_linejoin()</i>	114
<i>gf_context_set_pattern()</i>	116
<i>gf_context_set_penwidth()</i>	118
<i>gf_context_set_rop()</i>	120
<i>gf_context_set_surface()</i>	133
<i>gf_context_set_transform()</i>	135
<i>gf_context_set_translation()</i>	137
<i>gf_cursor_disable()</i>	139
<i>gf_cursor_enable()</i>	141
<i>gf_cursor_set()</i>	143
<i>gf_cursor_set_pos()</i>	146
<i>gf_dev_attach()</i>	148
<i>gf_dev_detach()</i>	151
<i>gf_dev_register_thread()</i>	152
gf_dim_t	153
<i>gf_display_attach()</i>	154

<i>gf_display_detach()</i>	158
<i>gf_display_set_layer_order()</i>	159
<i>gf_draw_begin()</i>	162
<i>gf_draw_bitmap()</i>	165
<i>gf_draw_blit1()</i>	167
<i>gf_draw_blit2()</i>	169
<i>gf_draw_blitscaled()</i>	171
<i>gf_draw_end()</i>	173
<i>gf_draw_finish()</i>	175
<i>gf_draw_flush()</i>	177
<i>gf_draw_poly_fill()</i>	179
<i>gf_draw_polyline()</i>	182
<i>gf_draw_rect()</i>	184
<i>gf_draw_string()</i>	186
<i>gf_font_attach()</i>	188
<i>gf_font_detach()</i>	190
<i>gf_layer_attach()</i>	191
<i>gf_layer_detach()</i>	193
<i>gf_layer_disable()</i>	194
<i>gf_layer_enable()</i>	196
<i>gf_layer_query()</i>	198
<i>gf_layer_set_blending()</i>	205
<i>gf_layer_set_brightness()</i>	207
<i>gf_layer_set_chroma()</i>	209
<i>gf_layer_set_contrast()</i>	211
<i>gf_layer_set_dst_viewport()</i>	213
<i>gf_layer_set_saturation()</i>	215
<i>gf_layer_set_src_viewport()</i>	217
<i>gf_layer_set_surfaces()</i>	219
<i>gf_layer_update()</i>	221
gf_palette_t	223
gf_point_t	225

gf_surface_attach() 226
gf_surface_attach_by_sid() 229
gf_surface_create() 231
gf_surface_create_layer() 235
gf_surface_free() 239
gf_surface_get_info() 241

D QNX Image Library Reference 245

img_codec_list() 248
img_codec_list_byext() 250
img_codec_list_bymime() 252
img_convert_data() 254
img_decode_callouts_t 258
img_decode_frame() 262
img_info_t 264
img_lib_attach() 267
img_lib_detach() 269
img_load_file() 270
io_fd_init_input() 272
io_input_t 274
io_mem_init_input() 277

Glossary 279



List of Figures

GF application architecture	4
The monitor process	5
Device, display, and layer hierarchy	6
Hardware concepts	7
Filling a simple polygon.	180
Filling an overlapping polygon.	180



Beta Documentation Review



Title:	<i>Developer's Guide</i>
Product:	Graphics Framework and 3D
Document Version:	Version 6.3.x
Date:	Fri May 13 09:56:37 2005
Editor:	Drew Avis (aavis@qnx.com)
Send comments to:	the appropriate <code>quics.beta.*</code> newsgroup

We need your feedback...

If you have any comments, questions, or changes, please let us know as soon as possible.

Question notes indicate a part of the manual that needs resolution. We'd especially like any feedback you can give on these areas.



About This Guide



Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications. The following table summarizes our conventions:

Reference	Example
Code examples	<code>if(stream == NULL)</code>
Command options	<code>-lR</code>
Commands	<code>make</code>
Environment variables	<code>PATH</code>
File and pathnames	<code>/dev/null</code>
Function names	<code>exit()</code>
Keyboard chords	Ctrl – Alt – Delete
Keyboard input	<code>something you type</code>
Keyboard keys	Enter
Program output	<code>login:</code>
Programming constants	<code>NULL</code>
Programming data types	<code>unsigned short</code>
Programming literals	<code>0xFF, "message string"</code>
Variable names	<code>stdin</code>
User-interface components	<code>Cancel</code>

We format single-step instructions like this:

- To reload the current page, press Ctrl – R.

We use an arrow (→) in directions for accessing menu items, like this:

Typographical conventions

You'll find the **Other...** menu item under **Perspective→Show View**.

We use notes, cautions, and warnings to highlight important messages:



Notes point out something important or useful.



CAUTION: Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.



WARNING: Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

What you'll find in this guide

The QNX Graphics Framework and *3D Developers's Guide* accompanies the QNX Graphics Framework Development System and is intended for application developers. It contains both a programming guide and an API reference. It describes how to create graphical applications that use the Graphics Framework API and the OpenGL[®] ES API.

This table may help you find what you need in this book:

For information about:	See:
The Graphics Framework	Overview
Basic steps to use the GF API	Basic drawing
Loading images	Working with images
Layers	Working with layers and surfaces
User input	Handling user input with io-hid
Fonts	Managing fonts
3D rendering	Creating OpenGL ES targets
Embedding	Embedding your application
Hardware capabilities	Hardware capabilities of supported devices
GF API	A list of all the functions in the Graphics Framework API
GF API (by functionality)	A list of all GF functions, grouped by areas of functionality

continued...

What you'll find in this guide

For information about:	See:
Image Library API	A list of all the functions in the Image Library API

Chapter 1

Introduction

In this chapter...

Overview of the GF architecture	3
GF compared to Photon	7



This chapter introduces you to the basic terms and concepts you'll use when developing a GF application.

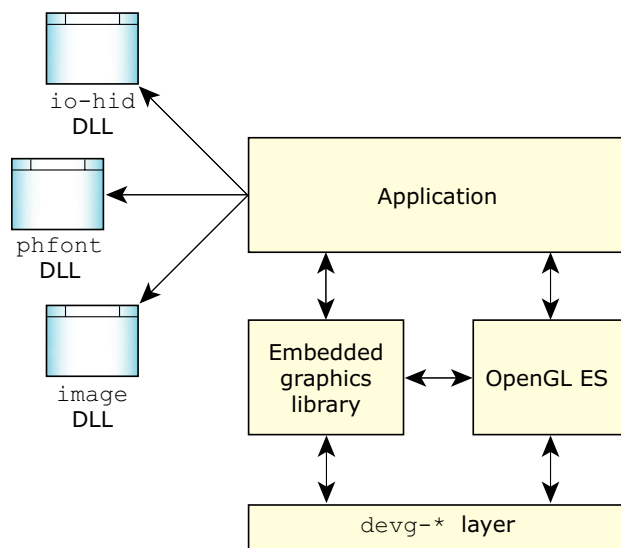
Overview of the GF architecture

The QNX Neutrino Graphics Framework (GF) is a framework for creating application user interfaces without the overhead of a full windowing system. Its architecture allows it to render directly to hardware, making it faster and more responsive in an embedded environment where resources are limited. The GF achieves this by providing direct access to graphics drivers (there's no message passing or context switching while rendering), and by using hardware acceleration where possible. This makes it perfect for embedded environments, where it can be used as the basic graphics layer for anything from a simple fullscreen UI to a complex windowing system for multiple application GUIs. It can also act as a porting layer for existing UIs.

For embedded 3D requirements, the GF supports the OpenGL ES 1.0 specification, a well-defined subset of OpenGL designed for embedded applications. The Neutrino implementation supports the Common profile and EGL platform interface layers.

The GF is just one of several components that provide services for Neutrino applications. For a typical GF application, keyboard and mouse input is managed via `io-hid`, font rendering can be handled by Bitstream's Font Fusion library, and image loading can be handled by the Neutrino image library.

Overview of the GF architecture



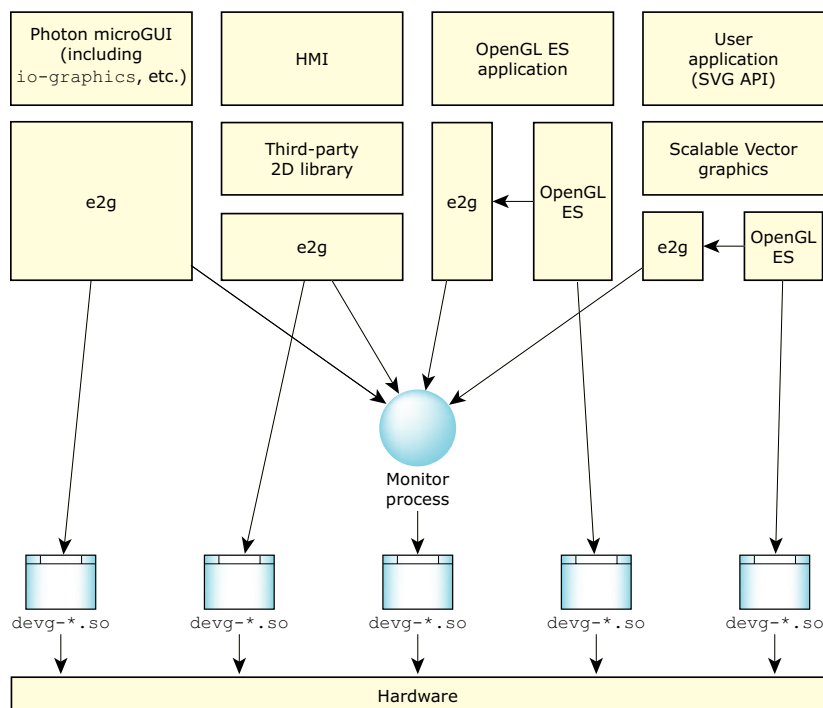
GF application architecture

While the Graphics Framework is intended for fullscreen application user interfaces, it can handle multiple threads, multiple applications, and you can even run GF/OpenGL-ES applications under Photon if required.

The Graphics Framework uses a client/server model, with the server (**io-display**) managing device resources (such as memory and layers) and hardware access. The client (the GF application) renders directly to the hardware. Clients obtain a lock by calling *gf_draw_begin()* to ensure that only one thread or application has access to the hardware at a time. An independent monitor process provides increased robustness by catching abnormal client termination, cleaning up resources after termination, and reviving the shared mutex if it was held by an abnormally terminated client.

All hardware access by GF applications is brokered by the monitor process, **io-display**. The monitor also allocates all video memory, and handles application termination (both normal and abnormal) by releasing allocated memory, and releasing locked hardware (if

necessary). The monitor is “invisible” to GF clients, as all interactions are handled internally by the library. This diagram illustrates the relationship of the monitor process to GF clients:



The monitor process

For more information about configuring the monitor process, and what components you need to distribute on target systems, see the Embedding GF Applications chapter.

GF components

Let’s look at the conceptual components of the Graphics Framework, devices, displays, layers, surfaces, and contexts.

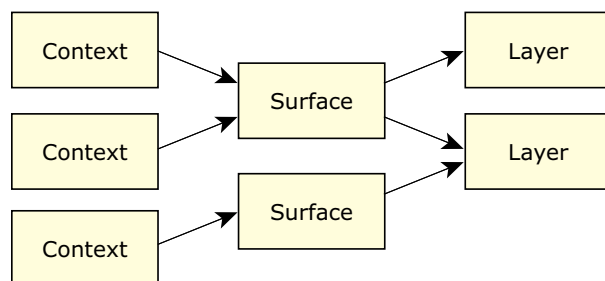
Devices and displays are the hardware components of the GF. A device is the graphics hardware, which is attached to and managed by

`io-display` via the appropriate `devg-*` driver. Each device has an entry in `/dev/io-display`.

Each device has at least one display, which represents the video display viewed by the end-user. Some devices are *multi-headed*, meaning they support multiple displays (that is, two or more monitors).

Each display has at least one layer. Some modern graphics hardware supports multiple layers, though from the GF perspective all devices have at least one layer (layer 0), even if they don't support layering. Layers provide a great deal of flexibility in laying out content on a display. For example, you can put a scrolling map on a background layer, and GUI controls for scrolling the map on a front layer. The application can scroll the map smoothly without redrawing the GUI controls, eliminating flicker and reducing CPU load. In any situation where you want to display text or a GUI over top of visual information, layering can be very useful.

The relationship between a device, its displays, and their layers is illustrated in this figure:



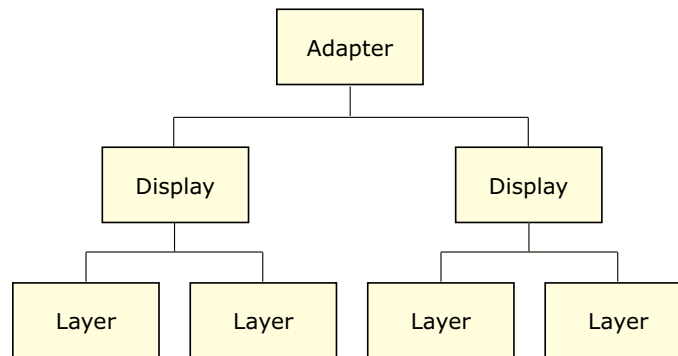
Device, display, and layer hierarchy

A *surface* is piece of memory that the GF library can render to. If the surface is connected to a layer (and the layer is visible), anything rendered on the surface appears on the layer's display. The surface must have a pixel format that is supported by the layer so that objects rendered onto the surface appear correctly. Generally, a layer *targets* (or is associated with) a single surface. An exception is planar data,

such as planar YUV, which requires three surfaces (one for each video component). In general, the GF allocates a surface in video memory, though when you create the surface you can request that it be optimized for CPU access, which generally means it will be allocated in system RAM.

A *context* is a structure that maintains information about a rendering target between calls to rendering functions, such as the pen width, foreground and background colors, and clipping rectangle. A context targets a surface, and is an argument to all GF drawing functions.

This figure illustrates how contexts, surfaces, and layers are related:



Hardware concepts

GF compared to Photon

Should you use Photon or GF as your UI framework? Let's compare the strengths and weaknesses of two systems:

GF compared to Photon

Feature	Photon	GF
Intended for	Multiple applications, heavy windowing requirements, desktop-like environments	Fullscreen UI, no additional windowing requirements, embedded environments
Hardware access	Via message passing to io-graphics	Direct access to hardware via devg layer, no message passing while rendering
Font support	Full, via Phfont and the <i>Pf*()</i> library	Full, via the Bitstream Font Fusion library
Widget support	Full, via the Pt* library	None
3D support	OpenGL [®] ES	OpenGL [®] ES

Chapter 2

Basic Drawing



This chapter shows you how to use the basic functions in the Graphics Framework library.

Steps to use GF

Let's look at the steps your application needs to perform to use the GF for 2D drawing.



The steps for using the 3D library, OpenGL ES, is slightly different. See the Using OpenGL ES chapter for more details.

- 1 Attach to a graphics device using *gf_dev_attach()*. This gives you a handle to the device which you need to create draw contexts and surfaces, and to attach to displays. It's also required to attach to the font library.
- 2 Attach to the device's display (or displays if there are more than one) using *gf_display_attach()*. This gives you a display handle, which you require to use hardware cursors and layers for the display.
- 3 Attach to one or more of the device's layers using *gf_layer_attach()*. In the Graphics Framework, a device has at least one layer (layer 0), even if the hardware doesn't support multi-layering.

You need the handle for an attached layer to control any of the layer properties (such as brightness, contrast, and saturation), chroma and alpha blending, and viewports. You also use this handle to attach a surface to the layer.
- 4 Create a surface for each layer using *gf_surface_create_layer()*. You require at least one surface per layer in order to display an image on that layer.
- 5 Set the visible surface for each layer using *gf_layer_set_surfaces()*.
- 6 If you set any of the layer's parameters, commit the changes with *gf_layer_update()*. Any call to a *gf_layer_set*()* function

must be followed by *gf_layer_update()* to make the changes take effect.

- 7 Create a draw context using *gf_context_create()*.
- 8 Associate the rendering surface with the draw context using *gf_context_set_surface()*.



Never pass NULL as the handle for a device, display, layer, surface, or context in functions that have a handle parameter. Doing so may cause your application to crash.

Let's look at the details of each of these steps.

Attaching to a device and display

In the example below, you attach to the first device in **dev/io-display** (*GF_DEVICE_INDEX(0)*). Then, you attach to each of the device's displays, and print out some information about that display's resolution, refresh rate, and number of layers. Finally, when the application is ready to exit, you call *gf_dev_detach()* to disconnect from the device.

```
gf_dev_t          gdev;
gf_dev_info_t     gdev_info;
gf_display_t      display;
gf_display_info_t display_info;

if (gf_dev_attach(&gdev,GF_DEVICE_INDEX(0),&gdev_info) == -1) {
    printf("gf_dev_attach() failed\n");
    return (-1);
}

printf("Number of displays: %d\n",gdev_info.ndisplays);

for (i = gdev_info.ndisplays;i;) {
    printf("Display %d: ",i--);
    if (gf_display_attach(&display,gdev,i,&display_info) != -1) {
        printf("%dX%d, refresh = %dHz\n",display_info.xres,display_info.yres,display_info.refresh);
        printf("Number of layers: %d\n",display_info.nlayers);
    }
}
```

```

        } else {
            printf("gf_display_attach() failed\n");
        }
    }

    ...

    gf_dev_detach(gdev);

```

Attaching to display layers

This code snippet, you attach to the main layer (the *main_layer_index* member of the display's information structure) of the display returned by *gf_display_attach()*.

```

if (display != NULL) {
    gf_layer_t layer;

    if (gf_layer_attach(&layer, display,
                        display_info.main_layer_index, 0) != -1) {

        ...

    } else {
        printf("gf_layer_attach() failed\n");
    }
}

```

Create and set surfaces

In this snippet, the application creates a surface for the layer attached in the previous step using *gf_surface_create_layer()*. This surface has the same dimensions as the display (the *xres* and *yres* members of *display_info*), and the same format (which you can retrieve from an attached layer using *gf_layer_query()*, and inspecting the *format* member of the information structure @@@ doesn't look like all

formats are supported? @@@). The surface is then displayed on the layer using *gf_layer_set_surfaces()*, and the change is committed with *gf_layer_update()*. Finally, you can fill in a **gf_surface_info** structure with the surface information (most importantly, the surface ID), using *gf_surface_get_info()*.

```
if (gf_surface_create_layer(&surface, &layer, 1, 0,
                           display_info.xres,
                           display_info.yres, 1,
                           layer_format, NULL, 0) != -1) {
    gf_layer_set_surfaces(layer, layer_format, &surface, 1);
    if (gf_layer_update(layer, 0) != -1) {
        gf_surface_get_info(surface, &surface_info);
        printf("sid = %ul\n", surface_info.sid);

        ...

    } else {
        printf("gf_layer_update() failed\n");
    }
} else {
    printf("gf_surface_create_layer() failed\n");
}
```

Create a draw context

The final step is to create a draw context, and set the previously created surface for the context.

```
if (gf_context_create(&context) != -1) {
    if (gf_context_set_surface(context, surface) != -1) {

        /* draw stuff here */
        ...

        gf_context_free(context);
    }
}
```

At this point your application can start calling draw functions (see below). To draw you need to:

- 1 Call *gf_draw_begin()* to gain access to the graphics hardware. This function blocks until the graphics hardware is released by any other rendering threads.
- 2 Call any *gf_draw*()* functions.
- 3 Call *gf_draw_flush()* to flush any commands in the draw buffer to the graphics hardware, if you require all previous draw commands to be completed (for example, before waiting on user input that depends on a rendered image).
- 4 Call *gf_draw_end()* to release the graphics hardware. Every successful call to *gf_draw_begin()* must be matched by a call to *gf_draw_end()*.

Before your application exits, you should clean up allocated resources. At minimum, you need to:

- Free any contexts with *gf_context_free()*
- Detach from attached displays and layers.
- Free any surfaces.
- Detach from the graphics device with *gf_dev_detach()*



If an application terminates abnormally, **io-display** automatically releases any GF resources that were allocated by the application.

Drawing rectangles

Use *gf_draw_rect()* to draw a filled rectangle. This function takes a handle for a draw context, and the coordinates of the upper-left and lower-right corner of rectangle (inclusive). The fill color is the foreground color for the context.



To draw an unfilled rectangle, use *gf_draw_polyline()*.

Here's an example which draws a small black rectangle inside a larger white rectangle:



The code examples in this chapter assume you've already performed all the steps required to start drawing using GF.

```
gf_context_set_fgcolor(context, 0xFFFFFFFF);  
gf_draw_rect(context, 10, 10, 50, 50);  
gf_context_set_fgcolor(context, 0x000000);  
gf_draw_rect(context, 20, 20, 40, 40);
```

Drawing lines and polygons

Use *gf_draw_polyline()* and *gf_draw_poly_fill()* to draw polygons.

Both of these functions take the draw context, an array of points, and the number of points in the array as arguments. *gf_draw_polyline()* takes a flag as the fourth argument to indicate whether the last point should be joined to the first. If you simply pass 0, this function draws lines. Here's an example that draws two triangles using these functions:

```
...  
gf_context_set_fgcolor(context, 0xFFFFFFFF);  
pts[0].x = 20;  
pts[0].y = 20;  
pts[1].x = 10;  
pts[1].y = 30;  
pts[2].x = 40;  
pts[2].y = 30;  
  
// draws two lines, the first two sides of the triangle  
// set the flags to GF_DRAW_POLYLINE_CLOSED to complete the triangle:  
gf_draw_polyline(context, pts, 3, NULL);
```

```
// draws a complete, filled triangle:
gf_draw_poly_fill(context, pts, 3);
...
```

Bitmaps

Use *gf_draw_bitmap()* to draw a bitmap.

```
uint8_t image[] = {0x08,0x1C,0x3E,0x7F,0x3E,0x1C,0x08,0x00};
gf_bitmap_t btm;
btm.image = image;
btm.stride = 1;
btm.bit0_offset = 0;
btm.transparent = 0;
btm.dim.w = 8;
btm.dim.h = 8;

...

gf_context_set_bgcolor(context, 0xFF0000);
gf_context_set_fgcolor(context, 0xFFFFFFFF);
gf_draw_bitmap(context, &btm, 10, 10);
```

Blitting

Use *gf_draw_blit1()* to blit between areas on the same surface, or *gf_draw_blit2()* to blit between two surfaces. If you want to scale the area (increasing or decreasing its size), use *gf_draw_blitscaled()*.

For example, to blit the bitmap drawn in the previous example, add these two lines of code:

```
...
gf_draw_blit1(context, 10, 10, 18, 18, 25, 25);
gf_draw_blitscaled(context, NULL, NULL, 10, 10, 18, 18, 30, 30, 60, 60);
```

Note that we pass NULL for the source and destination surface for the scaled blit. This selects the default surface, which is the surface for

the current context. You can set these parameters if you want to blit between surfaces.

Multi-threaded applications

The GF library is thread safe, if you ensure that your application puts all rendering functions in a *gf_draw_begin()* and *gf_draw_end()* block. This ensures that your thread or process has exclusive access to the hardware, and leaves the hardware in a known state.



Because the hardware lock is associated with the draw context and not the thread, it is possible for one thread sharing a context to render without calling *gf_draw_begin()*, if the other thread in the same process has already done so. This is very dangerous, and can cause the graphics hardware to lock up. Even when sharing a draw context, all threads must call *gf_draw_begin()* before calling any *gf_draw_**() or *gf_context_set_**() functions. The only exception is *gf_context_set_surface()*, which must be called outside a *gf_draw_begin()* and *gf_draw_end()* block.

Threads, whether in the same or different processes, can share a surface. When the threads are in different processes, only one thread needs to attach to the display and the layer the shared surface is associated with. To share a surface, one thread creates the surface and communicates its surface ID (the *sid* member of the surface information structure filled in by *gf_surface_get_info()*). The second thread then calls *gf_surface_attach_by_sid()* and passes the shared surface ID. Keep in mind that there is nothing preventing one thread from rendering over top of the other thread's work.

Threads within the same process can share a connection to a graphics device to reduce the memory required to allocate device handles. The first thread attaches to the device using *gf_dev_attach()*, and other threads call *gf_dev_register_thread()* to register themselves as users of the device.

Chapter 3

Working With Images



This chapter shows you how to load and render images.

GF doesn't have any image handling functionality, aside from blitting images once they've been opened, decoded, and rendered to a surface. Instead, GF applications can use the Neutrino image library (*libimage()*) to accomplish these tasks.

The image library is a static library that provides a common interface for image codecs. This means that while the library is linked into your executable, you need to put any required image codecs, plus the image configuration file, onto targets running your application. You'll need to include (at least):

- the configuration file at `/etc/system/config/img.conf`
- the image codecs (`img_codec_*.so`) in `processor/lib/dll`.

For a sample buildfile that includes a GF application and the required image codecs, see the Embedding GF chapter.



Code examples in this chapter are taken from the `gf_loading` example application shipped with the 3D TDK. This GF application opens and displays images in any of the supported formats. See the `src/demos @@@` or whatever we're calling the directory `@@@` for the full application source.

Before your GF application can render images, it must perform all the steps required to prepare for rendering, as outlined in the Steps to using GF section of the Basic Drawing chapter.

Steps to display an image:

- attach to the image lib
- load the image file
- attach to a GF surface for the image
- blit the image onto the target surface

Let's look at each of these steps in a little more detail.

Attaching to the image library

Use *img_lib_attach()*:

```
img_lib_t ilib;
...
if (img_lib_attach(&ilib) == -1) {
    printf("img_lib_attach() failed\n");
    return -1;
}
```

Loading the image

Use *img_load_file()*:

```
img_info_t img;
...
if (img_load_file(ilib, argv[optind], NULL, 0, &img, NULL, 0) == -1) {
    printf("img_load_file() failed: %s\n", strerror(errno));
}
```

Attaching to a surface

This step is more involved. First, you need to determine the pixel format of the image so you can attach to a surface with a compatible format. You do this by looking at the *format* member of the *img_info_t* structure filled in by *img_load_file()*:

```
char *s = NULL;

switch (img.format) {
    case IMG_FMT_PAL8:           s = "PAL8";       break;
    case IMG_FMT_PACKED_RGB565:  s = "RGB565";     break;
    case IMG_FMT_PACKED_RGB555:  s = "RGB555";     break;
    case IMG_FMT_PACKED_ARGB1555: s = "ARGB1555"; break;
    case IMG_FMT_RGB888:         s = "RGB888";     break;
    case IMG_FMT_PACKED_XRGB8888: s = "RGB8888";  break;
    case IMG_FMT_PACKED_ARGB8888:
    case IMG_FMT_RGBA8888:

```

```

        case IMG_FMT_ARGB8888:          s = "ARGB8888"; break;
    }

```

Next, if your image is palette-based, your application needs to create a matching palette for the GF surface:

```

gf_palette_t _p, *palette;
...

if (img->palette.ncolors) {
    palette = &_p;
    palette->colors = img->palette.data;
    palette->ncolors = img->palette.ncolors;
    palette->format = convert_format(img->palette.format);
} else {
    palette = NULL;
}

```

Finally, you can now wrap the image in a GF surface, so that it can be used with the rest of the GF API. To do this, use *gf_surface_attach()*, passing the image's format, palette (if applicable), dimensions, stride, and image data:

```

gf_surface_attach(&surface, gfx,
                 img->w, img->h, img->stride,
                 convert_format(img->format), palette,
                 img->data, 0)

```

Note that a surface created with *gf_surface_attach()* isn't like other GF surfaces; there are some restrictions:

- Any rendering or blitting to or from this surface is done in software, not hardware. This is because the memory isn't allocated by the video driver, so the driver doesn't know anything about it.
- For that same reason, your application is responsible for freeing the image data. However, it should do so only after the attached surface is freed with *gf_surface_free()*.
- This surface can't be shared with other processes because it isn't managed by *io-display*.

Blitting the image

Before you blit the image onto a target surface, you should check to see if there's a transparency flag or alpha channel:

```
if (img->flags & IMG_TRANSPARENCY_VALID) {
    gf_chroma_t chroma;
    chroma.mode = GF_CHROMA_OP_SRC_MATCH | GF_CHROMA_OP_NO_DRAW;
    memcpy(&chroma.color0, &img->transparency, sizeof chroma.color0);
    gf_context_set_chroma(context, &chroma);
}

if (img->format & IMG_FMT_ALPHA) {
    gf_alpha_t alpha;
    alpha.mode = GF_ALPHA_M1_SRC_PIXEL_ALPHA | GF_BLEND_DST_1mM1 | GF_BLEND_SRC_1mM1;
    gf_context_set_alpha(context, &alpha);
}
```

@@@ not sure what's going on there, get more info from Dave
@@@

Finally, you can blit the image onto a target surface using *gf_draw_blit2()*:

```
gf_draw_blit2(context, surface, NULL, 0, 0, img->w - 1, img->h - 1, 0, 0,
```

Chapter 4

Working with Layers, Surfaces and Contexts



This chapter shows you how to work with layers and surfaces using the Graphics Framework.

Let's look at some of the capabilities of layers, surfaces, and contexts.

Using Layers

Once you've attached to a display, you can get the handle to any of the supported layers using *gf_layer_attach()*, passing the index of the layer you want. When you attach to a display, *gf_display_attach()* fills in a *gf_display_info_t* structure. Two members contain information about supported layers; *main_layer_index* tells you which layer is the main layer (the main layer index is hardware-dependent), and *nlayers* tells you the number of layers.

After you've attached to a layer, you can query the layer about the formats it supports. You need this information to attach to a surface with the correct format. Here's an example where you know your hardware supports a subset of pixel formats, and you want to find the layer format that matches one of them:

```
static int
find_rgb_format(gf_layer_t layer)
{
    gf_layer_info_t info;
    int i;

    for (i = 0; gf_layer_query(layer, i, &info) != -1; ++i) {
        switch (info.format) {
            case GF_LAYER_FORMAT_ARGB1555:
            case GF_LAYER_FORMAT_RGB565:
            case GF_LAYER_FORMAT_BGR888:
            case GF_LAYER_FORMAT_BGRA8888:
                return info.format;
            default:
                continue;
        }
    }

    return -1;
}
```

The `gf_layer_info_t` structure filled in by `gf_layer_query()` also contains information about other layer capabilities which your application may need to know about. @@@ Need more info here, once the layer caps are defined @@@

Once you've attached to a surface with a matching format, you set the target surface (or surfaces if there's more than one, such as with planar YUV data) for the layer using `gf_layer_set_surfaces()`.

Layer visibility

By default, the main layer is visible (enabled), and all other layers are not visible (disabled). You can make any layer except the main layer non-visible by calling `gf_layer_disable()`, and visible with `gf_layer_enable()`.

You can check to see if the hardware allows a layer to be disabled by checking the `GF_LAYER_CAP_DISABLE` flag bit in the `gf_layer_info_t.caps` returned by `gf_layer_query()`. The main layer can't be disabled.

Layer brightness, contrast, and saturation

You can set the brightness, contrast, or saturation attributes for a layer by calling `gf_layer_set_brightness()`, `gf_layer_set_saturation()`, and `gf_layer_set_saturation()`. These functions take an integer in the range of 127 to -128, where 0 is the normal (default) value. These functions will return -1 if the hardware doesn't support the attribute, 0 otherwise.



All of these functions require that your application calls `gf_layer_update()` to make any attribute changes take effect.

You can check to see if the hardware allows a layer to set these attributes by checking the `GF_LAYER_CAP_SET_BRIGHTNESS`, `GF_LAYER_CAP_SET_CONTRAST`, and `GF_LAYER_CAP_SET_SATURATION` flag bits in the `gf_layer_info_t.caps` returned by `gf_layer_query()`.

Viewports

You can create a viewport for a layer, which is an area on a source surface that's displayed on the destination layer. The *gf_layer_set_src_viewport()* function sets the area of the source surface that's displayed, and *gf_layer_set_dst_viewport()* sets the “window” on the main display.

A viewport is a convenient way to scroll or pan an image, or provide a thumbnail-type image of a larger area (such as a map). To scale an image, increase or decrease the size of the destination viewport. To scroll or pan the image, move the position of the source viewport.



- The rectangle that describes the viewport must be inside the bounds of the surface (for the source) and the layer (for the target).
- Both of these functions require that your application calls *gf_layer_update()* to make the attribute change take effect.

There may be hardware limitations on the viewports. Check these flag bits in **gf_layer_info_t.caps** filled in by *gf_layer_query()*:

- **GF_LAYER_CAP_PAN_SOURCE** — the source viewport position can be adjusted.
- **GF_LAYER_CAP_SIZE_DEST** — the size of destination viewport can be different than size of source viewport.
- **GF_LAYER_CAP_PAN_DEST** — the position of destination viewport can be adjusted

Layer alpha blending

You can determine how the hardware blends a layer with the layers behind it on the display by setting the layer's alpha and chroma parameters with:

- *gf_layer_set_blending()*
- *gf_layer_set_chroma()*

@@@ Need some sample code and more content here – are you more likely to do this on the context-level? @@@

Using Surfaces

Surfaces are areas of memory that you can render to using the GF or GLES APIs. A surface can be associated with a layer (in which case, whatever is rendered to the surface appears on the display associated with the layer), or can exist on its own to facilitate offscreen rendering.

The region of memory in which a surface is located largely depends on the hardware, and the flags passed to the surface create function. Setting the `GF_SURFACE_CREATE_CPU_FAST_ACCESS` flag bit for *gf_surface_create()* means that the library attempts to optimize the surface for CPU access speed, which generally means allocating system RAM. By default, the library attempts to optimize the surface for hardware accelerated rendering, which generally means allocating video RAM.

The functions you can use to create surfaces are:

- *gf_surface_create()* — create a buffer that’s not targetted by a layer (for example, an area of memory used for offscreen rendering).
- *gf_surface_create_layer()* — create a surface that will be targetted (displayed) by a layer.
- *gf_surface_attach()* — create a GF surface from an existing area of memory (for example, an image).
- *gf_surface_attach_by_sid()* — create another surface handle for a previously allocated surface. This function is useful for sharing a surface between processes.

All surfaces are freed by the GF library when an application exits, though if your application creates a surface and then no longer requires it, it should free the memory by calling *gf_surface_free()*. Note that memory “wrapped” by *gf_surface_attach_by_sid()* isn’t freed by the GF library and needs to be deallocated by the application.

If you need to query a surface about its capabilities, use *gf_surface_get_info()*. This function fills in a **gf_surface_info_t** structure with information about the surface's:

- id (sid) — which you require to use *gf_surface_attach_by_sid()* to attach to the surface from another process
- width and height
- pixel format
- stride
- physical address (this should only be used if the surface was created with the GF_SURFACE_PHYS_CONTIG flag bit set)
- virtual address (this should only be used if the surface was created with the GF_SURFACE_CREATE_CPU_LINEAR_ACCESSIBLE flag bit set)
- palette (if one exists)

Using Contexts

A context is a structure that maintains parameters for a surface. You set the parameters on the context rather than the surface itself. This allows you to use one context for several surfaces, if you want to.

To create a context, use *gf_context_create()*, then use *gf_context_set_surface()* to set the context to a specific surface. You should free the context with *gf_context_free()* when you're done with it.

Let's look at the parameters you can set for a context.



Many of the “set” functions discussed in this section have a corresponding “disable” function to turn the parameter off for the context. For example, *gf_context_set_alpha()* also has a *gf_context_disable_alpha()*.

Alpha and chroma

Set alpha blending for a context using *gf_context_set_alpha()*, and set chroma operations using *gf_context_set_chroma()*. Each of these functions takes a structure that defines the alpha or chroma parameters.

For example, if you are importing an image with an alpha channel and transparency, you’d set these parameters for the current draw context, as in this function from the image loading sample project @@@ name and path here @@@:

```
static void
draw_image(gf_context_t context, const img_info_t *img, gf_surface_t sur
    if (img->flags & IMG_TRANSPARENCY_VALID) {
        gf_chroma_t chroma;
        chroma.mode = GF_CHROMA_OP_SRC_MATCH | GF_CHROMA_OP_NO_DRAW;
        memcpy(&chroma.color0, &img->transparency, sizeof chroma.color0)
        gf_context_set_chroma(context, &chroma);
    }

    if (img->format & IMG_FMT_ALPHA) {
        gf_alpha_t alpha;
        alpha.mode = GF_ALPHA_M1_SRC_PIXEL_ALPHA | GF_BLEND_DST_1mM1 | GF
        gf_context_set_alpha(context, &alpha);
    }

    gf_draw_blit2(context, surface, NULL, 0, 0, img->w - 1, img->h - 1, 1
}
```

Foreground and background colors

The foreground color is the color applied to most draw primitives, such as rectangles, polygons, bitmaps, and polylines. It's set with *gf_context_set_fgcolor()*, which takes a 32-bit ARGB color (for more information, see **gf_palette_t**). The background color, set with *gf_context_set_bgcolor()*, is applied where a second color is required, for example as the background for a dashed polyline.

Line attributes

Aside from the foreground and background colors, polylines also take attributes that define their width, dashing, and join style between line segments.

The width of a line is set with *gf_context_set_penwidth()*. Note that line thickness is only supported for hardware that has accelerated thick line support. The line width can be a maximum of 32, or the hardware limit for thick lines (see the Hardware Capabilities appendix for more information on supported hardware). If you set the linewidth to 0, the hardware's default width for thin lines is used.

The join style between segments of a polyline is set with *gf_context_set_linejoin()*. Two line join styles are supported, butt joints and bevel joints: @@@ insert screen capture here @@@

Line dashing applies a stipple effect to polylines, giving them a “dashed” effect. Use *gf_context_set_linedash()* to set line dashing. For example:

```
gf_point_t p[] = { { 50,250},{w-100,250}};  
gf_context_set_fgcolor(context,0x00ffff);  
gf_context_set_bgcolor(context,0xffffffff);  
gf_context_set_linedash(context,0x00ffff, 32, GF_CONTEXT_LINEDASH_BA  
gf_context_set_penwidth( context,10);  
gf_draw_polyline(context,p,2,0);
```

Clipping

A clipping rectangle limits what is rendered on the final display to the contents of the rectangle, even if rendering commands for a surface extend beyond the borders of the rectangle. You set a clipping rectangle with *gf_context_set_clipping()*. In this example, several polygons are drawn with some points outside of a clipping rectangle, which are clipped on the final display:

```
gf_context_set_fgcolor(context, 0xffff00);
gf_draw_rect(context, 0, 0, w, h);

gf_context_set_clipping(context, 50, 50, 400, 400);

{
    gf_point_t p[][4] = { {80, 8, 150, 150, 80, 80, 40, 160},
                          {240, 8, 290, 80, 360, 10, 290, 160},
                          {40, 190, 150, 270, 50, 320, 90, 270},
                          {210, 260, 450, 190, 370, 260, 450, 340} };

    gf_context_set_fgcolor(context, 0x0000ff);
    gf_context_set_penwidth(context, 1);
    gf_draw_poly_fill(context, &p[0][0], 4);

    gf_context_set_fgcolor(context, 0x00ff00);
    gf_context_set_penwidth(context, 5);
    gf_draw_polyline(context, &p[1][0], 4, GF_DRAW_POLYLINE_CLOSED);

    gf_context_set_fgcolor(context, 0xf00f00);
    gf_context_set_penwidth(context, 10);
    gf_draw_poly_fill(context, &p[2][0], 4);

    gf_context_set_fgcolor(context, 0x00ffff);
    gf_context_set_penwidth(context, 20);
    gf_draw_polyline(context, &p[3][0], 4, 0);
}
```

Raster operations

Raster operations (rops) are set with two functions:

gf_context_set_rop() sets the actual operation while

gf_context_set_pattern() sets the pattern the operation uses (if applicable).

```
gf_context_set_fgcolor(context, 0x00ff00);
gf_draw_rect(context, 0, 0, w, h);
gf_draw_flush(context);
```

```
{
    uint8_t pattern[9]={ 0x00, 0x01, 0x03, 0x07,
                        0x0f, 0x1f, 0x3f, 0x7f, 0xff};

    unsigned short rop[][3] = {
        {GF_ROP_DPon,      /* 0x05 ~ (D | P) */
         GF_ROP_DPo,      /* 0xFA D | P */
         GF_ROP_DPa},     /* 0xA0 /* (D & P) */
        {GF_ROP_DPx,      /* 0x5A /* D ^ P */
         GF_ROP_Psx,      /* 0x3C /* P ^ S */
         GF_ROP_PSna}     /* 0x30 /* P & (~ S) */
        };

    int i=0, j=0;
    short x_offset = 0;
    short y_offset = 0 ;
    unsigned flags = GF_CONTEXT_PATTERN_BACKFILL;
    gf_context_set_pattern(context, pattern, x_offset, y_offset, flags);
    gf_context_set_fgcolor(context, 0xff0000);
    gf_context_set_bgcolor(context, 0x0000ff);

    for (; i<2; i++){
        j = 0;
        for (; j<3; j++) {
            gf_context_set_rop(context, rop[i][j]);
            gf_draw_rect (context, j*w/3, i*h/2, (j+1)*w/3, (i+1)*h/2 )
        }
    }
    gf_draw_flush(context);
}
```

Antialiasing

Antialiasing allows you to visually approximate an ideal resolution display by varying the intensities of discrete pixels, making edges appear smoother. To turn on this parameter, use *gf_context_set_antialias()*.

Transform matrices and translations

A translation sets an x and y value which are added to all polygon, line, and polyline co-ordinates prior to rendering. A transformation matrix is a 2D matrix that is multiplied by all polygon, line, and polyline co-ordinates prior to rendering. Using a combination of the transform matrix and translations, various rotation, scaling and reflection operations are possible. To set a 2D transform matrix, use *gf_context_set_transform()*; to set a translation, use *gf_context_set_translation()*. @@@ Need example here @@@

Chapter 5

Handling User Input



This chapter shows you how to handle user input in the Graphics Framework. - @@@ overview of io-hid - @@@ attaching to io-hid - @@@ getting and processing data @@@ functions documented for sys/hiddi.h, but the rest needs to be developed



Chapter 6

Using Fonts



This chapter shows you how to use fonts in the Graphics Framework.

To render strings in the GF API, you must first attach to the font server using *gf_font_attach()*. You then call *gf_draw_string()* to render the strings. Finally, when you are finished, you should call *gf_font_detach()* to release the font server.

Here's an example:

```
if (gf_context_create(&context,gdev) != -1) {
    if (gf_context_set_surface(context,surface) != -1) {
        gf_font_t font;
        if (gf_font_attach(&font, gdev) == -1) {
            printf("gf_font_attach failed \n");
        }

        if (gf_draw_begin(context) == -1) {
            printf("gf_draw_begin failed\n");
        }

        gf_context_set_fgcolor(context, 0x000000);
        gf_draw_rect(context, x,y,w,h);

        gf_context_set_fgcolor(context, 0xFF00FF);
        gf_context_update(context);

        gf_draw_string(context, font, "Helvetica14",
            "This is a string", 0, 25, 25, NULL );

        gf_draw_flush(context);
        gf_font_detach(font);
        gf_draw_end(context);
        delay(rand() % 100);
    }

    gf_context_free(context);
}
```



Chapter 7

Using OpenGL ES



This chapter shows you how to use OpenGL ES in the Graphics Framework.

OpenGL ES is subset of the OpenGL API designed for 3D graphics on embedded systems. For more information about the standard, see <http://www.khronos.org/opengles/>.

Certain constructs within the EGL API are mapped to equivalent constructs within the GF API:

EGL construct	GF construct
<code>NativeDisplayType</code>	<code>gf_dev_t</code>
<code>NativeWindowType</code>	<code>gf_3d_target_t</code>
<code>NativePixmapType</code>	<code>gf_surface_t</code>
<code>EGL_NATIVE_VISUAL_ID</code>	GF layer format

Using OpenGL ES

In order to write an application that uses the OpenGL ES and EGL interfaces under QNX6, the application must also use the GF library to interface with the QNX6 Graphics Framework.

The application must first attach to a device using *gf_dev_attach()*. The resulting handle (a `gf_dev_t`) is passed to *eglGetDisplay()* to initialize an EGL display connection.



The code examples in this chapter are adapted from sample applications included with the GF TDK.

For example:

```
gf_dev_t      gf_dev;  
gf_dev_info_t info;  
  
/* initialize the graphics device */  
if (gf_dev_attach(&gf_dev, NULL, &info) != GF_ERR_OK) {
```

```

        perror("gf_dev_attach()");
        return -1;
    }

    ...

    /* get an EGL display connection */
    display = eglGetDisplay(gf_dev);

    if (display == EGL_NO_DISPLAY) {
        fprintf(stderr, "eglGetDisplay() failed\n");
        return -1;
    }

```

Next, the application uses the GF API to attach to a display, and then to a layer on that display. For an example, see the Steps to use GF section of the Basic Drawing chapter.

Using the EGL API, the application must now find a buffer configuration that's compatible with the layer it will use to display 3D content. The *color buffer* (the buffer being rendered to) and the *display buffer* (the buffer which holds the display contents being displayed on the selected layer) must have identical formats. This enables flicker-free double buffering with minimal overhead.

Using the GF API, the application can query the display buffer formats supported by the selected layer by calling *gf_layer_query()*. It can then call *eglChooseConfig()* to determine which buffer configurations, if any, are compatible with a given display buffer format. The key to performing this match is to use the EGL_NATIVE_VISUAL_ID attribute to select the EGL buffer configurations that support a given format. For example:

```

for (i = 0; ; i++) {
    /* Walk through all possible pixel formats for this layer */
    if (gf_layer_query(layer, i, &linfo) == -1) {
        fprintf(stderr, "Couldn't find a compatible frame "
            "buffer configuration on layer %d\n", layer_idx);
        exit(EXIT_FAILURE);
    }
}

```

```

    /*
     * We want the color buffer format to match the layer format,
     * so request the layer format through EGL_NATIVE_VISUAL_ID.
     */
    attribute_list[1] = linfo.format;

    /* Look for a compatible EGL frame buffer configuration */
    if (eglChooseConfig(display,
        attribute_list, &config, 1, &num_config) == EGL_TRUE) {
        if (num_config > 0) {
            format_idx = i;
            break;
        }
    }
}

```

Creating surfaces

Once the application has selected an EGL buffer configuration, it may now create rendering contexts to render with, and surfaces to render into. The surface types that you can render into are:

- window surfaces
- pixmap surfaces
- pbuffer surfaces

Window surfaces

To create a window surface, the application must first create a native GF 3D rendering target by calling *gf_3d_create_target()*. The application may specify a list of pre-allocated native GF surfaces to *gf_3d_create_target()*, for rendering into. Note that these surfaces must be 3D targetable. That is, the GF_SURFACE_3D_TARGETABLE flag must have been specified when they were created. Typically, the application would specify two surfaces for rendering into so it could use double buffering. If the application only passes a single surface, or if it does not pass a list of surfaces at all, then the GF library will

attempt to internally allocate as many buffers as are necessary. Upon successful creation of a 3D rendering target, the resulting handle can be passed to *eglCreateWindowSurface()* to create a surface that can be used for OpenGL ES rendering.

For example:

```
/* create a 3D rendering target */
if (gf_3d_target_create(&target, layer,
    NULL, 0, width, height, linfo.format) != GF_ERR_OK) {
    fprintf(stderr, "Unable to create rendering target\n");
    return NULL;
}

...

/* create an EGL window surface */
surface = eglCreateWindowSurface(display, config, target, NULL);

if (surface == EGL_NO_SURFACE) {
    fprintf(stderr, "Create surface failed: 0x%x\n", eglGetError());
    exit(EXIT_FAILURE);
}
```

Pixmap surfaces

To create a pixmap surface, the application must create a native GF surface using the GF API, and then pass the surface handle to *eglCreatePixmapSurface()* as the *pixmap* argument. In order to allocate the GF surface so that it is compatible with a given EGL configuration ID, the application should call *gf_3d_query_config()* to determine the surface format and flags that it must pass to the surface creation function.

```
/*
 * We want to allocate a surface that EGL can use to create
 * a Pixmap surface
 */
if (gf_3d_query_config(&cfinfo, gf_dev, display, chosen) == -1) {
    fprintf(stderr, "query native failed\n");
}
```

```

        exit(EXIT_FAILURE);
    }

    if (gf_surface_create(&gfsurface, gf_dev, WIDTH, HEIGHT,
        cfginfo.surface_format, NULL, cfginfo.create_flags) == -1) {
        fprintf(stderr, "create surface failed\n");
        exit(EXIT_FAILURE);
    }

    pmsurface = eglCreatePixmapSurface(display, chosen, gfsurface, NULL);

    if (pmsurface == EGL_NO_SURFACE) {
        fprintf(stderr, "Create Pixmap failed: 0x%x\n", eglGetError());
        exit(EXIT_FAILURE);
    }

```

PBuffer surfaces

To create a PBuffer surface, the application must specify the width and height of the surfaces via the `EGL_WIDTH` and `EGL_HEIGHT` attributes. In the case of a PBuffer surface, the actual surface memory is always allocated internally by OpenGL ES.

```

/* create an EGL pbuffer surface */
pbsurface = eglCreatePbufferSurface(display, chosen, pb_attrs);

if (pbsurface == EGL_NO_SURFACE) {
    fprintf(stderr, "Create PBuffer failed: 0x%x\n", eglGetError());
    exit(EXIT_FAILURE);
}

/* connect the context to the PBuffer surface */
if (eglMakeCurrent(display, pbsurface, pbsurface, context) == EGL_FALSE) {
    fprintf(stderr, "Make current failed: 0x%x\n", eglGetError());
    exit(EXIT_FAILURE);
}

```



Chapter 8

Embedding GF



This chapter covers the things you need to know to create a GF application that is run on an embedded target.

General configuration

- how to get the configuration parameters you need for your hardware

Required elements

- gf libraries / binaries - gles - monitor / io-display / io-hid - graphics driver - imagelib codecs and config file

Fonts

Sample buildfiles

- for most common boards - other configuration files



Appendix A

Hardware Capabilities

In this appendix...

Coral 59



This appendix contains information about supported hardware capabilities and limitations. Currently GF supports one graphics hardware set: Fujitsu Coral (`devg-coral.so`).

Coral

GF drawing on Coral hardware has these limitations:

- Two-color dashed line drawing is supported, but transparent dashed lines are not.
- Only these values are valid for the dash repeat count when setting line dashing with `gf_context_set_linedash()`: 1, 2, 3, 4, 6, 8, 12, 16, 24, 32.
- The line thickness may be no more than **32** pixels.
- Only butt and bevel joins on polylines are supported.



Appendix B

QNX Graphics Framework Library Reference



These functions handle operations that directly involve the QNX Graphics Framework. Using these functions and structures, you can:

- Attach your application to a device, display, layer, and surface.
- Set drawing context attributes.
- Create a 3D rendering target for OpenGL ES.
- Perform drawing operations.
- Control a cursor.
- Attach to a font server and render text strings.



Appendix C

Summary of Entries

In this appendix...

3D rendering	69
Contexts	69
Cursors	70
Devices and displays	71
Drawing	71
Fonts	72
Layers	72
Surfaces	73
<i>gf_3d_target_create()</i>	75
<i>gf_3d_target_free()</i>	77
<i>gf_3d_query_config()</i>	79
gf_alpha_t	80
gf_chroma_t	86
<i>gf_context_create()</i>	88
<i>gf_context_disable_alpha()</i>	90
<i>gf_context_disable_antialias()</i>	91
<i>gf_context_disable_chroma()</i>	93
<i>gf_context_disable_clipping()</i>	94
<i>gf_context_disable_linedash()</i>	95
<i>gf_context_disable_pattern()</i>	96
<i>gf_context_disable_rop()</i>	97
<i>gf_context_disable_transform()</i>	98
<i>gf_context_disable_translation()</i>	99
<i>gf_context_free()</i>	100
<i>gf_context_set_alpha()</i>	101
<i>gf_context_set_antialias()</i>	103
<i>gf_context_set_bgcolor()</i>	105
<i>gf_context_set_chroma()</i>	107
<i>gf_context_set_clipping()</i>	109

<i>gf_context_set_fgcolor()</i>	111
<i>gf_context_set_linedash()</i>	112
<i>gf_context_set_linejoin()</i>	114
<i>gf_context_set_pattern()</i>	116
<i>gf_context_set_penwidth()</i>	118
<i>gf_context_set_rop()</i>	120
<i>gf_context_set_surface()</i>	133
<i>gf_context_set_transform()</i>	135
<i>gf_context_set_translation()</i>	137
<i>gf_cursor_disable()</i>	139
<i>gf_cursor_enable()</i>	141
<i>gf_cursor_set()</i>	143
<i>gf_cursor_set_pos()</i>	146
<i>gf_dev_attach()</i>	148
<i>gf_dev_detach()</i>	151
<i>gf_dev_register_thread()</i>	152
gf_dim_t	153
<i>gf_display_attach()</i>	154
<i>gf_display_detach()</i>	158
<i>gf_display_set_layer_order()</i>	159
<i>gf_draw_begin()</i>	162
<i>gf_draw_bitmap()</i>	165
<i>gf_draw_blit1()</i>	167
<i>gf_draw_blit2()</i>	169
<i>gf_draw_blitscaled()</i>	171
<i>gf_draw_end()</i>	173
<i>gf_draw_finish()</i>	175
<i>gf_draw_flush()</i>	177
<i>gf_draw_poly_fill()</i>	179
<i>gf_draw_polyline()</i>	182
<i>gf_draw_rect()</i>	184
<i>gf_draw_string()</i>	186
<i>gf_font_attach()</i>	188
<i>gf_font_detach()</i>	190
<i>gf_layer_attach()</i>	191
<i>gf_layer_detach()</i>	193
<i>gf_layer_disable()</i>	194
<i>gf_layer_enable()</i>	196
<i>gf_layer_query()</i>	198
<i>gf_layer_set_blending()</i>	205
<i>gf_layer_set_brightness()</i>	207
<i>gf_layer_set_chroma()</i>	209
<i>gf_layer_set_contrast()</i>	211
<i>gf_layer_set_dst_viewport()</i>	213
<i>gf_layer_set_saturation()</i>	215

gf_layer_set_src_viewport() 217
gf_layer_set_surfaces() 219
gf_layer_update() 221
gf_palette_t 223
gf_point_t 225
gf_surface_attach() 226
gf_surface_attach_by_sid() 229
gf_surface_create() 231
gf_surface_create_layer() 235
gf_surface_free() 239
gf_surface_get_info() 241



This chapter groups the datatypes and functions according to their purpose. You can use this chapter to determine what you need to perform a task.

3D rendering

gf_3d_query_config()

Query the current 3D configuration

gf_3d_target_create()

Create a 3D rendering target

gf_3d_target_free()

Free a 3D rendering target

Contexts

gf_context_create()

Create a context

gf_context_free()

Free a context

gf_context_set_alpha(), *gf_context_disable_alpha()*

Set or disable the alpha blending parameters for a context

gf_context_set_antialias(), *gf_context_disable_antialias()*

Set or disable antialias for a context

gf_context_set_bgcolor()

Set the background color for a context

gf_context_set_chroma(), *gf_context_disable_chroma()*

Set or disable the chroma key parameters for a context

gf_context_set_clipping(), *gf_context_disable_clipping()*

Set or disable the clipping rectangle for a context

gf_context_set_fgcolor()

Set the foreground color for a context

gf_context_set_linedash(), *gf_context_disable_linedash()*

Set or disable the dash style for polylines

gf_context_set_linejoin()

Set the join style for polylines

gf_context_set_pattern(), *gf_context_disable_pattern()*

Set or disable the pattern for use with raster operations

gf_context_set_penwidth()

Set the width for polylines

gf_context_set_rop(), *gf_context_disable_rop()*

Set or disable the raster operation

gf_context_set_surface()

Associate a surface with the draw context

gf_context_set_transform(), *gf_context_disable_transform()*

Set or disable the 2D transform matrix for a context

gf_context_set_translation(), *gf_context_disable_translation()*

Set or disable the translation for a context

Cursors

gf_cursor_enable()

Enable a hardware cursor

gf_cursor_disable()

Disable a hardware cursor

gf_cursor_set() Replace the current hardware cursor bitmap definition

gf_cursor_set_pos()

Set the hardware cursor position

Devices and displays

gf_dev_attach()

Attach to a device

gf_dev_detach()

Detach from a device

gf_dev_register_thread()

Register a graphics device initialized in another thread

gf_display_attach()

Attach to a display

gf_display_detach()

Detach from a display

gf_display_detach()

Detach from a display

gf_display_set_layer_order()

Set the ordering of the layers for a display

Drawing

gf_draw_begin() Begin rendering

gf_draw_end() Finish rendering

gf_draw_finish() Wait for the rendering hardware to finish

gf_draw_flush() Flush the draw buffer

gf_draw_bitmap()

Draw a bitmap

gf_draw_blit1() Blit from one area to another in a context

gf_draw_blit2() Blit between contexts

gf_draw_blitscaled() Blit and scale an area

gf_draw_flush() Flush the draw buffer

gf_draw_poly_fill() Draw a filled polygon

gf_draw_polyline() Draw an unfilled polygon

gf_draw_rect() Draw a rectangle

Fonts

gf_font_attach() Attach to a font server

gf_font_detach() Detach from a font server

gf_draw_string() Render a string

Layers

gf_layer_attach() Attach to a layer

gf_layer_detach() Detach from a layer

gf_layer_disable() Disable a layer

gf_layer_enable() Enable a layer

gf_layer_query()

Query a layer's properties

gf_layer_set_blending()

Set layer blending

gf_layer_set_brightness()

Set a layer's brightness

gf_layer_set_chroma()

Set a layer's chroma operation

gf_layer_set_contrast()

Set a layer's contrast level

gf_layer_set_dst_viewport()

Set a layer's destination viewport

gf_layer_set_saturation()

Set a layer's saturation level

gf_layer_set_src_viewport()

Set a layer's source viewport

gf_layer_set_surfaces()

Set the surface for a layer

gf_layer_update()

Update layer parameters

Surfaces

gf_surface_attach()

Allocate and attach to a surface

gf_surface_attach_by_sid()

Attach to a previously allocated surface

Surfaces

gf_surface_create()

Create a new surface

gf_surface_create_layer()

Create a surface for a layer

gf_surface_free()

Free a surface

gf_surface_get_info()

Get surface parameters

gf_3d_target_create()

Create a target for 3D rendering

Synopsis:

```
int gf_3d_target_create( gf_3d_target_t *   ptarget,
                        gf_layer_t          layer,
                        gf_surface_t *      surfaces,
                        int                  nsurfaces,
                        int                  width,
                        int                  height,
                        unsigned             layer_format );
```

Arguments:

<i>ptarget</i>	A pointer to a gf_3d_target_t handle for a 3D rendering target which you want to create.
<i>layer</i>	The layer which will be used to display the 3D content.
<i>surfaces</i>	An array of pre-allocated surfaces for 3D rendering. If NULL, the function allocates the required surfaces.
<i>nsurfaces</i>	The number of items in the <i>surfaces</i> array.
<i>width, height</i>	
<i>layer_format</i>	

Library:

gf

Description:

This function creates a 3D rendering target. @@@ need link to the user guide on how to use a 3D target @@@

gf_3d_target_create()

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_3d_target_free(), *gf_3d_query_config()*

gf_3d_target_free()

Free 3D rendering target

Synopsis:

```
void gf_3d_target_free( gf_3d_target_t target );
```

Arguments:

target A pointer to a **gf_3d_target_t** handle for the 3D rendering target that you want to free.

Library:

gf

Description:

This function frees a 3D rendering target.

Returns:

0 Success.
-1 An error occurred.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_3d_target_free()

See also:

gf_3d_target_create(), *gf_3d_query_config()*

gf_3d_query_config()

Query the current 3D configuration

Synopsis:

```
int gf_3d_query_config( gf_3d_config_info_t*  info,
                        gf_dev_t              dev,
                        EGLDisplay            display,
                        EGLConfig             cfg );
```

Arguments:

<i>info</i>	@@@ ??? @@@
<i>dev</i>	@@@ ??? @@@
<i>display</i>	@@@ ??? @@@
<i>cfg</i>	@@@ ??? @@@

Library:

gf

Description:

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

gf_3d_target_create(), *gf_3d_target_free()*

gf_alpha_t

Alpha operation specification structure

Synopsis:

See below

Description:

The `gf_alpha_t` structure describes an alpha operation. It contains at least the following members:

`_uint32 mode` The alpha mode to determine the blending/testing operation. See the list of valid flags below.

`unsigned map_w, unsigned map_h`

The alpha map width (in bytes) and height (in rows). An alpha map is only used if the `GF_ALPHA_M1_MAP` or `GF_ALPHA_M2_MAP` mode bit is set.

`unsigned map_stride`

The number of bytes between scanlines in alpha map. The Alpha map is only used if the `GF_ALPHA_M1_MAP` or `GF_ALPHA_M2_MAP` mode bit is set.

`unsigned map_x_offset, unsigned map_y_offset`

The initial horizontal and vertical offset into alpha map.

`_uint8* map` A pointer to the alpha map, which is assumed to be a sequence of `map_h` rows of `map_w` bytes each. Each byte represents an alpha factor ranging from 0 to 255.

`_uint8m1` The global M1 factor used when the `GF_ALPHA_M1_GLOBAL` mode bit is set.

`_uint8m2` The global M2 factor used when the `GF_ALPHA_M2_GLOBAL` mode bit is set.

Alpha mode flags

These flags set the alpha blending operation. They are:

Alpha Blending multiplier origins:

GF_ALPHA_M1_SRC_PIXEL_ALPHA

“M1” from the source pixel alpha component

GF_ALPHA_M1_DST_PIXEL_ALPHA

“M1” from the destination pixel alpha component

GF_ALPHA_M1_GLOBAL

“M1” from the global M1

GF_ALPHA_M1_MAP

“M1” comes from the Alpha map

GF_ALPHA_M2_SRC_PIXEL_ALPHA

“M2” comes from the source pixels alpha component

GF_ALPHA_M2_DST_PIXEL_ALPHA

“M2” comes from the destination pixels alpha component

GF_ALPHA_M2_GLOBAL

“M2” comes from the global multiplier 2

GF_ALPHA_M2_MAP

“M2” comes from the Alpha map

GF_ALPHA_TEST

Alpha test instead of blend

Alpha Blending Source Factor:

GF_BLEND_SRC_0

(0,0,0,0)

gf_alpha_t

GF_BLEND_SRC_M1
(M1,M1,M1,M1)

GF_BLEND_SRC_1mM1
(1-M1,1-M1,1-M1,1-M1)

GF_BLEND_SRC_1mD
(1-M2,1-Rd,1-Gd,1-Bd)

GF_BLEND_SRC_M2
(M2,M2,M2,M2)

GF_BLEND_SRC_D
(M2,Rd,Gd,Bd)

GF_BLEND_SRC_1
(1,1,1,1)

GF_BLEND_SRC_A1M1
(1,M1,M1,M1)

GF_BLEND_SRC_1mM2
(1-M2,1-M2,1-M2,1-M2)

GF_BLEND_SRC_1mA1M1
(1-M1,1-M1,1-M1,1-M1)

GF_BLEND_SRC_A1M2
(1,M2,M2,M2)

GF_BLEND_SRC_1mA1M2
(0,1-M2,1-M2,1-M2)

GF_BLEND_SRC_A0M1
(0,M1,M1,M1)

GF_BLEND_SRC_1mA0M1
(1,1-M1,1-M1,1-M1)

GF_BLEND_SRC_A0M2

(0,M2,M2,M2)

GF_BLEND_SRC_1mA0M2

(1,1-M2,1-M2,1-M2)

Alpha Blending Destination Factor:

GF_BLEND_DST_0

(0,0,0,0)

GF_BLEND_DST_M1

(M1,M1,M1,M1)

GF_BLEND_DST_1mM1

(1-M1,1-M1,1-M1,1-M1)

GF_BLEND_DST_1mS

(1-M1,1-Rs,1-Gs,1-Bs)

GF_BLEND_DST_M2

(M2,M2,M2,M2)

GF_BLEND_DST_S

(M1,Rs,Gs,Bs)

GF_BLEND_DST_1

(1,1,1,1)

GF_BLEND_DST_A1M1

(1,M1,M1,M1)

GF_BLEND_DST_1mM2

(1-M2,1-M2,1-M2,1-M2)

GF_BLEND_DST_1mA1M1

(0,1-M1,1-M1,1-M1)

gf_alpha_t

GF_BLEND_DST_A1M2

(1,M2,M2,M2)

GF_BLEND_DST_1mA1M2

(0,1-M2,1-M2,1-M2)

GF_BLEND_DST_A0M1

(0,M1,M1,M1)

GF_BLEND_DST_1mA0M1

(1,1-M1,1-M1,1-M1)

GF_BLEND_DST_A0M2

(0,M2,M2,M2)

GF_BLEND_DST_1mA0M2

(1,1-M2,1-M2,1-M2)

Alpha Tests Pixels are written in destination:

GF_TEST_NEVER

Never

GF_TEST_ALWAYS

Always

GF_TEST_LESS_THAN

if $M1 < M2$

GF_TEST_LESS_THAN_OR_EQUAL

if $M1 \leq M2$

GF_TEST_EQUAL

if $M1 == M2$

GF_TEST_GREATER_THAN_OR_EQUAL

if $M1 \geq M2$

GF_TEST_GREATER_THAN

if M1 > M2

GF_TEST_NOT_EQUAL

if M1 != M2

Classification:

Graphics Framework

See also:

gf_context_set_alpha(), gf_layer_set_blending()

gf_chroma_t

Specification of a chroma operation

Synopsis:

```
typedef struct {  
    _uint32 mode;  
    /* Key color to compare against test pixel */  
    gf_color_t color0;  
    gf_color_t color1;  
    gf_color_t mask;  
} gf_chroma_t;
```

Description:

The `gf_chroma_t` structure describes a chroma operation. It contains at least the following members:

<i>mode</i>	The chroma mode. This parameter must be a combination of one of: <ul style="list-style-type: none">• <code>GF_CHROMA_OP_SRC_MATCH</code> — test against source pixels• <code>GF_CHROMA_OP_DST_MATCH</code> — test against destination pixels and one of: <ul style="list-style-type: none">• <code>GF_CHROMA_OP_DRAW</code> — draw if <i>color0</i> matches the color of the test pixel• <code>GF_CHROMA_OP_NO_DRAW</code> — do not draw if <i>color0</i> matches the color of the test pixel
<i>color0</i>	Key color to compare against the test pixel for a single color match, or the first color in the range for a range match.
<i>color1</i>	Last color in a range, for a range match.
<i>mask</i>	@@@ ??? @@@

Classification:

Graphics Framework

See also:

gf_context_set_chroma(), *gf_layer_set_chroma()*

gf_context_create()

Create and initialize a new draw context

Synopsis:

```
int gf_context_create( gf_context_t * pcontext );
```

Arguments:

pcontext A pointer to a **gf_context_t** to store a handle for the new context.

Library:

gf

Description:

This function creates a graphics rendering context, or draw context. Draw contexts are used to target rendering surfaces, and preserve a state across multiple rendering calls.

Returns:

0 Success.

-1 An error occurred (*errno* is set).

Errors:

ENOMEM Insufficient memory to allocate structures.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_context_create()

See also:

gf_context_free(),

gf_context_disable_alpha()

Disable the current alpha operation

Synopsis:

```
void gf_context_disable_alpha( gf_context_t context );
```

Arguments:

context The graphics context to disable the alpha channel for.

Library:

gf

Description:

This function disables the alpha blending and alpha testing for subsequent draw operations for a draw context.



To re-enable alpha blending, use *gf_context_set_alpha()*.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_set_alpha()

gf_context_disable_antialias()

Disable antialiasing

Synopsis:

```
void gf_context_disable_antialias( gf_context_t context
                                  unsigned flags );
```

Arguments:

context The graphics context to disable the alpha channel for.

flags Flags to set the behavior. One flag is currently defined:
GF_CONTEXT_ANTIALIAS_POLYLINE — disable antialiasing for polylines.

Library:

gf

Description:

This function disables antialiasing for subsequent draw operations for a draw context.



To re-enable antialiasing, use *gf_context_set_antialias()*.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_context_disable_antialias()

See also:

gf_context_set_antialias()

gf_context_disable_chroma()

Disable the current chroma operation

Synopsis:

```
void gf_context_disable_chroma( gf_context_t context );
```

Arguments:

context The graphics context to disable chroma for.

Library:

gf

Description:

This function disables chroma key operations for subsequent draw operations for a draw context.



To re-enable chroma keying, use *gf_context_set_chroma()*.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_set_chroma()

gf_context_disable_clipping()

Disable the clipping rectangle

Synopsis:

```
void gf_context_disable_clipping( gf_context_t context );
```

Arguments:

context The graphics context to disable clipping for.

Library:

gf

Description:

This function disables clipping for subsequent draw operations for a draw context. When clipping is disabled, draw primitives are still clipped to the bounds of the current surface.



To re-enable clipping, use *gf_context_set_clipping()*.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_set_clipping()

gf_context_disable_linedash()

Disable line dashing

Synopsis:

```
void gf_context_disable_linedash( gf_context_t context );
```

Arguments:

context The graphics context to disable line dashing for.

Library:

gf

Description:

This function disables line dashes for subsequent draw operations for a draw context.



To re-enable the line dashing, use *gf_context_set_linedash()*.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_set_bgcolor(), *gf_context_set_fgcolor()*,
gf_context_set_linedash(), *gf_context_set_linejoin()*,
gf_context_set_penwidth()

gf_context_disable_pattern()

Disable the pattern

Synopsis:

```
void gf_context_disable_pattern( gf_context_t context );
```

Arguments:

context The graphics context to disable the pattern for.

Library:

gf

Description:

This function disables the pattern for subsequent draw operations for a draw context.



To re-enable the pattern, use *gf_context_set_pattern()*.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_set_pattern()

gf_context_disable_rop()

Disable the ROP

Synopsis:

```
void gf_context_disable_rop( gf_context_t context );
```

Arguments:

context The graphics context to disable the ROP for.

Library:

gf

Description:

This function disables raster operations (ROPs) for subsequent draw operations for a draw context.



To re-enable the ROP, use *gf_context_set_rop()*.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_set_rop()

gf_context_disable_transform()

Disable 2D transform matrix

Synopsis:

```
void gf_context_disable_transform( gf_context_t context );
```

Arguments:

context The graphics context to disable the 2D transform matrix for.

Library:

gf

Description:

This function disables the 2D transform matrix for subsequent draw operations for a draw context.



To re-enable the transform matrix, use *gf_context_set_transform()*.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_set_transform()

gf_context_disable_translation()

Disable translation

Synopsis:

```
void gf_context_disable_translation( gf_context_t context );
```

Arguments:

context The graphics context to disable the x,y translation for.

Library:

gf

Description:

This function disables the translation of x, y coordinates for subsequent draw operations for a draw context.



To re-enable the translation, use *gf_context_set_translation()*.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_set_translation()

gf_context_free()

Destroy a draw context

Synopsis:

```
void gf_context_free( gf_context_t context );
```

Arguments:

context The graphics context to free.

Library:

gf

Description:

This function destroys a graphics rendering context created with *gf_context_create()*, release all resources associated with it.



This function will not release the surface which the context is currently targeting. You will need to also call *gf_surface_free()* to free the surface.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_create(),

gf_context_set_alpha()

Set the current alpha operation

Synopsis:

```
void gf_context_set_alpha( gf_context_t context,  
                           gf_alpha_t const *alpha );
```

Arguments:

context The graphics context to set the alpha channel for.

alpha A pointer to a **gf_alpha_t** that describes the alpha blending parameters. Do not pass NULL for this parameter.

Library:

gf

Description:

This function sets the alpha blending parameters for subsequent draw operations for a draw context. Alpha blending gives you control over the opacity of pixels.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_context_set_alpha()

See also:

`gf_alpha_t`, *gf_context_disable_alpha()*,

gf_context_set_antialias()

Set the antialias behavior

Synopsis:

```
void gf_context_set_antialias( gf_context_t context,  
                              unsigned flags );
```

Arguments:

- context* The graphics context to set the antialias flags for.
- flags* The flags to enable antialiasing. One flag is defined:
- GF_CONTEXT_ANTIALIAS_POLYLINE — Apply antialiasing to polylines

Library:

gf

Description:

This function sets the antialias flags for subsequent draw operations for a draw context. Antialiasing allows you to visually approximate an ideal resolution display by varying the intensities of discrete pixels, making edges appear smoother.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_context_set_antialias()

See also:

gf_context_disable_antialias()

gf_context_set_bgcolor()

Replace the background rendering color

Synopsis:

```
void gf_context_set_bgcolor( gf_context_t context,
                             gf_color_t color );
```

Arguments:

<i>context</i>	Draw context handle.
<i>color</i>	The new background color, in packed ARGB8888 format.

Library:

gf

Description:

This function replaces the current secondary (background) rendering *color* for the given graphics *context*. The background color applies only to a few rendering operations:

- patterns
- backfilled bitmaps
- dashed lines.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_context_set_bgcolor()

See also:

gf_context_set_fgcolor()

gf_context_set_chroma()

Replace the current chroma operation

Synopsis:

```
void gf_context_set_chroma( gf_context_t context,
                           gf_chroma_t const *chroma );
```

Arguments:

<i>context</i>	Draw context handle.
<i>chroma</i>	A pointer to a gf_chroma_t that describes the chroma operation parameters. Do not pass NULL for this parameter.

Library:

gf

Description:

This function replaces the chroma parameters for subsequent draw operations for a draw context. A chroma operation gives you extended control over which pixels are plotted in terms of chroma key colors and testing operations.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_context_set_chroma()

See also:

`gf_chroma_t`, `gf_context_disable_chroma()`

gf_context_set_clipping()

Replace the clipping rectangle

Synopsis:

```
void gf_context_set_clipping( gf_context_t context,
                             int x1,
                             int y1,
                             int x2,
                             int y2 );
```

Arguments:

<i>context</i>	The draw context to set the clipping for.
<i>x1, y1, x2, y2</i>	The coordinates of the upper-left corner (x1,y1) and lower right corner (x2,y2) of the clipping rectangle.

Library:

gf

Description:

This function replaces the clipping region for the specified draw *context*. Subsequent draw commands are clipped to this region, meaning any part of the command that extends beyond the boundaries of the clipping rectangle doesn't appear on the display.



Clipping rectangles in the Graphics Framework are inclusive. That is, draw operations are clipped to the area within the clipping rectangle, including the values that define the rectangle.

Classification:

Graphics Framework

gf_context_set_clipping()

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_disable_clipping()

gf_context_set_fgcolor()

Replace the foreground rendering color

Synopsis:

```
void gf_context_set_fgcolor( gf_context_t context,
                             gf_color_t color );
```

Arguments:

context The draw context to set the color for.

color The new foreground color, in packed ARGB8888 format.

Library:

gf

Description:

This function replaces the foreground *color* for the given graphics *context*. The foreground color applies to most draw operations, such as lines, filled rectangles and polygons, bitmaps, and so on.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_set_bgcolor()

gf_context_set_linedash()

Set the current line dash style

Synopsis:

```
void gf_context_set_linedash( gf_context_t context,
                             _uint32 line_pat,
                             unsigned line_pat_offset,
                             int line_repeat,
                             unsigned flags );
```

Arguments:

<i>context</i>	Draw context handle.
<i>line_pat</i>	A bitmask holding the stipple pattern for broken lines. The bits are in most-significant to least-significant order. A high bit (1) means render the pixel with the foreground color, whereas a low bit (0) means render the pixel with the background color or not at all, depending on the <i>flags</i> parameter.
<i>line_pat_offset</i>	Offset into the line pattern.
<i>line_repeat</i>	Specifies how many of the bits in <i>line_pat</i> to use, after which the pattern is repeated. If <i>line_repeat</i> is less than 32, then the most-significant bits of <i>line_pat</i> are ignored. This value cannot be more than 32.
<i>flags</i>	Sets how the background is rendered for low bits (0). Can be one of: <ul style="list-style-type: none">GF_CONTEXT_LINEDASH_BACKFILL — render "off" pixels (0) in the background color. The default is to treat these pixels as transparent.

Library:

gf

Description:

This function replaces the current line dashing style. Line dashing allows you to apply stipple patterns along the path of polylines to draw lines of varying dashes.

To unset dashing (that is, render solid lines), pass a value of 0 for *flags*.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_disable_linedash(), *gf_context_set_bgcolor()*,
gf_context_set_fgcolor(), *gf_context_set_linejoin()*,
gf_context_set_penwidth()

gf_context_set_linejoin()

Replace the line join style

Synopsis:

```
void gf_context_set_linejoin( gf_context_t context,
                             int line_join );
```

Arguments:

context Draw context handle.

line_join The join style. Can be one of:

- GF_CONTEXT_LINEJOIN_BUTT — the default style
- GF_CONTEXT_LINEJOIN_BEVEL

Library:

gf

Description:

This function replaces the current joining style. Line joining allows you to control how the joints between the segments of a polyline are rendered. @@@ Insert illustration showing styles here: @@@



Join styles only apply to polylines where penwidth is greater than 0.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_context_set_linejoin()

See also:

gf_context_set_pattern()

Set the current ROP pattern

Synopsis:

```
void gf_context_set_pattern( gf_context_t context,
                             const uint8_t *pattern,
                             unsigned short x_offset,
                             unsigned short y_offset,
                             unsigned flags )
```

Arguments:

context The draw context to set the pattern for.

pattern Pointer to the bitmap containing the pattern

x_offset, y_offset

 The horizontal and vertical offset for the pattern bitmap.

flags Flags controlling additional behavior. Valid flags are:

- GF_CONTEXT_PATTERN_BACKFILL — The off bits (0) in the pattern are rendered with the background color. The default is to render them as transparent.

Library:

gf

Description:

This function sets the current pattern. Patterns are used in conjunction with raster operations (ROPs) to allow a total of 256 possible operations — see *gf_context_set_rop()* for more information.

Classification:

Graphics Framework

gf_context_set_pattern()

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_create()

gf_context_set_penwidth()

Set the pen width

Synopsis:

```
void gf_context_set_penwidth( gf_context_t context,
                             unsigned w );
```

Arguments:

<i>context</i>	The draw context to set the pen width for.
<i>w</i>	The width of the line, in pixels. Set to 0 to reset the pen width to the default rendering of thin primitives.

Library:

gf

Description:

This function replaces the pixel width of lines for draw primitives such as lines and points.



Currently pen width applies only to polylines on hardware that has accelerated thick line support.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_context_set_penwidth()

See also:

gf_draw_polyline()

gf_context_set_rop()

Replace the current raster operation

Synopsis:

```
void gf_context_set_rop( gf_context_t context,
                        unsigned short rop );
```

Arguments:

context The graphics context to set the ROP for.

rop The new ROP; see the list below.

Library:

gf

Description:

This function sets the raster operation (ROP) for subsequent draw operations for a draw context. ROPs give you extended control over how pixels are plotted in the form of bitwise operations considering source and destination values, as well as patterns. You can set the pattern using *gf_context_set_pattern()*. The Graphics Framework supports a total of 256 ROPs.

Raster Operation Defines

The table below lists the defined ROPs for GF. For additional macros, see **public/gf/gf_rop.h** in the GF source directory.

In the ROP names, these letters have the following meanings:

D	Dest Bit
S	Source Bit
P	Pattern Bit
o	bitwise or operation
a	bitwise and operation

- x bitwise exclusive or operation
- n bitwise not operation

Name	Rop #	Operation
GF_ROP_ZERO	0x00	0
GF_ROP_DPSoon	0x01	$\sim ((D \mid (P \mid S)))$
GF_ROP_DPSona	0x02	$D \& (\sim (P \mid S))$
GF_ROP_PSon	0x03	$\sim (P \mid S)$
GF_ROP_SDPona	0x04	$S \& (\sim (D \mid P))$
GF_ROP_DPon	0x05	$\sim (D \mid P)$
GF_ROP_PDSxnon	0x06	$\sim (P \mid (\sim (D \wedge S)))$
GF_ROP_PDSaon	0x07	$\sim (P \mid (D \& S))$
GF_ROP_SDPnaa	0x08	$S \& (D \& (\sim P))$
GF_ROP_PDSxon	0x09	$\sim (P \mid (D \wedge S))$
GF_ROP_DPna	0x0A	$D \& (\sim P)$
GF_ROP_PSDnaon	0x0B	$\sim (P \mid (S \& (\sim D)))$
GF_ROP_SPna	0x0C	$S \& (\sim P)$
GF_ROP_PDSnaon	0x0D	$\sim (P \mid (D \& (\sim S)))$
GF_ROP_PDSonon	0x0E	$\sim (P \mid (\sim (D \mid S)))$
GF_ROP_Pn	0x0F	$\sim P$
GF_ROP_PDSona	0x10	$P \& (\sim (D \mid S))$
GF_ROP_DSon	0x11	$\sim (D \mid S)$
GF_ROP_SDPxnon	0x12	$\sim (S \mid (\sim (D \wedge P)))$
GF_ROP_SDPaon	0x13	$\sim (S \mid (D \& P))$

continued...

gf_context_set_rop()

Name	Rop #	Operation
GF_ROP_DPSxnon	0x14	$\sim (D \mid (\sim (P \wedge S)))$
GF_ROP_DPSaon	0x15	$\sim (D \mid (P \wedge S))$
GF_ROP_PSDPSanaxx	0x16	$P \wedge (S \wedge (D \wedge (\sim (P \wedge S))))$
GF_ROP_SSPxDSxaxn	0x17	$\sim (S \wedge ((S \wedge P) \wedge (D \wedge S)))$
GF_ROP_SPxPDxa	0x18	$(S \wedge P) \wedge (P \wedge D)$
GF_ROP_SDPSanaxn	0x19	$\sim (S \wedge (D \wedge (\sim (P \wedge S))))$
GF_ROP_PDSPaox	0x1A	$P \wedge (D \mid (S \wedge P))$
GF_ROP_SDPSxaxn	0x1B	$\sim (S \wedge (D \wedge (P \wedge S)))$
GF_ROP_PSDPaox	0x1C	$P \wedge (S \mid (D \wedge P))$
GF_ROP_DSPDxaxn	0x1D	$\sim (D \wedge (S \wedge (P \wedge D)))$
GF_ROP_PDSox	0x1E	$P \wedge (D \mid S)$
GF_ROP_PDSaon	0x1F	$\sim (P \wedge (D \mid S))$
GF_ROP_DPSnaa	0x20	$D \wedge (P \wedge (\sim S))$
GF_ROP_SDPxon	0x21	$\sim (S \mid (D \wedge P))$
GF_ROP_DSna	0x22	$\sim (D \wedge S)$
GF_ROP_SPDnaon	0x23	$\sim (S \mid (P \wedge (\sim D)))$
GF_ROP_SPxDSxa	0x24	$(S \wedge P) \wedge (D \wedge S)$
GF_ROP_PDSPanaxn	0x25	$\sim (P \wedge (D \wedge (\sim (S \wedge P))))$
GF_ROP_SDPSaox	0x26	$S \wedge (D \mid (P \wedge S))$
GF_ROP_SDPSxnox	0x27	$S \wedge (D \mid (\sim (P \wedge S)))$
GF_ROP_DPSxa	0x28	$D \wedge (P \wedge S)$
GF_ROP_PSDPSaoxxn	0x29	$\sim (P \wedge (S \wedge (D \mid (P \wedge S))))$
GF_ROP_DPSana	0x2A	$D \wedge (\sim (P \wedge S))$

continued...

Name	Rop #	Operation
GF_ROP_SSPPxPDxaxn	0x2B	$\sim (S \wedge ((S \wedge P) \& (P \wedge D)))$
GF_ROP_SPDSsoax	0x2C	$S \wedge (P \& (D \mid S))$
GF_ROP_PSDnox	0x2D	$P \wedge (S \mid (\sim D))$
GF_ROP_PSDPxox	0x2E	$P \wedge (S \mid (D \wedge P))$
GF_ROP_PSDnoan	0x2F	$\sim (P \& (S \mid (\sim D)))$
GF_ROP_PSnax	0x30	$P \& (\sim S)$
GF_ROP_SDPnaon	0x31	$\sim (S \mid (D \& (\sim P)))$
GF_ROP_SDPSoox	0x32	$S \wedge (D \mid (P \mid S))$
GF_ROP_Sn	0x33	$\sim S$
GF_ROP_SPDSsoax	0x34	$S \wedge (P \mid (D \& S))$
GF_ROP_SPDSxnox	0x35	$S \wedge (P \mid (\sim (D \wedge S)))$
GF_ROP_SDPox	0x36	$S \wedge (D \mid P)$
GF_ROP_SDPoan	0x37	$\sim (S \& (D \mid P))$
GF_ROP_PSDPoax	0x38	$P \wedge (S \& (D \mid P)))$
GF_ROP_SPDnox	0x39	$S \wedge (P \mid (\sim D)))$
GF_ROP_SPDSxox	0x3A	$S \wedge (P \mid (D \wedge S))$
GF_ROP_SPDnoan	0x3B	$\sim (S \& (P \mid (\sim D)))$
GF_ROP_PSn	0x3C	$P \wedge S$
GF_ROP_SPDSsoax	0x3D	$S \wedge (P \mid (\sim (D \mid S)))$
GF_ROP_SPDSnaox	0x3E	$S \wedge (P \mid (D \& (\sim S)))$
GF_ROP_PSan	0x3F	$\sim (P \& S)$
GF_ROP_PSDnaa	0x40	$P \& (S \& (\sim D))$
GF_ROP_DPSxon	0x41	$\sim (D \mid (P \wedge S))$

continued...

gf_context_set_rop()

Name	Rop #	Operation
GF_ROP_SDxPDxa	0x42	$(S \wedge D) \& (P \wedge D)$
GF_ROP_SPDSanaxn	0x43	$\sim (S \wedge (P \& (\sim (D \& S))))$
GF_ROP_SDna	0x44	$S \& (\sim D)$
GF_ROP_DPSnaon	0x45	$\sim (D \mid (P \& (\sim S)))$
GF_ROP_DSPDaon	0x46	$D \wedge (S \mid (P \& D))$
GF_ROP_PSDPxaxn	0x47	$\sim (P \wedge (S \& (D \wedge P)))$
GF_ROP_SDPxa	0x48	$S \& (D \wedge P)$
GF_ROP_PDSPDoaxxn	0x49	$\sim (P \wedge (D \wedge (S \& (P \mid D))))$
GF_ROP_DPSDoax	0x4A	$D \wedge (P \& (S \mid D))$
GF_ROP_PDSnox	0x4B	$P \wedge (D \mid (\sim S))$
GF_ROP_SDPana	0x4C	$S \& (\sim (D \& P))$
GF_ROP_SSPxDSxoxn	0x4D	$\sim (S \wedge ((S \wedge P) \mid (D \wedge S)))$
GF_ROP_PDSPxox	0x4E	$P \wedge (D \mid (S \wedge P))$
GF_ROP_PDSnoan	0x4F	$\sim (P \& (D \mid (\sim S)))$
GF_ROP_PDna	0x50	$(\sim D) \& P$
GF_ROP_DSPnaon	0x51	$\sim (D \mid (S \& (\sim P)))$
GF_ROP_DPSDaon	0x52	$D \wedge (P \mid (S \& D))$
GF_ROP_SPDSxaxn	0x53	$\sim (S \wedge (P \& (D \wedge S)))$
GF_ROP_DPSonon	0x54	$\sim (D \mid (\sim (P \mid S)))$
GF_ROP_Dn	0x55	$\sim D$
GF_ROP_DPSox	0x56	$D \wedge (P \mid S)$
GF_ROP_DPSoan	0x57	$\sim (D \& (P \mid S))$
GF_ROP_PDSPoax	0x58	$P \wedge (D \& (S \mid P))$

continued...

Name	Rop #	Operation
GF_ROP_DPSnox	0x59	$D \wedge (P \mid (\sim S))$
GF_ROP_DPx	0x5A	$D \wedge P$
GF_ROP_DPSDonox	0x5B	$D \wedge (P \mid (\sim (S \mid D)))$
GF_ROP_DPSDxox	0x5C	$D \wedge (P \mid (S \wedge D))$
GF_ROP_DPSnoan	0x5D	$\sim (D \& (P \mid (\sim S)))$
GF_ROP_DPSDnaox	0x5E	$D \wedge (P \mid (S \& (\sim D)))$
GF_ROP_DPan	0x5F	$\sim (D \& P)$
GF_ROP_PDSxa	0x60	$P \& (D \wedge S)$
GF_ROP_DSPDSaoxxn	0x61	$\sim (D \wedge (S \wedge (P \mid (D \& S))))$
GF_ROP_DSPDoax	0x62	$D \wedge (S \& (P \mid D))$
GF_ROP_SDPnox	0x63	$S \wedge (D \mid (\sim P))$
GF_ROP_SDPSoax	0x64	$S \wedge (D \& (P \mid S))$
GF_ROP_DSPnox	0x65	$D \wedge (S \mid (\sim P))$
GF_ROP_DSx	0x66	$D \wedge S$
GF_ROP_SDPSonox	0x67	$S \wedge (D \mid (\sim (P \mid S)))$
GF_ROP_DSPDSonoxxn	0x68	$\sim (D \wedge (S \wedge (P \mid (\sim (D \mid S)))))$
GF_ROP_PDSxxn	0x69	$\sim (P \wedge (D \wedge S))$
GF_ROP_DPSax	0x6A	$D \wedge (P \& S)$
GF_ROP_PSDPSoaxxn	0x6B	$\sim (P \wedge (S \wedge (D \& (P \mid S))))$
GF_ROP_SDPax	0x6C	$S \wedge (D \& P)$
GF_ROP_PDSPDoaxx	0x6D	$P \wedge (D \wedge (S \& (P \mid D)))$
GF_ROP_SDPSnoax	0x6E	$S \wedge (D \& (P \mid (\sim S)))$
GF_ROP_PDSxnan	0x6F	$\sim (P \& (\sim (D \wedge S)))$

continued...

gf_context_set_rop()

Name	Rop #	Operation
GF_ROP_PDSana	0x70	$P \& (\sim (D \& S))$
GF_ROP_SSDxPDxaxn	0x71	$\sim (S \wedge ((S \wedge D) \& (P \wedge D)))$
GF_ROP_SDPSxox	0x72	$S \wedge (D \mid (P \wedge S))$
GF_ROP_SDPnoan	0x73	$\sim (S \& (D \mid (\sim P)))$
GF_ROP_DSPDxox	0x74	$D \wedge (S \mid (P \wedge D))$
GF_ROP_DSPnoan	0x75	$\sim (D \& (S \mid (\sim P)))$
GF_ROP_SDPSnaox	0x76	$S \wedge (D \mid (P \& (\sim S)))$
GF_ROP_DSan	0x77	$\sim (D \& S)$
GF_ROP_PDSax	0x78	$P \wedge (D \& S)$
GF_ROP_DSPDSoaxxn	0x79	$\sim (D \wedge (S \wedge (P \& (D \mid S))))$
GF_ROP_DPSDnoax	0x7A	$D \wedge (P \& (S \mid (\sim D)))$
GF_ROP_SDPxnan	0x7B	$\sim (S \& (\sim (D \wedge P)))$
GF_ROP_SPDSnoax	0x7C	$S \wedge (P \& (D \mid (\sim S)))$
GF_ROP_DPSxnan	0x7D	$\sim (D \& (\sim (P \wedge S)))$
GF_ROP_SPxDSxo	0x7E	$(S \wedge P) \mid (D \wedge S)$
GF_ROP_DPSaan	0x7F	$\sim (D \& (P \& S))$
GF_ROP_DPSaa	0x80	$D \& (P \& S)$
GF_ROP_SPxDSxon	0x81	$\sim ((P \wedge S) \mid (D \wedge S))$
GF_ROP_DPSxna	0x82	$D \& (\sim (P \wedge S))$
GF_ROP_SPDSnoaxn	0x83	$\sim (S \wedge (P \& (D \mid (\sim S))))$
GF_ROP_SDPxna	0x84	$S \& (\sim (D \wedge P))$
GF_ROP_PDSPnoaxn	0x85	$\sim (P \wedge (D \& (S \mid (\sim P))))$
GF_ROP_DSPDSoaxx	0x86	$D \wedge (S \wedge (P \& (D \mid S)))$

continued...

Name	Rop #	Operation
GF_ROP_PDSaxn	0x87	$\sim (P \wedge (D \& S))$
GF_ROP_DSa	0x88	$D \& S$
GF_ROP_SDPSnaoxn	0x89	$\sim (S \wedge (D \mid (P \& (\sim S))))$
GF_ROP_DSPnoa	0x8A	$D \& (S \mid (\sim P))$
GF_ROP_DSPSxoxn	0x8B	$\sim (D \wedge (S \mid (P \wedge S)))$
GF_ROP_SDPnoa	0x8C	$S \& (D \mid (\sim P))$
GF_ROP_SDPSxoxn	0x8D	$\sim (S \wedge (D \mid (P \wedge S)))$
GF_ROP_SSDxPDxax	0x8E	$S \wedge ((S \wedge D) \& (P \wedge D))$
GF_ROP_PDSanan	0x8F	$\sim (P \& (\sim (D \& S)))$
GF_ROP_PDSxna	0x90	$P \& (\sim (D \wedge S))$
GF_ROP_SDPSnoaxn	0x91	$\sim (S \wedge (D \& (P \mid (\sim S))))$
GF_ROP_DPSPDpoaxx	0x92	$D \wedge (P \wedge (S \& (D \mid P)))$
GF_ROP_SPDaxn	0x93	$\sim (S \wedge (P \& D))$
GF_ROP_PSDPSoaxx	0x94	$P \wedge (S \wedge (D \& (P \mid S)))$
GF_ROP_DPSaxn	0x95	$\sim (D \wedge (P \& S))$
GF_ROP_DPSxx	0x96	$D \wedge (P \wedge S)$
GF_ROP_PSDPSonoxx	0x97	$P \wedge (S \wedge (D \mid (\sim (P \mid S))))$
GF_ROP_SDPSonoxn	0x98	$\sim (S \wedge (D \mid (\sim (P \mid S))))$
GF_ROP_DSxn	0x99	$\sim (D \wedge S)$
GF_ROP_DPSnax	0x9A	$D \wedge (P \& (\sim S))$
GF_ROP_SDPSoaxn	0x9B	$\sim (S \wedge (D \& (P \mid S)))$
GF_ROP_SPDnax	0x9C	$S \wedge (P \& (\sim D))$
GF_ROP_DSPDoaxn	0x9D	$\sim (D \wedge (S \& (P \mid D)))$

continued...

gf_context_set_rop()

Name	Rop #	Operation
GF_ROP_DSPDSaoxx	0x9E	$D \wedge (S \wedge (P \mid (D \& S)))$
GF_ROP_PDSxan	0x9F	$\sim (P \& (D \wedge S))$
GF_ROP_DPa	0xA0	$(D \& P)$
GF_ROP_PDSPnaoxn	0xA1	$\sim (P \wedge (D \mid (S \& (\sim P))))$
GF_ROP_DPSnoa	0xA2	$D \& (P \mid (\sim S))$
GF_ROP_DPSDxoxn	0xA3	$\sim (D \wedge (P \mid (S \wedge D)))$
GF_ROP_PDSPonoxn	0xA4	$\sim (P \wedge (D \mid (\sim (S \mid P))))$
GF_ROP_PDxn	0xA5	$\sim (P \wedge D)$
GF_ROP_DSPnax	0xA6	$D \wedge (S \& (\sim P))$
GF_ROP_PDSPoaxn	0xA7	$\sim (P \wedge (D \& (S \mid P)))$
GF_ROP_DPSoa	0xA8	$D \& (P \mid S)$
GF_ROP_DPSoxn	0xA9	$\sim (D \wedge (P \mid S))$
GF_ROP_D	0xAA	D
GF_ROP_DPSono	0xAB	$D \mid (\sim (P \mid S))$
GF_ROP_SPDSxax	0xAC	$S \wedge (P \& (D \wedge S))$
GF_ROP_DPSDaoxn	0xAD	$\sim (D \wedge (P \mid (S \& D)))$
GF_ROP_DSPnao	0xAE	$D \mid (S \& (\sim P))$
GF_ROP_DPno	0xAF	$D \mid (\sim P)$
GF_ROP_PDSnoa	0xB0	$P \& (D \mid (\sim S))$
GF_ROP_PDSPxoxn	0xB1	$\sim (P \wedge (D \mid (S \wedge P)))$
GF_ROP_SSPxDSxox	0xB2	$S \wedge ((S \wedge P) \mid (D \wedge S))$
GF_ROP_SDPanan	0xB3	$\sim (S \& (\sim (D \& P)))$
GF_ROP_PSDnax	0xB4	$P \wedge (S \& (\sim D))$

continued...

Name	Rop #	Operation
GF_ROP_DPSDoaxn	0xB5	$\sim (D \wedge (P \& (S \mid D)))$
GF_ROP_DPSDPaoxx	0xB6	$D \wedge (P \wedge (S \mid (D \& P)))$
GF_ROP_SDPxan	0xB7	$\sim (S \& (D \wedge P))$
GF_ROP_PSDPxax	0xB8	$P \wedge (S \& (D \wedge P))$
GF_ROP_DSPDaoxn	0xB9	$\sim (D \wedge (S \mid (P \& D)))$
GF_ROP_DPSnao	0xBA	$D \mid (P \& (\sim S))$
GF_ROP_DSno	0xBB	$D \mid (\sim S)$
GF_ROP_SPDSanax	0xBC	$S \wedge (P \& (\sim (D \& S)))$
GF_ROP_SDxPDxan	0xBD	$\sim ((S \wedge D) \& (P \wedge D))$
GF_ROP_DPSxo	0xBE	$D \mid (P \wedge S)$
GF_ROP_DPSano	0xBF	$D \mid (\sim (P \& S))$
GF_ROP_PSa	0xC0	$P \& S$
GF_ROP_SPDSnaoxn	0xC1	$\sim (S \wedge (P \mid (D \& (\sim S))))$
GF_ROP_SPDSonoxn	0xC2	$\sim (S \wedge (P \mid (\sim (D \mid S))))$
GF_ROP_PSexn	0xC3	$\sim (P \wedge S)$
GF_ROP_SPDnoa	0xC4	$S \& (P \mid (\sim D))$
GF_ROP_SPDSxoxn	0xC5	$\sim (S \wedge (P \mid (D \wedge S)))$
GF_ROP_SDPnax	0xC6	$S \wedge (D \& (\sim P))$
GF_ROP_PSDPoaxn	0xC7	$\sim (P \wedge (S \& (D \mid P)))$
GF_ROP_SDPoa	0xC8	$S \& (D \mid P)$
GF_ROP_SPDoxn	0xC9	$\sim (S \wedge (P \mid D))$
GF_ROP_DPSDxax	0xCA	$D \wedge (P \& (S \wedge D))$
GF_ROP_SPDSaoxn	0xCB	$\sim (S \wedge (P \mid (D \& S)))$

continued...

gf_context_set_rop()

Name	Rop #	Operation
GF_ROP_S	0xCC	S
GF_ROP_SDPono	0xCD	$S \mid (\sim(D \mid P))$
GF_ROP_SDPnao	0xCE	$S \mid (D \& (\sim P))$
GF_ROP_SPno	0xCF	$S \mid (\sim P)$
GF_ROP_PSDnoa	0xD0	$P \& (S \mid (\sim D))$
GF_ROP_PSDPxoxn	0xD1	$\sim(P \wedge (S \mid (D \wedge P)))$
GF_ROP_PDSnax	0xD2	$P \wedge (D \& (\sim S))$
GF_ROP_SPDSsoaxn	0xD3	$\sim(S \wedge (P \& (D \mid S)))$
GF_ROP_SSPxPDxax	0xD4	$S \wedge ((S \wedge P) \& (P \wedge D))$
GF_ROP_DPSanan	0xD5	$\sim(D \& (\sim(P \& S)))$
GF_ROP_PSDPSaoxx	0xD6	$P \wedge (S \wedge (D \mid (P \& S)))$
GF_ROP_DPSxan	0xD7	$\sim(D \& (P \wedge S))$
GF_ROP_PDSPxax	0xD8	$P \wedge (D \& (S \wedge P))$
GF_ROP_SDPSaoxn	0xD9	$\sim(S \wedge (D \mid (P \& S)))$
GF_ROP_DPSDanax	0xDA	$D \wedge (P \& (\sim(S \& D)))$
GF_ROP_SPxDSxan	0xDB	$\sim((S \wedge P) \& (D \wedge S))$
GF_ROP_SPDnao	0xDC	$S \mid (P \& (\sim D))$
GF_ROP_SDno	0xDD	$S \mid (\sim D)$
GF_ROP_SDPxo	0xDE	$S \mid (D \wedge P)$
GF_ROP_SDPano	0xDF	$S \mid (\sim(D \& P))$
GF_ROP_PDSoa	0xE0	$P \& (D \mid S)$
GF_ROP_PDSoxn	0xE1	$\sim(P \wedge (D \mid S))$
GF_ROP_DSPDxax	0xE2	$D \wedge (S \& (P \wedge D))$

continued...

Name	Rop #	Operation
GF_ROP_PSDPaoxn	0xE3	$\sim (P \wedge (S \mid (D \& P)))$
GF_ROP_SDPSxax	0xE4	$S \wedge (S \& (P \wedge S))$
GF_ROP_PDSPaoxn	0xE5	$\sim (P \wedge (D \mid (S \& P)))$
GF_ROP_SDPSanax	0xE6	$S \wedge (D \& (\sim (P \& S)))$
GF_ROP_SPxDPxan	0xE7	$\sim ((S \wedge P) \& (D \wedge P))$
GF_ROP_SSPxDSxax	0xE8	$S \wedge ((S \wedge P) \& (D \wedge S))$
GF_ROP_DSPDSanaxxn	0xE9	$\sim (D \wedge (S \wedge (P \& (\sim (D \& S)))))$
GF_ROP_DPSao	0xEA	$D \mid (P \& S)$
GF_ROP_DPSxno	0xEB	$D \mid (\sim (P \wedge S))$
GF_ROP_SDPao	0xEC	$S \mid (D \& P)$
GF_ROP_SDPxno	0xED	$S \mid (\sim (D \wedge P))$
GF_ROP_DSso	0xEE	$D \mid S$
GF_ROP_SDPnoo	0xEF	$S \mid (D \mid (\sim P))$
GF_ROP_P	0xF0	P
GF_ROP_PDSono	0xF1	$P \mid (\sim (D \mid S))$
GF_ROP_PDSnao	0xF2	$P \mid (D \& (\sim S))$
GF_ROP_PSno	0xF3	$P \mid (\sim S)$
GF_ROP_PSDnao	0xF4	$P \mid (S \& (\sim D))$
GF_ROP_PDno	0xF5	$P \mid (\sim D)$
GF_ROP_PDSxo	0xF6	$P \mid (D \wedge S)$
GF_ROP_PDSano	0xF7	$P \mid (\sim (D \& S))$
GF_ROP_PDSao	0xF8	$P \mid (D \& S)$
GF_ROP_PDSxno	0xF9	$P \mid (\sim (D \wedge S))$

continued...

gf_context_set_rop()

Name	Rop #	Operation
GF_ROP_DPo	0xFA	D P
GF_ROP_DPSnoo	0xFB	D (P (~ S))
GF_ROP_PSo	0xFC	P S
GF_ROP_PSDnoo	0xFD	P (S (~ D))
GF_ROP_DPSoo	0xFE	D (P S)
GF_ROP_ONE	0xFF	1

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_disable_rop(),

gf_context_set_surface()

Associate a surface with the draw context

Synopsis:

```
int gf_context_set_surface( gf_context_t context,
                           gf_surface_t surface );
```

Arguments:

<i>context</i>	The draw context to associate a surface with.
<i>surface</i>	The handle for the surface to target. This handle is returned by <i>gf_surface_create()</i> .

Library:

gf

Description:

This function associates a surface for the draw context (or if there already is a surface associated with the context, this function allows you to target a new one). Any subsequent rendering that is done via this context affects the newly targeted surface.



While it is acceptable to call any other *gf_context_set_**() function while you are rendering (between calls to *gf_draw_begin()* and *gf_draw_end()*), calling *gf_context_set_surface()* during this time has no effect.

Returns:

0	Success.
-1	An error occurred (<i>errno</i> is set).

gf_context_set_surface()

Errors:

EBUSY	The context is currently rendering and cannot switch surfaces (call <i>gf_draw_end()</i> first).
ENOMEM	Could not allocate sufficient space for working memory.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_create(), *gf_context_free()*, *gf_surface_create()*

gf_context_set_transform()

Set the current 2D transform matrix

Synopsis:

```
void gf_context_set_transform(  
    gf_context_t context,  
    const gf_fixed_t *xform_matrix );
```

Arguments:

<i>context</i>	The draw context to set the transform for.
<i>xform_matrix</i>	Pointer to an array of 4 gf_fixed_t (signed 16.16 fixed point) elements composing the 2x2 matrix. Don't pass NULL for this argument.

Library:

gf

Description:

This function sets the current 2D transform matrix. This 2x2 matrix is multiplied by all polygon, line, and polyline co-ordinates prior to rendering. Using a combination of the transform matrix and translations, various rotation, scaling and reflection operations are possible. See *gf_context_set_translation()* for information on translations.



The 2D transform matrix applies only to geometrical primitives such as polygons, lines, and polylines. The algorithm is *not* applied to spans, rectangles, bitmaps, fonts, etcetera.

The **gf_fixed_t** type is a 32 bit number representing signed 16.16 fixed point values used in transformation matrices. The composition of this type is native endian with the low 16 bits representing the fractional component, the high 16 bits representing the integral component, in two's complement notation.

gf_context_set_transform()

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_disable_transform(), *gf_context_set_translation()*

gf_context_set_translation()

Set the current translation values

Synopsis:

```
void gf_context_set_translation( gf_context_t context,  
                                int tx,  
                                int ty );
```

Arguments:

context The draw context to set the translation for.
tx, ty The horizontal and vertical translation.

Library:

gf

Description:

This function sets the current x and y translation values. These values are added to all polygon, line, and polyline co-ordinates prior to rendering. Using a combination of the transform matrix and translations, various rotation, scaling and reflection operations are possible. For more information on transforms, see *gf_context_set_transform()*.



The translations apply only to geometrical primitives such as polygons, lines, and polylines. The algorithm is NOT applied to spans, rectangles, bitmaps, text etc.

@@@ Need a rotation example here @@@

Classification:

Graphics Framework

gf_context_set_translation()

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_context_disable_translation()

gf_cursor_disable()

Disable a hardware cursor

Synopsis:

```
int gf_cursor_disable( gf_display_t display );
```

Arguments:

display The display you want to disable the hardware cursor for.

Library:

gf

Description:

This function disables the hardware cursor for a display. You can enable the cursor using *gf_cursor_enable()*.

Returns:

0 Success.

-1 An error occurred communicating with **io-display**.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_cursor_disable()

See also:

gf_cursor_enable(), *gf_cursor_set()*, *gf_cursor_set_pos()*

gf_cursor_enable()

Enable a hardware cursor

Synopsis:

```
int gf_cursor_enable( gf_display_t display );
```

Arguments:

display The display you want to enable the hardware cursor for.

Library:

gf

Description:

This function enables the hardware cursor for a display. You need to set the cursor first, using *gf_cursor_set()*. You can disable the cursor using *gf_cursor_disable()*.

Returns:

0 Success.

-1 An error occurred communicating with **io-display**.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_cursor_enable()

See also:

gf_cursor_disable(), *gf_cursor_set()*, *gf_cursor_set_pos()*

gf_cursor_set()

Set the current hardware cursor bitmap definition

Synopsis:

```
int gf_cursor_set( gf_display_t display,
                  const gf_cursor_t *cursor );
```

Arguments:

<i>display</i>	The display you want to set the hardware cursor for.
<i>cursor</i>	A pointer to a gf_cursor_t that describes the cursor to set (see below).

Library:

gf

Description:

This function replaces the hardware cursor bitmap definition for a display.



By default, the hardware cursor is disabled. If the cursor is not already enabled on the display, you must explicitly enable it using *gf_cursor_enable()*.

The **gf_cursor_t** structure defines a cursor:

```
typedef struct {

    enum {
        GF_CURSOR_BITMAP,
    }
    type;
    gf_point_t hotspot;
    union {
        struct {
            int w;
            int h;
            int stride;
            gf_color_t color0;
        }
    }
};
```

gf_cursor_set()

```
        const _uint8 *image0;
        const _uint8 *image1;
        gf_color_t color1;
    } bitmap;
} cursor;
} gf_cursor_t;
```

It has at least these members:

<i>type</i>	The type of the cursor. The value of this field dictates which member of the <code>gf_cursor_t::cursor</code> union is valid. Only bitmap cursors are currently supported so you must always set this value to <code>GF_CURSOR_BITMAP</code> .								
<i>hotspot</i>	Offset into the cursor top left corner to consider as the actual pointer position. See <code>gf_point_t</code> .								
<i>cursor</i>	A union of type-dependent data describing the cursor entity. Since only bitmap cursors are supported, this union has one possible value, a struct that describes a cursor as two planar bitmaps: <table><tr><td><i>w, h</i></td><td>Width and height of the bitmaps, in pixels. The same values are assumed for both bitmaps.</td></tr><tr><td><i>stride</i></td><td>Bytes per line for the bitmaps. This includes the bytes required to represent a single line as well as any padding imposed by the application. The same stride is assumed for both bitmaps).</td></tr><tr><td><i>color0</i></td><td>The color for the first bitmap.</td></tr><tr><td><i>image0</i></td><td>A pointer to the bitmap data for the first plane. The data represents a $w \times h$, single bit per pixel bitmap, packing 8 pixels per byte. There should be h contiguous lines of $stride$ bytes each. On bits (1) are rendered in the color specified by <i>color0</i>, and off bits (0) are treated as transparent.</td></tr></table>	<i>w, h</i>	Width and height of the bitmaps, in pixels. The same values are assumed for both bitmaps.	<i>stride</i>	Bytes per line for the bitmaps. This includes the bytes required to represent a single line as well as any padding imposed by the application. The same stride is assumed for both bitmaps).	<i>color0</i>	The color for the first bitmap.	<i>image0</i>	A pointer to the bitmap data for the first plane. The data represents a $w \times h$, single bit per pixel bitmap, packing 8 pixels per byte. There should be h contiguous lines of $stride$ bytes each. On bits (1) are rendered in the color specified by <i>color0</i> , and off bits (0) are treated as transparent.
<i>w, h</i>	Width and height of the bitmaps, in pixels. The same values are assumed for both bitmaps.								
<i>stride</i>	Bytes per line for the bitmaps. This includes the bytes required to represent a single line as well as any padding imposed by the application. The same stride is assumed for both bitmaps).								
<i>color0</i>	The color for the first bitmap.								
<i>image0</i>	A pointer to the bitmap data for the first plane. The data represents a $w \times h$, single bit per pixel bitmap, packing 8 pixels per byte. There should be h contiguous lines of $stride$ bytes each. On bits (1) are rendered in the color specified by <i>color0</i> , and off bits (0) are treated as transparent.								

image1 Pointer to the bitmap data for the second plane, which will be laid over the first. This image is treated identically to *image0*, except color is specified by *color1*.

color1 The color for the second bitmap.

Returns:

0 Success.

-1 An error occurred (*errno* is set).

Errors:

ENOTSUP The cursor format isn't supported by the hardware.

Any other error

There was a problem communicating with **io-display**.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_cursor_disable(), *gf_cursor_enable()*, *gf_cursor_set_pos()*

gf_point_t

gf_cursor_set_pos()

Set the hardware cursor position

Synopsis:

```
int gf_cursor_set_pos( gf_display_t display,
                      int x,
                      int y );
```

Arguments:

<i>display</i>	The display you want to set the hardware cursor position for.
<i>x, y</i>	Coordinates that describe the new hardware cursor position.

Library:

gf

Description:

This function sets the position of the hardware cursor for a *display*.

Returns:

0	Success.
-1	An error occurred communicating with io-display .

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_cursor_set_pos()

See also:

gf_cursor_disable(), *gf_cursor_enable()*, *gf_cursor_set()*

gf_dev_attach()

Connect to a graphics device

Synopsis:

```
int gf_dev_attach( gf_dev_t * pdev,
                  const char *name,
                  gf_dev_info_t *info );
```

Arguments:

- | | |
|-------------|--|
| <i>pdev</i> | Address for a gf_dev_t where the function stores a handle for the graphics device. |
| <i>name</i> | The name of the device to which to attach. This could be a string matching an entry in /dev/io-display , or it could be the manifest GF_DEVICE_INDEX(n) to specify the <i>n</i> th device in the /dev/io-display directory, where <i>n</i> can be from 0 to 64. Normally you would simply specify GF_DEVICE_INDEX(0) to connect to the first device found. |
| <i>info</i> | A gf_dev_info_t structure filled in with information about the graphics device you want to attach to (see below). |

Library:

gf

Description:

This function initializes and connects to a graphics device. It provides you with a handle to facilitate subsequent communication with the display hardware in a manner that promotes thread safety. Hardware parameters are configured by a separate configuration file and maintained by a separate server, **io-display**. @@@ add link here @@@



The display server, **io-display**, must be running before you call this function.

The *info* argument contains information about the attached device:


```
typedef struct {
    unsigned ndisplays;
    unsigned flags;
} gf_dev_info_t;
```

It contains at least the following members:

<i>ndisplays</i>	The number of displays the device supports, which is never less than 1.
<i>flags</i>	Device capability flags. This member is currently unused and set to 0.

Returns:

0	Success.
-1	An error occurred.

Errors:

ENXIO	The device with specified GF_DEVICE_INDEX() doesn't exist.
ENOMEM	Insufficient memory to allocate structures.
Other	See the error codes for <i>opendir()</i> .

Examples:

```
gf_dev_t gdev;
gf_dev_info_t gdev_info;

if (gf_dev_attach(&gdev, GF_DEVICE_INDEX(0), &gdev_info) == -1) {
    printf("gf_dev_attach() failed\n");
    return (-1);
}
```

gf_dev_attach()

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_dev_detach()

gf_dev_detach()

Disconnect from a graphics device

Synopsis:

```
void gf_dev_detach( gf_dev_t gdev );
```

Arguments:

gdev The handle for the graphics device to detach from. This is the handle acquired by *gf_dev_attach()*.

Library:

gf

Description:

This function disconnects from a graphics device, releasing all associated resources and shutting down the hardware if no other clients are connected to it.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_dev_attach()

gf_dev_register_thread()

Register a device with a thread

Synopsis:

```
int gf_dev_register_thread( gf_dev_t gdev );
```

Arguments:

gdev The handle for the graphics device to register. This is the handle acquired by *gf_dev_attach()*.

Library:

gf

Description:

This function should be called by threads that wish to register and use an already initialized graphics device. In a multi-threaded application, one thread can call *gf_dev_attach()* to attach to the graphics device, and other threads can call *gf_dev_register_thread()* to use the same device.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_dev_attach()

Synopsis:

```
typedef struct {  
    uint16_t w;  
    uint16_t h;  
} gf_dim_t;
```

Description:

The **gf_dim_t** structure defines the dimensions of an area. It contains at least the following members:

w Width of the area.
h Height of the area.

Classification:

Graphics Framework

See also:

gf_cursor_set(), *gf_draw_bitmap()*

gf_display_attach()

Attach to a display on a graphics device

Synopsis:

```
int gf_display_attach( gf_display_t * pdisplay,
                      gf_dev_t gfx,
                      int display_index,
                      gf_display_info_t *info );
```

Arguments:

<i>pdisplay</i>	A pointer to a gf_display_t where the function stores a handle for a graphics display.
<i>gfx</i>	A handle for the graphics device, acquired by <i>gf_dev_attach()</i> .
<i>display_index</i>	The index of the desired display. A graphics device can have one or more displays, with the index starting at 0.
<i>info</i>	A pointer to a gf_display_info_t structure which the function fills in with information about the display (see below). Set to NULL if you don't need to obtain this information.

Library:

gf

Description:

This function attaches to a display on a graphics device. This function provides you with a handle which allows you to use the display while maintaining thread safety. Actual parameters for the individual display (or displays) on a graphics device are configured via a separate configuration file and maintained by a separate server, **io-display**.

A device typically drives one display, although some hardware is fitted with multiple displays, each of which can be attached to via this

function. With a separate handle for each display, each can be addressed and manipulated on an individual basis.

The *info* argument contains information about the attached display:

```
typedef struct
{
    unsigned nlayers;
    unsigned main_layer_index;
    _uint16 xres;
    _uint16 yres;
    gf_layer_format_t layer_format;
    int refresh;
} gf_display_info_t;
```

It contains at least the following members:

nlayers The number of layers the display supports. There's always at least one layer present (the main display layer), although some hardware has support for multiple layers per display.

main_layer_index The index of the main display layer. Every display has the concept of a main display layer, which encapsulates the frame buffer for that display. Generally, control of the main display layer is limited in comparison to other layers. This zero-based index allows you to identify the main display layer.

xres, yres The horizontal and vertical resolution, in pixels.

layer_format The main layer's format.
Layer formats should not be confused with pixel formats. Pixel formats provide detailed information about the formatting of pixel data in a buffer, while layer formats are derived from the way in which layers are implemented. Typically, drivers separate

gf_display_attach()

layer capabilities into distinct formats (analogous to the different video modes a display can support). The layer format type lets you easily identify which of these formats you're interested in.

For a list of supported layer formats, see *gf_surface_create_layer()*.

refresh The current refresh rate of the display (in Hz).

Returns:

- 0 Success.
- 1 An error occurred (*errno* is set).

Errors:

- ENOMEM Insufficient memory to allocate structures.
- Any other error
 There was a problem communicating with **io-display**.

Examples:

In this example, we attach to a device, and then use the device information structure **gf_dev_info_t** to attach to the device's displays:

```
gf_dev_t      gdev;
gf_dev_info_t gdev_info;
gf_display_t   display;
gf_display_info_t display_info;

int i;

if (gf_dev_attach(&gdev, GF_DEVICE_INDEX(0), &gdev_info) == -1) {
    printf("gf_dev_attach() failed\n");
    return (-1);
}
```



```
printf("Number of displays: %d\n",gdev_info.ndisplays);

for (i = gdev_info.ndisplays;i;) {
    printf("Display %d: ",i--);
    if (gf_display_attach(&display,gdev,i,&display_info) != -1) {
        printf("dX%d, refresh = %dHz\n",display_info.xres,display_info.yres,displa
        printf("Number of layers: %d\n",display_info.nlayers);
    } else {
        printf("gf_display_attach() failed\n");
    }
}
```

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_display_detach(), *gf_display_set_layer_order()*

gf_display_detach()

Detach from a display

Synopsis:

```
void gf_display_detach( gf_display_t display );
```

Arguments:

display The handle for the display to detach from. This handle is returned by *gf_display_attach()*.

Library:

gf

Description:

This function detaches from an attached display (a display attached using a call to *gf_display_attach()*), releasing any associated resources.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_display_attach()

gf_display_set_layer_order()

Set the ordering of the layers for a display

Synopsis:

```
int gf_display_set_layer_order( gf_display_t display,
                               const unsigned order[],
                               unsigned flags );
```

Arguments:

- | | |
|----------------|---|
| <i>display</i> | The handle for the display to set the layer order for. This handle is returned by <i>gf_display_attach()</i> . |
| <i>order</i> | An array of type unsigned specifying the new order of the layers. For each layer of the display, the value at the corresponding index in the array specifies the new z-position for that layer. Therefore, there must be the correct number of elements in the array, which is the number of available layers for that display. If there aren't enough elements in the array, the results aren't defined. |
| <i>flags</i> | Flags controlling additional options. Valid flags are: <ul style="list-style-type: none">• GF_LAYER_UPDATE_NO_WAIT_VSYNC — Perform the operation asynchronously; the default behaviour is to block until the next vertical synchronization before changing the layer order.• GF_LAYER_UPDATE_NO_WAIT_IDLE — Perform the operation immediately; the default behaviour is to wait for the draw hardware to finish before changing the layer order. |

Library:

gf

Description:

This function set the ordering of the layers for a display. Each display is always equipped with at least one layer (known as the main display layer), but some hardware is equipped with additional layers which

gf_display_set_layer_order()

can provide advanced visual effects without incurring additional CPU overhead.

Layers are stacked and therefore have an implicit z-order. Some hardware allows the reordering of these layers. The default ordering is numerical by layer index, with layer 0 being the furthest away from the user.

For example, to reverse the default order of the layers on a 2-layer display, you would specify an order array of { 1, 0 }.

Returns:

- 0 Success.
- 1 An error occurred (*errno* is set).

Errors:

- ENOTSUP Layer ordering isn't supported on the specified display.
- EINVAL Invalid z-order for a layer or duplicate entries in the array (two layers cannot be ordered in the same position). Some layers have z-position restrictions; check the *order_caps* member of the **gf_layer_info_t** filled in by *gf_layer_attach()*.
- Any other error
 There was a problem communicating with **io-display**.

Classification:

Graphics Framework

Safety

- Interrupt handler No
- Signal handler No

continued...

gf_display_set_layer_order()

Safety

Thread	Yes
--------	-----

See also:

gf_display_attach(), *gf_display_detach()*

gf_draw_begin()

Begin rendering

Synopsis:

```
int gf_draw_begin( gf_context_t context );
```

Arguments:

context The graphics context handle to render to.

Library:

gf

Description:

This function begins rendering using the given draw *context*. It ensures that the calling thread has exclusive access to the hardware, and prepares the hardware to receive draw commands. It also ensures that all the current context settings are applied. Once you no longer require exclusive access to the hardware, you should call *gf_draw_end()*.



CAUTION: Although a common address space makes it easy to share contexts across threads of the same process, you should exercise caution when doing so by ensuring that each thread calls *gf_draw_begin()* before it begins rendering using the shared context. Without following this practice, the two threads may hit the hardware simultaneously, which could leave the hardware in an undefined state, causing your application to hang.

Because this function locks the hardware for exclusive access by the calling thread, you should follow these practices to ensure fairness to other threads waiting to render, and to avoid potential deadlock:

- call *gf_draw_begin()* only when your thread is ready to begin rendering
- call *gf_draw_end()* as soon as your thread is done rendering

- avoid calling GF functions that are not directly related to rendering between *gf_draw_begin()* and *gf_draw_end()*. This means you should only call functions beginning with *gf_draw* or *gf_context* (with the exception of *gf_context_set_surface()* which cannot be called while you are rendering).



In order to render, there must be a surface associated with the given context (using *gf_context_set_surface()*). If the context has no associated surface, the function fails.

You need to call *gf_draw_begin()* each time you begin drawing using a different context.

Returns:

- | | |
|----|---|
| 0 | Success. The hardware is locked, call <i>gf_draw_end()</i> when done. |
| -1 | An error occurred. The hardware is <i>not</i> locked. |

Errors:

- | | |
|---------|--|
| EINVAL | The surface cannot be rendered to, either because no surface is currently targeted, or because the surface has a format that is not targetable by the 2D engine. |
| EAGAIN | Insufficient resource to lock the hardware mutex. |
| EDEADLK | The calling thread has already successfully called <i>gf_draw_begin()</i> ; recursive behaviour is not permitted. |

Classification:

Graphics Framework

gf_draw_begin()

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_draw_end(), *gf_context_set_surface()*

gf_draw_bitmap()

Draw a bitmap

Synopsis:

```
void gf_draw_bitmap( gf_context_t context,
                    const uint8_t *image,
                    int stride,
                    int bit0_offset,
                    int x,
                    int y,
                    int w,
                    int h,
                    unsigned flags );
```

Arguments:

<i>context</i>	Handle for the draw context to use.
<i>image</i>	A buffer that contains the bitmap data.
<i>stride</i>	The width of a line of pixel data, in bytes.
<i>bit0_offset</i>	Specifies an index into the first byte of the source bitmap. For each scanline of the bitmap that's drawn, this index specifies the bit within the first byte of the scanline's source data that corresponds to the first pixel of the scanline that's drawn.
<i>x, y</i>	The coordinates of the pixel offset within the draw surface where the bitmap is drawn.
<i>w, h</i>	The width and height of the bitmap, in pixels.
<i>flags</i>	Flags affecting rendering. One flag is defined: <ul style="list-style-type: none">• GF_DRAW_BITMAP_BACKFILL — render low bits (0) as background color; default is to treat low bits as transparent.

gf_draw_bitmap()

Library:

gf

Description:

This function draws a bitmap using given draw *context* at the position *x*, *y*. Bitmaps are simple, one bit/pixel images that are described by a series of bytes, each packing 8 pixels per byte. There may be padding bytes between successive lines. These padding bytes are accounted for in the *stride*. The bitmap is using the current foreground color as the “on” color (high bits) and the current background color as the “off” color (low bits — if the transparent member in the bitmap structure isn’t set).

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_draw_blit1(), *gf_draw_blit2()*, *gf_draw_blitscaled()*,

gf_draw_blit1()

Blit an area within a surface

Synopsis:

```
void gf_draw_blit1( gf_context_t context,
                    int sx1,
                    int sy1,
                    int sx2,
                    int sy2,
                    int dx,
                    int dy );
```

Arguments:

<i>context</i>	Handle for the draw context to use. The targeted surface for this context is used as the source and destination surface.
<i>sx1, sy1</i>	The coordinates of the upper-left corner of the source area to blit from.
<i>sx2, sy2</i>	The coordinates of the lower-right corner of the source area to blit from.
<i>dx, dy</i>	The coordinates of the offset point on the display to blit the area to.

Library:

gf

Description:

This function performs a simple BLock Image Transfer (BLIT, or pixel copy) from one area to another area of a surface associated with the given *context*. The source and destination areas may overlap.

gf_draw_blit1()



Clipping doesn't apply to blits, although blits are clipped by the boundary of the target surface.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_draw_blit2(), *gf_draw_blitscaled()*

gf_draw_blit2()

Blit an area from one surface to another

Synopsis:

```
void gf_draw_blit2( gf_context_t context,
                    gf_surface_t ssurface,
                    gf_surface_t dsurface,
                    int sx1,
                    int sy1,
                    int sx2,
                    int sy2,
                    int dx,
                    int dy );
```

Arguments:

<i>context</i>	The handle for the draw context to blit.
<i>ssurface</i>	The handle of the surface containing the source pixels. If NULL, the context's currently targeted surface is used.
<i>dsurface</i>	The handle of the destination surface to copy to. If NULL, the context's currently targeted surface is used.
<i>sx1, sy1</i>	The coordinates of the upper-left corner of the source area to blit from.
<i>sx2, sy2</i>	The coordinates of the lower-right corner of the source area to blit from.
<i>dx, dy</i>	The coordinates of the offset point on the display to blit the area to.

Library:

gf

Description:

This function copies the area defined by *src*, from the source surface to the same sized area in the destination surface, offset by *dst*. If the

gf_draw_blit2()

source and destination surfaces are the same, overlapping blits are supported.



Clipping does not apply to blits, although blits are always clipped by the boundary of the target surface. @@@ Make sure this is true @@@

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_draw_blit1(), *gf_draw_blitscaled()*

gf_draw_blitscaled()

Perform a scaled blit from one surface to another

Synopsis:

```
void gf_draw_blitscaled( gf_context_t context,
                        gf_surface_t  ssurface,
                        gf_surface_t  dsurface,
                        int  sx1,
                        int  sy1,
                        int  sx2,
                        int  sy2,
                        int  dx1,
                        int  dy1,
                        int  dx2,
                        int  dy2 );
```

Arguments:

<i>context</i>	The handle for draw context.
<i>ssurface</i>	The surface to copy from (source surface).If NULL, the context's currently targeted surface is used.
<i>dsurface</i>	The surface to copy to (destination surface).If NULL, the context's currently targeted surface is used.
<i>sx1, sy1, sx2, sy2</i>	The coordinates of the source area to blit from. The coordinates <i>sx1,sy1</i> is the upper-left corner, and <i>sx2,sy2</i> is the lower left.
<i>dx1, dy1, dx2, dy2</i>	The coordinates of the destination area to blit to. This area may be larger or smaller than the source area — the blitted pixels are scaled accordingly.

Library:

gf

gf_draw_blitscaled()

Description:

This function copies the rectangle defined by *src*, from a source surface to a destination surface, scaling the source rectangle as necessary to fit the destination.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_draw_blit1(), *gf_draw_blit2()*,

gf_draw_end()

Finish rendering

Synopsis:

```
void gf_draw_end( gf_context_t context );
```

Arguments:

context The draw context handle.

Library:

gf

Description:

This function ends a gf draw operation for the given *context*. All *gf_draw_**() functions must be preceded by a call to *gf_draw_begin()* and followed by *gf_draw_end()*.

This function finishes rendering with the specified *context*. It relinquishes access to the rendering hardware, permitting other threads to acquire it. As with all the other drawing functions, you should only call this function following a successful call to *gf_draw_begin()*. Conversely, it's very important to balance each successful call to *gf_draw_begin()* with a corresponding call to *gf_draw_end()*, otherwise no other threads are able to render and your thread could eventually end up deadlocking itself.



Because a successful call to *gf_draw_begin()* locks the hardware for exclusive access by the calling thread, you should call *gf_draw_end()* as soon as you are done rendering, to ensure hardware access by other threads or applications that are ready to render. Don't confuse this function with *gf_draw_finish()*.

Classification:

Graphics Framework

gf_draw_end()

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_draw_begin(), *gf_draw_finish()*

gf_draw_finish()

Wait for the rendering hardware to finish

Synopsis:

```
void gf_draw_finish( gf_context_t context );
```

Arguments:

context The handle for the draw context to wait on.

Library:

gf

Description:

This function waits until all draw operations have actually been carried out. It's blocking and will not return until the hardware becomes idle.

This function flushes out buffered commands prior to waiting for them to be processed. Therefore, you do not need to call *gf_draw_flush()* explicitly prior to calling *gf_draw_finish()*.



Don't confuse this function with *gf_draw_end()*. This function must be called between *gf_draw_begin()* and *gf_draw_end()*.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_draw_finish()

See also:

gf_draw_begin(), *gf_draw_end()*,

gf_draw_flush()

Flush the draw buffer

Synopsis:

```
void gf_draw_flush( gf_context_t context );
```

Arguments:

context The handle of the draw context to flush.

Library:

gf

Description:

This function flushes the draw buffer to the surface. Some graphics hardware puts draw events into a queue that must be explicitly flushed before the draw events are rendered.

Call this function when you need to know that all previous draw commands have been rendered, for example before you wait for user input based on a rendered image. You do not normally need to call this function, and should call it only when required.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_draw_flush()

See also:

gf_draw_rect(), *gf_draw_poly_fill()*, *gf_draw_polyline()*,
gf_draw_blit1(), *gf_draw_blit2()*, *gf_draw_blitscaled()*,
gf_draw_bitmap(), *gf_draw_string()*

gf_draw_poly_fill()

Draw a filled polygon

Synopsis:

```
void gf_draw_poly_fill( gf_context_t context,
                        gf_point_t *pts,
                        size_t npoints );
```

Arguments:

<i>context</i>	The graphics context to use.
<i>pts</i>	An array of gf_point_t structures that represent the vertices of the polygon.
<i>npoints</i>	The number of items in <i>pts</i> .

Library:

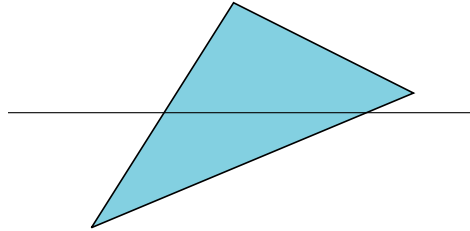
gf

Description:

This function draws a filled polygon using the given rendering *context*. The filled polygon is rendered using the context's current foreground color.

Polygons that overlap themselves are filled using the so-called *even-odd* rule: if an area overlaps an odd number of times, it isn't filled. Another way of looking at this is to draw a horizontal line across the polygon. As you travel along this line and cross the first line, you're inside the polygon; as you cross the second line, you're outside. As an example, consider a simple polygon:

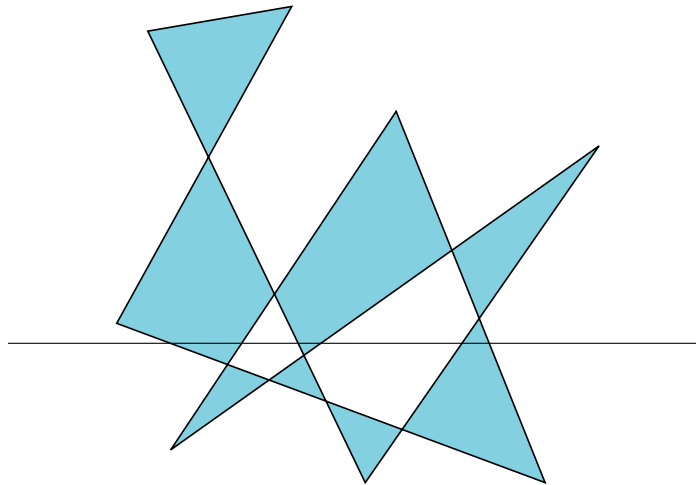
gf_draw_poly_fill()



Filling a simple polygon.

This rule can be extended for more complicated polygons:

- When you cross an odd number of lines, you're inside the polygon, so the area is filled.
- When you cross an even number of lines, you're outside the polygon, so the area isn't filled.



Filling an overlapping polygon.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_draw_rect(), *gf_draw_polyline()*, *gf_draw_blit1()*, *gf_draw_blit2()*,
gf_draw_blitscaled(), *gf_draw_flush()*, *gf_draw_bitmap()*,
gf_draw_string(), **gf_point_t**

gf_draw_polyline()

Draw an unfilled polygon

Synopsis:

```
void gf_draw_polyline( gf_context_t context,
                       gf_point_t *pts,
                       size_t npoints,
                       uint32_t flags );
```

Arguments:

<i>context</i>	The graphics context to use.
<i>pts</i>	An array of gf_point_t structures that represent the vertices of the polygon.
<i>npoints</i>	The number of items in <i>pts</i> .
<i>flags</i>	Flags controlling the functions behavior. Currently only one flag is defined: GF_DRAW_POLYLINE_CLOSED — set this bit to join the last point in the polygon to the first.

Library:

gf

Description:

This function draws a polyline, using the specified draw context. The polyline is rendered using the current foreground color. You can also use this function to render a single line or a stroked outline rectangle.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_draw_polyline()

See also:

gf_draw_poly_fill(), *gf_context_set_linedash()*, *gf_context_set_linejoin()*,
gf_point_t

gf_draw_rect()

Draw a filled rectangle

Synopsis:

```
void gf_draw_rect( gf_context_t context,
                  int x1,
                  int y1,
                  int x2,
                  int y2 );
```

Arguments:

<i>context</i>	The handle for the draw context.
<i>x1, y1</i>	The coordinates of the upper-left corner of the rectangle.
<i>x2, y2</i>	The coordinates of the lower-right corner of the rectangle.

Library:

gf

Description:

This function draws a filled rectangle using the given rendering *context*. The filled rectangle is rendered using the context's current foreground color.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

*gf_draw_poly_fill(), gf_draw_polyline(), gf_draw_blit1(),
gf_draw_blit2(), gf_draw_blitscaled(), gf_draw_flush(),
gf_draw_bitmap(), gf_draw_string()*

gf_draw_string()

Render a text string

Synopsis:

```
int gf_draw_string( gf_context_t context,
                    gf_font_t font,
                    const char *fontname,
                    const char *string,
                    int string_len,
                    const gf_point_t *pos,
                    uint32_t flags );
```

Arguments:

<i>context</i>	A handle for the draw context.
<i>font</i>	A handle for the font manager, returned by <i>gf_font_attach()</i> .
<i>fontname</i>	A font name generated by a call to <i>PfGenerateFontName*</i> ().
<i>string</i>	A pointer to a NULL-terminated UTF8 string you want to render.
<i>string_len</i>	The number of characters (bytes) in <i>string</i> . If you set to 0, the length is calculated for you.
<i>pos</i>	A pointer to a gf_point_t structure which describes the position of the rendered text.
<i>flags</i>	Rendering flags. There is one bit you can set: GF_DRAW_STRING_BACKFILL . If this bit is set, the areas within the bounding box of the text that aren't text are filled with the current background color. Otherwise these areas are unmodified.

Library:

gf

Description:

This function renders a text string with the given *context*. The string is rendered using the current foreground color, and with the current background colour if the GF_DRAW_STRING_BACKFILL *flags* bit is set.



You need to attach to the font manager by calling *gf_font_attach()* before calling this function.

Returns:

- 0 Success.
- 1 An error occurred.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_draw_rect(), *gf_draw_poly_fill()*, *gf_draw_polyline()*,
gf_draw_blit1(), *gf_draw_blit2()*, *gf_draw_blitscaled()*, *gf_draw_flush()*,
gf_draw_bitmap(), *gf_font_attach()*, *gf_font_detach()*, **gf_point_t**

gf_font_attach()

Attach to the font manager

Synopsis:

```
int gf_font_attach( gf_font_t * pfont,  
                   gf_dev_t  gfx );
```

Arguments:

<i>pfont</i>	The address for a gf_font_t where the function can store the handle for the font manager.
<i>gfx</i>	A handle for the graphics device to associate the font manager with. This is the handle acquired by <i>gf_dev_attach()</i> .

Library:

gf

Description:

This function initializes and attaches to the font manager. You need to attach to the font manager to render text using *gf_draw_string()*. This function loads the font engine **phfont.so** and its associated libraries. @@@ Need more information here about where phfont.so / libfont needs to be located, etc @@@

Returns:

0	Success.
-1	An error occurred.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_draw_string(), *gf_font_detach()*

gf_font_detach()

Detach from the font manager

Synopsis:

```
void gf_font_detach( gf_font_t font );
```

Arguments:

font A handle for the font manager to detach from. This is the handle returned from *gf_font_attach()*.

Library:

gf

Description:

This function detaches from the font manager and unloads the associated font libraries.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_draw_string(), *gf_font_attach()*

gf_layer_attach()

Attach to a layer

Synopsis:

```
int gf_layer_attach( gf_layer_t * player,
                    gf_display_t display,
                    unsigned layer_index,
                    unsigned flags );
```

Arguments:

<i>player</i>	A pointer to a gf_layer_t where the function can store the handle for the layer.
<i>display</i>	A handle for the display you want the layer for, returned by <i>gf_display_attach()</i> .
<i>layer_index</i>	The index of the layer to attach to. Layers are indexed numerically, starting at 0. A display always has at least one layer.
<i>flags</i>	Flags to control behavior. Currently one flag is defined: <ul style="list-style-type: none">• GF_LAYER_ATTACH_NODEFAULTS — attach only; do not reset layer parameters to initial values.

Library:

gf

Description:

This function attaches a GF application to a layer within the specified *display*. This function provides you with a handle, *player*, which you require to manipulate the layer.

If you pass 0 for *flags*, some layer settings are set to their defaults:

- the layer is enabled

gf_layer_attach()

- the source viewport is set to entire surface
- the destination viewport is set to entire display



If you call *gf_layer_attach()* for an already attached layer index, set the `GF_LAYER_ATTACH_NODEFAULTS` flag bit to prevent any layer settings being reset to their defaults. If you attach to a layer that's disabled with this flag bit set, the layer remains disabled and you'll need to explicitly enable it with *gf_layer_enable()*.

To get information about an attached layer's capabilities, call *gf_layer_query()*.

Returns:

- 0 Success.
- 1 An error occurred.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_display_attach(), *gf_layer_detach()*, *gf_layer_disable()*,
gf_layer_enable(), *gf_layer_query()*

gf_layer_detach()

Detach from a layer

Synopsis:

```
void gf_layer_detach( gf_layer_t layer );
```

Arguments:

layer The handle for the layer to detach from. This is the handle returned by *gf_layer_attach()*.

Library:

gf

Description:

This function detaches from a display layer.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_layer_attach(), *gf_layer_disable()*, *gf_layer_enable()*,
gf_layer_query()

gf_layer_disable()

Disable a display layer

Synopsis:

```
void gf_layer_disable( gf_layer_t layer );
```

Arguments:

layer The handle for the layer to disable.

Library:

gf

Description:

This function disables a *layer*. A layer can be either:

- *enabled* — displayed to the user
- *disabled* — not visible to the user



In some cases it's not possible to disable a layer. Main display layers, for instance, typically can't be disabled.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_layer_attach(), *gf_layer_detach()*, *gf_layer_enable()*,
gf_layer_query()

gf_layer_enable()

Enable a display layer

Synopsis:

```
void gf_layer_enable( gf_layer_t layer );
```

Arguments:

layer The handle for the layer to enable.

Library:

gf

Description:

This function enables a *layer*. A layer can be either:

- *enabled* — displayed to the user
- *disabled* — not visible to the user



In some cases it's not possible to disable a layer. Main display layers, for instance, typically can't be disabled.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_layer_attach(), *gf_layer_detach()*, *gf_layer_disable()*,
gf_layer_query()

gf_layer_query()

Query a layer's capabilities

Synopsis:

```
int gf_layer_query( gf_layer_t layer,
                   int format_index,
                   gf_layer_info_t *info );
```

Arguments:

<i>layer</i>	The layer you want to query.
<i>format_index</i>	The index of the format you want to query. Formats are indexed numerically starting at 0.
<i>info</i>	The address of a gf_layer_info_t structure where the function stores information about the <i>layer</i> (see below). Don't pass NULL for this parameter.

Library:

gf

Description:

This function queries a *layer* for a given format index, and fills in the *info* argument with the result. Normally, you'd perform a loop, querying each format index starting at 0 until you either found the *format* you're looking for, or the function returns -1.

The *info* member is a pointer to a **gf_layer_info_t** structure. It contains at least these members:

gf_layer_format_t *format*

The layer format. See *gf_surface_create_layer()* for a list of valid formats.

unsigned *caps*

The layer's capabilities. Valid bits include:

GF_LAYER_CAP_FILTER

The layer can apply a filtering technique to the image as it's being displayed in order to produce a smoother image. You can use filtering techniques to reduce artifacts when scaling images.

GF_LAYER_CAP_SCALE_REPLICATE

A simple pixel replication scaling technique is available for a source image that's scaled before it's displayed in the destination viewport.

GF_LAYER_CAP_PAN_SOURCE

Source viewport position can be adjusted via *gf_layer_set_src_viewport()*.

GF_LAYER_CAP_SIZE_DEST

Size of destination viewport can be different than size of source viewport.

GF_LAYER_CAP_PAN_DEST

Position of destination viewport can be adjusted via *gf_layer_set_dst_viewport()*.

GF_LAYER_CAP_EDGE_CLAMP

If the image being displayed isn't large enough to fill the destination viewport, the unfilled right and bottom portions of the viewport can be filled. The last pixel that was displayed can be replicated to the edge of the viewport.

GF_LAYER_CAP_EDGE_WRAP

If the image being displayed isn't large enough to fill the destination viewport, the unfilled right and bottom portions of the viewport can be filled. The right and bottom portions can be wrapped around to the top left portions of the image.

GF_LAYER_CAP_DISABLE

Layer can be disabled via *gf_layer_disable()*.

GF_LAYER_CAP_SET_BRIGHTNESS

Brightness can be adjusted via *gf_layer_set_brightness()*.

gf_layer_query()

GF_LAYER_CAP_SET_CONTRAST

Contrast can be adjusted via *gf_layer_set_contrast()*.

GF_LAYER_CAP_SET_SATURATION

Saturation can be adjusted via *gf_layer_set_saturation()*.

GF_LAYER_CAP_ALPHA_WITH_CHROMA

Layer can apply alpha and chroma operations simultaneously.

GF_LAYER_CAP_BLEND_WITH_FRONT

The blending parameters normally control how the contents of the layer are blended with intersecting layers that are behind the specified layer. However, if the `GF_LAYER_CAP_BLEND_WITH_FRONT` flag is set for the layer, then the blending parameters control how the layer is blended with intersecting layers in front.

GF_LAYER_CAP_PAN_DEST_NEGATIVE

@@@ ??? @@@

GF_LAYER_CAP_MAIN_DISPLAY

This layer is the main display layer.

unsigned *alpha_valid_flags*;

Flags that may be specified in the *mode* parameter of the `gf_alpha_t` argument to *gf_layer_set_blending()*.

unsigned *alpha_combinations*

The capabilities of the alpha-blending hardware for this layer.
Valid bits include:

GF_ALPHA_CAP_SPP_WITH_GLOBAL

Source per-pixel blending can be used in conjunction with a global alpha multiplier.

GF_ALPHA_CAP_GLOBAL_WITH_DPP

Destination per-pixel blending can be used in conjunction with a global alpha multiplier.

GF_ALPHA_CAP_SPP_WITH_DPP

Source per-pixel blending can be used in conjunction with destination per-pixel blending.

GF_ALPHA_CAP_GLOBAL_WITH_GLOBAL

Source global alpha multiplier can be used in conjunction with a destination global alpha multiplier.

unsigned *chromakey_caps*

The capabilities of the chroma-key hardware for this layer. Valid bits include:

GF_LAYER_CHROMAKEY_CAP_SRC_SINGLE

The layer supports chroma-keying based on an exact match between the source pixel value and a single key color.

GF_LAYER_CHROMAKEY_CAP_SRC_RANGE

@@@??@ @

GF_LAYER_CHROMAKEY_CAP_SRC_MASK

@@@??@ @

GF_LAYER_CHROMAKEY_CAP_DST_SINGLE

The layer supports chroma-keying based on an exact match between the destination pixel value.

GF_LAYER_CHROMAKEY_CAP_DST_RANGE

@@@??@ @

GF_LAYER_CHROMAKEY_CAP_DST_MASK

@@@??@ @

GF_LAYER_CHROMAKEY_CAP_SHOWTHROUGH

The layer can be configured so that when a chroma-key comparison is made, and the colors match, the pixel displayed comes from the behind the layer. When the colors don't match, the pixel that appears comes from the layer displayed.

GF_LAYER_CHROMAKEY_CAP_BLOCK

The layer can be configured so that when a chroma-key comparison is made, and the colors match, the pixel

gf_layer_query()

displayed comes from the layer displayed. When the colors don't match, the pixel that appears comes from behind the layer.

_uint64 order_caps

The z-order slots which this layer can occupy. For each slot, a corresponding bit in *order_caps* will be either ON or OFF to reflect whether or not this layer can be ordered to that slot. Slot 0 (furthest from the user) is represented by the least significant bit 0, slot 1 by bit 1, and so on.

int src_max_height;

The maximum height (in pixels) of the source.

int src_max_width

The maximum width (in pixels) of the source.

int src_max_viewport_height

The maximum height (in pixels) of the source viewport.

int src_max_viewport_width

The maximum width (in pixels) of the source viewport.

int dst_max_height

The maximum height (in pixels) of the destination.

int dst_max_width

The maximum width (in pixels) of the destination.

int dst_min_height

The maximum height (in pixels) of the destination viewport.

int dst_min_width

The maximum width (in pixels) of the destination viewport.

int max_scaleup_x, int max_scaleup_y

The maximum scaling factor for image upscaling in the horizontal and vertical directions. A value of 1 means upscaling

can't be performed. A value `< 1` is invalid. A value `> 1` for *max_scaleup_x* means that the destination viewport width can be up to *max_scaleup_x* times the source viewport width. Similarly, a value `> 1` for *max_scaleup_y* means that the destination viewport height can be up to *max_scaleup_y* times the source viewport height.

`int max_scaledown_x, int max_scaledown_y`

Maximum scaling factor for image upscaling in the horizontal and vertical directions. A value of `1` means upscaling can't be performed. A value `< 1` is invalid. A value `> 1` for *max_scaledown_x* means that the destination viewport width can be up to *max_scaledown_x* times the source viewport width. Similarly, A value `> 1` for *max_scaledown_y* means that the destination viewport height can be up to *max_scaledown_y* times the source viewport height.

Returns:

- `0` Success.
- `-1` An error occurred.

Examples:

See Using Layers in the Using Layers, Surfaces, and Contexts chapter.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_layer_query()

See also:

gf_layer_attach(), *gf_layer_detach()*, *gf_layer_disable()*,
gf_layer_enable()

gf_layer_set_blending()

Set layer blending

Synopsis:

```
void gf_layer_set_blending( gf_layer_t layer,
                           gf_alpha_t * alpha );
```

Arguments:

layer A pointer to a **gf_layer_t** handle of the layer you want to set alpha blending for.

alpha A **gf_alpha_t** structure describing the new blending.

Library:

gf

Description:

This function gives you control over layer blending (that is, opacity), allowing you to make a layer fully opaque, transparent, or semitransparent.



Changes to the layer's parameters are enqueued. You need to call *gf_layer_update()* to make them take effect.

The blending parameters normally control how the contents of the layer are blended with intersecting layers that are behind the specified layer. However, if the **GF_LAYER_CAP_BLEND_WITH_FRONT** flag was set for the layer, then the blending parameters control how the layer is blended with intersecting layers in front.



For the blending operation, the front most layer(s) can be thought of as the “source” while the layer(s) behind can be thought of as the “destination”.

gf_layer_set_blending()

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_alpha_t, ***gf_layer_query()***, ***gf_layer_update()***

gf_layer_set_brightness()

Set a layer's brightness

Synopsis:

```
void gf_layer_set_brightness( gf_layer_t layer,  
                             int brightness );
```

Arguments:

<i>layer</i>	A pointer to the gf_layer_t handle of the layer you want to set the brightness for.
<i>brightness</i>	The brightness level of layer, in the range of -127 to 128, with 0 indicating normal brightness.

Library:

gf

Description:

This function controls the brightness of a layer.



Changes to the layer's parameters are enqueued. You need to call *gf_layer_update()* to make them take effect.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_layer_set_brightness()

See also:

gf_display_attach(), *gf_layer_detach()*, *gf_layer_disable()*,
gf_layer_enable(), *gf_layer_query()*

gf_layer_set_chroma()

Set a layer's chroma operation

Synopsis:

```
void gf_layer_set_chroma( gf_layer_t layer,  
                          gf_chroma_t * chroma );
```

Arguments:

<i>layer</i>	A pointer to a gf_layer_t handle of the layer you want to set the chroma operation for.
<i>chroma</i>	A pointer to a gf_chroma_t describing the new chroma operation.

Library:

gf

Description:

This function sets a layer's chroma operation. Chroma allows you extended control over how the layer is overlaid into the overall display.



Changes to the layer's parameters are enqueued. You need to call *gf_layer_update()* to make them take effect.

The chroma parameters normally control how the contents of the layer are compared with intersecting layers that are behind the specified layer. However, if the **GF_LAYER_CAP_BLEND_WITH_FRONT** flag was set for the layer, then the chroma parameters control how the layer is compared with intersecting layers in front.

gf_layer_set_chroma()



For the chroma operation, the front-most layer(s) can be thought of as the “source” while the layer(s) behind can be thought of as the “destination”.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_chroma_t, *gf_layer_set_blending()*, *gf_layer_query()*,
gf_layer_update()

gf_layer_set_contrast()

Set a layer's contrast level

Synopsis:

```
void gf_layer_set_contrast( gf_layer_t layer,
                           int contrast );
```

Arguments:

<i>layer</i>	A pointer to a gf_layer_t handle of the layer you want to set the contrast for.
<i>contrast</i>	The contrast level of layer, in the range of -127 to 128, with 0 indicating normal contrast.

Library:

gf

Description:

This function controls the contrast of a layer.



Changes to the layer's parameters are enqueued. You need to call *gf_layer_update()* to make them take effect.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_layer_set_contrast()

See also:

gf_display_attach(), *gf_layer_detach()*, *gf_layer_disable()*,
gf_layer_enable(), *gf_layer_query()*

gf_layer_set_dst_viewport()

Set a layer's destination viewport

Synopsis:

```
void gf_layer_set_dst_viewport( gf_layer_t layer,
                                int x1,
                                int y1,
                                int x2,
                                int y2 );
```

Arguments:

<i>layer</i>	A pointer to a gf_layer_t handle of the layer you want to set the destination viewport for.
<i>x1, y1, x2, y2</i>	The coordinates of the bounding rectangle for the destination viewport. The <i>x1, y1</i> points define the upper-left corner of the rectangle, and the <i>x2, y2</i> points define the lower-right corner. The rectangle's coordinates must be within the bounds of the target display.

Library:

gf

Description:

This function sets a layer's destination viewport.

There may be hardware limitations on viewports. Check these flag bits in **gf_layer_info_t.caps** filled in by *gf_layer_query()*:

- **GF_LAYER_CAP_PAN_SOURCE** — the source viewport position can be adjusted.
- **GF_LAYER_CAP_SIZE_DEST** — the size of destination viewport can be different than size of source viewport.
- **GF_LAYER_CAP_PAN_DEST** — the position of destination viewport can be adjusted

gf_layer_set_dst_viewport()



Changes to the layer’s parameters are enqueued. You need to call *gf_layer_update()* to make them take effect.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_layer_set_src_viewport(), *gf_layer_query()*

The “Viewports” section of the Working with Layers, Surfaces and Contexts chapter.

gf_layer_set_saturation()

Set a layer's saturation level

Synopsis:

```
void gf_layer_set_saturation( gf_layer_t layer,
                             int saturation );
```

Arguments:

<i>layer</i>	A pointer to a gf_layer_t handle of the layer you want to set the saturation for.
<i>saturation</i>	The saturation level for the layer, in the range of -127 to 128 , with 0 indicating normal saturation.

Library:

gf

Description:

This function controls the saturation level of a layer.



Changes to the layer's parameters are enqueued. You need to call *gf_layer_update()* to make them take effect.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_layer_set_saturation()

See also:

gf_display_attach(), *gf_layer_detach()*, *gf_layer_disable()*,
gf_layer_enable(), *gf_layer_query()*

gf_layer_set_src_viewport()

Set a layer's source viewport

Synopsis:

```
void gf_layer_set_src_viewport( gf_layer_t layer,
                                int x1,
                                int y1,
                                int x2,
                                int y2 );
```

Arguments:

<i>layer</i>	A pointer to a gf_layer_t handle of the layer you want to set the source viewport for.
<i>x1, y1, x2, y2</i>	The coordinates of the bounding rectangle for the destination viewport. The <i>x1, y1</i> points define the upper-left corner of the rectangle, and the <i>x2, y2</i> points define the lower-right corner.

Library:

gf

Description:

This function sets a layer's source viewport. The *source viewport* specifies the area of the target surface which the layer displays in the destination viewport.

There may be hardware limitations on viewports. Check these flag bits in **gf_layer_info_t.caps** filled in by *gf_layer_query()*:

- **GF_LAYER_CAP_PAN_SOURCE** — the source viewport position can be adjusted.
- **GF_LAYER_CAP_SIZE_DEST** — the size of destination viewport can be different than size of source viewport.
- **GF_LAYER_CAP_PAN_DEST** — the position of destination viewport can be adjusted

gf_layer_set_src_viewport()



Changes to the layer's parameters are enqueued. You need to call *gf_layer_update()* to make them take effect.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_layer_set_dst_viewport(), *gf_layer_query()*

The “Viewports” section of the Working with Layers, Surfaces and Contexts chapter.

gf_layer_set_surfaces()

Set the target surface for a layer

Synopsis:

```
void gf_layer_set_surfaces( gf_layer_t layer,
                           unsigned layer_format,
                           igf_surface_t surface[],
                           int nsurfs );
```

Arguments:

<i>layer</i>	The handle for a layer.
<i>surface</i>	An array of surface handles for the layer to use.
<i>nsurfs</i>	The number of items in the <i>surface</i> array.

Library:

gf

Description:

This function sets the target surface or surfaces for a layer.



In general a layer targets a single surface. However some planar formats (for example, YUV) use one surface per plane.



Changes to the layer's parameters are enqueued. You need to call *gf_layer_update()* to make them take effect.

Classification:

Graphics Framework

Safety

Interrupt handler No

continued...

gf_layer_set_surfaces()

Safety	
Signal handler	No
Thread	Yes

See also:

gf_display_attach(), *gf_layer_detach()*, *gf_layer_disable()*,
gf_layer_enable(), *gf_layer_query()*

gf_layer_update()

Update layer parameters

Synopsis:

```
int gf_layer_update( gf_layer_t layer,
                    unsigned flags );
```

Arguments:

- layer* The handle for the layer to update.
- flags* Flags to control the function's behavior. Valid flags are:
- GF_LAYER_UPDATE_NO_WAIT_VSYNC — Perform the operation asynchronously; the default behavior is to block until the next vertical synchronization.
 - GF_LAYER_UPDATE_NO_WAIT_IDLE — Perform the operation immediately; the default behavior is to wait for the draw hardware to finish.

Library:

gf

Description:

This function updates a layer's parameters. Changes made to a layer with the *gf_layer_**() set of functions are enqueued and don't take effect until you explicitly call this function.

Returns:

- 0 Success.
- 1 An error occurred.

Classification:

Graphics Framework

gf_layer_update()

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_layer_detach(), *gf_layer_disable()*, *gf_layer_enable()*,
gf_layer_query()

gf_palette_t

A graphics framework palette

Synopsis:

```
typedef struct {  
    int ncolors;  
    gf_color_t *colors;  
} gf_palette_t;
```

Description:

The **gf_palette_t** structure defines a color palette associated with a surface. It contains at least the following members:

<i>ncolors</i>	The number of colors in the <i>colors</i> array.
<i>colors</i>	Pointer to an array of <i>ncolors</i> colors, of type gf_color_t .

A **gf_color_t** is a generic 32 bit color argument used by the GF library. This type is assumed to be native endian, with the following components:

- red — an 8-bit value specifying red intensity, occupies byte 2 (bits 16-23).
- green — an 8-bit value specifying green intensity, occupies byte 1 (bits 8-15).
- blue — an 8-bit value specifying blue intensity, occupies the least significant byte (bits 0-7)
- alpha — and 8-bit value specifying alpha factor, where applicable. This value occupies the most significant byte (bits 24-31).

Classification:

Graphics Framework

`gf_palette_t`

See also:

gf_surface_attach(), *gf_surface_attach_by_sid()*, *gf_surface_create()*,
gf_surface_create_layer()

gf_point_t

Coordinates of a single point

Synopsis:

```
typedef struct {  
    _int16_t x;  
    _int16_t y;  
} gf_point_t;
```

Description:

The **gf_point_t** structure describes a 2D coordinate in a Cartesian system where *x* progresses from the left to the right and *y* progresses from the top to the bottom. It contains at least the following members:

- x* X-axis coordinate, which is the horizontal position of the point.
- y* Y-axis coordinate, which is the vertical position of the point.

Classification:

Graphics Framework

See also:

*gf_cursor_set(), gf_cursor_set_pos(), gf_draw_bitmap(),
gf_draw_blit1(), gf_draw_blit2(), gf_draw_poly_fill(),
gf_draw_polyline(), gf_draw_string()*

gf_surface_attach()

Create a new surface from existing memory

Synopsis:

```
int gf_surface_attach( gf_surface_t *psurface,
                      gf_dev_t gdev,
                      int w,
                      int h,
                      int stride,
                      gf_pixel_format_t format,
                      const gf_palette_t *palette,
                      uint8_t *ptr,
                      unsigned flags );
```

Arguments:

<i>psurface</i>	Address to store a handle for the surface.
<i>gdev</i>	The graphics context that's responsible for managing the surface, returned by <i>gf_dev_attach()</i> .
<i>w, h</i>	The height and width of the surface, in pixels.
<i>stride</i>	Number of bytes per scanline
<i>format</i>	Pixel format of the surface. See <i>gf_surface_create()</i> for possible values.
<i>palette</i>	A pointer to a gf_palette_t structure, which represents the surface palette. You may pass NULL only if the surface format is GF_FORMAT_PAL8.
<i>ptr</i>	A pointer to the memory buffer containing image data.
<i>flags</i>	Flags affecting the surface properties. None are defined; pass 0.

Library:

gf

Description:

This function creates a new surface and attaches it to an existing area of memory. This wraps your own area of memory (or pre-existing frame data, such as a previously loaded image) in a format that can be manipulated by the GF API as a regular surface (with some restrictions).

Because the driver knows nothing about the nature of the memory nor how it was allocated, the hardware renderer isn't able to target attached surfaces directly. Any rendering or blitting to or from the surface is done in software.

For the same reason, the application is responsible for deallocating the memory containing the frame data. This should be done only after all attached surfaces are done with the memory (that is, after all attached surfaces have been deallocated with *gf_surface_free()*).

Because the memory isn't managed by **io-display** it can't be shared. Therefore, other processes aren't able to access attached surfaces via *gf_surface_attach_by_sid()*.

Returns:

- 0 Success.
- 1 An error occurred.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_surface_attach()

See also:

gf_palette_t, *gf_surface_create()*, *gf_surface_create_layer()*,
gf_surface_free()

gf_surface_attach_by_sid()

Attach to a previously allocated surface

Synopsis:

```
int gf_surface_attach_by_sid( gf_surface_t * psurface,
                             gf_dev_t gdev,
                             gf_sid_t sid );
```

Arguments:

<i>psurface</i>	Address where the function can store a handle to the surface.
<i>gdev</i>	A handle for the graphical device (acquired by <i>gf_dev_attach()</i> that will manage the surface.
<i>sid</i>	The surface ID of the desired surface.

Library:

gf

Description:

This function creates a new surface which is attached to a previously allocated surface.



No control over surface parameters is provided via this function since the surface already exists and these parameters are already defined.

Any surface that is managed by **io-display** (that is, a surface that's been created using *gf_surface_create()* or *gf_surface_create_layer()* with default settings) can be shared across process boundaries by its surface ID (SID), a unique numerical ID assigned by **io-display** to each managed surface. Multiple processes can render to the same surface by sharing surfaces in this manner.

gf_surface_attach_by_sid()



Coordination of rendering (ensuring one process doesn't render over another) is up to the application.

You can get the SID for a surface by using *gf_surface_get_info()*.

Returns:

- 0 Success.
- 1 An error occurred.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_surface_attach(), *gf_surface_create()*, *gf_surface_get_info()*

gf_surface_create()

Create a new surface

Synopsis:

```
int gf_surface_create( gf_surface_t *psurface,
                      gf_dev_t gdev,
                      int w,
                      int h,
                      gf_pixel_format_t format,
                      const gf_palette_t *palette,
                      unsigned flags );
```

Arguments:

<i>psurface</i>	An address where the function can store a handle to the surface.
<i>gdev</i>	The handle to the graphical device (acquired by <i>gf_dev_attach()</i>) that will manage the surface.
<i>w, h</i>	The width and height of the surface, in pixels.
<i>format</i>	The pixel format of the surface. See below
<i>palette</i>	A pointer to a gf_palette_t structure, which represents the surface palette. Pass NULL if the surface format isn't palette-based (GF_FORMAT_PAL8).
<i>flags</i>	Flags for the surface's properties. These can be: <ul style="list-style-type: none">• GF_SURFACE_CREATE_2D_ACCESSIBLE — surface will be used by the 2D API (that is, rendered to and blitted from).• GF_SURFACE_CREATE_3D_ACCESSIBLE — surface will be used by the 3D API (GL ES).• GF_SURFACE_CREATE_CPU_FAST_ACCESS — surface is allocated so that CPU access speed is optimized. Satisfying this request typically precludes accelerated hardware rendering. The default is to allocate a surface optimize for rendering performance; that is, try to allocate such that hardware accelerated rendering is possible.

gf_surface_create()

- `GF_SURFACE_CREATE_CPU_LINEAR_ACCESSIBLE` — surface memory is linearly accessible. Note that this request may preclude hardware accelerated rendering if, for example, the hardware uses tiled video memory. In general, though, this is not the case.
- `GF_SURFACE_PAGE_ALIGNED` — the allocated surface must be aligned on a page boundary. You may require this if you are doing your own advanced manipulation of allocated memory, or if you have some hardware with alignment restrictions. Otherwise this is not usually a requirement.
- `GF_SURFACE_PHYS_CONTIG` — surface memory must be physically contiguous. This is not a usual requirement; however it may be needed in some cases. Some video capture hardware, for instance, requires that the memory be physically contiguous.

Library:

`gf`

Description:

Create a new surface. A surface provides an area of memory which can be rendered to via draw contexts. Use this function for “offscreen” buffers that will not be targeted by any layers (that is, you will only render to the surface, then blit the results somewhere else). If you plan to directly target the surface with any layers, you must use *gf_surface_create_layer()* instead.

The `gf_pixel_format_t` enumerates supported pixel formats. Pixel formats identify the format of pixel data in a buffer.

The possible values are:

`GF_FORMAT_BYTE`

8 bit per pixel format with context-dependent meaning

GF_FORMAT_PAL8

8 bit per pixel index into a palette of up to 256 entries

GF_FORMAT_PKLE_ARGB1555

16-bit per pixel ARGB packed into 16-bit little-endian integer type with bits 0-4 for blue, 5-9 for green, 10-14 for red and most significant bit for alpha

GF_FORMAT_PKLE_RGB565

16-bit per pixel RGB packed into 16-bit little-endian integer type with bits 0-4 for blue, 5-10 for green, and 11-15 for red

GF_FORMAT_BGR888

24-bit per pixel BGR with 8 bits per channel as an ordered byte sequence

GF_FORMAT_BGRA8888

32-bit per pixel BGRA with 8 bits per channel as an ordered byte sequence

GF_FORMAT_PKLE_YUV_UYVY

16-bit packed YUV (FourCC code UYVY)

GF_FORMAT_PKLE_YUV_YUY2

16-bit packed YUV (FourCC code YUY2)

GF_FORMAT_PKLE_YUV_YVYU

16-bit packed YUV (FourCC code YVYU)

GF_FORMAT_PKLE_YUV_V422

16-bit packed YUV (FourCC code V422)

Returns:

- 0** Success.
- 1** An error occurred.

gf_surface_create()

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_palette_t, *gf_surface_attach()*, *gf_surface_create_layer()*,
gf_surface_free()

gf_surface_create_layer()

Create a surface that a layer can target

Synopsis:

```
int gf_surface_create_layer( gf_surface_t *psurface,
                             gf_layer_t *layer,
                             int nlayers,
                             int surface_index,
                             int w,
                             int h,
                             gf_layer_format_t layer_format,
                             const gf_palette_t *palette,
                             uint32_t flags );
```

Arguments:

<i>psurface</i>	An address where the function can store a handle to the surface.
<i>layer</i>	An array of handles to the layers which you would like to be able to safely target the surface.
<i>nlayers</i>	The number of handles in the <i>layer</i> array.
<i>surface_index</i>	The surface index, which used only in the case of planar layer formats (for example YUV) which requires a surface for each plane. For non-planar surfaces (the usual case) you should always pass 0 for this parameter.
<i>w, h</i>	The width and height of the surface, in pixels.
<i>layer_format</i>	The layer format of the surface. See below.
<i>palette</i>	A pointer to a gf_palette_t structure, which represents the surface palette. You can pass NULL if the surface format isn't palette-based (GF_LAYER_FORMAT_PAL8).
<i>flags</i>	Flags affecting the surface allocation. See <i>gf_surface_create()</i> .

gf_surface_create_layer()

Library:

gf

Description:

This function creates a new surface that can be targetted by the specified *layer*. You must use this function if you plan on binding the surface directly to one or more layers via *gf_layer_set_surfaces()*. It provides additional hardware checking to verify that the layer can target the surface. If the surface is only going to be used as a rendering “scratchpad” for offscreen rendering, you can use *gf_surface_create()* instead.

The **gf_layer_format_t** enumerates supported layer formats. Layer formats should not be confused with pixel formats. Pixel formats provide detailed information about the formatting of pixel data in a buffer, while layer formats are derived from the way in which layers are implemented. Typically, drivers separate layer capabilities into distinct formats (analagous to the different video modes a display can support). The layer format type lets you easily identify which of these formats you are interested in.

The possible values for **gf_layer_format_t** are:

GF_LAYER_FORMAT_PAL8

8 bit/pixel indexed into a palette of up to 256 entries

GF_LAYER_FORMAT_ARGB1555

16 bit/pixel ARGB model

GF_LAYER_FORMAT_RGB565

16 bit/pixel RGB model

GF_LAYER_FORMAT_BGR888

24 bit/pixel RGB model

GF_LAYER_FORMAT_BGRA8888

32 bit/pixel ARGB model

gf_surface_create_layer()

GF_LAYER_FORMAT_YUY2

16-bit packed YUV (FourCC code YUY2)

GF_LAYER_FORMAT_UYVY

16-bit packed YUV (FourCC code UYVY)

GF_LAYER_FORMAT_YVYU

16-bit packed YUV (FourCC code YVYU)

GF_LAYER_FORMAT_V422

16-bit packed YUV (FourCC code V422)

GF_LAYER_FORMAT_YVU9

Planar YUV (FourCC code YVU9)

GF_LAYER_FORMAT_YV12

Planar YUV (FourCC code YV12)

GF_LAYER_FORMAT_YUV420

Planar YUV

Returns:

- 0 Success.
- 1 An error occurred.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_surface_create_layer()

See also:

gf_palette_t, *gf_layer_set_surfaces()*, *gf_surface_attach()*,
gf_surface_create(), *gf_surface_free()*

gf_surface_free()

Free a surface

Synopsis:

```
void gf_surface_free( gf_surface_t surface );
```

Arguments:

psurface The handle for the surface you want to free.

Library:

gf

Description:

This function frees a surface created with *gf_surface_create()*, *gf_surface_create_layer()*, or *gf_surface_attach()*. Call this only once you are done with a surface and no remaining contexts are targeting it. If the frame data buffer was allocated automatically it also will be freed.



When deallocating surfaces created with *gf_surface_attach()* or *gf_surface_attach_by_sid()*, the frame data buffer won't be automatically freed. It is up to the application to free this memory at an appropriate time.

gf_dev_detach() automatically frees all associated surfaces.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

gf_surface_free()

See also:

gf_surface_attach(), *gf_surface_create()*, *gf_surface_create_layer()*

gf_surface_get_info()

Get surface parameters

Synopsis:

```
gf_surface_info_t *gf_surface_get_info( gf_surface_t surface,  
                                        gf_surface_info_t *info );
```

Arguments:

surface The handle for the surface.

info Address of structure that the function can fill with surface information (see below). Do not pass NULL.

Library:

gf

Description:

This function retrieves surface parameters.



If the surface ID (SID) is reported as GF_SID_INVALID by this function, then the surface isn't being managed by **io-display** and therefore can't be shared across process boundaries.

The *info* parameter contains information about the surface:

```
typedef struct {  
    gf_sid_t          sid;  
    int               w;  
    int               h;  
    gf_pixel_format_t pixel_format;  
    unsigned          stride;  
    _uint64           paddr;  
    _uint8*           vaddr;  
    gf_palette_t      palette;  
    unsigned          flags;  
} gf_surface_info_t;
```

It contains at least these parameters:

gf_surface_get_info()

<i>sid</i>	If this surface was allocated by io-display , this field specifies the numerical identifier that is assigned to this surface, which is guaranteed to be unique across the entire device. If the surface was not allocated by io-display (as in the case of <i>gf_surface_attach()</i> , this field is set to a value of GF_SID_INVALID .
<i>w, h</i>	Width and height of the surface, in pixels.
<i>pixel_format</i>	Pixel format of the surface. See <i>gf_display_attach()</i> for a list of possible pixel formats.
<i>stride</i>	Number of bytes per scanline.
<i>paddr</i>	Physical address of the surface memory buffer. This should only be used if the surface was created with the GF_SURFACE_PHYS_CONTIG flag bit set.
<i>vaddr</i>	Virtual address of the surface memory buffer. This should only be used if the surface was created with the GF_SURFACE_CREATE_CPU_LINEAR_ACCESSIBLE flag bit set.
<i>palette</i>	The surface palette, of type gf_palette_t . This member is valid only if the surface format is set to GF_FORMAT_PAL8 .
<i>flags</i>	Flags that provide additional information regarding the surface. Valid bits include: GF_SURFACE_DISPLAYABLE The surface can be displayed via a CRT controller. GF_SURFACE_CPU_LINEAR_READABLE The CPU can read this surface directly. GF_SURFACE_CPU_LINEAR_WRITEABLE The CPU can write to this surface directly.

gf_surface_get_info()

GF_SURFACE_2D_READABLE

The surface is read-accessible by 2D engine.

GF_SURFACE_3D_READABLE

The surface is read-accessible by 3D engine.

GF_SURFACE_2D_TARGETABLE

The 2D engine can render into surface.

GF_SURFACE_3D_TARGETABLE

The 3D engine can render into surface.

GF_SURFACE_SCALER_DISPLAYABLE

The surface can be displayed via video overlay scaler.

GF_SURFACE_VML_TARGETABLE

Video in hardware can write frames into surface.

GF_SURFACE_DMA_SAFE

The DMA engine can treat the surface memory as one contiguous block.

GF_SURFACE_PAGE_ALIGNED

The surface memory starts on a page boundary, and its linear size is a multiple of the page size.

GF_SURFACE_BYTES_REVERSED

The byte order is reversed. This is valid only for 16 and 32 bpp surfaces.

GF_SURFACE_NON_CACHEABLE

The memory should be mapped non-cacheable.

GF_SURFACE_WT_CACHEABLE

The memory can be mapped write-through cacheable (but not write-back cacheable).

GF_SURFACE_PHYS_CONTIG

The memory is physically contiguous.

gf_surface_get_info()

GF_SURFACE_DRIVER_NOT_OWNER

The driver did not create or allocate this surface.

Returns:

The address passed in.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

gf_surface_attach(), *gf_surface_create()*, *gf_surface_create_layer()*,
gf_palette_t

Appendix D

QNX Image Library Reference



These functions handle operations that directly involve the QNX Image Library (**libimg**). Using these functions and structures, you can load and decode images from a file.

img_codec_list()

Enumerate codecs

Synopsis:

```
size_t img_codec_list( img_lib_t    ilib,  
                      img_codec_t* buf,  
                      size_t       nbuf,  
                      img_codec_t* exclude,  
                      size_t       nexclude );
```

Arguments:

<i>ilib</i>	The handle for the image library, returned by <i>img_lib_attach()</i> .
<i>buf</i>	The address of an array that the function populates with handles for available codecs.
<i>nbuf</i>	Number of items in the <i>buf</i> array.
<i>exclude</i>	The address of an array of codec handles that you'd like the function to exclude from the list.
<i>nexclude</i>	Number of items in the <i>exclude</i> array.

Library:

img

Description:

This function lists all codecs installed, without regard to any specific criteria. The codecs listed in the *exclude* array aren't included in the list.

Returns:

The number of items copied into the *buf* array.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

img_lib_attach(), *img_codec_list_byext()*, *img_codec_list_bymime()*

img_codec_list_byext()

Enumerate codecs by file extension

Synopsis:

```
size_t img_codec_list_byext( img_lib_t    ilib,  
                             const char*  string,  
                             img_codec_t* buf,  
                             size_t       nbuf);
```

Arguments:

<i>ilib</i>	The handle for the image library, returned by <i>img_lib_attach()</i> .
<i>string</i>	String containing the file extension to identify. For example “jpg” for files named filename.jpg .
<i>buf</i>	The address of an array that the function populates with handles for available codecs.
<i>nbuf</i>	Number of items in the <i>buf</i> array.

Library:

img

Description:

This function enumerates codecs which handle files with the specified extension.

While there are no standards defining what extensions are and how they should be named, there seems to be a de facto standard governing their use. The term “extension” originates as an intrinsic filename property although most contemporary implementations no longer treat them as separate components of the filename. Thus the term has evolved to loosely describe the portion of characters in a filename that follow the last occurrence of the `.` character. This principle can be seen in use throughout the world wide web, and through many OSes which use extensions as a content recognition mechanism. Although extensions do not in themselves impose any guarantee on the nature of data a file contains, they are generally appropriately assigned from

img_codec_list_byext()

a well known set, and as such they represent reasonable criteria in deciding which codec should handle the data, at least in an initial pass.

Returns:

The number of items copied into the *buf* array.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

img_lib_attach(), *img_codec_list()*, *img_codec_list_bymime()*

img_codec_list_bymime()

Enumerate codecs by MIME type

Synopsis:

```
size_t img_codec_list_bymime( img_lib_t    ilib,  
                             const char*   mime,  
                             img_codec_t*  buf,  
                             size_t        nbuf );
```

Arguments:

<i>ilib</i>	The handle for the image library, returned by <i>img_lib_attach()</i> .
<i>mime</i>	String describing the desired MIME type (in accordance with RFC 2046).
<i>buf</i>	The address of an array that the function populates with handles for available codecs.
<i>nbuf</i>	Number of items in the <i>buf</i> array.

Library:

img

Description:

This function enumerate codecs which handle a specified MIME type.

Returns:

The number of items copied into the *buf* array.

Classification:

Graphics Framework

Safety

Interrupt handler No

continued...

img_codec_list_bymime()

Safety

Signal handler	No
Thread	No

See also:

img_lib_attach(), *img_codec_list()*, *img_codec_list_byext()*

img_convert_data()

Convert data from one image format to another

Synopsis:

```
int img_convert_data( img_format_t      sformat,
                     const uint8_t*    src,
                     img_format_t      dformat,
                     uint8_t*          dst,
                     size_t            n,
                     const img_color_t* palette );
```

Arguments:

<i>sformat</i>	The format of the data you are converting from (see below).
<i>src</i>	A pointer to a buffer containing the source data.
<i>dformat</i>	The format you would like to convert the data to.
<i>dst</i>	A pointer to a buffer to store the converted data. This may point to a different buffer, or it can point to the same buffer as <i>src</i> , as long as you've ensured that the source buffer is large enough to store the converted data (the <i>IMG_FMT_BPL()</i> macro can help you with this).
<i>n</i>	The number of samples to convert.
<i>palette</i>	A table of colors specifying the palette entries associated with the source data. This is needed only when the IMG_FMT_PALETTE bit of <i>sformat</i> is set; otherwise you may pass NULL for this argument.

Library:

img

Description:

This function converts data from one image format to another. The conversion may be done from one buffer to another, or in place.



The destination format *dformat* cannot be a palette-based format (for example IMG_FMT_PAL8 or IMG_FMT_PAL4). Palettizing samples is beyond the scope of a simple data conversion utility.

The `img_format_t` structure

The `img_format_t` is an enumeration of these possible image formats:

IMG_FMT_MONO

Monochromatic bitmap with 1 bit/pixel, packing 8 pixels per byte.

IMG_FMT_G8 8 bit/pixel graymap.

IMG_FMT_PAL1 1 bit/pixel index into a palette of 2 entries, packing 8 pixels per byte.

IMG_FMT_PAL4 4 bit/pixel index into a palette of up to 16 entries, packing 2 pixels per byte.

IMG_FMT_PAL8 8 bit/pixel index into a palette of up to 256 entries.

IMG_FMT_PKLE_RGB565

16-bit/pixel RGB packed into 16-bit little-endian integer type with bits 0-4 for B, 5-10 for G, and 11-15 for R.

IMG_FMT_PKLE_ARGB1555

16-bit/pixel ARGB packed into 16-bit little-endian integer type with bits 0-4 for B, 5-9 for G, 10-14 for R and most significant bit for A.

IMG_FMT_BGR888

24-bit/pixel BGR with 8 bits per channel as an ordered byte sequence.

img_convert_data()

IMG_FMT_RGB888

24-bit/pixel RGB with 8 bits per channel as an ordered byte sequence.

IMG_FMT_BGRX8888

24-bit/pixel BGR with 8 bits per channel as an ordered byte sequence, followed by a single byte of padding.

IMG_FMT_BGRA8888

32-bit/pixel BGRA with 8 bits per channel as an ordered byte sequence.

IMG_FMT_RGBA8888

32-bit/pixel RGBA with 8 bits per channel as an ordered byte sequence.

Returns:

- 0 Success.
- 1 One of the formats specified is invalid.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	No

img_convert_data()

See also:

img_lib_attach(),

img_decode_callouts_t

Decoder callout table

Synopsis:

```
typedef struct {
    img_decode_choose_format_f *choose_format_f;
    img_decode_alloc_f *alloc_f;
    img_decode_release_f *release_f;
    img_decode_scanline_f *scanline_f;
    img_decode_notify_f *notify_f;
    _Uinptrt data;
} img_decode_callouts_t;
```

Description:

The **gf_palette_t** structure defines a decoder callout table. It provides the decoder with a list of callouts for it to invoke at various stages of the decode.

img_decode_choose_format_f* *choose_format_f*

Choose image data format. Prior to allocating the buffer for the frame data, the decoder will generate a list of possible data formats that it can provide the data in, based on the input data. This callout is expected to select an appropriate format.

This callout is optional. If you do not provide one, the library provides a default that selects the first format that the decoder offers.

The function takes this form:

```
int img_decode_choose_format_f( _Uinptrt data,
                               img_info_t* info,
                               img_format_t const* formats,
                               size_t nformats );
```

The arguments for this function are:

<i>data</i>	Application data
<i>info</i>	Pointer to a partially filled img_info_t structure providing vital information about the frame. This callout is expected to fill the <i>format</i> field of that structure.

formats An array of possible `img_format_t` formats to choose from. The callout must choose one of the formats from the list and copy the selection into the *format* field of the *info* structure.

nformats The number of elements in the *formats* array.

It should return:

- 0 — Format selected, proceed.
- -1 — No formats were acceptable, don't proceed.

`img_decode_alloc_f* alloc_f`

A pointer to a function that allocates memory for the frame. This callout is invoked after the header information for the frame has been read, but before any palette or image data is read.

This callout is optional. If you do not supply one, the library provides a default that allocates memory from system RAM for the frame and palette using `malloc()`. In order to free the data, you may simply call `free()` passing the *data* field of `img_info_t`. In this case, this will take care of freeing both the image data and the palette (where applicable). You do *not* have to free the palette explicitly when relying on this default callout.

The function takes this form:

```
int img_decode_alloc_f( _Uintptr_t data,
                      img_info_t *info );
```

The arguments for this function are:

data Application data

info Pointer to a partially filled `img_info_t` structure providing vital information about the frame. The callout is responsible for filling in the following remaining fields:

data Set this to point to a buffer where the image data should be decoded. This buffer must be

`img_decode_callouts_t`

sufficiently large to, at a minimum, hold a single scanline of image data (if you have set the *stride* member to 0). The `IMG_FMT_BPL()` macro can be used to determine the minimum requirements for a single scanline. If *stride* is set to a nonzero value, then the buffer must provide a buffer of *h* rows of *stride* bytes each, and scanlines will be decoded from the top moving down.

stride The number of bytes between scanlines in your allocated buffer. You should use the `IMG_FMT_BPL()` macro to determine a minimum limit for this value. Note that you may set this to zero if you wish to receive and process the data one scanline at a time.

palette Pointer to an area of memory to store the palette. You only need to set this if the `IMG_FMT_PALETTE` bit is set in the *format* field.

`img_decode_release_f* release_f`

A pointer to a function that releases memory allocated for a frame. It's invoked if an error occurs during the decode phase and any frame resources allocated by the setup callout need to be released.

This callout is optional, however not supplying one could result in a resource leak unless you're careful to explicitly free your allocated data in the event of a decode error.

If you don't supply an allocate callout, the library provides its own corresponding release callout in addition to the allocate callout it provides.

The function takes this form:

```
void img_decode_release_f( _Uinptrt data,
                          img_info_t* info );
```

The arguments for this function are:

data Application data.

info A pointer to an `img_info_t` structure that needs to be released.

`img_decode_scanline_f`* *scanline_f*

A pointer to a function that's invoked to notify the application when a scanline has been decoded.

This callout is optional, although it's important if the *stride* member of *info* is set to 0. If you set stride to 0 and don't provide a scanline callout, the scanline data is overwritten by subsequent scanlines.

The function takes this form:

```
int img_decode_scanline_f( _Uinptrt data,
                           img_info_t *info,
                           int row );
```

The arguments for this function are:

data Application data.

info A pointer to an `img_info_t` describing the frame.

row Index of the scanline that has been decoded. Scanlines are numbered starting at 0 (topmost scanline) to (*h* - 1) where *h* represents the height of the image.

`_Uinptrt data`

User-defined data passed as an additional argument to callouts.

Classification:

Image library

See also:

`img_decode_frame()`,
`img_load_file()`

img_decode_frame()

Decode a frame

Synopsis:

```
int img_decode_frame( const img_codec_t*   codecs,
                      size_t              ncodecs,
                      const io_input_t*    input,
                      const img_decode_callouts_t* callouts,
                      img_info_t*         info );
```

Arguments:

<i>codecs</i>	A pointer to an array of img_codec_t handles providing a list of codecs to try. The function will try each codec in order until it finds one that validates the data in the stream.
<i>ncodecs</i>	The number of items in the <i>codecs</i> array.
<i>input</i>	A pointer to an io_input_t structure that describes how to access the data stream. Don't pass NULL for this value. This structure is filled in by <i>io_fd_init_input()</i> or <i>io_mem_init_input()</i> .
<i>callouts</i>	<p>A pointer to an img_decode_callouts_t structure that provides system callouts for the decoder. If you pass NULL for this value a set of default callouts is used that do the following:</p> <ul style="list-style-type: none">• accept the data in the first format that the decoder offers• store the decoded frame data in system RAM• ignore all warning messages
<i>info</i>	The address of an img_info_t structure to fill with information regarding the decoded frame.

Library:

`img`

Description:

This function decodes a frame.

Returns:

- ≥ 0 Success. The returned value is the index of the codec in the provided list which successfully decoded the frame. The index will be in the range 0 through (*ncodecs* - 1).
- 1 An error occurred.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

`img_decode_callouts_t`,
`img_info_t`,
`io_fd_init_input()`,
`io_input_t`,
`io_mem_init_input()`

img_info_t

Information describing a decoded frame

Synopsis:

```
typedef struct {
    _uint8 *data;
    unsigned stride;
    unsigned w;
    unsigned h;
    img_format_t format;
    unsigned npalette;
    img_color_t *palette;
    unsigned int flags;
    img_color_t transparency;
    int x;
    int y;
    unsigned interval;
} img_info_t;
```

Description:

The `img_info_t` structure describes a decoded frame. The members of the `img_info_t` structure include:

<i>data</i>	Pointer to the beginning of the image data. Frame data is a linear buffer containing <i>h</i> scanlines of <i>stride</i> bytes each, beginning with the topmost line of the image and moving down.
<i>stride</i>	The number of bytes per scanline, including data and padding.
<i>w, h</i>	The width and height of the image frame, in pixels.
<i>format</i>	The <code>img_format_t</code> format of the image's pixel data.
<i>npalette</i>	The number of colors in the image <i>palette</i> color table.
<i>palette</i>	The palette color table.

<i>flags</i>	<p>Flags indicating which of the “extended” fields in the structure are valid. Can be one or more of:</p> <p>IMG_TRANSPARENCY_VALID</p> <p>The <i>transparency</i> field is valid and the specified color within the image should be treated as transparent.</p> <p>IMG_POSITION_VALID</p> <p>The <i>x</i>, <i>y</i> fields are valid and the frame should be rendered at an offset of (<i>x</i>, <i>y</i>) from the baseline position of the image rendering. Typically used only by some multiframe formats.</p> <p>IMG_INTERVAL_VALID</p> <p>The <i>interval</i> field is valid and frame should be displayed for the specified number of milliseconds prior to rendering the next frame in the sequence. This field is typically valid only for some multiframe formats.</p>
<i>transparency</i>	<p>The transparency color. The format of this color corresponds to the format of the image data. This is only valid if the IMG_TRANSPARENCY_VALID flag bit is set.</p>
<i>x</i> , <i>y</i>	<p>The vertical and horizontal offset of the frame, in pixels. These fields are only valid if the IMG_POSITION_VALID flag bit is set.</p>
<i>interval</i>	<p>The duration (in milliseconds) that this frame should be displayed. This field is only valid if the IMG_INTERVAL_VALID flag bit is set.</p>

Classification:

Image library

`img_info_t`

See also:

img_lib_attach()

Initialize the image library

Synopsis:

```
int img_lib_attach( img_lib_t*  ilib );
```

Arguments:

ilib The address where the function stores a handle to the library.

Library:

img

Description:

This function initializes the image library.

Returns:

0 Success.
-1 An error occurred (*errno* is set).

Errors:

ENOMEM Insufficient memory to allocate structures.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	No

img_lib_attach()

See also:

img_lib_detach()

img_lib_detach()

Detach from the library

Synopsis:

```
void img_lib_detach( img_lib_t  ilib );
```

Arguments:

ilib The library handle filled in by *img_lib_attach()*. The handle will no longer be valid.

Library:

img

Description:

This function detaches from the image library and frees all associated resources.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

img_lib_attach()

img_load_file()

Decode a frame from a file on the filesystem

Synopsis:

```
int img_load_file( img_lib_t      ilib,
                  const char*    path,
                  const img_decode_callouts_t* callouts,
                  img_info_t*    info );
```

Arguments:

<i>ilib</i>	A handle for the image library, returned by <i>img_lib_attach()</i> .
<i>path</i>	The full path to the file from which the data can be read.
<i>callouts</i>	<p>A pointer to an img_decode_callouts_t structure that provides system callouts for the decoder. If you specify NULL for this value a set of default callouts is supplied that do the following:</p> <ul style="list-style-type: none">• accept the data in the first format that the decoder offers• store the image data into system RAM• ignore all warning messages
<i>info</i>	The address of an img_info_t structure the function fills in with information about the decoded frame.

Library:

img

Description:

This function decodes a frame from a file on the filesystem. This function decodes only the first frame encountered.

Returns:

- ≥ 0 Success. The returned value is the index of the codec that successfully decoded the frame. The index will be within 0 and (ncodecs - 1), inclusive. @ @ @ ??? @ @ @
- 1 An error occurred.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

`img_decode_callouts_t`,
`img_info_t`,
`img_lib_attach()`,

io_fd_init_input()

Initialize an input stream based on a standard UNIX fd

Synopsis:

```
int io_fd_init_input( io_input_t*      input,
                     io_input_destroy_f** destroy_f,
                     int               fd );
```

Arguments:

<i>input</i>	A pointer to an img_info_t structure to populate with data and callouts describing how to access the data stream.
<i>destroy_f</i>	The address of an io_destroy_f function pointer (see below). This function will use this memory to pass back a pointer to the function which should be called once all input is done and you need to release resources associated with <i>input</i> .
<i>fd</i>	A standard UNIX file descriptor that has been opened for reading. The image library assumes the encoded image data will be readily available by reading from this file descriptor.

Library:

img

Description:

This function initializes an input stream based on a standard UNIX fd.

The destroy function simply releases resources associated with the **img_info_t**. It doesn't actually close the file descriptor.

The function has this form:

```
typedef void() io_input_destroy_f(io_input_t *input)
```

The function should release resources associated with an input stream. You should only invoke such a function after you have finished reading from the associated input stream.

The function's parameters are:

input A pointer to the `io_input_t` structure describing the stream to be released.

Returns:

0 Success.

-1 An error occurred (*errno* is set).

Errors:

ENOMEM Insufficient memory to allocate structures.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

`io_input_t`,
`io_mem_init_input()`

io_input_t

Input stream encapsulation

Synopsis:

```
typedef struct {
    io_read_f *read_f;
    io_unread_f *unread_f;
    io_skip_f *skip_f;
    _Uinptrt data;
} io_input_t;
```

Description:

The **io_input_t** structure provides a list of functions for accessing a data input stream. The members of the **io_input_t** structure include:

read_f A function that reads bytes from the input stream into the specified buffer. Following a successful read, the file pointer should be moved to the byte following the data which was read.

The function takes this form:

```
typedef size_t() io_read_f(_Uinptrt data, void *buf, size
```

The arguments are:

data User-defined data describing your input stream context

buf Pointer to the buffer where the read data is to be stored

nbytes Number of bytes to read

This function should return the number of bytes read; *errno* should be set appropriately if the number of bytes read is less than *nbytes*.

unread_f A function that rewinds the input stream. A buffer containing the previously read characters is provided in the case where the underlying input source does not support backward seeking and therefore must buffer unread bytes.

The function takes this form:

```
typedef size_t() io_unread_f(_Uinptrt data, void *buf,
```

The arguments are:

data User-defined data describing your input stream context.

buf Pointer to a buffer containing the bytes that should be pushed back onto the stream, if the source does not support backward seeking.

nbytes Number of bytes to rewind.

This function should return the number of bytes rewound; *errno* should be set appropriately if the number of bytes rewound is less than *nbytes*.

skip_f This function advances the input stream, ignoring the bytes beng skipped.

The function takes this form:

```
typedef size_t() io_skip_f(_Uinptrt data, size_t nbytes
```

The arguments are:

data User-defined data describing your input stream context

nbytes Number of bytes to skip

This function should return the number of bytes skipped; *errno* should be set appropriately if the number of bytes skipped is less than *nbytes*.

`io_input_t`

data User-defined data describing your input stream context.

Classification:

Image library

See also:

img_decode_frame(),
io_fd_init_input(),
io_mem_init_input()

io_mem_init_input()

Initialize an input stream based on data contained in a memory buffer

Synopsis:

```
int io_mem_init_input( io_input_t*      input,
                      io_input_destroy_f** destroy_f,
                      const void*      buf,
                      size_t           nbuf );
```

Arguments:

<i>input</i>	A pointer to an img_info_t structure to populate with data and callouts describing how to access the data stream.
<i>destroy_f</i>	The address of an io_destroy_f function pointer (see below). This function will use this memory to pass back a pointer to the function which should be called once all input is done and you need to release resources associated with <i>input</i> .
<i>buf</i>	A pointer to the area in memory where the encoded image data begins.
<i>nbuf</i>	The number of bytes available to be read from the <i>buf</i> buffer.

Library:

img

Description:

This function initializes an input stream based on data contained in a memory buffer.

The destroy function simply releases resources associated with the **img_info_t**. It doesn't actually close the file descriptor.

The destroy function has this form:

```
typedef void() io_input_destroy_f(io_input_t *input)
```

io_mem_init_input()

The function should release resources associated with an input stream. You should only invoke such a function after you have finished reading from the associated input stream.

The function's parameters are:

input A pointer to the `io_input_t` structure describing the stream to be released.

Returns:

0 Success.

-1 An error occurred (*errno* is set).

Errors:

ENOMEM Insufficient memory to allocate structures.

Classification:

Graphics Framework

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

`io_input_t`,
`io_fd_init_input()`

Glossary



blit

An efficient way to copy an entire area of memory to another. Blitting from an off-screen context to an on-screen context can reduce the appearance of flicker as a scene is rendered compared to rendering directly to an on-screen context.

context

A structure that maintains draw attributes between function calls.

device

Any graphics hardware. Also called an adaptor.

display

Any graphics display. Some devices support more than one display.

layer

A surface displayed by a device. A device may support multiple layers. In the GF, you can assume a device has at least one layer, whether or not the hardware supports layering. Layers are numbered starting at 0, which is furthest from the viewer. Layers are combined for alpha blending and/or chroma keying.

offscreen memory

@@@ need def @@@

stride

The byte offset from the start of one line of bitmap data to the next.

surface

A surface is an area of memory. You can draw on a surface, blit to a surface, and so on. A surface may be "visible" (on-screen) or off-screen; the GF API does not make a distinction.

target

Applies to 3D support. A 3D target is a handle created via the GF API that can render OpenGL ES.