QNX® Momentics® 6.3.0

Driver Development Kit

*Graphics Devices*

*For targets running QNX® Neutrino® 6.3.0 or later*

© 2006, QNX Software Systems GmbH & Co. KG.
Contents

About the Graphics DDK ix
What you’ll find in this guide xi
Assumptions xi
Building DDKs xii

1 Introduction to Graphics Drivers 1
Graphics drivers 3

2 Writing a Graphics Driver 7
Overview 9
Sample drivers 9
The modules 10
The graphics driver subsystem 11
Font manager and render library 12
Modeswitching and enumeration 13
2D drawing 13
Offscreen memory manager 14
Video overlay control 14
Layer control 15
Surfaces 15
Viewports 16
Binding your driver to the graphics framework 16
Calling sequence 17
Conventions 18
Colors 18
Coordinate system  18
Core vs. context 2D functions  19
Context information  20
Supplying draw functions and software fallback  20
Patterns  22
Pixel formats  27
Pixel formats for layers  29

3  Debugging a Graphics Driver   33

A simple setup  35
Location of the Graphics DDK  35
Compiling shared objects  35
   Building for a particular OS or architecture  36
   Shipping modified libs with your product  37
   Setting the _LD_LIBRARY_ PATH environment variable  37
The graphics file  38
Making a debug version of a driver  38
Running the debug driver and setting a breakpoint  39

4  Graphics Driver API  41

devg_get_contextfuncs()  46
devg_get_corefuncs()  48
devg_get_memfuncs()  50
devg_get_miscfuncs()  52
devg_get_modefuncs()  54
devg_get_vcapfuncs()  56
devg_get_vdfs()  58
disp_2d_caps_t  60
disp_adapter_t  63
DISP_ADD_FUNC()  67
disp_crtc_settings_t  70
disp_draw_context_t  74
disp_draw_contextfuncs_t  81
5 Libraries 169
DISPUTIL library — utility functions 171
  Miscellaneous display driver functions 171
  PCI configuration access functions 177
  Memory manager functions 180
  Video memory management functions 184
  Video BIOS services (x86 only) 187
FFB library — 2D software fallback routines 193

Glossary 199

Index 205
List of Figures

Directory structure for this DDK. xiii
How a driver interacts with Photon. 4
Three surfaces. 24
Typical pattern. 25
Pattern filling a surface. 26
Memory layout. 140
About the Graphics DDK
What you’ll find in this guide

In this preface you’ll find:

- Assumptions
- Building DDKs

The following table may help you find information quickly:

<table>
<thead>
<tr>
<th>For information about:</th>
<th>See:</th>
</tr>
</thead>
<tbody>
<tr>
<td>How Photon uses graphics drivers</td>
<td>Introduction to Graphics Drivers</td>
</tr>
<tr>
<td>Graphics drivers</td>
<td>Writing Your Own Driver</td>
</tr>
<tr>
<td>The entry points your driver provides, and macros and data it uses</td>
<td>Graphics Driver API</td>
</tr>
<tr>
<td>Other useful functions</td>
<td>Libraries</td>
</tr>
<tr>
<td>How to debug a graphics driver</td>
<td>Debugging a Graphics Driver</td>
</tr>
<tr>
<td>Definitions of terms used in this guide</td>
<td>Glossary</td>
</tr>
</tbody>
</table>

You must use this DDK with QNX Neutrino 6.3.0 or later.

Assumptions

We assume you have a basic familiarity with graphics cards, concepts, and terminology (e.g. pixels, spans, blitting, alpha-blending, chroma-keying, and raster operations).

You’ll also need sufficient hardware documentation for your graphics chip in order to be able to program all the registers. A working knowledge of the C programming language is essential.
Building DDKs

You can compile the DDK from the IDE or the command line.

- To compile the DDK from the IDE:
  Please refer to the Managing Source Code chapter, and “QNX Source Package” in the Common Wizards Reference chapter of the IDE User’s Guide.

- To compile the DDK from the command line:
  Please refer to the release notes or the installation notes for information on the location of the DDK archives.

DDKs are simple zipped archives, with no special requirements. You must manually expand their directory structure from the archive. You can install them into whichever directory you choose, assuming you have write permissions for the chosen directory.

Historically, DDKs were placed in /usr/src/ddk_VERSION directory, e.g. /usr/src/ddk-6.2.1. This method is no longer required, as each DDK archive is completely self-contained.

The following example indicates how you create a directory and unzip the archive file:

```
# cd ~
# mkdir my_DDK
# cd my_DDK
# unzip /path_to_ddks/ddk-device_type.zip
```

The top-level directory structure for the DDK looks like this:
Directory structure for this DDK.
You must run:

. ./setenv.sh

before running make, or make install.

Additionally, on Windows hosts you’ll need to run the Bash shell (bash.exe) before you run the . ./setenv.sh command.

If you fail to run the . ./setenv.sh shell script prior to building the DDK, you can overwrite existing binaries or libs that are installed in $QNX_TARGET.

Each time you start a new shell, run the . ./setenv.sh command. The shell needs to be initialized before you can compile the archive.

The script will be located in the same directory where you unzipped the archive file. It must be run in such a way that it modifies the current shell’s environment, not a sub-shell environment.

In ksh and bash shells, All shell scripts are executed in a sub-shell by default. Therefore, it’s important that you use the syntax

. <script>

which will prevent a sub-shell from being used.

Each DDK is rooted in whatever directory you copy it to. If you type make within this directory, you’ll generate all of the buildable entities within that DDK no matter where you move the directory. all binaries are placed in a scratch area within the DDK directory that mimics the layout of a target system.

When you build a DDK, everything it needs, aside from standard system headers, is pulled in from within its own directory. Nothing that’s built is installed outside of the DDK’s directory. The makefiles shipped with the DDKs copy the contents of the prebuilt directory into the install directory. The binaries are built from the source using include files and link libraries in the install directory.
Chapter 1

Introduction to Graphics Drivers

In this chapter...

Graphics drivers 3
Graphics drivers

The graphics drivers are independent of the Photon microGUI; the driver that you supply is implemented as one or more shared objects (your choice) that can be used by Photon or by any other application requiring the services of a graphics driver.

You provide a set of well defined entry points, and the appropriate graphics system dynamically loads your driver and calls the entry points.

By way of example, this is how your driver interacts with Photon and the graphics driver subsystem, \texttt{io-graphics}, under QNX Neutrino:
How a driver interacts with Photon.

As you can see from the above diagram, a set of Photon infrastructure components are responsible for the interface to Photon:

Connector Presents the graphical region to Photon. This is the area that’s defined to be shown on the graphical screen.
Draw stream interpreter
Interprets Photon’s *draw stream* and decodes the graphical commands into scans, bitmaps, images, and fills, and passes them to the graphics driver. The interpreter also converts the draw stream from whatever endian format it’s in to native-endian format.

Render library
Converts complex shapes (such as circles) into lower-level drawing primitives (scans, bitmaps, images, and fills) that the graphics driver can handle.

Font manager
Converts textual information into bitmaps.

Rasterizer (FFB)
Converts lower-level drawing primitives into a raster format, using the Flat Frame Buffer (FFB) library.

Graphics driver
Your graphics driver, supplied as one or more shared objects.
Chapter 2
Writing a Graphics Driver

In this chapter...

Overview 9
Binding your driver to the graphics framework 16
Calling sequence 17
Conventions 18
This chapter describes how to write a graphics driver.

Overview

Before looking at the data structures and functions, it’s important to understand the “big picture” for the Photon Graphics Driver Development Kit.

The purpose of the Graphics DDK is to allow third parties to write accelerated drivers without requiring QNX Software Systems to become involved in doing the work.

Sample drivers

The Graphics DDK includes these sample drivers:

- a driver for IBM VGA-compatible adapters
- a driver for banked Super VGA adapters
- a generic VESA 2.00 linear frame buffer driver

The Professional Edition also includes:

- an accelerated driver for 3dfx VooDoo Banshee and VooDoo3 chipsets
- an accelerated driver for Intel 82810 (i810) chipsets
- an accelerated driver for Chips and Technologies 655xx, 690xx chipsets.

The 3dfx, Intel, and Chips and Technologies chipsets were chosen as the basis for the examples because the register-level programming documentation is available to anyone without signing a non-disclosure agreement.

These examples provide a good starting point for nearly any modern card. To write a new driver, you typically start with an existing driver (whichever is closest in terms of functionality to the chipset that you are targeting) and then modify the hardware-dependent pieces of the driver to drive your hardware.
Note that the driver framework includes a library called the FFB (Flat Frame Buffer). This library serves these main purposes:

- It provides a software fallback for drawing routines that can’t be accelerated in hardware.
- It serves as reference when implementing an accelerated draw function in your driver.

For every 2D drawing entry point in a graphics driver, there’s an equivalent software version in the FFB that takes identical parameters. Thus your driver isn’t required to provide any functions. When the graphics framework asks your driver to supply its 2D drawing functions, the driver can return a mixture of its own accelerated functions and software-implemented FFB functions.

For more information, see “FFB library — 2D software fallback routines” in the Libraries chapter.

**The modules**

Photon graphics drivers are implemented in a modular fashion. Graphics driver functionality is broken down into various groups. A graphics driver’s shared object may supply one or more groups of functionality. The shared object contains one module per functional group that it implements.

A graphics driver exposes its functionality by supplying a well-known entry point per module.

The modules currently defined are:

- Modeswitcher / mode enumerator
- 2D drawing module
- Offscreen memory manager
- Video overlay / scaler module
- Video capture / TV tuner module
Future modules can be implemented by defining a new well known entry point.

Because a graphics driver consists of one or more modules, it’s possible to package a complete driver solution that consists of multiple shared objects. For example:

- One shared object could contain an overlay scaler module for an external video scaler device, while another shared object could contain the 2D graphics driver and modeswitcher for the desktop display device. Photon could use the scaler and desktop devices together, with a separate shared object supplied to control each device. However, a single shared object typically controls a single integrated device with desktop graphics and video scaler functionality.

- If you want to write a generic modeswitcher (e.g. a modeswitcher to support any adapter with a VESA compliant BIOS), you could build the VESA BIOS modeswitcher as a single shared object and use it in conjunction with a shared object with a 2D module for a specific chipset, which might, or might not, have its own built-in modeswitcher.

It’s possible to load multiple shared objects such that there’s more than one instance of a given module type. In this case, the order in which to load the shared objects can be specified to the graphics subsystem; modules found in later-loaded shared objects are used in preference to those in earlier-loaded shared objects.

**The graphics driver subsystem**

With QNX Neutrino, the graphics driver subsystem consists of a main application and multiple shared objects. The main application is called `io-graphics` under QNX Neutrino, and is responsible for:

- connecting to the appropriate Photon server
- locating the correct set of shared objects to use for a particular set of hardware
• loading those shared objects and using them to fulfill the instructions encoded into the Photon draw stream.

For more information about starting \texttt{io-graphics}, see the QNX Neutrino \textit{Utilities Reference}.

Although the current implementation of \texttt{io-graphics} is limited to driving a single Photon graphical region, eventually it will simultaneously handle an arbitrary number of graphical regions and Photon servers.

\textbf{Font manager and render library}

There are many operations defined in the Photon high-level API that are extremely unlikely to be handled by any kind of graphics hardware. For example, a graphics card is unlikely to handle circles and scaled fonts. Even if a graphics card \textit{could} handle them, it would probably draw them in a card-dependent way that would be inconsistent with other cards.

The \texttt{io-graphics} subsystem solves these problems by using the render library and the font manager to turn high-level entities into lower-level primitives that all hardware can draw consistently.

The font manager is obviously used for rendering any sort of text objects. It’s currently designed to return raster style output that the driver draws as bitmaps or images, but eventually it will be enhanced to return vector information that future drivers could use directly.

The render library is used to simplify operations (other than fonts) that are defined in the Photon API, but that make little sense to implement in chipset-specific code. These operations include drawing circles as well as things like thick dashed lines.

The current implementation of the render library breaks down its output into scanlines, rectangles, bitmaps and images, but future plans call for it to be upgraded to return other kinds of data, such as lists of vertices representing a polygonal area to fill.

To this end, extra draw commands will be added to the 2D driver specification, but the main thing to remember is that you should have
to worry only about implementing the routines currently described in this guide.

**Modeswitching and enumeration**

*Enumeration* is the process of discovering what kind of video card you have, what its capabilities are. *Modeswitching* is the process of putting the video card into one of its supported modes.

With a standard installation of Photon for QNX Neutrino, cards are detected with the help of a generic utility called the enumerator (*enum-devices*). This utility uses a mapping file to detect video chips by their PCI or AGP vendor and device IDs. Depending on what video hardware it recognizes, if any, it builds an intermediate file that’s used by the *crttrap* utility, which is normally invoked when Photon is started, and performs the secondary phase of hardware detection. For more information about *enum-devices* and *crttrap*, see the QNX Neutrino *Utilities Reference*.

At the driver level, enumeration of the video modes supported by a card roughly corresponds to the VESA BIOS model. A list of numbers is returned corresponding to the modes the card can do, and a function is called for each of the mode numbers, returning information about that mode.

Switching to a given mode is accomplished by calling a driver entry point with one of the supported mode numbers.

**2D drawing**

2D drawing routines are the functions that actually produce or manipulate a two-dimensional image.

Operations that fall into this category include:

- filled rectangle routines
- scanline operation routines
- BLIT (BLock Image Transfer) routines
- hardware cursor routines
BLIT routines include operations that render an image that’s in system RAM into the framebuffer and routines that move a rectangular portion of the screen from one place to another.

**Offscreen memory manager**

Offscreen memory management routines are the code that allows *io-graphics* to manage multiple 2D objects, or surfaces, and to use the graphics driver to draw into various surfaces, whether the surfaces are on the visible display, or not.

Offscreen memory is the most important new API feature in Photon, and is what allows applications to take advantage of more advanced hardware features.

Most modern video cards have far more memory than is actually needed for the display. Most of them also allow the graphics hardware to draw into this unused memory, and then copy the offscreen object onto the visible screen, and vice-versa.

The offscreen management module deals with managing this memory. The routines in this module deal with allocating and deallocating 2D surfaces.

**Video overlay control**

Video overlay control routines manage the process of initializing and using video overlay hardware to do things like show MPEG content.

A video overlay scaler is a hardware feature that allows a rectangular area of the visible screen to be replaced by a scaled version of a different image. The prescaled video frames are typically stored in offscreen memory, and are fetched from memory and overlaid on top of the desktop display image in real time, by the overlay scaler.

Chroma keying is used to control what parts of the video frame are visible. Typically, the application picks a color to be the chroma-key color and draws a rectangle of this color where video content is to appear. When another application’s window is placed on top of the video playback application, the chroma-colored rectangle is obscured. Since the video hardware is programmed to display video content
only where the chroma-key color is drawn, video doesn’t show through where the chroma-colored rectangle is obscured.

Most of the routines in this module deal with letting applications know what kind of features the particular hardware supports and then setting the overlay up to cover a specific area of the screen and to accept an input stream of a particular size.

The rest of the overlay routines deal with implementing a protocol so that the application knows when a given frame has been dealt with and when it can send new frames to be displayed.

Layer control

Some display controllers allow you to transparently overlay multiple "screens" on a single display. Each overlay is called a layer. Layers can be used to combine independent display elements. Because overlaying is performed by the graphics hardware, this can be more efficient than rendering all of the display elements onto a single memory surface. For example, a fast navigational display can be implemented with the scrolling navigational map on a background layer and pop-up GUI elements, such as menus or a web browser, on the foreground layer.

Layer capabilities vary, depending on the display controller and the driver. Some display controllers don’t support layers. Different layers on the same display may have different capabilities. Layers are indexed per display, starting from 0, from back to front in the default overlay order.

Surfaces

The image on a layer is fetched from one or more offscreen contexts, also called surfaces. The layer format determines the number of surfaces needed by a layer. For example, a layer whose format is DISP_LAYER_FORMAT_ARGB888 requires one surface, while a layer whose format is DISP_LAYER_FORMAT_YUV420 requires three surfaces for a complete image.
**Viewports**

The source viewport defines a rectangular window into the surface data. This window is used to extract a portion of the surface data for display by the layer. The destination viewport defines a rectangular window on the display. This window defines where the layer displays its image. Scrolling and scaling (if supported by the layer) can be implemented by adjusting these viewports.

**Binding your driver to the graphics framework**

You must include the file `<display.h>`, which contains structures that you use to bind your driver to the graphics framework.

The graphics framework binds your graphics driver by calling `dlopen()` to load your driver, and then finding your entry point(s). The name of the entry point depends on which functional module(s) your shared object is providing; a single shared object can provide more than one functional block, hence the names are unique. The following table applies:

<table>
<thead>
<tr>
<th>Functional block</th>
<th>Name of function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 2D drawing functions</td>
<td><code>devg_get_corefuncs()</code></td>
</tr>
<tr>
<td>Context 2D drawing functions</td>
<td><code>devg_get_contextfuncs()</code></td>
</tr>
<tr>
<td>Miscellaneous 2D drawing functions</td>
<td><code>devg_get_miscfuncs()</code></td>
</tr>
<tr>
<td>Modeswitcher and layer control</td>
<td><code>devg_get_modefuncs()</code></td>
</tr>
<tr>
<td>Memory manager / frame buffer</td>
<td><code>devg_get_memfuncs()</code></td>
</tr>
<tr>
<td>Video overlay</td>
<td><code>devg_get_vidfuncs()</code></td>
</tr>
<tr>
<td>Video capture</td>
<td><code>devg_get_vcapfuncs()</code></td>
</tr>
</tbody>
</table>
The three functions, `devg_get_miscfuncs()`, `devg_get_corefuncs()`, and `devg_get_contextfuncs()` must be supplied in the same shared object — they constitute one group.

All the functions in the table have a similar structure: the graphics framework passes to each a pointer to a `disp_adapter_t` structure, a pointer to a set of functions (the type of which depends on the function being called), and a table size in `tabsize` (plus other arguments as appropriate).

The `disp_adapter_t` structure is the main “glue” that the graphics framework uses to hold everything together. It describes the adapter for the graphics hardware.

Your function is expected to fill the function table with all the available functions — this is how the graphics framework finds out about the functions supported by each functional block module.

Finally, the table size (`tabsize`) parameter indicates the size of the structure, in bytes. This is so that your initialization function doesn’t overwrite the area provided. You should use the `DISP_ADD_FUNC()` macro (defined in `<display.h>`) to add function pointers to the tables; it automatically checks the `tabsize` parameter.

The idea is that newer drivers that supply more functions will run properly with older versions of Photon that supply smaller function tables.

For more information, see the Graphics Driver API chapter.

### Calling sequence

The graphics framework calls your driver’s functions as follows:

```c
modeswitch->init ();
  modeswitch->set_mode ();
mem->init ();
  misc->init ();
...
// graphics functions get called here
...
```

November 2, 2006  Chapter 2 • Writing a Graphics Driver  17
// user requests a new mode; shut everything down
misc->fini ();
mem->fini ();
// at this point no more graphics functions will be called

modeswitch->set_mode ();
mem->init ();
misc->init ();

...  
// graphics functions get called here

...  
// shutdown of graphics drivers requested here
misc->fini ();
mem->fini ();
// at this point no more graphics functions will be called
modeswitch->fini ();

You can call the devg_get_* functions at any time.

Conventions

Before looking at the function descriptions, there are some conventions that you should be aware of.

Colors

RGB colors that are passed to 2D drawing functions are in the same pixel format as the surface that’s the target of the rendering operation. However, chroma-key colors are passed with the 24-bit true-color value stored in the least significant three bytes. However, if the target of the draw operation is a palette-based surface, the palette index that corresponds to the color-key value is also stored in the most significant byte.

Coordinate system

The coordinate (0, 0) is the top left of the displayed area. Coordinates extend to the bottom right of the displayed area. For example, if your graphics card has a resolution of 1280 (horizontal) by 1024 (vertical), the coordinates are (0, 0) for the top left corner, through to (1279, 1023), for the bottom right.
Order

The graphics framework passes only sorted coordinates to your driver. For example, if a draw-span function is called to draw a horizontal line from \((x1, y)\) to \((x2, y)\), the graphics framework always passes the coordinates such that \(x1 \leq x2\); it never passes them such that \(x1 > x2\).

Inclusiveness

All coordinates given are inclusive, meaning, for example, that a call to draw a line from \((5, 12)\) to \((7, 12)\) shall produce three pixels (that is, \((5, 12)\), \((6, 12)\), and \((7, 12)\)) in the image, and not two. Be careful to avoid this common coding mistake:

\[
\text{...}
// \text{WRONG!}
\text{for (x = x1; x < x2; x++) {}
\text{...}
\]

and instead use:

\[
\text{...}
// \text{CORRECT!}
\text{for (x = x1; x <= x2; x++) {}
\text{...}
\]

Core vs. context 2D functions

Core 2D drawing functions are typically expected to perform very simple operations. With the exception of pattern information, and information regarding the surface that’s being drawn to, the core functions can ignore most of the information in the draw context structure.

Context 2D drawing functions are expected to be able to handle more complex operations, such as chroma-keying, alpha-blending, and complex raster operations.

Core functions may be coded to handle a single pixel format. However, context functions must be prepared to handle drawing to any of the possible drawable surface types.
Another difference is that the context functions are allowed to make calls to core drawing functions, but not the other way around. The draw state structure contains a pointer to the function table that contains pointers to the core functions, which allows the context functions access to the core functions.

Since context functions can be expected to perform complex operations, it often makes sense to perform the operation in multiple stages. Future optimizations in the FFB may entail having the FFB versions of the context functions making calls to the core functions.

Context information

The graphics framework passes to every function, as its first argument, a pointer to a `disp_draw_context_t` structure that gives the function access to the draw state context block.

*If your functions modify any of the context blocks during their operation, they must restore them before they return.*

If the graphics framework modifies the context blocks between calls to the draw functions, it then calls the appropriate `update_*()` function to inform you which parts of the context data have been modified. The graphics framework doesn’t modify the context blocks while your function is running.

Supplying draw functions and software fallback

When a 2D draw function (i.e. a function that’s been supplied by the driver in either the `disp_draw_corefuncs_t` or the `disp_draw_contextfuncs_t` structure) is called, it’s expected to perform the draw operation correctly before returning (i.e. it may not fail).

The FFB (Flat Frame Buffer) library serves as a reference as to how 2D primitives are to be rendered. The draw functions that your driver supplies in its `devg_get_corefuncs()` and `devg_get_contextfuncs()` entry points are expected to carry out the draw operation as the FFB does.
With typical graphics hardware, not all primitives can be rendered using hardware, while producing the same resulting pixels in the targeted draw surface as the FFB does. In order to perform the draw operation correctly, it’s often necessary to call the FFB library functions to carry out a draw operation. This is called falling back on software.

Falling back on software can be achieved in these ways:

- Supply direct pointers to FFB functions in the `disp_draw_corefuncs_t` and `disp_draw_contextfuncs_t` structures. This is possible because, for each drawing entry point that your driver is expected to supply, there’s an equivalent version of the function in the FFB.

  This has pleasant implications for the driver writer: it means that none of the draw functions are mandatory, that is, the driver implementor can simply supply only FFB functions. Obviously this would lead to a far-from-optimal driver, since none of the draw functions would be taking advantage any hardware acceleration features.

  Or:

- Supply function pointers to one or more of your driver’s internal 2D rendering functions. When asked to perform a draw operation, the driver checks the contents of the draw context structure to determine whether or not it needs to fall back on the FFB functions.

  Typically, the `devg_get_corefuncs()` and `devg_get_contextfuncs()` routines function as follows:

1. Retrieve the software versions of the rendering functions by calling `ffb_get_corefuncs()` or `ffb_get_contextfuncs()` as appropriate.

2. Overwrite the function pointers within the function table with pointers to accelerated routines, using the `DISP_ADD_FUNC()` macro.
Using the above method makes your driver forward-compatible with future versions of the driver framework. If more draw functions are added to the specification, \texttt{io-graphics} will pass in a larger structure, and an updated FFB library will fill in software versions of the new functions. The graphics driver doesn’t need to be rebuilt or reshipped. At your discretion, you can update the graphics driver to supply accelerated versions of the new functions.

When a driver function is called to perform a draw operation, it typically checks members of the draw context structure, in order to determine if it needs to fallback on software. However, note that anytime the framework changes the draw context structure, it notifies the driver by means of one of the update-notification entry points.

As an optimization, your driver can perform these checks in its \texttt{update()} function (e.g. the \texttt{disp_draw_contextfuncs.t}’s \texttt{update_rop3()} function) and set flags to itself (in its private context structure) that indicate whether or not it can render various graphics primitives without falling back on software. This saves each and every context function from having to perform this work at runtime; it just checks the flags in a single test.

\section*{Patterns}

Patterns are stored as a monochrome $8\times 8$ array. Since many of the driver routines work with patterns, they’re passed in 8-bit chunks (an \texttt{unsigned char}), with each bit representing one pixel. The most significant bit (MSB) represents the leftmost pixel, through to the least significant bit (LSB) representing the rightmost pixel. If a bit is on (1), the pixel is considered \textit{active}, whereas if the bit is off (0), the pixel is considered \textit{inactive}. The specific definitions of active and inactive, however, depend on the context where the pattern is used.

As an example, the binary pattern \texttt{11000001} (hex \texttt{0xC1}) indicates three active pixels: the leftmost, the second leftmost, and the rightmost.

Note that functions that have \texttt{8x1} in their function names deal with a single byte of pattern data (one horizontal line), whereas functions
that have 8x8 in their function names deal with an 8 by 8 array (eight horizontal lines).

The pattern is circular, meaning that if additional bits are required of the pattern past the end of the pattern definition (for that line) the beginning of the pattern (for that line) is reused. For example, if the pattern is 11100000 and 15 bits of pattern are required, then the first eight bits come from the pattern (i.e. 11100000) and the next 7 bits once again come from the beginning of the pattern (i.e. 1110000) for a total pattern of 1110000111000. See “Pattern rotation,” below for more details about the initial offset into the pattern buffer.

A similar discussion applies to the vertical direction: if an 8-byte pattern is used and more pattern definitions are required past the bottom of the pattern buffer, the pattern starts again at the top.

Pattern rotation on a filled surface

In order to ensure a consistent look to anything that’s drawn with a pattern, you need to understand the relationships among the x and y coordinates of the beginning of the object to be drawn, the origin of the screen, and the pat_xoff and pat_yoff members of the disp_draw_contextfuncs_t context structure.
Three surfaces.

The diagram above shows three overlapping rectangles, representing three separate regions (for example, three pterm windows). If an application draws three rectangles within one Photon region, it’s up to the application to draw the three rectangles in the appropriate order — the discussion here about clipping applies only to separate regions managed by Photon.

If only the middle rectangle is present (i.e. there are no other rectangles obscuring it), your function to draw a rectangle with a pattern (e.g. draw_rect_pat8x8()), is called once, with the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>3</td>
</tr>
<tr>
<td>y1</td>
<td>2</td>
</tr>
<tr>
<td>x2</td>
<td>6</td>
</tr>
</tbody>
</table>

continued...
### Conventions

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_2$</td>
<td>6</td>
</tr>
<tr>
<td>$\text{pat}_x\text{off}$</td>
<td>0</td>
</tr>
<tr>
<td>$\text{pat}_y\text{off}$</td>
<td>0</td>
</tr>
</tbody>
</table>

Note that the $x_1, y_1, x_2$ and $y_2$ parameters are passed to the function call itself, while the $\text{pat}_x\text{off}$ and $\text{pat}_y\text{off}$ parameters are part of a data structure that the function has access to.

The values for $x_1, y_1, x_2$ and $y_2$ are reasonably self-explanatory; draw a rectangle from $(x_1, y_1)$ to $(x_2, y_2)$. The $\text{pat}_x\text{off}$ and $\text{pat}_y\text{off}$ values are both zero. This indicates that you should begin drawing with the very first bit of the very first byte of the pattern. If the pattern looks like this:

```
0 1 2 3 4 5 6 7
0 0xCC 0xCC
1 0x33 0x33
2 0xAA 0x55
3 0xF0 0xF0
```

*Typical pattern.*

then the rectangle drawn looks like this:
Pattern filling a surface.

If the values of the `pat_xoff` and `pat_yoff` arguments are anything other than zero (specifically, if these variables are functions of the location of the rectangle) then the pattern appears to creep along with the change of the location.

In the case where the other two rectangles are partially obscuring the rectangle of interest, Photon automatically transforms the single middle rectangle into a set of three rectangles, corresponding to the area that’s still visible (this is called clipping):

- (3, 2) to (4, 3)
- (3, 4) to (6, 4)
- (5, 5) to (6, 6)

This means that `draw_rect_pat8x8()` is called three times with these parameters:
<table>
<thead>
<tr>
<th>Parameter</th>
<th>First call</th>
<th>Second</th>
<th>Third</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>$y_1$</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$x_2$</td>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>$y_2$</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>$\text{pat} _\text{xoff}$</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$\text{pat} _\text{yoff}$</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Notice how the $\text{pat} \_\text{xoff}$ and $\text{pat} \_\text{yoff}$ pattern offset values are different in each call (first $(0, 0)$, then $(0, 2)$ and finally $(2, 2)$) in order to present the same window on the pattern regardless of where the rectangle being drawn begins. This is called pattern rotation.

### Pattern rotation on an image

To find the right bit in the pattern for a rectangle at point $(X, Y)$:

\[
\text{x\_index} = (x + \text{pat\_xoff}) \mod 8;
\]
\[
\text{y\_index} = (y + \text{pat\_yoff}) \mod 8;
\]

The BLIT functions take a $dx$ and $dy$ parameter, so you should substitute that in the equations above.

### Pixel formats

The pixel formats are defined below.

---

You aren’t expected to be able to render into the formats tagged with an asterisk (*) — these can only act as sources for operations, not as destinations. Therefore, these formats are never specified as parameters to `devg\_get\_corefuncs()`.

In any case, if you receive a pixel format that you don’t know what to do with (or don’t want to handle yourself), simply call `ffb\_get\_corefuncs()` to populate the function table with software rendering routines.
For RGB formats, the name indicates the layout of the color components. For example, for DISP_SURFACE_FORMAT_ARGB1555, the Alpha bit is stored in the most significant bit of the pixel, and the Blue component is stored in the least significant 5 bits.

DISP_SURFACE_FORMAT_MONO (*)
Each pixel has 1 bit (monochrome).

DISP_SURFACE_FORMAT_PAL8
Each pixel has 8 bits, and is selected from a palette of 256 (8-bit) colors.

DISP_SURFACE_FORMAT_ARGB1555
Each pixel has 16 bits, and the color components for red, green, and blue are 5 bits each (the top bit, 0x80 will be used for alpha operations in the future).

DISP_SURFACE_FORMAT_RGB565
Each pixel has 16 bits, and the color components for red and blue are 5 bits each, while green is 6 bits.

DISP_SURFACE_FORMAT_RGB888
Each pixel has 24 bits, and the color components for red, green, and blue are 8 bits each.

DISP_SURFACE_FORMAT_ARGB8888
Each pixel has 32 bits, and the color components for red, green, and blue are 8 bits each, with the other 8 bits to be used for alpha operations in the future.

DISP_SURFACE_FORMAT_PACKEDYUV_UYVY (*)
Effectively 16 bits per pixel, organized as UYVY, two pixels packed per 32-bit quantity.

DISP_SURFACE_FORMAT_PACKEDYUV_YUY2 (*)
Effectively 16 bits per pixel, organized as YUYV, two pixels packed per 32-bit quantity.
DISP_SURFACE_FORMAT_PACKED_YUV_YVYU (*)
   Effectively 16 bits per pixel, organized as YVYU, two pixels packed per 32-bit quantity.

DISP_SURFACE_FORMAT_PACKED_YUV_V422 (*)
   Same as YUY2, above.

DISP_SURFACE_FORMAT_YPLANE (*)
   Surface contains the Y component of planar YUV data.

DISP_SURFACE_FORMAT_UPLANE (*)
   Surface contains the U component of planar YUV data.

DISP_SURFACE_FORMAT_VPLANE (*)
   Surface contains the V component of planar YUV data.

DISP_SURFACE_FORMAT_BYTES (*)
   Surface is a collection of bytes with no defined format (for example, unallocated frame buffer memory).

You can use the DISP_BITS_PER_PIXEL() and DISP_BYTES_PER_PIXEL() macros in <display.h> to determine the number of bits or bytes of a packed surface format, including packed RGB and packed YUV format.

These macros don’t work for planar formats. Before using them, you should examine their definitions in <display.h> to see exactly what they do!

Pixel formats for layers

The pixel formats for layers are defined below.

DISP_LAYERVER_FORMAT_PAL8
   Each pixel has 8 bits.
Each pixel has 16 bits, with 1 bit for alpha, and 5 bits for red, green, and blue.

Each pixel has 16 bits, with 5 bits of red, 6 bits of green, and 5 bits of blue.

Each pixel has 24 bits, with 8 bits each of red, green, and blue.

Each pixel has 32 bits, with 8 bits of alpha, and 8 bits each of red, green, and blue.

Each pixel has 16 bits. Every 4 bytes of YUV colorspace data is arranged as YUYV. While the Y data uses one byte of data per pixel, the U and V subsampled data uses one byte of data each, shared between two pixels.

Each pixel has 16 bits. Every 4 bytes of YUV colorspace data is arranged as UYVY. While the Y data uses one byte of data per pixel, the U and V subsampled data uses one byte of data each, shared between two pixels.

Each pixel has 16 bits. Every 4 bytes of YUV colorspace data is arranged as YVYU. While the Y data uses one byte of data per pixel, the U and V subsampled data uses one byte of data each, shared between two pixels.

Each pixel has 16 bits. Every 4 bytes of YUV colorspace data is organized as VYUY. While the Y data uses one byte of data per pixel, the U and V subsampled data uses one byte of data each, shared between two pixels.
DISP_LAYER_FORMAT_YVU9

Three memory surfaces (the Y, U, and V planes) are needed to make a valid image. The Y plane index is 0, the U plane index is 1, and the V plane index is 2. The U and V planes are one quarter the width and height of the Y plane. One byte of U data and one byte of V data is subsampled across a 4x4 (16 pixel) grid. For every 16 pixels, 144 associated bits of data are created. This averages out to 9 bits per pixel.

DISP_LAYER_FORMAT_YV12

Three memory surfaces (the Y, U, and V planes) are needed to make a valid image. The Y plane index is 0, the U plane index is 1, and the V plane index is 2. The U and V planes are one half the width and height of the Y plane. One byte of U data and one byte of V data is subsampled across a 2x2 (4 pixel) grid. For every 4 pixels, 48 associated bits of data are created. This averages out to 12 bits per pixel.

DISP_LAYER_FORMAT_YVU420

Three memory surfaces (the Y, U, and V planes) are needed to make a valid image. The Y plane index is 0, the U plane index is 1, and the V plane index is 2. The U and V planes are one half the width of the Y plane. One byte of U data and one byte of V data is subsampled for every adjacent pair of Y samples. For every 2 pixels, 32 associated bits of data are created. This averages out to 16 bits per pixel.
Chapter 3

Debugging a Graphics Driver

In this chapter...

A simple setup 35
Location of the Graphics DDK 35
Compiling shared objects 35
Making a debug version of a driver 38
Running the debug driver and setting a breakpoint 39
A simple setup

Let’s assume you have two machines: a remote debug terminal (Machine A) and your QNX Neutrino target (Machine B).

The remote debug terminal, Machine A, can be running QNX Neutrino or any system that supports a telnet connection into Machine B. (Some prefer to run QNX Neutrino on Machine A as well.)

In order to support a telnet session on Machine B, be sure to run the inetd services (e.g. as root, type: inetd &).

Once you have a telnet session, you should be able to login to Machine B from Machine A and get a shell prompt.

As an alternative to telnet, you could use qtalk to communicate to both machines over a serial link.

Unless noted otherwise, you should run all the commands described in the following sections on Machine B, your QNX Neutrino target system.

Location of the Graphics DDK

The Graphics Driver Development Kit as downloaded (version 1.00) is in the form of a package, which should install to $ddk_install_dir/ddk-graphics$.

Compiling shared objects

After installation, you’ll first need to compile the shared objects needed by the graphics driver.

1. cd to the $ddk_install_dir/ddk-graphics$ directory.

2. Type:
   make
You can individually go in to each subdirectory (ffb, disputil, etc.) and make from the lowest level if you choose.

**Building for a particular OS or architecture**

You can easily specify the hardware platform (e.g. x86):

<table>
<thead>
<tr>
<th>To build for:</th>
<th>Use this command:</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 only</td>
<td>CPULIST=x86 make</td>
</tr>
<tr>
<td>PPC only</td>
<td>CPULIST=ppc make</td>
</tr>
</tbody>
</table>

The *libdisputil.so* and *libffb.so* objects that we created have slightly different symbol information than the original ones that exist in `/usr/lib`.

Note also that you need to add the extension “.2” to the files *libdisputil.so* and *libffb.so* in order for the loader to locate them.

In order to run our debug driver, we’ll need these newer libraries that we just compiled. We’ll need to compile against them as well as have them available in our library path so that they’re loaded in when we start our driver.

We recommend that you make a backup of the original *libdisputil.so* and *libffb.so* objects and then copy the newly compiled ones on top of the existing versions. For example:

```
    cd /usr/lib
    cp libdisputil.so.1 libdisputil.orig.so.1
    cp libffb.so.1 libffb.orig.so.1
    cp ddk_working_dir/prebuilt/cpu variant/libdisputil.so /usr/lib/libdisputil.so.1
    cp ddk_working_dir/prebuilt/cpu variant/lib/ffb/nto/x86/so/libffb.so /usr/lib/libffb.so.1
```
Note that the make environment used for the Graphics DDK is the same as the recursive make as described in the Conventions for Makefiles and Directories in appendix of the QNX Neutrino Programmer’s Guide.

Shipping modified libs with your product

If you wish to ship modified versions of the ffb or disputil libraries with your product, you should rename them so that they’re not confused with the QNX-supplied libraries.

You’ll also need to modify your driver’s common.mk, so that it’s linked against the renamed library. Generally, however, you link and test your driver against the QNX-supplied library binaries.

Setting the LD_LIBRARY_PATH environment variable

To make sure the loader can find all the binaries when Photon and the driver are started, you should set the LD_LIBRARY_PATH variable to point at their location. (Or, you could simply install them in /lib/dll, but using the environment variable is probably better for testing purposes).

For example, before starting Photon, type:

```
export LD_LIBRARY_PATH=/home/barney/test_drivers
```

Now, running `crttrap trap` should create a trap file with the entries required to have Photon use the sample drivers.

The command at the top line of the trap file is the command that’s run to start the graphics driver. In order to specify which mode is used when Photon starts, you may select a line for a different mode and place it at the top of the trap file.
The graphics file

Another important file is /etc/system/enum/devices/graphics. The system determines which particular driver to use with a particular chip, using the mappings in this graphics file.

If you want to add a new driver to the system, you need to add an entry to this file.

Making a debug version of a driver

In this example, we’ll make a debug version of the devg-banshee driver.

1. cd to the devg/banshee/nto/x86 directory. You’ll see a dll directory.

2. Make a dll.g directory at the same level as the dll directory.

3. Copy the Makefile from inside the dll directory to the dll.g directory.

Both dll and dll.g use the very same Makefile — the directory-naming convention is what tells make to compile with the -g debugging option.

4. cd to dll.g and type:

   make

   This should create a devg-banshee_g.so shared object.

We recommend that you copy the debug object to the same spot on disk as the non-debug object. This is okay because they have different names, and you’ll invoke them differently.

5. Copy the debug object:

   cp devg-banshee_g.so /lib/dll

   The gdb debug example assumes that the debug object resides under the /lib/dll directory.
Running the debug driver and setting a breakpoint

Now let’s try to run your debug driver (`devg-banshee_g.so`) and set a breakpoint on `banshee_init()`, which is a function within the `devg-banshee_g.so` shared object.

There’s a slight complication here: we can’t set the breakpoint under `gdb` until after the banshee DLL is loaded. What we want to do is freeze `io-graphics` after it has loaded `devg-banshee_g.so`, but before `banshee_init()` has been called.

Note how `io-graphics` gets the address of the `banshee_init()` function before it can call it: it obtains it via the `devg_get_modefuncs()` entry point. This means that `devg_get_modefuncs()` has to get called before `banshee_init()` (or any other modeswitcher entry point for that matter).

So in order to gain control under `gdb` before any of the modeswitcher entry points get called, we can add the following line of code to the `devg_get_modefuncs()` function:

```c
kill(0, SIGSTOP);
```

What we’re doing here is dropping a “stop” signal on our own process, `io-graphics`. Once `io-graphics` is in a stopped state, we can attach to it with `gdb`.

Now we’re ready to start Photon.

If you’re starting Photon manually, or via a custom script, you could launch `io-graphics` with the following command:

```
io-graphics -g1024x768x32 -d/devg-banshee_g.so -I0 -d0x121a,0x5 -R75
```

If you’re using the `ph` script (which launches `io-graphics` via `crttrap`) you could place the above line at the top of the file `/etc/system/config/graphics-modes`.

When `io-graphics` runs, it loads your debug driver, but when it hits the `kill()` function in `devg_get_modefuncs()`, it goes into a stopped state.
Running the debug driver and setting a breakpoint © 2006, QNX Software Systems GmbH & Co. KG.

From a telnet or serial session, type:

```
pidin | grep io-graphics
```

to find the process ID of io-graphics, so that we can attach to it with gdb.

You need to ensure that gdb looks in the same location for your driver shared object as the run timer loader (otherwise the symbol table in gdb won’t match the binary you’re trying to debug!)

You can do this by setting the solib-search-path variable inside gdb. Here’s an example for an x86 target:

```
(gdb) set solib-search-path /x86/lib:/x86/usr/lib:/x86/usr/dll:/home/barney/test_drivers
```

You could place the above command line in your .gdbinit file (note the leading period) so that it’s automatically executed when gdb starts. This file is located in your home directory.

Now we can start gdb, attach to io-graphics using the process ID that we got from pidin, load the symbol table from the debug driver, and set a breakpoint:

```
(gdb) attach 123456
(gdb) shared
(gdb) break banshee_init
(gdb) cont
```

The gdb debugger should regain control when io-graphics hits the breakpoint in banshee_init().
Chapter 4

Graphics Driver API
This chapter includes reference pages for the entry points your driver must provide, as well as for the macros and data structures you’ll use when writing a graphics driver:

`devg_get_contextfuncs()`  
Get a table of your driver’s context drawing functions

`devg_get_corefuncs()`  
Get a table of your driver’s core 2D drawing functions

`devg_get_memfuncs()`  
Get a table of your driver’s video memory-management functions

`devg_get_misclfunc()`  
Get a table of your driver’s miscellaneous drawing functions

`devg_get_modefuncs()`  
Get a table of your driver’s modeswitcher functions

`devg_get_vcapfuncs()`  
Get a table of your driver’s video capture functions

`devg_get_vidfuncs()`  
Get a table of your driver’s video-overlay functions

`disp_2d_caps_t`  
Description of your driver’s 2D capabilities

`disp_adapter_t`  
Description of the graphics hardware’s adapter

`DISP_ADD_FUNC()`  
Add a function pointer to a table

`disp_crtc_settings_t`  
CRT Controller settings
disp_draw_context_t
Draw context for a graphics driver

disp_draw_contextfuncs_t
Table of a driver’s context drawing functions

disp_draw_corefuncs_t
Table of a driver’s core drawing functions

disp_draw_miscfuncs_t
Table of miscellaneous drawing functions

disp_layer_query_t
Query the capabilities of a given layer

disp_memfuncs_t
Table of memory-management functions

disp_mode_info_t
Information for a display mode

disp_modefuncs_t
Table of your driver’s modeswitcher functions

disp_module_info_t
Driver module information

disp_surface_t
Description of a two-dimensional surface

disp_vcap_channel_caps_t
General capabilities of a video capture unit

disp_vcap_channel_props_t
Configurable properties of a video capture unit

disp_vcap_channel_status_t
Status of a video capture unit
disp_vcapfuncs_t
Table of video capture functions

disp_vid_alpha_t
A region of the video viewport to be blended with the desktop

disp_vid_channel_caps_t
General capabilities of a video scaler

disp_vid_channel_props_t
Configurable properties of a video scaler channel

disp_vidfuncs_t
Table of video overlay functions
**Synopsis:**

```c
int devg_get_contextfuncs (  
    disp_adapter_t *ctx,  
    disp_draw_contextfuncs_t *fns,  
    int tabsize) ;
```

**Description:**

A 2D driver must provide an entry point called `devg_get_contextfuncs()`. The graphics framework calls this function to get a table of your driver's context 2D drawing functions.

The arguments are:

- `ctx` A pointer to the `disp_adapter_t` structure that describes the graphics adapter.
- `fns` A pointer to a `disp_draw_contextfuncs_t` structure that this function must fill with pointers to your driver’s context drawing functions.
- `tabsize` The size of the table, in bytes.

Use the `DISP_ADD_FUNC()` macro to add function pointers to the table. It checks the `tabsize` argument and adds a function pointer only if there’s space for it in the table.

**Returns:**

- 0 Success.
- -1 An error occurred.

**Classification:**

Photon
Safety

<table>
<thead>
<tr>
<th>Interrupt handler</th>
<th>Not applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Thread</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

See also:

`devg_get_corefuncs()`, `devg_get_memfuncs()`, `devg_get_mscfuncs()`, `devg_get_modefuncs()`, `devg_get_vcapfuncs()`, `devg_get_vidfuncs()`,
`disp_adapter_t`, `DISP_ADD_FUNC()`, `disp_draw_contextfuncs_t`
**devg_get_corefuncs()**

Get a table of your driver’s core 2D drawing functions

**Synopsis:**

```c
int devg_get_corefuncs (disp_adapter_t *ctx,
        unsigned pixel_format,
        disp_draw_corefuncs_t *fns,
        int tabsize);
```

**Description:**

A 2D driver must provide an entry point called `devg_get_corefuncs()`. The graphics framework calls it to get a table of your driver’s core 2D drawing functions for a particular pixel format.

The arguments are:

- `ctx` A pointer to the `disp_adapter_t` structure that describes the graphics adapter.
- `pixel_format` The required pixel format. For more information, see “Pixel formats” in the Writing a Graphics Driver chapter.
- `fns` A pointer to a `disp_draw_corefuncs_t` structure that your driver must fill with pointers to its core 2D drawing functions.
- `tabsize` The size of the table, in bytes.

Use the `DISP_ADD_FUNC()` macro to add function pointers to the table. It checks the `tabsize` argument and adds a function pointer only if there’s space for it in the table.

**Returns:**

- 0 Success.
- -1 The driver doesn’t support the given pixel format, or an error occurred.
Classification:

Photon

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Thread</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

See also:

devg_get_contextfuncs(), devg_get_memfuncs(), devg_get_mscfuns(),
devg_get_modefuncs(), devg_get_vcapfuncs(), devg_get_yidfuncs(),
disp_adapter_t, DISP_ADD_FUNC(),
disp_draw_corefuncs_t
devg_get_memfuncs()

Get a table of your driver's video memory management functions

Synopsis:

```c
int devg_get_memfuncs (disp_adapter_t *ctx,
                        disp_memfuncs_t *funcs,
                        int tabsize);
```

Description:

A driver that contains a memory-management module must provide an entry point called `devg_get_memfuncs()`. The graphics framework calls it to get a table of your driver’s video memory management functions.

The arguments are:

- `ctx` A pointer to the `disp_adapter_t` structure that describes the graphics adapter.
- `funcs` A pointer to a `disp_memfuncs_t` structure that your driver must fill with pointers to its video memory management functions.
- `tabsize` The size of the table, in bytes.

Use the `DISP_ADD_FUNC()` macro to add function pointers to the table. It checks the `tabsize` argument and adds a function pointer only if there’s space for it in the table.

The video driver is responsible for keeping track of the allocation of video memory. The framework requests the allocation of 2D surfaces, and asks the driver to free them when they’re no longer in use. The onus is on the framework to free all surfaces; that is, for each surface that’s allocated, the framework explicitly asks the driver to free the surface at some point before the driver is unloaded.

In general, you don’t need to deal with all the complexities of writing a memory manager. Library routines in the DISPUTIL library can perform the bulk of the work. For more information, see the Libraries chapter.
devg_get_memfuncs()

Returns:

0  Success.

-1  An error occurred.

Classification:

Photon

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Thread</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

See also:

devg_get_contextfuncs(), devg_get_corefuncs(), devg_get_misgfuncs(),
devg_get_modefuncs(), devg_get_vcapfuncs(), devg_get_videofuncs(),
disp_adapter_t, DISP_ADD_FUNC(), disp_memfuncs_t
**Synopsis:**

```c
int devg_get_miscfuncs (disp_adapter_t *ctx,
                        disp_draw_miscfuncs_t *fns,
                        int tabsize);
```

**Description:**

A 2D driver must provide an entry point called `devg_get_miscfuncs()`. The graphics framework calls it to get a table of your driver’s miscellaneous 2D drawing functions.

The arguments are:

- `ctx` A pointer to the `disp_adapter_t` structure that describes the graphics adapter.
- `fns` A pointer to a `disp_draw_miscfuncs_t` structure that your driver must fill with pointers to its miscellaneous drawing functions.
- `tabsize` The size of the table, in bytes.

Use the `DISP_ADD_FUNC()` macro to add function pointers to the table. It checks the `tabsize` argument and adds a function pointer only if there’s space for it in the table.

**Returns:**

- 0 Success.
- -1 An error occurred.

**Classification:**

Photon
devg_get_miscfuncs()

Safety

<table>
<thead>
<tr>
<th></th>
<th>Not applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td></td>
</tr>
<tr>
<td>Signal handler</td>
<td></td>
</tr>
<tr>
<td>Thread</td>
<td></td>
</tr>
</tbody>
</table>

See also:

devg_get_contextfuncs(), devg_get_corefuncs(), devg_get_memfuncs(),
devg_get_modefuncs(), devg_get_vcapfuncs(), devg_get_yidfuncs(),
disp_adapter_t, DISP_ADD_FUNC(),
disp_draw_miscfuncs_t
Get a table of your driver's modeswitcher functions

Synopsis:

```c
int devg_get_modefuncs (disp_adapter_t *ctx,
                        disp_modefuncs_t *fns,
                        int tabsize);
```

Description:

A driver that contains a modeswitcher module must provide an entry point called `devg_get_modefuncs()`. The graphics framework calls it to get a table of your driver’s modeswitcher functions.

The arguments are:

- `ctx` A pointer to the `disp_adapter_t` structure that describes the graphics adapter.
- `fns` A pointer to a `disp_modefuncs_t` structure that your driver must fill with pointers to its modeswitcher functions.
- `tabsize` The size of the table, in bytes.

Use the `DISP_ADD_FUNC()` macro to add function pointers to the table. It checks the `tabsize` argument and adds a function pointer only if there’s space for it in the table.

Returns:

- 0 Success.
- -1 An error occurred.

Classification:

Photon
Safety

<table>
<thead>
<tr>
<th>Interrupt handler</th>
<th>Not applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Thread</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

See also:

devg_get_contextfuncs(), devg_get_corefuncs(), devg_get_memfuncs(),
devg_get_miscfuncs(), devg_get_ycapfuncs(), devg_get_vidfuncs(),
disp_adapter_t, DISP_ADD_FUNC(), disp_modefuncs_t
**devg_get_vcapfuncs()**  
Get a table of your driver's video capture functions

**Synopsis:**

```c
int devg_get_vcapfuncs (disp_adapter_t *ctx,
                       disp_vcapfuncs_t *funcs,
                       int tabsize);
```

**Arguments:**

- **ctx**: A pointer to the `disp_adapter_t` structure that describes the graphics adapter.
- **fns**: A pointer to a `disp_vcapfuncs_t` structure that your driver must fill with pointers to its video capture functions.
- **tabsize**: The size of the table, in bytes.

**Description:**

A driver that contains a video capture module must provide an entry point called `devg_get_vcapfuncs()`. The graphics framework calls it to get a table of your driver's video capture functions.

Use the `DISP_ADD_FUNC()` macro to add function pointers to the table. It checks the `tabsize` argument and adds a function pointer only if there’s space for it in the table.

**Returns:**

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

**Classification:**

- **Photon**
- **Safety**
  - Interrupt handler: Not applicable

continued . . .
Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Thread</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

See also:

devg_get_contextfuncs(), devg_get_corefuncs(), devg_get_memfuncs(), devg_get_mscifuncs(), devg_get_modefuncs(), devg_get_vidfuncs(), disp_adapter_t, DISP_ADD_FUNC(), disp_vcapfuncs_t
Get a table of your driver's video overlay functions

Synopsis:

```c
int devg_get_vidfuncs (disp_adapter_t *ctx,
                        disp_vidfuncs_t *funcs,
                        int tabsize);
```

Description:

A driver that contains a video overlay module must provide an entry point called `devg_get_vidfuncs()`. The graphics framework calls it to get a table of your driver's video overlay functions:

The arguments are:

- `ctx` A pointer to the `disp_adapter_t` structure that describes the graphics adapter.
- `funcs` A pointer to a `disp_vidfuncs_t` structure that your driver must fill with pointers to its video overlay functions.
- `tabsize` The size of the table, in bytes.

Use the `DISP_ADD_FUNC()` macro to add function pointers to the table. It checks the `tabsize` argument and adds a function pointer only if there's space for it in the table.

Returns:

- 0 Success.
- -1 An error occurred.

Classification:

Photon
Safety

<table>
<thead>
<tr>
<th>Interrupt handler</th>
<th>Not applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Thread</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

See also:

devg_get_contextfuncs(), devg_get_corefuncs(), devg_get_memfuncs(),
devg_get_miscfuncs(), devg_get_modefuncs(), devg_get_vcapfuncs(),
disp_adapter_t, DISP_ADD_FUNC(), disp_vidfuncs_t
**disp_2d_caps_t**

Description of your driver's 2D capabilities

**Synopsis:**

```c
#include <draw.h>

typedef struct disp_2d_caps {
    int size;
    unsigned accel_flags;
    unsigned flags;
    int min_stride;
    int max_stride;
    int stride_gran;
} disp_2d_caps_t;
```

**Description:**

The `disp_2d_caps_t` structure describes a driver's 2D capabilities. Your driver fills it in when the graphics framework calls the `get_2d_caps()` function defined in `disp_draw_miscfuncs_t`.

The `disp_2d_caps_t` structure includes the following members:

- **size**
  The size of this structure, in bytes.

- **accel_flags**
  Flags (defined in `<draw.h>`) describing which draw operations are performed with hardware acceleration:
  - DISP_2D_ACCEL_OPAQUE_BLIT — simple, opaque blit operations
  - DISP_2D_ACCEL_OPAQUE_FILL — simple, opaque fill operations
  - DISP_2D_ACCEL_MONO_PAT — simple draw operations with two colors and a pattern
  - DISP_2D_ACCEL_TRANS_PAT — simple draw operations with a transparency pattern
  - DISP_2D_ACCEL_SIMPLE_ROPS — copy, XOR, AND and OR with a pattern
  - DISP_2D_ACCEL_COMPLEX_ROPS — all 256 rop3 codes are accelerated
• DISP_2D_ACCEL_SRCALPHA_BLEND_GLOBAL — alpha blending with a global source factor is supported
• DISP_2D_ACCEL_SRCALPHA_BLEND_PIXEL — alpha blending with a per-pixel source factor is supported
• DISP_2D_ACCEL_SRCALPHA_BLEND_MAP — alpha blending with an alpha map as source is supported
• DISP_2D_ACCEL_DSTALPHA_BLEND_GLOBAL — alpha blending with a global destination factor
• DISP_2D_ACCEL_DSTALPHA_BLEND_PIXEL — alpha blending with a per-pixel destination factor
• DISP_2D_ACCEL_SRC_CHROMA — source image chroma keying is supported
• DISP_2D_ACCEL_DST_CHROMA — destination image chroma keying is supported

flags Flags (also defined in `<draw.h>`) describing miscellaneous properties of the 2D renderer:
• DISP_2D_SRC_DST_STRIDE_EQUAL — separate source and destination strides can’t be specified

min_stride The minimum stride that a 2D surface can be.
max_stride The maximum stride that a 2D surface can be.
strideGran A value that the stride of the 2D surface must be a multiple of.

Classification:
Photon
See also:

disp_draw_miscfuncs_t
Synopsis:

```c
#include <display.h>

typedef struct disp_adapter {
    
} disp_adapter_t;
```

Description:

This structure describes the graphics hardware’s adapter. There’s one instance of this structure for each device.

The `disp_adapter_t` structure includes some members that aren’t described here; don’t use or change any undocumented members.

Each driver module has its own context block — these are the members whose name ends with `_ctx`. Your driver can use these context blocks to store any data it requires.

The structures pointed to by the `modefuncs` and `memfuncs` structures contain the entry points of the memory manager and modeswitcher modules. Through these, it’s possible for one module to call functions within another. Since all entry points have access to the `disp_adapter_t` structure, each module’s entry point is always able to find its own private data structures.

The members of `disp_adapter_t` include:

- `int size` Size of this structure.
- `void *gd_ctx` Context block for graphics (2D) drivers.
- `void *ms_ctx` Context block for the modeswitch function group.
- `void *mm_ctx` Context block for the memory manager function group.
- `void *vid_ctx` Context block for the video overlay function group.
disp_adapter_t

**void** *vcap_ctx*  
Context block for the video capture function group.

**int** *bus_type*  
Identifies the type of bus interface that connects the device to the rest of the system:

- DISP_BUS_TYPE_UNKNOWN — the driver can’t determine the type of bus to which the device is connected, or the bus type isn’t one of the following.
- DISP_BUS_TYPE_PCI — the device is connected to a PCI bus.
- DISP_BUS_TYPE_AGP — the device is connected to an AGP bus.
- DISP_BUS_TYPE_ISA — device is connected to an ISA bus.
- DISP_BUS_TYPE_VL — device is connected to a VESA local bus.

**uintptr_t** *bus.pci.base[6]*  
An array of up to six physical base addresses that correspond to the device’s (PCI) aperture bases. This value is defined only if the *bus_type* is DISP_BUS_TYPE_PCI or DISP_BUS_TYPE_AGP.

**uintptr_t** *bus.pci.apsize[6]*  
An array of up to six aperture sizes that correspond to the device’s (PCI) aperture bases. This value is defined only if the *bus_type* is DISP_BUS_TYPE_PCI or DISP_BUS_TYPE_AGP.

**unsigned short** *bus.pci.pci_vendor_id*  
The PCI Vendor Identification number of the device that the driver interfaces with. This value is defined only if the *bus_type* is DISP_BUS_TYPE_PCI or DISP_BUS_TYPE_AGP.
disp_adapter_t

**unsigned short** `bus.pci.pci_device_id`

The PCI Device Identification number of the device that the driver interfaces with. This value is defined only if the `bus.type` is `DISP_BUS_TYPE_PCI` or `DISP_BUS_TYPE_AGP`.

**short** `bus.pci.pci_index`

The PCI Index of the device that the driver interfaces with. Together, the three fields `pci_vendor_id`, `pci_device_id`, and `pci_index` uniquely identify a hardware device in the system. This value is defined only if the `bus.type` is `DISP_BUS_TYPE_PCI` or `DISP_BUS_TYPE_AGP`.

**struct pci_dev_info **`*bus.pci.pci_devinfo`

A pointer to a structure containing extra PCI device information. For more details, see `pci_attach_device()` in the QNX Neutrino Library Reference. This value is defined only if the `bus.type` is `DISP_BUS_TYPE_PCI` or `DISP_BUS_TYPE_AGP`, and is present only under QNX Neutrino.

**unsigned caps**

Capabilities; a bitmap of the following values:

- `DISP_CAP_MULTI_MONI_MONITOR_SAFE` — the card can work with other VGA cards in the same system
- `DISP_CAP_2D_ACCEL` — the device provides 2D driver acceleration
- `DISP_CAP_3D_ACCEL` — the device provides 3D driver acceleration

The modeswitcher ORs in the multimonitor safe flag, if appropriate, and the other modules OR in their own capability flags if supported.

**FILE **`*dbgfile`

A file pointer that indicates to which file debugging output is to be sent. The `disp_perror()` function
and disp_printf() functions send their output to this file, unless its value is NULL.

```c
struct disp_modefuncs *modelfuncs
```

A pointer to a disp_modefuncs_t table containing the modeswitcher’s entry points. This member lets the 2D driver invoke certain modeswitcher functionality.

```c
struct disp_memfuncs *memfuncs
```

A pointer to a disp_memfuncs_t table containing the memory manager’s entry points. This member lets the 2D driver allocate and free memory surfaces.

The 2D driver is responsible for freeing any surfaces that it allocates on its own behalf. If the graphics framework asks the 2D driver to allocate any surfaces, the framework explicitly asks the driver to free them.

```c
int adapter_ram
```

The amount of video RAM on the card, in bytes.

```c
struct vbios_context *vbios
```

The handle set by vbios_register() that you need to pass to the other vbios_* functions. For more information, see the Libraries chapter.

**Classification:**

Photon

**See also:**

disp_memfuncs_t, disp_modefuncs_t
Synopsis:

```c
#include <display.h>

#define DISP_ADD_FUNC(tabletype, table, entry, func, limit) ...
```

Description:

When the graphics framework calls your driver’s entry points:

- `devg_get_contextfuncs()`
- `devg_get_corefuncs()`
- `devg_get_memfuncs()`
- `devg_get_miscfuncs()`
- `devg_get_modefuncs()`
- `devg_get_vcapfuncs()`
- `devg_get_vidfuncs()`

use the `DISP_ADD_FUNC()` macro to add your driver’s functions to the appropriate table:

- `disp_draw_contextfuncs_t`
- `disp_draw_corefuncs_t`
- `disp_draw_miscfuncs_t`
- `disp_memfuncs_t`
- `disp_modefuncs_t`
- `disp_vcapfuncs_t`
- `disp_vidfuncs_t`
The `DISP_ADD_FUNC()` macro adds the function only if there's room for it in the table. If you use this macro, newer drivers that supply more functions will run properly with older versions of Photon that supply smaller function tables.

The arguments are:

- `tabletype` The data type for the table (e.g. `disp_draw_contextfuncs_t`).
- `table` A pointer to the instance of the table that you want to add the function to.
- `entry` The name of the entry you want to set in the table (e.g. `draw_span`).
- `func` A pointer to the function that your driver provides.
- `limit` The size of the table, in bytes.

**Examples:**

```c
DISP_ADD_FUNC (disp_draw_corefuncs_t,
    &my_contextfuncs, blit1,
    &my_blit1_fn, tabsize);
```

**Classification:**

- Photon

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
See also:

`devg_get_contextfuncs()`, `devg_get_corefuncs()`, `devg_get_memfuncs()`,
`devg_get_miscfuncs()`, `devg_get_modefuncs()`, `devg_get_vcapfuncs()`,
`devg_get_vidfuncs()`, `disp_draw_contextfuncs_t`,
`disp_draw_corefuncs_t`, `disp_draw_miscfuncs_t`,
`disp_memfuncs_t`, `disp_modefuncs_t`, `disp_vcapfuncs_t`,
`disp_vidfuncs_t`
**disp_crtc_settings_t** © 2006, QNX Software Systems GmbH & Co. KG.

**CRT Controller settings**

**Synopsis:**

```c
#include <mode.h>

typedef struct disp_crtc_settings {
    short xres;
    short yres;
    short refresh;
    unsigned pixel_clock;

    uint8_t sync_polarity;
    uint8_t h_granularity;
    uint8_t v_granularity;

    short h_total;
    short h_blank_start;
    short h_blank_len;
    short h_sync_start;
    short h_sync_len;

    short v_total;
    short v_blank_start;
    short v_blank_len;
    short v_sync_start;
    short v_sync_len;

    unsigned flags;
} disp_crtc_settings_t;
```

**Description:**

The `disp_crtc_settings_t` structure contains the CRT Controller (CRTC) settings.
These members are used in conjunction with generic modes only (with the refresh member applicable to both generic and fixed modes):

- $h_{\text{granularity}}$
- $v_{\text{granularity}}$
- $\text{pixel}\_\text{clock}$
- $\text{sync}\_\text{polarity}$
- $h_{\text{total}}$
- $h_{\text{blank}\_\text{start}}$
- $h_{\text{blank}\_\text{len}}$
- $h_{\text{sync}\_\text{start}}$
- $h_{\text{sync}\_\text{len}}$
- $v_{\text{total}}$
- $v_{\text{blank}\_\text{start}}$
- $v_{\text{blank}\_\text{len}}$
- $v_{\text{sync}\_\text{start}}$
- $v_{\text{sync}\_\text{len}}$

For more information, see the get_modelist() function in the description of $\text{disp}\_\text{modefuncs}\_\text{t}$.

The members include:

- $xres$, $yres$ The horizontal and vertical resolution, in pixels.
- refresh The refresh rate (in Hz)
- pixel\_clock The pixel clock rate (in kHz)
disp_crtc_settings_t © 2006, QNX Software Systems GmbH & Co. KG.

**sync_polarity** Any combination of the following bits:

- DISP_SYNC_POLARITY_V_POS — vertical synchronization is indicated by a positive signal if this bit is on, else negative.
- DISP_SYNC_POLARITY_H_POS — horizontal synchronization is indicated by a positive signal if this bit is on, else negative.

Or, you can use the following manifest constants (composed of the bits from the above):

- DISP_SYNC_POLARITY_NN — both synchronization signals are negative.
- DISP_SYNC_POLARITY_NP — horizontal negative, vertical positive.
- DISP_SYNC_POLARITY_PN — horizontal positive, vertical negative.
- DISP_SYNC_POLARITY_PP — both synchronization signals are positive.

**h_granularity, v_granularity**

The horizontal and vertical granularity; the values of the other h_* and v_* members must be multiples of these.

**h_total, h_blank_start, h_blank_len, h_sync_start, h_sync_len**

Detailed monitor timings indicating the horizontal total, blanking start, length of blanking, horizontal sync start and length, given in units of h_granularity.

**v_total, v_blank_start, v_blank_len, v_sync_start, v_sync_len**

Detailed monitor timings indicating the vertical total, blanking start, length of blanking, horizontal sync start and length; given in units of lines.

**flags**

There are currently no flags defined.
disp_crtc_settings_t

Classification:

Photon

See also:

disp_modefuncs_t
 Synopsis:

```
#include <draw.h>

typedef struct disp_draw_context {
    int size;
    disp_adapter_t *adapter;
    void *gd_ctx;
    struct disp_draw_corefuncs *cfuncs;
    unsigned flags;
    disp_color_t fgcolor;
    disp_color_t bgcolor;
    uint8_t *pat;
    unsigned short pat_xoff;
    unsigned short pat_yoff;
    unsigned short pattern_format;
    unsigned short rop3;
    unsigned short chroma_mode;
    disp_color_t chroma_color0;
    disp_color_t chroma_color1;
    disp_color_t chroma_mask;
    unsigned alpha_mode;
    unsigned s_alpha;
    unsigned d_alpha;
    unsigned alpha_map_width;
    unsigned alpha_map_height;
    unsigned alpha_map_xoff;
    unsigned alpha_map_yoff;
    unsigned char *alpha_map;
    disp_surface_t *dsurf;
    char *sysram_workspace;
    int sysram_workspace_size;
} disp_draw_context_t;
```

Description:

The `disp_draw_context_t` structure defines the graphics driver’s draw context. The graphics framework passes this structure to of all the 2D drawing entry points.

The members include:
disp_draw_context_t

size
The size of the structure, in bytes.

adapter
A pointer to the disp_adapter_t structure.

gd_ctx
The 2D module’s private context structure.

cfuncs
A pointer to the disp_draw_corefuncs_t structure that lists the core functions for rendering into the currently targeted draw surface. This surface is of the type specified by the dsurf structure’s pixel_format member.

flags
Flags that indicate what kind of operations should be performed in all subsequent “context draw” functions. Selected from one or more of the following (bitmap):

- DISP_DRAW_FLAG_SIMPLE_ROP
- DISP_DRAW_FLAG_COMPLEX_ROP
- DISP_DRAW_FLAG_USE_ALPHA
- DISP_DRAW_FLAG_USE_CHROMA
- DISP_DRAW_FLAG_MONO_PATTERN
- DISP_DRAW_FLAG_TRANS_PATTERN

fgcolor
The foreground color.

bgcolor
The background color.

pat
Pattern buffer; see the description in “Patterns” (in the “Conventions” section of the Writing a Graphics Driver chapter), as well as the context functions draw_rect_pat8x8(), and draw_rect_trans8x8().

pat_xoff, pat_yoff
Horizontal and vertical offsets for the pattern to cause it to be shifted. For more information, see “Patterns” in the Writing a Graphics Driver chapter.
pattern_format  One of DISP_PATTERN_FORMAT_MONO_8x1 or  
DISP_PATTERN_FORMAT_MONO_8x8 (from draw.h).

rop3        Bitmapped raster operations, range between 0 and  
255 inclusive. See the memcpy_x.c file in the flat  
framebuffer library source for a sample implementation.

chroma_mode Selected from the following, see “Chroma mode  
bits,” below: either  
DISP_CHROMA_OP_SRC_MATCH or  
DISP_CHROMA_OP_DST_MATCH, and/or either  
DISP_CHROMA_OP_DRAW or  
DISP_CHROMA_OP_NO_DRAW. (In other words,  
SRC and DST are mutually exclusive, as are DRAW  
and NO_DRAW.)

chroma_color0  The chroma key; indicates the color to test on.

chroma_color1, chroma_mask  
Reserved; don’t examine or modify.

alpha_mode  A bitmask indicating alpha blending operations,  
see “Alpha mode bits,” below. For unrecognized  
alpha operations, call the supplied flat frame buffer  
functions.

s_alpha  The source alpha blending factor. For layers’  
blending configurations, multiplier 1, (M1), is the  
global alpha multiplier equivalent to s_alpha.

d_alpha  The destination alpha blending factor. For layers’  
blending configuration, multiplier 2 M2, is the  
global alpha multiplier equivalent to d_alpha.

alpha_map_width, alpha_map_height  
The width and height of the alpha map (below) in  
pixels.
The X and Y offset of the alpha map (below). See the discussion above in “Patterns” for more information.

The alpha mapping grid, whose size is determined by alpha_map_width and alpha_map_height (above). Each element of the map is one byte, corresponding to one pixel. If this member is NULL, there’s no alpha map. The stride here is equal to the width, i.e. one byte per element.

A pointer to a disp_surface_t structure that contains the definition of the currently targeted draw surface. All draw operations target this surface by default, unless parameters to the draw function explicitly override this.

A “scratch” area that the 2D driver and FFB library routines may use for temporary storage.

The size of the workspace, in bytes. If the driver wishes, it may reallocate the workspace in order to increase its size. This member should be updated to reflect the change in size. The driver should never decrease the size of the workspace.

When using an alpha map, blending factors come from the alpha_map, and not from the s_alpha or d_alpha members.

The following bits apply to the chroma mode flag mode, which performs a per-pixel test:

DISP_CHROMA_OP_SRC_MATCH

Perform match on source image.
DISP_CHROMA_OP_DST_MATCH
   Perform match on destination image.

DISP_CHROMA_OP_DRAW
   If match, draw pixel.

DISP_CHROMA_OP_NO_DRAW
   If match, don’t draw pixel.

Note that DISP_CHROMA_OP_SRC_MATCH and
DISP_CHROMA_OP_DST_MATCH are mutually exclusive, as are
DISP_CHROMA_OP_DRAW and DISP_CHROMA_OP_NO_DRAW.

Alpha mode bits (alpha_mode)

The (Group 1) alpha modes are:

DISP_ALPHA_M1_SRC_PIXEL_ALPHA
   Use the M1 multiplier for the Alpha component of the source
   pixels.

DISP_ALPHA_M1_DST_PIXEL_ALPHA
   Use the M1 multiplier for the Alpha component of the
   destination pixels.

DISP_ALPHA_M1_GLOBAL
   Use global blend factor from the M1 multiplier.

The (Group 2) alpha modes are:

DISP_ALPHA_M2_SRC_PIXEL_ALPHA
   Use the M2 multiplier for the Alpha component of the source
   pixels.

DISP_ALPHA_M2_DST_PIXEL_ALPHA
   Use the M2 multiplier for the Alpha component of the
   destination pixels.
DISP_ALPHA_M2_GLOBAL

Use global destination blend factor from the \textit{M2} multiplier.

\textbf{Alpha mode blending (source) operation}

The (Group 3) \textit{alpha mode} source operations are:

\begin{verbatim}
DISP_BLEND_SRC_M1_ALPHA
  Ms = M1
DISP_BLEND_SRC_ONE_MINUS_M1_ALPHA
  Ms = 1 -M1
DISP_BLEND_SRC_M2_ALPHA
  Ms = M2
DISP_BLEND_SRC_ONE_MINUS_M2_ALPHA
  Ms = 1 - M2
\end{verbatim}

\textbf{Alpha mode blending (destination) operation}

The (Group 4) \textit{alpha mode} destination operations are:

\begin{verbatim}
DISP_BLEND_DST_M1_ALPHA
  Md =M1
DISP_BLEND_DST_ONE_MINUS_M1_ALPHA
  Md = 1 -M1
DISP_BLEND_DST_M2_ALPHA
  Md = M2
DISP_BLEND_DST_ONE_MINUS_M2_ALPHA
  Md = 1 -M2
\end{verbatim}
For each pixel, the value of the blended pixel is derived by combining the source and the destination pixels with the multipliers, as shown in the following equation: \( P_d = P_s \times M_s + P_d \times M_d \), where \( P_d \) is the destination pixel value, and \( P_s \) is the source pixel value.

**Classification:**

Photon

**See also:**

disp_adapter_t, disp_draw_corefuncs_t, disp_surface_t
Synopsis:

```c
#include <draw.h>

typedef struct disp_draw_contextfuncs {
    void (*draw_span)(...);
    void (*draw_span_list)(...);
    void (*draw_rect)(...);
    void (*blit)(...);
    void (*update_general)(...);
    void (*update_color)(...);
    void (*update_rop3)(...);
    void (*update_chroma)(...);
    void (*update_alpha)(...);
} disp_draw_contextfuncs_t;
```

Description:

The `disp_draw_contextfuncs_t` structure is a table that your driver uses to define the context 2D drawing functions that it provides to the graphics framework. Your driver’s `devg_get_contextfuncs()` entry point must fill in this structure.

All functions in the context drawing structure must obey the members of the `disp_draw_context_t` structure (e.g. the current foreground color, and the alpha- and chroma-related members); check the `flags` to see which members of the context structure need to be obeyed. Note also that the core functions `update_pattern()` and `update_draw_surface()` affect the operation of these (the context) functions.

**draw_span()**

The graphics framework calls this function to draw a single, horizontal line from \((x1, y)\) to \((x2, y)\). The prototype is:

```c
void (*draw_span)(disp_draw_context_t *context,
                   int x1,
                   int x2,
                   int y);
```
**draw_span_list()**

The graphics framework calls this function to draw `count` number of horizontal lines as given by the arrays `x1`, `x2`, and `y`. The prototype is:

```c
void (*draw_span_list) (disp_draw_context_t *context, int count, int *x1, int *x2, int *y)
```

**draw_rect()**

The graphics framework calls this function to draw a rectangle from `(x1, y1)` to `(x2, y2)`. The prototype is:

```c
void (*draw_rect) (disp_draw_context_t *context, int x1, int y1, int x2, int y2)
```

**blit()**

The graphics framework calls this function to perform a blit. The prototype is:

```c
void (*blit) (disp_draw_context_t *context, disp_surface_t *src, disp_surface_t *dst, int sx, int sy, int dx, int dy, int width, int height)
```
disp_draw_contextfuncs_t

This function should move the pixels from the source surface (src) specified by the rectangle beginning at (sx, sy) for the specified size (length and height) to the destination surface beginning with the rectangle at (dx, dy) for the same size.

The disp_surface_t structures pointed to by src and dst may actually describe the same 2D surface. In this case, the driver must be prepared to handle the case where the source and destination areas intersect (i.e. this function must handle overlapping blits).

Fallback function: ffb_ctx_blit()

update_general()

The graphics framework calls this function to notify the driver that potentially all the members of the context structure have changed. The prototype is:

```c
void (*update_general) (disp_draw_context_t *context)
```

Fallback function: ffb_ctx_update_general()

update_color()

The graphics framework calls this function to notify the driver that the fg_color and bg_color members of the context structure could have changed. The prototype is:

```c
void (*update_color) (disp_draw_context_t *context)
```

Fallback function: ffb_ctx_update_color()

update_rop3()

The graphics framework calls this function to notify the driver that the rop3 member of the context structure could have changed. The prototype is:

```c
void (*update_rop3) (disp_draw_context_t *context)
```

Fallback function: ffb_ctx_update_rop3()
**update_chroma()**

The graphics framework calls this function to notify the driver that the chroma-related members of the `context` structure could have changed. This includes the `flags`, `chroma_mode` and `chroma_color0` members. The prototype is:

```c
void (*update_chroma) (disp_draw_context_t *context)
```

Fallback function: `ffb_ctx_update_chroma()`

**update_alpha()**

The graphics framework calls this function to notify the driver that the alpha-related members of the `context` structure could have changed. This includes the `flags`, `alpha_mode`, `d_alpha`, `alpha_map_width`, `alpha_map_height`, `alpha_map_xoff`, `alpha_map_yoff`, and `alpha_map` members. The prototype is:

```c
void (*update_alpha) (disp_draw_context_t *context)
```

Fallback function: `ffb_ctx_update_alpha()`

**Classification:**

Photon

**See also:**

`disp_draw_context_t`, `disp_surface_t`

“FFB library — 2D software fallback routines” in the Libraries chapter
Synopsis:

```
#include <draw.h>

typedef struct disp_draw_corefuncs {
    void (*wait_idle) (...);
    void (*hw_idle) (...);
    void (*draw_span) (...);
    void (*draw_span_list) (...);
    void (*draw_solid_rect) (...);
    void (*draw_line_pat8x1) (...);
    void (*draw_line_trans8x1) (...);
    void (*draw_rect_pat8x8) (...);
    void (*draw_rect_trans8x8) (...);
    void (*blit1) (...);
    void (*blit2) (...);
    void (*draw_bitmap) (...);
    void (*update_draw_surface) (...);
    void (*update_pattern) (...);
} disp_draw_corefuncs_t;
```

Description:

The `disp_draw_corefuncs_t` structure is a table that your driver uses to define the core 2D drawing functions that it provides to the graphics framework. Your driver’s `dev_get_corefuncs()` entry point must fill in this structure.

The core functions need to obey only the target information from the `disp_draw_context_t` structure, unless otherwise noted.

`wait_idle()`

The graphics framework calls this function when it needs to wait for the hardware to become idle. The prototype is:

```
void (*wait_idle) (disp_draw_context_t *context);
```
This function must not return until the hardware is idle. It’s safe to
directly access the draw target surface after this function returns.

Fallback function: ffb_wait_idle()

**hw_idle()**

The graphics framework calls this function to determine whether or
not the 2D hardware is idle. The prototype is:

```c
void (*hw_idle) (disp_adapter_t *context,
                  void *ignored)
```

This function should return a nonzero value if the 2D hardware is idle,
in which case it’s safe to directly access the draw target surface. If the
hardware isn’t idle, this function should return 0.

Fallback function: ffb_hw_idle()

**draw_span()**

The graphics framework calls this function to draw a single line. The
prototype is:

```c
void (*draw_span) (disp_draw_context_t *context,
                   disp_color_t color,
                   int x1,
                   int x2,
                   int y)
```

This function should draw a solid, opaque, horizontal line with the
given color from \((x1, y)\) to \((x2, y)\). It doesn’t use any pattern
information — the line is a single, solid color.

Fallback functions: ffb_draw_span_8(), ffb_draw_span_16(),
                  ffb_draw_span_24(), ffb_draw_span_32()

**draw_span_list()**

The graphics framework calls this function to draw a list of lines. The
prototype is:

```c
void (*draw_span_list) {
    disp_draw_context_t *context,
```
int count,
disp_color_t color,
int *x1,
int *x2,
int *y)

It’s identical to draw_span() above, except a list of lines is passed, with count indicating how many elements are present in the x1, x2, and y arrays.

Fallback functions: ffb_draw_span_list_8(), ffb_draw_span_list_16(), ffb_draw_span_list_24(), ffb_draw_span_list_32()

draw_solid_rect()

The graphics framework calls this function to draw a solid rectangle. The prototype is:

void (*draw_solid_rect) (  
disp_draw_context_t *context,
disp_color_t color,
int x1,
int y1,
int x2,
int y2)

It draws a solid, opaque rectangle with the given color (in color), from (x1, y1) to (x2, y2). It doesn’t use any pattern information — the rectangle is a single, solid color.

Fallback functions: ffb_draw_solid_rect_8(), ffb_draw_solid_rect_16(), ffb_draw_solid_rect_24(), ffb_draw_solid_rect_32()

draw_line_pat8x1()

The graphics framework calls this function to draw an opaque, patterned line. The prototype is:

void (*draw_line_pat8x1) (  
disp_draw_context_t *context,
disp_color_t bgcolor,
disp_color_t fgcolor,
int x1,
int x2,
int y,
uint8_t pattern)
It uses the pattern argument to determine the color to use for each pixel. An active bit (1) is drawn with the fgcolor color, and an inactive bit (0) is drawn with the bgcolor color. The pattern is consumed from left to right, with the most significant bit of the pattern being used for the first pixel drawn.

For more information, see “Patterns” in the Writing a Graphics Driver chapter.

Fallback functions: 

```c
ffb_draw_line_pat8x1_8(),
ffb_draw_line_pat8x1_16(), ffb_draw_line_pat8x1_24(),
ffb_draw_line_pat8x1_32()
```

**draw_line_trans8x1()**

The graphics framework calls this function to draw a transparent, patterned line. The prototype is:

```c
void (*draw_line_trans8x1) (  
disp_draw_context_t *context,
  disp_color_t color,
  int x1,
  int x2,
  int y,
  uint8_t pattern)
```

It uses the passed pattern to determine which pixels to draw. An active bit (1) is drawn with the color color, and an inactive bit (0) doesn’t affect existing pixels. The pattern is consumed from left to right, with the most significant bit of the pattern being used for the first pixel drawn.

Fallback functions: 

```c
ffb_draw_line_trans8x1_8(),
ffb_draw_line_trans8x1_16(), ffb_draw_line_trans8x1_24(),
ffb_draw_line_trans8x1_32()
```

**draw_rect_pat8x8()**

The graphics framework calls this function to draw an opaque, patterned rectangle. The prototype is:

```c
void (*draw_rect_pat8x8) {
```
It uses the draw context structure’s members \texttt{pat}, \texttt{pat\_xoff}, \texttt{pat\_yoff}, (but not \texttt{pattern\_format} as it’s already defined implicitly by virtue of this function being called). The pattern is used as described in the “Patterns” section of “Conventions,” in the Writing a Graphics Driver chapter. An active bit is drawn with the \texttt{fgcolor} color, and an inactive bit is drawn with the \texttt{bgcolor} color.

Fallback functions: \texttt{ffb\_draw\_rect\_pat8x8\_8()}, \texttt{ffb\_draw\_rect\_pat8x8\_16()}, \texttt{ffb\_draw\_rect\_pat8x8\_24()}, \texttt{ffb\_draw\_rect\_pat8x8\_32()}

\textbf{\textit{draw\_rect\_trans8x8()}}

The graphics framework calls this function to draw a transparent, patterned rectangle. The prototype is:

\begin{verbatim}
void (*draw_rect_trans8x8)(
    disp_draw_context_t *context,
    disp_color_t color,
    int x1, int y1, int x2, int y2)
\end{verbatim}

It uses the context structure’s members \texttt{pat}, \texttt{pat\_xoff}, \texttt{pat\_yoff}, (but not \texttt{pattern\_format} as it’s already defined implicitly by virtue of this function being called). The pattern is used as described in the “Patterns” section of “Conventions,” in the Writing a Graphics Driver chapter. An active bit is drawn with the \texttt{color} color, and an inactive bit doesn’t affect existing pixels.

Fallback functions: \texttt{ffb\_draw\_rect\_trans8x8\_8()}, \texttt{ffb\_draw\_rect\_trans8x8\_16()}, \texttt{ffb\_draw\_rect\_trans8x8\_24()}, \texttt{ffb\_draw\_rect\_trans8x8\_32()}
**blit1()**

The graphics framework calls this function to blit an area within a surface. The prototype is:

```c
void (*blit1)(disp_draw_context_t *context,
    int sx, int sy, int dx, int dy, int width, int height)
```

It blits within the surface defined by the context structure’s `dsurf` member (i.e., the source and destination are within the same surface). The contents of the area defined by the coordinates `(sx, sy)` for width `width` and height `height` are copied to the same-sized area defined by the coordinates `(dx, dy)`.

This function must be able to deal with overlapping blits, i.e. where the source area intersects with the destination area.

Fallback function: `fb_core_blit1()`

**blit2()**

The graphics framework calls this function to blit an area from one surface to another. The prototype is:

```c
void (*blit2)(disp_draw_context_t *context,
    disp_surface_t *src, disp_surface_t *dst,
    int sx, int sy, int dx, int dy, int width, int height)
```

This function blits from the source surface specified by the `disp_surface_t` structure pointed to by `src` to the destination surface specified by `dst`. The contents of the area defined by the
coordinates \((sx, sy)\) for width \(width\) and height \(height\), within the source surface, are transferred to the same-sized area defined by the coordinates \((dx, dy)\), within the destination surface.

The src and dst surfaces are guaranteed to be different, whereas in \textit{blit1()}, the operation takes place on the \textit{same} surface (as implied by the lack of a destination surface parameter). Your driver may need to check the surface flags to see where the src and dst images are (either in system memory or video memory) before performing the operation, since the draw engine may not be able to copy the image directly from system RAM.

Fallback function: \textit{ffbl}\_core\_blit2()

\textbf{draw\_bitmap()}

The graphics framework calls this function to draw an image in the destination surface by expanding the monochrome bitmap data in the buffer pointed to by “image”. The prototype is:

\begin{verbatim}
void (*draw_bitmap) (disp_draw_context_t *context,
    uint8_t *image,
    int sstride,
    int bit0_offset,
    disp_color_t fgcolor,
    disp_color_t bgcolor,
    int transparent,
    int dx,
    int dy,
    int width,
    int height)
\end{verbatim}

The \textit{sstride} argument specifies the stride of the source bitmap image, in bytes. The \textit{bit0_offset} 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.

The \textit{fgcolor} specifies the color to use when drawing a pixel when the corresponding bit in the source image is a 1. The \textit{bgcolor} argument specifies the color to use when drawing a pixel when the
For the transparent argument, a value of 0 specifies that the bitmap is a monochrome bitmap. Otherwise, the bitmap is a transparent bitmap. For transparent bitmaps, a bit value of 0 in the source image means that the corresponding pixel shouldn’t be drawn. For monochrome bitmaps, a bit value of 0 in the source image means that the corresponding pixel should be drawn using the color specified by bgcolor.

The point (dx, dy) specifies the pixel offset within the draw surface where the image is to be drawn, and the width and height arguments specify the size of the bitmap, in pixels.

Fallback functions: ffb_draw_bitmap_8(), ffb_draw_bitmap_16(), ffb_draw_bitmap_24(), ffb_draw_bitmap_32()

**update_draw_surface()**

The graphics framework calls this function whenever the members of the structure pointed to by the dsurf member of the context structure have changed. The prototype is:

```c
void (*update_draw_surface) (     disp_draw_context_t *context)
```

All subsequent 2D drawing operations should be performed on the surface specified by dsurf (except for functions where the target surfaces are specified explicitly by parameters to the function).

Fallback function: ffb_update_draw_surface()

**update_pattern()**

The graphics framework calls this function whenever pattern information in the context structure has changed. The prototype is:

```c
void (*update_pattern) (     disp_draw_context_t *context)
```
This information includes the `flags`, `pat`, `pat_xoff`, `pat_yoff` and `pattern_format` members.

Fallback function: `ffb_update_pattern()`

**Classification:**

Photon

**See also:**

`devg_get_corefuncs()`, `disp_draw_context_t`, `disp_surface_t`

“FFB library — 2D software fallback routines” in the Libraries chapter
Table of miscellaneous drawing functions

Synopsis:

```c
#include <draw.h>

typedef struct disp_draw_miscfuncs {
    int (*init)(...);
    void (*fini)(...);
    void (*module_info)(...);
    void (*set_palette)(...);
    int (*set_hw_cursor)(...);
    void (*enable_hw_cursor)(...);
    void (*disable_hw_cursor)(...);
    void (*set_hw_cursor_pos)(...);
    void (*flushrect)(...);
    void (*get_2d_caps)(...);
    int (*get_corefuncs_sw)(...);
    int (*get_contextfuncs_sw)(...);
} disp_draw_miscfuncs_t;
```

Description:

The `disp_draw_miscfuncs_t` structure is a table that your driver uses to define the miscellaneous drawing functions that it provides to the graphics framework. Your driver’s `devg_get_miscfuncs()` entry point must fill in this structure.

Note that if the driver doesn’t support hardware cursors, it should set all of the hardware cursor entry points in this structure to NULL. If any one of the hardware cursor entry points is non-NULL, your driver must provide all the hardware cursor entry points.
init()

The graphics framework calls this function to initialize the drawing hardware, allocate resources, and so on. The prototype is:

```c
int (*init) (disp_adapter_t *adapter,
            char *optstring )
```

For more information on where this function fits into the general flow, see “Calling sequence” in the Writing a Graphics Driver chapter.

fini()

The graphics framework calls this function to shut down the driver by shutting off hardware, freeing resources, and so on. The prototype is:

```c
void (*fini) (disp_adapter_t *adapter)
```

For more information on where this function fits into the general flow, see “Calling sequence” in the Writing a Graphics Driver chapter.

module_info()

The graphics framework calls this function to get information about the 2D driver module. The prototype is:

```c
void (*module_info) {
    disp_adapter_t *adapter,
    disp_module_info_t *module_info
}
```

This function must fill in the disp_module_info_t structure pointed to by module_info.

set_palette()

The graphics framework calls this function to set the palette. The prototype is:

```c
void (*set_palette) (disp_adapter_t *ctx,
                     int index,
                     int count,
                     disp_color_t *pal)
```
If the modeswitcher version of this function
(disp_modefuncs->set_palette) is present, it’s called instead (i.e. the modeswitcher function overrides this one).

set_hw_cursor()

The graphics framework calls this function to set the attributes of the hardware cursor. The prototype is:

```c
int (*set_hw_cursor) (disp_adapter_t *ctx,
    uint8_t *bmp0,
    uint8_t *bmp1,
    unsigned color0,
    unsigned color1,
    int hotspot_x,
    int hotspot_y,
    int size_x,
    int size_y,
    int bmp_stride)
```

The `hotspot` represents the “active” point of the cursor (e.g. the tip of the arrow in case of an arrow cursor, or the center of the crosshairs in case of a crosshair cursor).

If the cursor can’t be displayed properly, this function should return -1, which causes the framework to show a software cursor instead. For example, if `size_x` or `size_y` is too big for the hardware to handle, this function should return -1.

The cursor image itself is defined by two bitmaps. The two colors, `color0` and `color1` apply to the two bitmaps `bmp0` and `bmp1`. Both bitmaps have the same width (`size_x`), height (`size_y`), and stride (`bmp_stride`) values.

For a given pixel within the cursor image, a 0 in both bitmap locations means this pixel is transparent. A 1 in `bmp0` means draw the corresponding pixel using the color given by `color0`. A 1 in `bmp1` means draw the corresponding pixel using the color given by `color1`. If there’s a 1 in both bitmaps, then `color1` is to be used.
enable_hardware_cursor()

The graphics framework calls this function to make the cursor visible. The prototype is:

```c
void (*enable_hardware_cursor)(disp_adapter_t *ctx)
```

disable_hardware_cursor()

The graphics framework calls this function to make the cursor invisible. The prototype is:

```c
void (*disable_hardware_cursor)(disp_adapter_t *ctx)
```

set_hardware_cursor_pos()

The graphics framework calls this function to set the position of the hardware cursor. The prototype is:

```c
void (*set_hardware_cursor_pos)(disp_adapter_t *ctx, int x, int y)
```

Position the cursor such that the hotspot is located at (x, y) in screen coordinates.

flushrect()

The graphics framework calls this function to flush a modified area to the draw surface. The prototype is:

```c
void (*flushrect)(disp_draw_context_t *ctx, int x1, int y1, int x2, int y2)
```

The area defined by (x1, y1), (x2, y2) has been modified. Higher level software keeps track of which parts of the screen have been modified.
disp_draw_miscfuncts_t © 2006, QNX Software Systems GmbH & Co. KG.

(or “dirtied”), since the last call to this function, and this function is called to flush the dirtied area to the targeted draw surface.

This function is optional; supply it only if you wish to implement a “dirty rectangle” driver. The sample VGA driver provided with the DDK is a good example of a dirty rectangle driver.

Even though the frame buffer organization is non-linear, the VGA driver is still able to use the standard rendering functions from the FFB lib. It does this by keeping a “shadow” copy of the frame buffer in system RAM. The format of the shadow buffer is linear (8-bit palette format). The FFB renders into the shadow buffer, then this function is used to flush the appropriate area of the shadow buffer out to the planar VGA frame buffer, by converting the data from linear to planar format.

get_2d_caps()

The graphics framework calls this function to get information about the 2D capabilities of the graphics hardware’s accelerator. The prototype is:

```c
void (*get_2d_caps) (disp_adapter_t *ctx,
                     disp_surface_t *surface,
                     disp_2d_caps_t *caps)
```

The surface argument points to a disp_surface_t structure that describes a targetable 2D surface. Your driver must fill in the disp_2d_caps_t structure pointed to by caps with information about how 2D drawing to that surface is to be performed.

get_corefuncs_sw()

The graphics framework calls this function to get your driver’s core 2D drawing functions. The prototype is:

```c
int (*get_corefuncs_sw) (
    disp_adapter_t *adapter,
    unsigned pixel_format,
    disp_draw_corefuncs_t *fas,
    int tabsize );
```
This function should be similar to `devg_get_corefuncs()`, but `get_corefuncs_sw()` should get only the ones that are guaranteed to render into system RAM.

**get_contextfuncs_sw()**

The graphics framework calls this function to get your driver’s context 2D drawing functions. The prototype is:

```c
int (*get_contextfuncs_sw)(
    disp_adapter_t *adapter,
    disp_draw_contextfuncs_t *fns,
    int tabsize);
```

This function should be similar to `devg_get_contextfuncs()`, but `get_contextfuncs_sw()` should get only the ones that are guaranteed to render into system RAM.

**Classification:**

Photon

**See also:**

`devg_get_miscfuncs()`, `disp_2d_caps_t`, `disp_adapter_t`, `disp_modefuncs_t`, `disp_module_info_t`, `disp_surface_t`
disp_layer_query_t
Query capabilities of a given layer

Synopsis:
#include <display.h>

typedef struct {
    int size;
    unsigned pixel_format;
    unsigned caps;
    unsigned alpha_valid_flags;
    unsigned alpha_combinations;
    unsigned chromakey_caps;
    int src_max_height;
    int src_max_width;
    int src_max_viewport_height;
    int src_max_viewport_width;
    int dst_max_height;
    int dst_max_width;
    int dst_min_height;
    int dst_min_width;
    int max_scaleup_x;
    int max_scaleup_y;
    int max_scaledown_x;
    int max_scaledown_y;
} disp_layer_query_t *info;

Description:
The disp_layer_query_t structure defines the graphics driver’s
capabilities. The graphics framework passes this structure to the layer
query entry point.

The members of disp_layer_query_t include:

size The sizeof disp_surface_info_t structure.
disp_layer_query_t

format

The pixel format; see “Pixel formats for layers” in the Writing a Graphics Driver chapter.

caps

The flags that describe the capabilities of a layer for a given format:

- DISP_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. Filtering techniques may be used to reduce artifacts when scaling images.
- DISP_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.
- DISP_LAYER_CAP_PAN_SOURCE — the source viewpoint can be moved within the source image for this layer. If this flag isn’t set, the viewport can be located only at the top left corner of the image.
- DISP_LAYER_CAP_PAN_DEST — the destination viewpoint can be moved within the display area for this layer. If this flag isn’t set, the viewport can be located only at the top left corner of the image.
- DISP_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.
- DISP_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.
• DISP_LAYER_CAP_DISABLE — you can disable this layer and make it invisible. This layer can also be enabled.
• DISP_LAYER_CAP_SET_BRIGHTNESS — you can adjust the intensity of the displayed image.
• DISP_LAYER_CAP_SET_CONTRAST — you can adjust the contrast of the displayed image.
• DISP_LAYER_CAP_SET_SATURATION — you can adjust the color saturation of the displayed image.
• DISP_LAYER_CAP_ALPHA_WITH_CHROMA — chroma-keying and alpha-blending operations can be used simultaneously when you combine this layer with other layers.
• DISP_LAYER_CAP_MAIN_DISPLAY — according to the mode-switcher, this layer is the “main” layer. Mode-switcher calls such as set_display_offset() that don’t target a specific layer affect this layer. Typically, a driver for a “main” layer only reports that a single pixel format is available. This format is the same as the format that was selected when set_mode() was called.

alpha_valid_flags These are the flags that may be specified in the alpha_mode parameter to layer_set_blending().

alpha_combinations
alpha_combinations are used to define some capabilities of the alpha-blending hardware for this layer. Defined values are:

• DISP_ALPHA_BLEND_CAP_SPP_WITH_GLOBAL — source per-pixel alpha blending can be used in conjunction with a global alpha multiplier.
DISP_ALPHA_BLEND_CAP_GLOBAL_WITH_DPP — destination per-pixel blending can be used in conjunction with a global alpha multiplier.

DISP_ALPHA_BLEND_CAP_SPP_WITH_DPP — source per-pixel alpha blending can be used in conjunction with destination per-pixel alpha blending.

**chromakey_caps**

These flags define some capabilities of the chroma-key hardware for this layer. Defined values are:

- **DISP_LAYER_CHROMAKEY_CAP_SRC_SINGLE** — chroma-keying based on an exact match between the source-pixel value and a single key color is supported.

- **DISP_LAYER_CHROMAKEY_CAP_DST_SINGLE** — chroma-keying based on an exact match between the destination-pixel value and a single key color is supported.

- **DISP_LAYER_CHROMAKEY_CAP_SHOWTHROUGH** — a 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.

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

*src_max_height*  
The maximum height of the source image that can be displayed. The surface to be displayed may be taller than this value. When this is the case, it isn’t possible to display part of the image at the bottom of the surface.

*src_max_width*  
The maximum width of the source image that can be displayed. The surface to be displayed may be wider than this value. When this is the case, it isn’t possible to display part of the image at the right of the surface.

*src_max_viewport_height*  
Maximum height of the viewport into the source image. The layer isn’t capable of fetching an image area for a display that’s taller than this value.

*src_max_viewport_width*  
Maximum width of the viewport into the source image. The layer isn’t capable of fetching an image area for a display that’s wider than this value.

*dst_max_height*  
Maximum height of the destination viewport on the display.

*dst_max_width*  
Maximum width of the destination viewport on the display.

*dst_min_height*  
Minimum height of the destination viewport on the display.

*dst_min_width*  
Minimum width of the destination viewport on the display.
**disp_layer_query_t**

- **max_scaleup_x**: Maximum scaling factor for image upscaling in the horizontal direction. A value of 1 means upscaling can’t be performed. A value < 1 is invalid. A value > 1 means that the destination viewport width can be up to `max_scaleup_x` times the source viewport width.

- **max_scaleup_y**: Maximum scaling factor for image upscaling in the vertical direction. A value of 1 means upscaling can’t be performed. A value < 1 is invalid. A value > 1 means that the destination viewport height can be up to `max_scaleup_y` times the source viewport height.

- **max_scaledown_x**: Maximum scaling factor for image downscaling in the horizontal direction. A value of 1 means downscaling can’t be performed. A value < 1 is invalid. A value > 1 means that the source viewport width can be up to `max_scaledown_x` times the destination viewport width.

- **max_scaledown_y**: Maximum scaling factor for image downscaling in the vertical direction. A value of 1 means downscaling can’t be performed. A value < 1 is invalid. A value > 1 means that the source viewport height can be up to `max_scaledown_y` times the destination viewport height.

**Classification:**

Photon
Table of memory management functions

Synopsis:

```
#include <vmem.h>

typedef struct disp_memfuncs {
  int (*init) (...);
  void (*fini) (...);
  void (*module_info) (...);
  int (*reset) (...);
  int (*alloc_surface) (...);
  int (*alloc_layer_surface) (...);
  int (*free_surface) (...);
  int (*mem_avail) (...);
} disp_memfuncs_t;
```

Description:

The `disp_memfuncs_t` structure is a table that your driver uses to define the memory management functions that it provides to the graphics framework. Your driver’s `devg_get_memfuncs()` entry point must fill in this structure.

`init()`

The graphics framework calls this function to initialize the memory manager. The prototype is:

```
int (*init) (disp_adapter_t *adapter,
             char *optstring)
```

The graphics framework calls this function before any of the other functions in the `disp_memfuncs_t` structure. For more information on where this function fits into the general flow, see “Calling sequence” in the Writing a Graphics Driver chapter.

`fini()`

The graphics framework calls this function to shut down the memory management module, freeing any resources that it allocated. The prototype is:
For more information on where this function fits into the general flow, see “Calling sequence” in the Writing a Graphics Driver chapter.

**module_info()**

The prototype is:

```c
int (*module_info) (disp_adapter_t *adapter,
                    disp_module_info_t *info)
```

This function must fill in the `disp_module_info_t` structure pointed to by `module_info`.

**reset()**

The graphics framework calls this function to reset the memory management module to its initial state. The prototype is:

```c
int (*reset) (disp_adapter_t *adapter,
              disp_surface_t *surf)
```

**alloc_surface()**

The graphics framework calls this function to allocate video memory to contain a 2D surface. The prototype is:

```c
disp_surface_t * (*alloc_surface) (disp_adapter_t *adapter,
                                   int width,
                                   int height,
                                   unsigned format,
                                   unsigned flags,
                                   unsigned user_flags)
```

The size of the surface is `width` pixels by `height` scanlines. This function must allocate a `disp_surface_t` structure and initialize it to describe the allocated surface memory. Set the structure’s `adapter` member to point to the driver’s `disp_adapter_t` structure.
If there isn’t enough video RAM, your driver can allocate the surface from system RAM, but when the driver works with surfaces, it must use the flags defined for `disp_surface_t` to determine where the surfaces are defined. The driver must also be able to work with surfaces that aren’t all in video RAM.

The function must return a pointer to the allocated `disp_surface_t` and must ensure that the memory it allocates conforms to the surface properties requested by the `flags` parameter. This implies that when the function returns, any flags set in `flags` are also set in the `flags` member of the returned surface.

There are currently no user flags defined; your driver should ignore the `user_flags` argument.

### alloc_layer_surface()

The graphics framework calls this function to allocate video memory to a layer surface. The prototype is:

```c
disp_surface_t * (*alloc_layer_surface) (  
  disp_adapter_t *adapter,  
  int dispno,  
  int width,  
  int height,  
  unsigned sflags,  
  unsigned hint_flags,  
  int layer_idx,  
  unsigned fmt_idx,  
  unsigned surface_idx );
```

The size of the buffer to be allocated is specified in pixels, by the `width` and `height` arguments. This function must allocate a `disp_surface_t` structure and initialize it to describe the allocated memory surface. Set the structure’s `adapter` member to point to the driver’s `disp_adapter_t` structure.

In the case of planar YUV data, the width and height argument would be the same, regardless of whether a Y, a U, or a V plane were being allocated for a given image. It is up to the driver to know how big the actual buffer size which needs to be allocated, should be.
The function must return a pointer to the allocated `disp_surface_t` and must ensure that the memory it allocates conforms to the surface properties requested by the `sflags` parameter. This implies that when the function returns, any flags set in `sflags` are also set in the `flags` member of the returned surface.

The `sflags` argument is one of the surface flags as defined in the `disp_surface_t` section. However, the following flags have no meaning here:

- DISP_SURFACE_DISPLAYABLE
- DISP_SURFACE_SCALER_DISPLAYABLE

The `fmt_idx` and `surface_idx` arguments select the data format of the surface. Some layer formats split the image components across multiple buffers. A planar YUV surface, for example, requires three buffers to store a valid image. The `surface_idx` argument specifies whether a Y, U, or V buffer gets allocated.

The `fmt_idx` argument selects the data format of the memory to be allocated.

This function should return a pointer to a structure describing the allocated memory surface, or NULL to indicate failure.

The only valid value for `hint_flags` is 0.

### free_surface()

The graphics framework calls this function to free the video memory associated with the `disp_surface_t` structure that `surf` points to, as well as freeing the structure itself. The prototype is:

```c
int (*free_surface) (disp_adapter_t *adapter,
                     disp_surface_t *surf)
```
mem_avail()

The graphics framework calls this function to determine how much memory is available. The prototype is:

```c
int (*mem_avail) (disp_adapter_t *adapter,
                  unsigned sflags)
```

This function returns the amount of video memory available, in bytes. However, the memory that’s reported as being available must conform to the surface properties specified by `sflags`.

Classification:

Photon

See also:

`devg_get_memfuncs()`, `disp_adapter_t`, `disp_mode_info_t`, `disp_surface_t`
disp_mode_info_t
Information for a display mode

Synopsis:

#include <mode.h>

typedef struct disp_mode_info {
    short size;
    disp_mode_t mode;
    int xres, yres;
    unsigned pixel_format;
    unsigned flags;
    unsigned crtc_start_gran;
    unsigned caps;
    union {
        struct {
            short refresh [DISP_MODE_NUM_REFRESH];
        } fixed;
        struct {
            int min_vfreq, max_vfreq;
            int min_hfreq, max_hfreq;
            int min_pixel_clock;
            int max_pixel_clock;
            uint8_t h_granularity;
            uint8_t v_granularity;
            uint8_t sync_polarity;
        } generic;
    } u;
    int num_colors;
    unsigned crtc_pitch_gran;
} disp_mode_info_t;

Description:

The disp_mode_info_t structure holds information about a display mode. Your driver fills in this structure when the graphics framework calls the get_modeinfo function defined in the disp_modefuncs_t structure.

The members of disp_mode_info_t include:

size The size of this structure.
`disp_mode_info_t` © 2006, QNX Software Systems GmbH & Co. KG.

- **mode**: The unique mode ID; see the `get_modelist` function defined in the `disp_modefuncs_t` structure.

- **xres, yres**: The display dimensions, in pixels.

- **pixel_format**: The frame buffer’s pixel format. For more information, see “Pixel formats” in the Writing a Graphics Driver chapter.

- **flags**: Flags that specify various attributes of this mode, selected from the following:
  - `DISP_MODE_TVOUT` — this mode drives a TV, and not a monitor.
  - `DISP_MODE_TVOUT_OR_MONITOR` — this mode can drive a TV or a monitor, but not both simultaneously.
  - `DISP_MODE_TVOUT_WITH_MONITOR` — this mode can drive a TV and a monitor simultaneously.
  - `DISP_MODE_TVOUT_OVERSCAN` — the overscan goes beyond the edge of the TV (i.e. there are no borders at the edges).
  - `DISP_MODE_TVOUT_NTSC` — this mode generates NTSC format video signal.
  - `DISP_MODE_TVOUT_PAL` — this mode generates PAL format video signal.
  - `DISP_MODE_TVOUT_SECAM` — this mode generates SECAM format video signal.
  - `DISP_MODE GENERIC` — this mode is a generic mode. That is, the mode has a given `pixel_format` but can handle any resolution or refresh rate, within certain constraints. If this flag is set, the members of the `u.generic` structure are defined. Otherwise, the members of the `u.fixed` structure are defined.
Note that there’s a macro, 
\texttt{DISP_TVOUT_STANDARD()} that returns just the type of output (PAL, NTSC, SECAM).

\texttt{crtc\_start\_gran} The granularity of the frame buffer base address. Values passed in the \texttt{offset} parameter to the \texttt{set\_display\_offset()} function in the \texttt{disp\_modefuncs\_t} structure are multiples of this value.

\texttt{caps} The list of available features:

- \texttt{DISP\_MCAP\_SET\_DISPLAY\_OFFSET} — the display controller can point to different areas of the video RAM. This indicates that its offset into video RAM isn’t hard-coded, meaning that it can perform double-buffering operations.
- \texttt{DISP\_MCAP\_DPMS\_SUPPORTED} — the display supports DPMS (if this bit set), else no support.
- \texttt{DISP\_MCAP\_VIRTUAL\_PANNING} — the display supports virtual resolutions that are larger than the physical display, and supports “panning” of the physical viewport into the virtual display.

\texttt{u\_fixed\_refresh} An array of possible refresh rates (in Hz) for this mode. The size of this array is given by \texttt{DISP\_MODE\_NUM\_REFRESH} (i.e. it’s the maximum number of refresh rates that can be supported for a given mode).

\texttt{u\_generic\_min\_vfreq}, \texttt{u\_generic\_max\_vfreq} The monitor vertical frequency limits, in Hz.

\texttt{u\_generic\_min\_hfreq}, \texttt{u\_generic\_max\_hfreq} The monitor horizontal frequency limits, in kHz.

\texttt{u\_generic\_min\_pixel\_clock}, \texttt{u\_generic\_max\_pixel\_clock} The pixel clock limits, in kHz.
The horizontal granularity; X resolution must be a multiple of this.

The vertical granularity; Y resolution must be a multiple of this.

Defined only if the display format is palette-based. It specifies the maximum palette index that the display can handle. For example, the example 16-color VGA driver sets this to 16.

**Classification:**

Photon

**See also:**

`disp_modefuncs_t`
disp_modefuncs_t

Table of your driver’s modeswitcher functions

Synopsis:

```c
#include <mode.h>

typedef struct disp_modefuncs {
    int (*init)(...);
    void (*fini)(...);
    void (*module_info)(...);
    int (*get_modeinfo)(...);
    void (*get_modelist)(...);
    int (*set_mode)(...);
    int (*wait_vsync)(...);
    int (*set_dpms_mode)(...);
    int (*set_display_offset)(...);
    int (*set_palette)(...);
    void (*set_scroll_position)(...);
    int (*layer_query)(...);
    int (*layer_select_format)(...);
    int (*layer_enable)(...);
    int (*layer_disable)(...);
    int (*layer_set_surface)(...);
    int (*layer_set_source_viewport)(...);
    int (*layer_set_dest_viewport)(...);
    int (*layer_set_blending)(...);
    int (*layer_set_chromakey)(...);
    int (*layer_set_brightness)(...);
    int (*layer_set_saturation)(...);
    int (*layer_set_contrast)(...);
    int (*layer_set_flags)(...);
    void (*layer_update_begin)(...);
    void (*layer_update_end)(...);
    void (*layer_reset)(...);
} disp_modefuncs_t;
```

Description:

The `disp_modefuncs_t` structure is a table that your driver uses to define the modeswitcher functions that it provides to the graphics...
framework. Your driver’s `devg_get_modefuncs()` entry point must fill in this structure.

**init()**

The graphics framework calls this function to initialize your hardware. The prototype is:

```c
int (*init) (disp_adapter_t *ctx, char *optstring)
```

This function should return the number of displays that this modeswitcher controls, or -1 to indicate an error. For example, if a display card controls both a flat-panel and a monitor simultaneously, this function should return 2.

For more information on where this function fits into the general flow, see “Calling sequence” in the Writing a Graphics Driver chapter.

**fini()**

The graphics framework calls this function to return your hardware to the uninitialized state, deallocate resources, and so on. The prototype is:

```c
void (*fini) (disp_adapter_t *ctx)
```

For more information on where this function fits into the general flow, see “Calling sequence” in the Writing a Graphics Driver chapter.

**module_info()**

The graphics framework calls this function to get information about the module, such as its description and revision numbers. The prototype is:

```c
void (*module_info) (disp_adapter_t *adapter, disp_module_info_t *info);
```

This function should fill in the `disp_module_info_t` structure pointed to by `info`. 
**disp_modefuncs_t**

**get_modeinfo()**

The graphics framework calls this function to get mode information. The prototype is:

```c
int (*get_modeinfo) (disp_adapter_t *ctx,
    int dispno,
    unsigned mode,
    disp_mode_info_t *info)
```

This function should populate the `disp_module_info_t` structure pointed to by `info` with information pertaining to the mode specified in `mode` for the display referenced by `dispno`. See the note about modes in `get_modelist()` below for more information.

**get_modelist()**

The graphics framework calls this function to get a list of the modes that your driver supports. The prototype is:

```c
void (*get_modelist) (disp_adapter_t *ctx,
    int dispno,
    disp_mode_t *list,
    int index,
    int size)
```

This function should place a maximum of `size` modes into the array `list`, starting at the index `index`, for the display referenced by `dispno`. The `index` parameter is in place to allow multiple calls to the `get_modelist()` function in case there are more modes than will fit into the `list` array on any given call.

The list of modes is terminated with the constant `DISP.MODE_LISTEND` — don’t return this as a valid mode. The list of modes must be returned in the exact same order each time, but there’s no requirement to arrange the list by any sorting criteria.
It’s the mode number (the content of the list array) that’s important for subsequent calls, and not the mode index itself.

For example, if your driver returns the following array:

```c
list [0] = 0x1234;
list [1] = 0x070B;
list [2] = 0x4321;
list [3] = DISP_MODE_LISTEND;  // terminate list
```

then your `get_modeinfo()` and `set_mode()` functions are called with, for example, `0x4321` and not the index 2.

The driver should use only the 15 least significant bits of the `disp_mode_t` type. Thus each entry in `list` must be less than or equal to `0x7fff`.

### set_mode()

The graphics framework calls this function to set the hardware for the display referenced by `dispno` to the mode specified by `mode`. The prototype is:

```c
int (*set_mode)(disp_adapter_t *ctx,
               int dispno,
               unsigned mode,
               disp_crtc_settings_t *settings,
               disp_surface_t *surf,
               unsigned flags)
```

See the note about modes in `get_modelist()` above for more information.

The `settings` parameter is valid only if the mode is a generic mode, which means that the framework can pass an arbitrary X and Y resolution and refresh rate. The driver advertises that it can support generic modes by setting the appropriate flag in the `disp_mode_info_t` structure when `get_modeinfo()` is called.
If your driver supports virtual panning, the virtual screen might be larger than the physical screen resolution of the requested mode. The virtual x and y resolutions are passed in `surf->width` and `surf->height`. The `surf->stride` member specifies the stride, in bytes, that the driver should use to program the CRTC controller.

For more information on where this function fits into the general flow, see “Calling sequence” in the Writing a Graphics Driver chapter.

**wait_vsync()**

This function must wait until the display controller for the display referenced by `dispno` enters the vertical-retrace period. The prototype is:

```c
int (*wait_vsync)(disp_adapter_t *adapter,
                  int dispno);
```

**set_dpms_mode()**

The graphics framework calls this function to set the DPMS mode. The prototype is:

```c
void (*set_dpms_mode)(disp_adapter_t *ctx,
                      int dispno,
                      int mode)
```

Select a DPMS mode for the display referenced by `dispno` as follows:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>On</td>
</tr>
<tr>
<td>1</td>
<td>Standby</td>
</tr>
<tr>
<td>2</td>
<td>Suspend</td>
</tr>
<tr>
<td>4 (not a typo)</td>
<td>Off</td>
</tr>
</tbody>
</table>
set_display_offset()

The graphics framework calls this function to set the frame buffer base of the visible display for the display controller referenced by dispno. The prototype is:

```c
void (*set_display_offset) (disp_adapter_t *ctx,
   int dispno,
   unsigned offset,
   int wait_vsync)
```

The offset member must be a multiple of the crtc_start_gran member of the disp_mode_info_t structure.

If wait_vsync is nonzero, your driver should wait for the next vertical retrace period before returning from this function.

set_palette()

The graphics framework calls this function to set the palette for the display referenced by dispno. The prototype is:

```c
void (*set_palette) (disp_adapter_t *ctx,
   int dispno,
   int index,
   int count,
   disp_color_t *pal)
```

One or more entries in the palette can be set at a time with this function call. The index indicates the starting palette index, and count indicates the number of entries being set. Finally, pal contains an array of color values, one per entry, to set.
If this function is specified (i.e. not NULL in the function pointer \texttt{set\_palette}), then it’s called regardless of whether or not the \texttt{set\_palette()} function in the miscellaneous callouts structure, \texttt{disp\_draw\_miscfuncs\_t}, has been supplied:

\begin{verbatim}
if (disp\_modefuncs \rightarrow set\_palette) {
    (*disp\_modefuncs \rightarrow set\_palette) (...);
} else if (disp\_draw\_miscfuncs \rightarrow set\_palette) {
    (*disp\_draw\_miscfuncs \rightarrow set\_palette (...);
}
\end{verbatim}

\textbf{set\_scroll\_position()}

The graphics framework calls this function to scroll, or pan around a virtual desktop when the virtual display surface is bigger than the physical display. If your driver supports Virtual Panning, you must provide this function. The prototype is:

\begin{verbatim}
void (*set\_scroll\_position) (disp\_adapter\_t *adapter,
                                int dispno,
                                int xoff,
                                int yoff )
\end{verbatim}

This function should set the physical viewport into the virtual display surface such that the point \((xoff, yoff)\) within the virtual display appears at the upper left corner of the physical display.

\textbf{layer\_query()}

The graphics framework calls this function to query the capabilities of a given layer. The prototype is:

\begin{verbatim}
int (*layer\_query) (disp\_adapter\_t *adapter,
                     int dispno,
                     int layer\_idx,
                     int fmt\_idx,
                     disp\_layer\_query\_t *info);
\end{verbatim}

The arguments to \texttt{layer\_query} are:
adapter A pointer to the `disp_adapter_t` structure.

dispno The display associated with the layer selected by `layer_idx`.

layer_idx The index of the layer being queried. Layers are indexed starting at 0.

fmt_idx The index that pertains to the data format that’s returned. Formats are indexed starting at 0.

info A pointer to a structure that the driver fills in based on the selected layer and data format. The information returned for a layer can vary for different pixel formats. For more information on pixel formats for layers, see “Pixel formats for layers” in the Writing a Graphics Driver chapter.

If the `layer_idx` or `fmt_idx` indexes are out of range, this function should return -1, otherwise it should return 0.

`layer_select_format()`

The graphics framework calls this function to select the pixel format of the the layer specified by `layer_idx`. The prototype is:

```c
int (*layer_select_format) ( 
    disp_adapter_t *adapter, 
    int dispno, 
    int layer_idx, 
    int fmt_idx )
```

The format specified by `fmt_idx` corresponds to the `fmt_idx` that was passed to `layer_query`.

The arguments to `layer_select_format` are:

adapter A pointer to the `disp_adapter_t` structure.

dispno The display associated with the layer selected by `layer_idx`. 
layer_idx

The index of the layer that corresponds to the pixel format selected. Layers are indexed starting at 0.

fmt_idx

The index that pertains to the pixel format selected. Formats are indexed starting at 0.

If the dispno, layer_idx, or fmt_idx indexes are out of range, this function should return -1 to indicate an error, otherwise it should return 0.

\textit{layer_enable()}

The graphics framework calls this function to make the layer specified by \textit{layer_idx} visible. The prototype is:

\begin{verbatim}
int (*layer_enable)(
    disp_adapter_t *adapter,
    int dispno,
    int layer_idx
)
\end{verbatim}

The arguments to \textit{layer_enable} are:

- \textit{adapter} A pointer to the \textit{disp_adapter_t} structure.
- \textit{dispno} The display associated with the layer selected by \textit{layer_idx}.
- \textit{layer_idx} The index of the layer that \textit{layer_idx} makes visible. Layers are indexed starting at 0.

If the \textit{dispno} or \textit{layer_idx} indexes are out of range, or the layer can’t be disabled or reenabled, this function should return -1 to indicate an error, otherwise it should return 0.

\textit{layer_disable()}

The graphics framework calls this function to make the layer specified by \textit{layer_idx} invisible. The prototype is:

\begin{verbatim}
int (*layer_enable)(
    disp_adapter_t *adapter,
    int dispno,
    int layer_idx
)
\end{verbatim}
The arguments to `layer_disable` are:

- `adapter`: A pointer to the `disp_adapter_t` structure.
- `dispno`: The display associated with the layer selected by `layer_idx`.
- `layer_idx`: The index of the layer that `layer_idx` makes invisible. Layers are indexed starting at 0.

If the `dispno`, or `layer_idx` indexes are out of range, or the layer can’t be disabled or re-enabled, this function should return -1 to indicate an error, otherwise it should return 0.

`layer_set_surface()`

The graphics framework calls this function to associate a memory surface with the layer specified by `layer_idx`. The prototype is:

```c
int (*layer_set_surface)(
    disp_adapter_t *adapter,
    int dispno,
    int layer_idx,
    int surface_idx,
    disp_surface_t *surf )
```

The arguments to `layer_set_surface` are:

- `adapter`: A pointer to the `disp_adapter_t` structure.
- `dispno`: The display associated with the layer selected by `layer_idx`.
- `layer_idx`: The index of the layer associated with the memory surface. Layers are indexed starting at 0.
- `surface_idx`: The meaning of this argument depends on the layer image format selected by `layer_select_format`.
  Some layer formats split the image components across multiple buffers. A planar YUV surface, for
example, requires three buffers to store a valid image. In this case, the `surface_idx` argument specifies whether a Y, U, or V buffer gets allocated.

`surf` A pointer to the surface created by `disp_surface_t`.

The memory surface was created by the driver’s `alloc_layer_surface()` management entry point. The `layer_select_format` entry point is called before this entry point.

If the `dispno` or `layer_idx` indexes are out of range, this function should return -1 to indicate an error, otherwise it should return 0.

**layer_set_source_viewport()**

The graphics framework calls this function to set the size of the source viewport for the layer that `layer_idx` specifies. The prototype is:

```c
int (*layer_set_source_viewport)(
    disp_adapter_t *adapter,
    int dispno,
    int layer_idx,
    int x1, y1,
    int x2, y2
)
```

The arguments to `layer_set_source_viewport` are:

- `adapter` A pointer to the `disp_adapter_t` structure.
- `dispno` The display associated with the layer selected by `layer_idx`.
- `layer_idx` The index of the layer that corresponds to the source viewport. Layers are indexed starting at 0.
- `x1, y1` The location of the top-left edge of the source viewport.
- `x2, y2` The location of the bottom-right edge of the source viewport.
This function should return -1 if:

- indexes are out of range
  Or:
- viewport width is outside range
  Or:
- viewport size is outside range

Otherwise the function should return 0.

**layer_set_dest_viewport()**

The graphics framework calls this function to set the size of the destination viewport for the layer that `layer_idx` specifies. The prototype is:

```c
int (*layer_set_dest_viewport) ( 
    disp_adapter_t *adapter,
    int dispno,
    int layer_idx,
    int x1, y1,
    int x2, y2 )
```

The arguments to `layer_set_dest_viewport` are:

- **adapter** A pointer to the `disp_adapter_t` structure.
- **dispno** The display associated with the layer selected by `layer_idx`.
- **layer_idx** The index of the layer that corresponds to the destination viewport. Layers are indexed starting at 0.
- **x1, y1** The location of the top-left edge of the destination viewport.
- **x2, y2** The location of the bottom-right edge of the destination viewport.
If the `dispno` or `layer_idx` indexes are out of range, or the viewport width for the capabilities structure for this layer is outside the ranges specified by `dst_max_height`, `dst_min_height`, `dst_max_width`, and `dst_min_width`, this function should return -1 to indicate an error, otherwise it should return 0.

**layer_set_blending()**

The graphics framework calls this function to set the layer’s blending configuration for the layer specified by `layer_idx`. The prototype is:

```c
int (*layer_set_blending)(
    disp_adapter_t *adapter,
    int dispno,
    int layer_idx,
    unsigned alpha_mode,
    int m1,
    int m2)
```

The arguments to `layer_set_blending` are:

- **adapter** A pointer to the `disp_adapter_t` structure.
- **dispno** The display associated with the layer selected by `layer_idx`.
- **layer_idx** The index of layer to blend. Layers are indexed starting at 0.
- **alpha_mode** The logical OR of up to four different flags. Four groups of flags are available, and either zero or one flags can be selected from each group. When a value of 0 is passed for `alpha_mode`, alpha blending is disabled.
- **m1** A primary alpha multiplier that’s used when there’s a request for blending using global alpha multipliers.
- **m2** A secondary alpha multiplier that’s used when there’s a request for blending using global alpha multipliers.
If the dispno or layer_idx indexes are out of range, this function should return -1 to indicate an error, otherwise it should return 0.

If the layer selected by layer_idx doesn’t support the combination of flags specified by alpha_mode, this function should return -1, otherwise it should return 0.

For more information on global alpha multipliers, and the currently defined flags, see “Alpha mode bits.”

*layer_set_chromakey()*

The graphics framework calls this function to set the layer’s chroma-key configuration for the layer specified by layer_idx. The prototype is:

```c
int (*layer_set_chromakey) ( 
    disp_adapter_t *adapter,
    int dispno,
    int layer_idx,
    unsigned chroma_mode,
    disp_color_t color0,
    disp_color_t color1,
    disp_color_t mask );
```

The arguments to layer_set_chromakey are:

*adapter*  
A pointer to the disp_adapter_t structure.

*dispno*  
The display associated with the layer selected by layer_idx.

*layer_idx*  
The layer index for the chroma-key configuration.

*chroma_mode*  
Selected from: DISP_CHROMA_OP_SRC_MATCH, DISP_CHROMA_OP_DST_MATCH, DISP_CHROMA_OP_DRAW, DISP_CHROMA_OP_NO_DRAW

DISP_CHROMA_OP_SRC_MATCH and DISP_CHROMA_OP_DST_MATCH are mutually exclusive.
DISP_CHROMA_OP_DRAW and DISP_CHROMA_OP_NO_DRAW are also mutually exclusive.

If 0 is passed for chroma-mode, chroma-keying is disabled.

For more information on chroma key mode bits, see “Chroma mode bits”

color0 The color to test on.
color1 Reserved; ignore this argument.
mask Reserved; ignore this argument.

If the layer selected by layer_idx doesn’t support the combination of flags specified by chroma_mode, this function should return -1, otherwise it should return 0.

layer_set_brightness()

The graphics framework calls this function to set the brightness attribute of the layer specified by layer_idx. The prototype is:

```c
int (*layer_set_brightness)(
    disp_adapter_t *adapter,
    int dispno,
    int layer_idx,
    int brightness );
```

The arguments to layer_set_brightness are:

adapter A pointer to the disp_adapter_t structure.
dispno The display associated with the layer selected by layer_idx.
layer_idx The layer index for the brightness adjustment. Layers are indexed starting at 0.
brightness  The brightness value, in the range from 127 to -128, where 127 is brightest, and -128 is darkest. A value of 0 is normal brightness.

If the selected layer doesn’t support brightness adjustment, this function should return -1 to indicate an error, otherwise it should return 0.

**layer_set_saturation()**

The graphics framework calls this function to set the color saturation attribute of the layer specified by layer_idx. The prototype is:

```c
int (*layer_set_saturation)(
    disp_adapter_t *adapter,
    int dispno,
    int layer_idx,
    int saturation );
```

The arguments to layer_set_saturation are:

- **adapter**  A pointer to the disp_adapter_t structure.
- **dispno**  The display associated with the layer selected by layer_idx.
- **layer_idx**  The layer index for the saturation adjustment. Layers are indexed starting at 0.
- **saturation**  The saturation value, in the range from 127 to -128, where 127 is most saturated, and -128 is least saturated. A value of 0 is normal saturation.

If the selected layer doesn’t support saturation adjustment, this function should return -1 to indicate an error, otherwise it should return 0.
layer_set_contrast()

The graphics framework calls this function to set the contrast attribute of the layer specified by layer_idx. The prototype is:

```c
int (*layer_set_contrast) (  
    disp_adapter_t *adapter,  
    int dispno,  
    int layer_idx,  
    int contrast );
```

The arguments to layer_set_contrast are:

- **adapter**: A pointer to the disp_adapter_t structure.
- **dispno**: The display associated with the layer selected by layer_idx.
- **layer_idx**: The layer index for the contrast adjustment. Layers are indexed starting at 0.
- **contrast**: The contrast value, in the range from 127 to -128, where 127 is the highest contrast, and -128 is the lowest contrast. A value of 0 is normal contrast.

If the selected layer doesn’t support contrast adjustment, this function should return -1 to indicate an error, otherwise it should return 0.

layer_set_flags()

The graphics framework calls this function to set various attributes of the layer specified by layer_idx. The prototype is:

```c
int (*layer_set_flags) (  
    disp_adapter_t *adapter,  
    int dispno,  
    int layer_idx,  
    unsigned flag_mask  
    unsigned flag_values );
```

The arguments to layer_set_flags are:
adaptorter A pointer to the disp_adapter_t structure.

dispno The display associated with the layer selected by layer_idx.

layer_idx The layer index for the attributes that are set. Layers are indexed starting at 0.

flag_mask The bits whose values are to be changed. If a bit isn’t set in flag_mask, the corresponding flag isn’t affected.

flag_values The new settings that correspond to the bits set in flag_mask.

The following flags are supported:

DISP_LAYER_FLAG_DISABLE_FILTERING
Disable any filtering algorithms that are applied to the image data before the image is displayed. When scaling is in effect, pixel replication is used instead of filtering to produce the scaled image, if pixel replication is supported.

DISP_LAYER_FLAG_EDGE_CLAMP
If the image being displayed isn’t large enough to fill the destination viewport, then the unfilled right and bottom portions of the viewport are filled by replicating the last pixel that was displayed to the edge of the viewport.

DISP_LAYER_FLAG_EDGE_WRAP
If the image being displayed isn’t large enough to fill the destination viewport, then the unfilled right and bottom portions of the viewport are filled by wrapping around to the top, left portions of the image.

If the selected layer isn’t supported by one or more of the flags specified by flag_mask, this function should return -1 to indicate an error, otherwise it should return 0.
**layer_update_begin()**

This function *must* be called before you make any adjustments to the layer configuration. The driver should attempt to make multiple adjustments atomic, by having the adjustments take effect at the time `layer_update_end()` is called. The prototype is:

```c
void (*layer_update_begin) (  
    disp_adapter_t *adapter,
    int dispno,
    int layer_idx );
```

The arguments to `layer_update_begin` are:

- **adapter** A pointer to the `disp_adapter_t` structure.
- **dispno** The display associated with the layer selected by `layer_idx`.
- **layer_idx** The layer index for the updates. Layers are indexed starting at 0.

**layer_update_end()**

The prototype is:

```c
void (*layer_update_end) (  
    disp_adapter_t *adapter,
    int dispno,
    int layer_idx );
```

The arguments to `layer_update_end` are:

- **adapter** A pointer to the `disp_adapter_t` structure.
- **dispno** The display associated with the layer selected by `layer_idx`.
- **layer_idx** The layer index for the updates. Layers are indexed starting at 0.
Before any calls are made that affect the layer configuration, `layer_update_begin()` is called, and subsequently `layer_update_end()` makes the changes effective.

This function runs after adjustments are made to the layer configuration. Any adjustments made since `layer_update_begin()` was called, will now take effect.

The `layer_query()` function doesn’t affect the layer configuration and is therefore an exception to this rule.

**layer_reset()**

The prototype is:

```c
void (*layer_reset)(
    disp_adapter_t *adapter,
    int dispno,
    int layer_idx);
```

The arguments to `layer_reset` are:

- **adapter** A pointer to the `disp_adapter_t` structure.
- **dispno** The display associated with the layer selected by `layer_idx`.
- **layer_idx** The index for the reset layers. Layers are indexed starting at 0.

This function resets the specified layer to its initial state.

A layer should be reset to its initial state after a modeswitch.

The initial state is defined as one where:

- a layer that can be disabled, is disabled (i.e. made invisible)
- the brightness, contrast, and color saturation of the layer are all set to 0
all the flags that can be set using `layer_set_flags()` are set to 0

- the chroma-key state, the alpha-blending state, and the size and position of the source and destination viewports are undefined.

**Classification:**

Photon

**See also:**

`devg_get_modefuncs()`, `disp_adapter_t`,
`disp_crtc_settings_t`, `disp_mode_info_t`,
`disp_module_info_t`, `disp_surface_t`
Synopsis:

```
#include <display.h>

typedef struct disp_module_info {
    uint8_t        ddk_version_major;
    uint8_t        ddk_version_minor;
    uint8_t        ddk_rev;
    uint8_t        driver_rev;
    char           *description;
    unsigned       reserved[16];
} disp_module_info_t;
```

Description:

The `disp_module_info_t` structure stores module information about your driver. Your driver must fill in this structure when the graphics framework calls the `module_info()` function that your driver defines in these structures:

- `disp_draw_miscfuncs_t`
- `disp_memfuncs_t`
- `disp_modefuncs_t`
- `disp_vidfuncs_t`

The members of the `disp_module_info_t` structure include:

- `ddk_version_major`
  The major version number of the Graphics DDK that you used to build your driver. Set this to DDK_VERSION_MAJOR.

- `ddk_version_minor`
  The minor version number of the DDK; set it to DDK_VERSION_MINOR.

- `ddk_rev`
  The revision number of the DDK; set it to DDK_REVISION.
disp_module_info_t

\textit{driver\_rev} \hspace{1cm} \text{The revision of the driver; use this member as you see fit.}

\textit{description} \hspace{1cm} \text{A string that describes the driver module. The recommended format is the driver name, followed by a space, a hyphen, another space, and then a generic description of the hardware that the driver should support. For example,} \texttt{vga - driver for VGA-compatible devices.}

\textbf{Classification:}

\text{Photon}

\textbf{See also:}

\texttt{disp\_draw\_miscfuncts\_t, disp\_memfuncts\_t, disp\_modefuncts\_t, disp\_vidfuncts\_t}
**disp_surface_t**

Description of a two-dimensional surface

**Synopsis:**

```c
#include <display.h>

typedef struct disp_surface {
    int size;
    unsigned pixel_format;
    unsigned offset;
    unsigned char *vidptr;
    uintptr_t paddr;
    unsigned stride;
    unsigned flags;
    int width;
    int height;
    disp_color_t *palette;
    int palette_valid_entries;
    disp_adapter_t *adapter;
    unsigned reserved;
} disp_surface_t;
```

**Description:**

The **disp_surface_t** structure describes a two-dimensional surface. It’s used as an argument to several functions, and is also used within other structures, such as **disp_draw_context_t**.

The members include:

- **size**
  
  The size of this structure. Any time a driver creates a structure of this type, it should set this member to `sizeof (disp_surface_t)`.

- **pixel_format**
  
  The pixel format. For more information, see “Pixel formats” in the Writing a Graphics Driver chapter.

- **offset**
  
  A device-dependent address that points to the start of the surface data.

- **vidptr**
  
  The virtual (CPU) address that points to the start of the surface data.
**disp_surface_t**

- **paddr**: The physical (bus) address that points to the start of the surface data.
- **stride**: The number of bytes from the beginning of one scanline to the beginning of the next (see diagram below).
- **flags**: Surface flags, defined below.
- **width**: The width of the surface, in pixels (see diagram below).
- **height**: The height of the surface, in scan lines (see diagram below).
- **pal_ptr**: A pointer to the palette for this surface. If not a palette type, this pointer is NULL.
- **pal_valid_entries**: The number of entries that are valid in the `pal_ptr` palette. For palette-based surface data, pixel values in the data shouldn’t exceed this value, otherwise they might reference past the end of the palette array.
- **adapter**: A pointer to the `disp_adapter_t` structure for the device that created this surface (if any).

**Relationship of stride, height, and width**

The three members, `stride`, `height`, and `width` are used to define a surface as follows:
Memory layout.

The entire content of the box represents the total memory area available, and the unshaded portions represent the memory area that’s actually used for the surface.

Don’t overwrite the “not used” areas; it might be used to store other surfaces.

flags

The flags member is a bitmap of the following values:

DISP_SURFACE_DISPLAYABLE

The surface can be displayed via CRT controller.

DISP_SURFACE_CPU_LINEAR_READABLE

The CPU can read this surface directly.

DISP_SURFACE_CPU_LINEAR_WRITEABLE

The CPU can write to this surface directly.
The 2D engine can render into this surface.

The surface is read-accessible by 2D engine.

The 3D engine can render into surface.

The surface is read-accessible by 3D engine.

The surface can be displayed via video overlay scaler.

Video capture hardware can write frames into this surface.

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

Surface memory starts on a page boundary.

Surface memory accesses by CPU are byte-swapped; the device on which the memory resides in the opposite “endian” from the CPU.

The CPU shouldn’t map the surface memory as cacheable.

The surface memory is physically contiguous, from a bus master device’s perspective.
Classification:

Photon

See also:

disp_adapter_t, disp_draw_context_t,
disp_draw_contextfuncs_t, disp_draw_corefuncs_t,
disp_draw_misccfuncs_t, disp_memfuncs_t,
disp_modefuncs_t, disp_vidfuncs_t
### Synopsis:

```c
typedef struct {  
    unsigned flags;

    unsigned input_format;
    int frame_width;
    int frame_height;
    int frame_rate;

    unsigned output_format;

    unsigned reserved[10];
} disp_vcap_channel_caps_t;
```

### Description:

The `disp_vcap_channel_caps_t` structure describes the capabilities of a video capture unit. The members include:

- **flags**  
  Flags that describe the capabilities:

  - DISP_VCAP_CAP_SOURCE_TUNER — can capture video from a TV tuner.
  - DISP_VCAP_CAP_SOURCE_SVIDEO — can capture video from an S-Video connector.
  - DISP_VCAP_CAP_SOURCE_COMPOSITE — can capture video from a Composite video connector.
  - DISP_VCAP_CAP_BRIGHTNESS_ADJUST — brightness can be adjusted.
  - DISP_VCAP_CAP_CONTRAST_ADJUST — contrast can be adjusted.
  - DISP_VCAP_CAP_SATURATION_ADJUST — saturation can be adjusted.
  - DISP_VCAP_CAP_HUE_ADJUST — green Hue can be adjusted.
• DISP_VCAP_CAP_AUDIO_SOURCE_MUTE — audio can be muted.
• DISP_VCAP_CAP_AUDIO_SOURCE_TUNER — audio can be routed from a TV tuner through the capture unit.
• DISP_VCAP_CAP_AUDIO_SOURCE_EXTERNAL — audio can be routed from an external line in through the capture unit.
• DISP_VCAP_CAP_DOWNSCALING — the capture unit can scale video frames downwards.
• DISP_VCAP_CAP_UPSCALING — the capture unit can scale video frames upwards.
• DISP_VCAP_CAP_CROPPING — the capture unit can crop frames prior to scaling.
• DISP_VCAP_CAP_DOUBLE_BUFFER — the capture unit can alternate its output between two buffers.

input_format The format of the incoming video signal; one of:

• DISP_TV_SIGNAL_NTSC
• DISP_TV_SIGNAL_NTSC_JAPAN
• DISP_TV_SIGNAL_PAL
• DISP_TV_SIGNAL_PAL_M
• DISP_TV_SIGNAL_PAL_N
• DISP_TV_SIGNAL_PAL_N_COMBO
• DISP_TV_SIGNAL_SECAM

frame_width The width of the output frames, in pixels, prior to scaling.

frame_height The height of the output frames, in scanlines, prior to scaling.

frame_rate The frequency of frames output, in Hz (e.g. 30 for NTSC).
disp_vcap_channel_caps_t

output_format The data format of the frames output by the scaler unit; see “Pixel formats” in the Writing a Graphics Driver chapter.

Classification:
Photon

See also:
disp_vcapfuncs_t
**Synopsis:**

```c
typedef struct {
    unsigned size;
    unsigned flags;
    unsigned video_source;
    unsigned audio_source;
    unsigned input_format;
    unsigned output_format;
    int tuner_channel;
    int Fif;
    short dst_width;
    short dst_height;
    short crop_top;
    short crop_bottom;
    short crop_left;
    short crop_right;
    short brightness;
    short contrast;
    short u_saturation;
    short v_saturation;
    short hue;
    short reserved;
    unsigned update_flags;
    int scaler_index;
    unsigned reserved2[6];
} disp_vcap_channel_props_t;
```

**Description:**

The `disp_vcap_channel_props_t` describes the configurable properties of a video capture unit. The members include:

- `size`: Filled in with `sizeof (disp_vcap_channel_props_t)` on instantiation.

- `flags`: Valid flags are:
  - `DISP_VCAP_FLAG_AFT_ON` — enable Auto Fine Tuning. After the driver programs the TV
tuner to the requested frequency and band, the
driver can use feedback from the tuner hardware
in order to adjust the frequency to improve
signal quality.

- DISP_VCAP_FLAG_RUNNING — the driver
  should enable streaming of incoming video
  frames only if this flag is set.

- DISP_VCAP_FLAG_DOUBLE_BUFFER — enable
double buffering of the incoming video frames.
The driver should direct each alternate frame of
data to the alternate memory buffer.

- DISP_VCAP_FLAG_SYNC_WITH_SCALER —
  valid when
  DISP_VCAP_FLAG_DOUBLE_BUFFER is also
  specified. The driver should attempt to
  configure the hardware such that whenever a
  complete frame of data arrives, the scaler is
  automatically set to display the new frame.

Note that the scaler hardware is already
configured up when the set_channel_props()
function defined in the disp_vcapfuncs_t
structure is called. If the driver can’t
synchronize the capture unit with the scaler
specified by the scaler_index member of the
disp_vcap_channel_props_t structure in
this manner, this function should simply act as
if this flag hadn’t been specified.

video_source

The source of the video signal to be captured:

- DISP_VCAP_SOURCE_TUNER — capture from
  the output of the TV tuner.

- DISP_VCAP_SOURCE_SVIDEO — capture from
  the SVideo connector.

- DISP_VCAP_SOURCE_COMPOSITE — capture
  from the Composite video connector.
If something other than one of the above is specified, the driver should default to something reasonable.

**audio_source**  
The source of the audio signal that’s routed through the capture unit:

- DISP.VCAP_AUDIO_SOURCE_MUTE — mute the audio output from the capture unit.
- DISP.VCAP_AUDIO_SOURCE_TUNER — route the audio signal from the TV tuner through the capture unit.
- DISP.VCAP_AUDIO_SOURCE_EXTERNAL — route the audio signal from the external line in through the capture unit.

**input_format**  
The encoding of the incoming signal to be captured:

- DISP.TV_SIGNAL_NTSC
- DISP.TV_SIGNAL_NTSC_JAPAN
- DISP.TV_SIGNAL_PAL
- DISP.TV_SIGNAL_PAL_M
- DISP.TV_SIGNAL_PAL_N
- DISP.TV_SIGNAL_PAL_N_COMBO
- DISP.TV_SIGNAL_SECAM

**output_format**  
The format of the data output by the capture unit; see “Pixel formats” in the Writing a Graphics Driver chapter.

**tuner_channel**  
The tuner frequency, in Hertz, to which the TV tuner should be programmed.

**Fif**  
The Intermediate Frequency, in Hertz, to be used when programming the TV tuner.

**dst_width**  
The output width of the frames after scaling.
### disp_vcap_channel_props_t

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dst_height</td>
<td>The output height of the frames after scaling.</td>
</tr>
<tr>
<td>crop_top</td>
<td>The number of lines to crop off the top of the captured before scaling.</td>
</tr>
<tr>
<td>crop_bottom</td>
<td>The number of lines to crop off the bottom of the captured before scaling.</td>
</tr>
<tr>
<td>crop_left</td>
<td>The number of lines to crop off the left of the captured before scaling.</td>
</tr>
<tr>
<td>crop_right</td>
<td>The number of lines to crop off the right of the captured before scaling.</td>
</tr>
<tr>
<td>brightness</td>
<td>The luminance adjustment to captured video. This is a signed value.</td>
</tr>
<tr>
<td></td>
<td>where 0 means no adjustment, 32767 is the maximum brightness, and -32768 is</td>
</tr>
<tr>
<td></td>
<td>the darkest.</td>
</tr>
<tr>
<td>contrast</td>
<td>The contrast adjustment to captured video. This is a signed value.</td>
</tr>
<tr>
<td></td>
<td>where 0 means normal contrast, 32767 is the maximum contrast, and -32768 is</td>
</tr>
<tr>
<td></td>
<td>the minimum.</td>
</tr>
<tr>
<td>u_saturation</td>
<td>The saturation level of the Cr chroma component of captured video.</td>
</tr>
<tr>
<td></td>
<td>This is a signed value, where 0 means normal saturation, 32767 is the</td>
</tr>
<tr>
<td></td>
<td>maximum saturation, and -32768 is the minimum.</td>
</tr>
<tr>
<td>v_saturation</td>
<td>The saturation level of the Cb chroma component of captured video.</td>
</tr>
<tr>
<td></td>
<td>This is a signed value, where 0 means normal saturation, 32767 is the</td>
</tr>
<tr>
<td></td>
<td>maximum saturation, and -32768 is the minimum.</td>
</tr>
<tr>
<td>hue</td>
<td>The level of green hue adjustment to captured video.</td>
</tr>
<tr>
<td></td>
<td>This is a signed value, where 0 means no adjustment, 32767 is the maximum</td>
</tr>
<tr>
<td></td>
<td>hue, and -32768 is the minimum.</td>
</tr>
<tr>
<td>update_flags</td>
<td>Flags that indicate which members of the disp_vcap_channel_props_t structure</td>
</tr>
<tr>
<td></td>
<td>contain</td>
</tr>
</tbody>
</table>
valid data that should be used to program the capture unit:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Valid member(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISP_VCAP_UPDATE_VIDEO_SOURCE</td>
<td>video_source</td>
</tr>
<tr>
<td>DISP_VCAP_UPDATE_AUDIO_SOURCE</td>
<td>audio_source</td>
</tr>
<tr>
<td>DISP_VCAP_UPDATE_INPUT_FORMAT</td>
<td>input_format</td>
</tr>
<tr>
<td>DISP_VCAP_UPDATE_OUTPUT_FORMAT</td>
<td>output_format</td>
</tr>
<tr>
<td>DISP_VCAP_UPDATE_TUNER</td>
<td>tuner_channel, Fif</td>
</tr>
<tr>
<td>DISP_VCAP_UPDATE_OUTPUT_SIZE</td>
<td>dst_width, dst_height,</td>
</tr>
<tr>
<td></td>
<td>crop_top, crop_bottom,</td>
</tr>
<tr>
<td></td>
<td>crop_left, crop_right</td>
</tr>
<tr>
<td>DISP_VCAP_UPDATE_BRIGHTNESS</td>
<td>brightness</td>
</tr>
<tr>
<td>DISP_VCAP_UPDATE_CONTRAST</td>
<td>contrast</td>
</tr>
<tr>
<td>DISP_VCAP_UPDATE_SATURATION</td>
<td>u_saturation, v_saturation</td>
</tr>
<tr>
<td>DISP_VCAP_UPDATE_HUE</td>
<td>hue</td>
</tr>
<tr>
<td>DISP_VCAP_UPDATE_OUTPUT_TARGET</td>
<td>One or more of the <em>plane</em> parameters is valid, and one or more of the buffers described by the <em>plane</em> parameters has changed since the last time this function was called.</td>
</tr>
</tbody>
</table>
disp_vcap_channel_props_t

Classification:

Photon

See also:

disp_vcapfuncs_t
Synopsis:

typedef struct {
  unsigned size;
  unsigned flags;
  unsigned reserved[8];
} disp_vcap_channel_status_t;

Description:

The disp_vcap_channel_status_t structure describes the status of a video capture unit. The members include:

size  Filled in with sizeof (disp_vcap_channel_status_t) on instantiation.

flags  Currently defined flags:

- DISP_VCAP_STATUS_TUNED — the TV tuner hardware has stabilized.
- DISP_VCAP_STATUS_CHANNEL_PRESENT — a good signal has been detected on the currently tuned channel.
- DISP_VCAP_STATUS_VIDEO_PRESENT — a video signal has been detected on the current channel.
- DISP_VCAP_STATUS_STEREO — stereo audio has been detected on the current channel.

Classification:

Photon

See also:

disp_vcapfuncts_t
**Synopsis:**

```c
#include <vid.h>

typedef struct disp_vcapfuncs {
    int (*init) (...);
    void (*fini) (...);
    void (*module_info) (...);
    int (*get_channel_caps) (...);
    int (*get_channel_status) (...);
    int (*set_channel_props) (...);
    int (*close_channel) (...);
} disp_vcapfuncs_t;
```

**Description:**

The `disp_vcapfuncs_t` structure is a table that your driver uses to define the video capture functions that it provides to the graphics framework. Your driver’s `devg_get_vcapfuncs()` entry point must fill in this structure.

### `init()`

The graphics framework calls this function to initialize the modeswitcher. The prototype is:

```c
int (*init) (disp_adapter_t *adapter,
              char *optstring)
```

This function should return the number of capture units available.

### `fini()`

The graphics framework calls this function to free resources and disable all scalers (making them invisible). The prototype is:

```c
void (*fini) (disp_adapter_t *adapter)
```

This function must free any offscreen memory that was allocated for frame data.
module_info()

The graphics framework calls this function to get information about the module. The prototype is:

```c
void (*module_info) (disp_adapter_t *adapter,
                     disp_module_info_t *info)
```

This function must fill the given `disp_module_info_t` structure.

get_channel_caps()

The graphics framework calls this function to get the scaler capabilities for a given pixel format. The prototype is:

```c
int (*get_channel_caps) (disp_adapter_t *adapter,
                         int channel,
                         int input_fmt_index,
                         int output_fmt_index,
                         disp_vcap_channel_caps_t *caps)
```

This function should fill in the `disp_vcap_channel_caps_t` structure pointed to by `caps`.

The video capture module is responsible for controlling video capture hardware, which takes a video stream from a source such as a TV tuner, and turns it into a sequence of video frames. Currently, the interface defined here is mainly intended to support devices that stream data into the video frame buffer, especially into buffers whose contents can be displayed by the video scaler.

Currently, viewing of TV or live video in a window can be supported by higher level software, but full video capture (e.g. capture to disk) isn’t yet supported.

A hardware device may integrate zero or more video capture units. The `channel` argument is a zero-based index that represents the device for which the device capabilities should be returned.

A video capture unit captures a video stream of a particular format (e.g. an analog NTSC signal) and converts it to a sequence of video frames that are of a particular format (e.g. packed YUV 422).
A capture unit may support one or more input formats, and one or more output formats. The input_fmt_index and output_fmt_index arguments are zero-based indexes that represent the input and output formats for which to provide information.

If input_fmt_index is greater than or equal to the number of input formats supported, or if output_fmt_index is greater than or equal to the number of output formats supported, this function should return -1, otherwise it should return 0.

**get_channel_status()**

The graphics framework calls this function to get the channel status. The prototype is:

```c
int (*get_channel_status) (  
disp_adapter_t *adapter,  
  int channel,  
  disp_vcap_channel_status_t *caps );
```

This function should fill in the `disp_vcap_channel_status_t` structure pointed to by `caps` with status information for the capture unit specified by `channel`.

The `get_channel_status()` function should return 0 on success, or -1 on failure.

**set_channel_props()**

The graphics framework calls this function to set the scaler capabilities. The prototype is:

```c
int (*set_channel_props) (  
disp_adapter_t *adapter,  
  int channel,  
  disp_vcap_channel_props_t *props,  
  disp_surface_t *plane1,  
  disp_surface_t *plane2,  
  disp_surface_t *unplane1,  
  disp_surface_t *unplane2) ;
```
This function should configure the the capture unit specified by `channel`, based on the contents of the `disp_vcap_channel_props_t` structure pointed to by `props`. The *plane* arguments describe the buffers where the incoming video is to be streamed. Depending on the properties specified by `props`, `set_channel_props()` may ignore some of the *plane* structures:

- When single-buffering, you should ignore the *plane2* arguments.
- For non-planar formats, you should ignore the `uplane*` and `vplane*` arguments.

In the case of packed RGB data, `yplane1` (and `yplane2`, if double buffering) specify where the data is to be streamed.

The `set_channel_props()` function should return 0 on success, or -1 on failure.

`close_channel()`

The graphics framework calls this function to close a channel. The prototype is:

```c
int (*close_channel) (disp_adapter_t *adapter,
                      int channel)
```

This function should halt the video capture unit specified by `channel`, and free any system resources that were associated with it.

**Classification:**

Photon

**See also:**

`devg_get_vcapfuncs()`, `disp_adapter_t`, `disp_module_info_t`, `disp_surface_t`, `disp_vcap_channel_caps_t`, `disp_vcap_channel<Props_t`, `disp_vcap_channel_status_t`
Synopsis:

```c
#include <vid.h>

typedef struct {
  unsigned char src_alpha;
  unsigned char dst_alpha;
  unsigned alpha_mode;
  unsigned short x1;
  unsigned short x2;
  unsigned short y1;
  unsigned short y2;
} disp_vid_alpha_t;
```

Description:

The `disp_vid_alpha_t` structure describes a region of the video viewport to be blended with desktop. The members include:

- `src_alpha` The source alpha blending factor.
- `dst_alpha` The destination alpha blending factor.
- `alpha_mode` The alpha blending mode. The description of this field is the same as for the `alpha_mode` member of the `disp_draw_context_t` structure.
- `x1` The offset of the left hand side of the region to be blended, in pixels, relative to the left hand side of the scaler viewport.
- `y1` The offset of the top scanline of the region to be blended, in pixels, relative to the top of the scaler viewport.
- `x2` The offset of the right-hand side of the region to be blended, in pixels, relative to the left hand side of the scaler viewport.
**disp_vid_alpha_t**

The offset of the bottom scanline of the region to be blended, in pixels, relative to the top of the scaler viewport.

**Classification:**

Photon

**See also:**

`disp_draw_context_t`, `disp_vidfuncs_t`, `disp_vid_channel Props_t`
Synopsis:

```c
#include <vid.h>

typedef struct {
    unsigned short  size;
    unsigned short  reserved0;
    unsigned       flags;
    unsigned       format;
    int            src_max_x;
    int            src_max_y;
    int            max_mag_factor_x;
    int            max_mag_factor_y;
    int            max_shrink_factor_x;
    int            max_shrink_factor_y;
    unsigned      reserved[8];
} disp_vid_channel_caps_t;
```

Description:

The `disp_vid_channel_caps_t` structure describes the general capabilities of a video scaler for a given format. The members include:

- **size** The size of this structure.

- **reserved0, reserved** Resolved, don’t examine or modify.

- **flags** Flags that indicate the capabilities:
  - DISP_VID_CAP_SRC_CHROMA_KEY — the video viewport supports chroma-keying on frame data.
  - DISP_VID_CAP_DST_CHROMA_KEY — the video viewport supports chroma-keying on desktop data.
  - DISP_VID_CAP_BUSMASTER — the scaler device can bus-master the data from system RAM.
  - DISP_VID_CAP_DOUBLE_BUFFER — the scaler channel can be double-buffered.
DISP_VID_CAP_BRIGHTNESS_ADJUST — the brightness of the video viewport can be adjusted.

DISP_VID_CAP_CONTRAST_ADJUST — the contrast of the video viewport can be adjusted.

*format*  The pixel format; see “Pixel formats” in the Writing a Graphics Driver chapter.

*src_max_x, src_max_y*  
The maximum width and height of source frames.

*max_mag_factor_x, max_mag_factor_y*  
The magnification — -1 means cannot scale upwards.

*max_shrink_factor_x, max_shrink_factor_y*  
1 means cannot scale downwards.

**Classification:**  
Photon

**See also:**  
disp_vidfuncs_t
Configurable properties of a video scaler channel

Synopsis:

```c
#include <vid.h>

typedef struct {
    unsigned short size;
    unsigned short reserved0;
    unsigned flags;
    unsigned format;
    disp_color_t chroma_key0;
    unsigned reserved1;
    unsigned chroma_flags;
    disp_color_t chroma_key_mask;
    disp_color_t chroma_mode;
    int x1, y1;
    int x2, y2;
    int src_width, src_height;
    unsigned fmt_index;
    short brightness;
    short contrast;
    disp_vid_alpha_t alpha [DISP_VID_MAX_ALPHA];
    unsigned reserved [8];
} disp_vid_channel_props_t;
```

Description:

The `disp_vid_channel_props_t` structure describes the configurable properties of a video scaler channel. The members include:

- **size**
  The size of this structure.

- **reserved0, reserved1, reserved**
  Reserved, don’t examine or modify.

- **flags**
  A combination of the following bits:
  - DISP_VID_FLAG_ENABLE — enable the video viewport.
**disp_vid_channel_props_t**

- DISP_VID_FLAG_CHROMA_ENABLE — perform chroma-keying.
- DISP_VID_FLAG_DOUBLE_BUFFER — perform double-buffering.

**format**
The format of the frame data.

**chroma_key0**
The chroma-key color.

**chroma_flags**
The chroma-key comparison operation.

**chroma_key_mask**
Colors are masked with this before chroma comparison.

**chroma_mode**
The type of chroma key match to perform:
- DISP_VID_CHROMA_FLAG_DST — perform chroma test on desktop data.
- DISP_VID_CHROMA_FLAG_SRC — perform chroma test on video frame data.

**x1, y1**
The top left corner of video viewport in display coordinates.

**x2, y2**
The bottom right corner of video viewport in display coordinates.

**src_width, src_height**
The dimensions of the video source data.

**fmt_index**
Selects the format of the source frame data.

**brightness**
Brightness adjustment, where 0x7fff is normal, 0 the darkest, and 0xffff the brightest.

**contrast**
Contrast adjustment, where 0x7fff is normal, 0 the minimum, and 0xffff the maximum.

**alpha**
An array of regions of the video viewport to be blended with desktop. For more information, see **disp_vid_alpha_t**.
disp_vid_channel_props_t

Classification:
Photon

See also:

disp_vid_alpha_t, disp_vidfuncs_t
disp_videofuncs_t

Table of video overlay functions

Synopsis:

```c
#include <vid.h>

typedef struct disp_videofuncs {
    int (*init) (...);
    void (*fini) (...);
    void (*module_info) (...);
    int (*get_channel_caps) (...);
    int (*set_channel_props) (...);
    int (*next_frame) (...);
    int (*close_channel) (...);
} disp_videofuncs_t;
```

Description:

The `disp_videofuncs_t` structure is a table that your driver uses to define the video overlay functions that it provides to the graphics framework. Your driver’s `devg_get_videofuncs()` entry point must fill in this structure.

`init()`

The graphics framework calls this function to initialize the modeswitcher. The prototype is:

```c
int (*init) (disp_adapter_t *adapter,
            char *optstring)
```

This function should return the number of scalers available (functions similarly to the way the modeswitcher’s `init()` function returns the number of display controllers available).

`fini()`

The graphics framework calls this function to free resources and disable all scalers (making them invisible). The prototype is:

```c
void (*fini) (disp_adapter_t *adapter)
```

This function must free any offscreen memory that was allocated for frame data.
module_info()

The graphics framework calls this function to get information about the module. The prototype is:

```c
void (*module_info)(disp_adapter_t *adapter,
                    disp_module_info_t *info)
```

This function must fill the given `disp_module_info_t` structure.

get_channel_caps()

The graphics framework calls this function to get the scaler capabilities for a given pixel format. The prototype is:

```c
int (*get_channel_caps)(
                        disp_adapter_t *adapter,
                        int channel,
                        int fmt_index,
                        disp_vid_channel_caps_t *caps)
```

This function should fill in the `disp_vid_channel_caps_t` structure pointed to by `caps`.

The overlay scaler module is responsible for controlling scaler hardware, which is typically used for motion video (e.g. MPEG) acceleration. A hardware device may integrate zero or more scalers. Each scaler provides a “channel” for a video output stream.

The driver framework starts with a `fmt_index` of 0, and keeps calling `get_channel_caps()`, increasing `fmt_index` until this function returns -1. Thus, the framework can retrieve information on each format supported by the scaler denoted by `channel`. Channels are 0-based, i.e. if `init()` indicates that there are 3 channels, the valid channel numbers are 0, 1, and 2.

set_channel_props()

The graphics framework calls this function to set the scaler capabilities. The prototype is:

```c
int (*set_channel_props)(
                         disp_adapter_t *adapter,
```
int channel,
    disp_vid_channel Props_t *props,
    disp_surface_t **yplane1,
    disp_surface_t **yplane2,
    disp_surface_t **uplane1,
    disp_surface_t **uplane2,
    disp_surface_t **vplane1,
    disp_surface_t **vplane2)

This function should configure the scaler channel channel, according to the contents of the disp_vid_channel Props_t structure pointed to by props. Typically, the driver allocates video memory surfaces to store the video frame data when this routine is called.

Each of the *plane* parameters it the address of a pointer to a disp_surface_t structure that describes a surface. Depending on the properties requested by props, the driver is required to return one or more surface pointers via the *plane* pointers. Unless the DISP_VID_FLAG_DOUBLE_BUFFER flag is set, your driver should ignore the *plane2 arguments. Unless the frame data format is planar (more than one surface needed), your driver should ignore the uplane* and vplane* arguments.

This function returns -1 on failure. Otherwise, it must return 1 if any of the attributes of the surfaces associated with the channel (e.g. the size or location of the surface memory) has been modified. If the state of the channel’s surfaces hasn’t been modified, set_channel Props() returns 0. Thus higher-level software can determine whether or not it’s necessary to remap any of the frame data buffers.

(Note that the scaler may also support RGB or other non-YUV formats, in which case yplane* points to the data).

next_frame()

The graphics framework calls this function every time the higher-level software has finished preparing a new frame for display. The prototype is:

int (*next_frame)(disp_adapter_t *adapter,
            int channel)
When double buffering, the driver typically moves the scaler’s frame base pointer, so that the scaler displays the new video frame during the next display refresh cycle.

This function should return the frame index (0 or 1) if double buffering, to specify whether the data for the next frame should be copied to the *plane1 surface set, or the *plane2 surface set that was returned by set_channel.props(). In all other cases, this function should return 0.

**close_channel()**

The graphics framework calls this function to close a channel. The prototype is:

```c
int (*close_channel) (disp_adapter_t *adapter,
                        int channel)
```

Disable the scaler specified by channel. You should free up any offscreen memory that you may have allocated for the frame data on this channel.

**Classification:**

Photon

**See also:**

`devg_get_vidfuncs()`, `disp_adapter_t`, `disp_module_info_t`, `disp_surface_t`, `disp_vid_channel_caps_t`, `disp_vid_channel_props_t`
Chapter 5
Libraries

In this chapter...

DISPUTIL library — utility functions 171
FFB library — 2D software fallback routines 193
This chapter describes the functions provided in the following libraries:

- DISPUTIL library — utility functions
- FFB library — 2D software fallback routines

**DISPUTIL library — utility functions**

The **disutil** (display utilities) library provides miscellaneous functions that can be useful when writing graphics drivers:

- Miscellaneous display driver functions
- PCI configuration access functions
- Virtual memory management functions
- Video memory management functions
- Video BIOS services (x86 only)

**Miscellaneous display driver functions**

The display driver utilities set includes:

- `disp_register_adapter()`
- `disp_unregister_adapter()`
- `disp_crtc_calc()`
- `disp_acquire_vga_resources()`
- `disp_release_vga_resources()`
- `disp_perror()`
- `disp_printf()`
- `disp_wait_condition()`
- `disp_usecspin()`
DISPUTIL library — utility functions

- disp_get_rom_image()
- disp_vga_save_state()
- disp_vga_restore_state()

**disp_register_adapter()**

This function registers your driver with the display utilities libraries. The prototype is:

```c
int disp_register_adapter (disp_adapter_t *adapter)
```

Call this function before calling any other disp_* functions. If your driver uses the video BIOS functions, it must call vbios_register() before calling disp_register_adapter().

**disp_unregister_adapter()**

This function frees any resources allocated by disp_register_adapter(). The prototype is:

```c
int disp_unregister_adapter (disp_adapter_t *adapter)
```

Your driver should call this function when it no longer requires the services of the DISPUTIL lib. Typically, one of the driver’s fini() routines calls disp_unregister_adapter().

**disp_crtc_calc()**

This function calculates the CRT controller settings. The prototype is:

```c
int disp_crtc_calc (disp_crtc_settings_t *display)
```

It uses the xres, yres, refresh, h_granularity, and v_granularity members of the disp_crtc_settings_t structure pointed to by display and computes the remaining members, according to the VESA GTF (Generalized Timing Formula).
**disp_acquire_vga_resources()**

This function acquires access to the VGA registers. The prototype is:

```c
int disp_acquire_vga_resources ( 
    disp_adapter_t *adapter)
```

You must call this function *before* activating any of the VGA registers.

Drivers that can’t deactivate their legacy VGA registers shouldn’t set the DISP_CAP_MULTI_MONITOR_SAFE flag in the caps member of the disp_adapter_t structure. At initialization time, the driver should ensure that the device doesn’t respond to I/O or memory cycles to legacy VGA addresses (e.g. the memory aperture from 0xa0000 to 0xbffff, or VGA I/O ports within in the 0x3b4 to 0x3d5 range). If the driver needs to activate the VGA legacy registers, and it has set the DISP_CAP_MULTI_MONITOR_SAFE flags, it should call this function before activating the registers.

For non-x86 architectures, VGA register conflicts aren’t a concern, and this function has no effect.

**disp_release_vga_resources()**

This function is the opposite of disp_acquire_vga_resources(); call it when you’re done with the VGA registers. The prototype is:

```c
int disp_release_vga_resources ( 
    disp_adapter_t *adapter)
```

You must deactivate the card’s response to legacy VGA cycles before calling this function.

For non-x86 architectures, VGA register conflicts aren’t a concern, and this function has no effect.
**disp_perror()**

This function is similar to the standard C library’s `perror()` function. The prototype is:

```c
void disp_perror (disp_adapter_t *adapter,
                  char *what)
```

It outputs the string given by `what` along with the string interpretation of the global `errno` to the graphics framework’s debug stream (as given in the `dbgfile` member of the `disp_adapter_t` structure).

**disp_printf()**

This function is similar to the standard C library’s `printf()` and `fprintf()` functions. The prototype is:

```c
void disp_printf (disp_adapter_t *adapter,
                  const char *fmt, ...)
```

It outputs the given string (starting with the `fmt` parameter and any additional parameters specified) to the graphics framework’s debug stream (as given in the `dbgfile` member of the `disp_adapter_t` structure).

**disp_wait_condition()**

This function provides a mechanism for waiting for a condition, while allowing other processes to be scheduled in the meantime. The prototype is:

```c
void disp_wait_condition (disp_adapter_t *adapter,
                          int (*check)(disp_adapter_t *,
                                       void *arg),
                          void *arg)
```

This is better than polling, or busy-waiting, since it allows other processes to access the CPU, instead of wasting CPU cycles. The `check` function is a callback that indicates whether or not the condition has occurred yet. The `check` function should return 0 if the condition hasn’t yet occurred, or nonzero if the condition has occurred. The `arg` parameter is passed as the `arg` parameter to the callback function.
disp_usecspin()

This function busy waits for at least usecs microseconds. The prototype is:

```c
void disp_usecspin (unsigned usecs)
```

While polling is generally discouraged, sometimes the hardware demands that registers be accessed only after a certain (small) delay. If longer delays are necessary, use the disp_wait_condition function, which allows other processes to be scheduled, instead of wasting CPU cycles.

disp_get_rom_image()

This function gets the adapter’s ROM image, if present. The prototype is:

```c
void disp_get_rom_image (disp_adapter_t *adapter,
    int code_type,
    int *size,
    int map_pci_base_index,
    int map_offset)
```

For PCI devices, this function first attempts to fetch the ROM from the PCI ROM aperture. Note that the ROM image in the PCI aperture may be different on x86 systems from the image that typically shows up at physical address \texttt{0xc0000}. This is because the POST (Power On Self Test) process may modify and shrink the ROM image. See the PCI specification for more details. If the ROM can’t be found in the PCI aperture, then, (x86 systems only) the function looks for a ROM image at physical address \texttt{0xc0000}.

The `code_type` parameter specifies the type of ROM image to fetch from the ROM aperture. The PCI specification defines the ROM types, and allows multiple ROM images to be present in the PCI ROM aperture, in order to allow multiple platforms to be supported. A value of 0 for `rom_type` specifies an x86 ROM image.

On some systems, the BIOS doesn’t set up the bridge chips correctly, such that it isn’t possible to map the ROM aperture. To work around
this, a trick is used, whereby the ROM is mapped inside of one of the
device’s memory apertures (usually the frame buffer). The
map_pci_base argument specifies in which of the six pci apertures the
ROM should be mapped. The map_map_offset argument specifies the
offset into this aperture at which the ROM should be mapped.

If size is non-NULL, disp_get_rom_image() sets *size to the size of the
ROM, in bytes.

disp_vga_save_state()

This function reads the values of the generic VGA registers and saves
them in the disp_vga_state_t structure pointed to by state. The
prototype is:

```c
void disp_vga_save_state( disp_adapter_t *ctx,
                         disp_vga_state_t *state);
```

disp_vga_restore_state()

This function restores the state of the VGA registers from the values
saved in the disp_vga_state_t structure pointed to by state. The
prototype is:

```c
void disp_vga_restore_state( 
                         disp_adapter_t *ctx,
                         disp_vga_state_t *state);
```

disp_vga_state_t

This structure is used to save the state of the VGA registers:

```c
typedef struct {
    uint8_t misc_out;
    uint8_t seq[5];
    uint8_t crtc[25];
    uint8_t gc[9];
    uint8_t attr[21];
    uint8_t pal[64*3];
} disp_vga_state_t;
```
PCI configuration access functions

The PCI configuration access utilities set includes:

- `disp_pci_init()`
- `disp_pci_shutdown()`
- `disp_pci_read_config()`
- `disp_pci_write_config()`
- `disp_pci_dev_find()`
- `disp_pci_dev_read_config()`
- `disp_pci_dev_write_config()`
- `disp_pci_info()`

**disp_pci_init()**

This function attaches the driver to a PCI device. The prototype is:

```c
int disp_pci_init (disp_adapter_t *adapter,
                   unsigned flags)
```

It attaches the driver to the PCI device specified by the `pci_vendor_id`, `pci_device_id`, and `pci_index` members of the `adapter` structure.

The `flags` argument may be one of:

- `DISP_PCI_INIT_IRQ`  
  Allocate an IRQ for the device.

- `DISP_PCI_BASE[0-5]`  
  Allocate a memory range and initialize the corresponding PCI apertures.

- `DISP_PCI_BASES`  
  Allocate memory ranges and initialize all PCI apertures that are implemented on the device.
DISP_PCI_ALL  Allocate memory ranges and initialize all PCI
apertures that are implemented on the device. Also
allocate a ROM aperture, and an IRQ, if
implemented by the device.

If successful, disp_pci_init() sets adapter->bus.type to DISP_BUS_PCI,
initializes the adapter->bus.pci.base, adapter->bus.pci.apsize, and
adapter->irq members, and returns 0. Otherwise, this function returns
-1.

Call this function before calling any other PCI-related functions in the
disputil library.

disp_pci_shutdown()

This function detaches the driver from the device and releases the
resources from a previous disp_pci_init() call. The prototype is:


int disp_pci_shutdown (disp_adapter_t *adapter)

disp_pci_read_config()

This function reads a PCI configuration register. The prototype is:


int disp_pci_read_config (disp_adapter_t *adapter,
unsigned offset,
unsigned count,
sizet size,
void *bufptr)

PCI configuration registers can be byte, word, or double-word. This
function reads a PCI configuration register (or registers if count is
greater than one), as given by offset and size, into the data area given
by bufptr. For details on the return values, see pci_read_config() in the
QNX Neutrino Library Reference.
**disp_pci_write_config()**

This function writes a PCI configuration register. The prototype is:

```c
int disp_pci_write_config (disp_adapter_t *adapter,
                        unsigned offset,
                        unsigned count,
                        size_t size,
                        void *bufptr)
```

It writes a PCI configuration register (or registers if `count` is greater than one), as given by `offset` and `size`, from the data area given by `bufptr`. For details on the return values, see `pci_write_config()` in the QNX Neutrino Library Reference.

**disp_pci_dev_find()**

This function is similar to `pci_find_device()`. The prototype is:

```c
int disp_pci_dev_find (unsigned devid,
                    unsigned venid,
                    unsigned index,
                    unsigned *bus,
                    unsigned *devfunc)
```

It discovers a device’s `bus` and `devfunc` values in order to let a driver talk to a PCI device other than the one specified in the `disp_adapter_t` structure.

**disp_pci_dev_read_config()**

This function reads a PCI configuration register. The prototype is:

```c
int disp_pci_dev_read_config (unsigned bus,
                        unsigned devfunc,
                        unsigned offset,
                        unsigned cnt,
                        size_t size,
                        void *bufptr)
```

This function reads a PCI configuration register (like `disp_pci_read_config()`, above), but from a specific `bus` and device (`devfunc`). Error return codes are documented in `pci_read_config()`.
**disp_pci_dev_write_config()**

This function writes a PCI configuration register. The prototype is:

```c
int disp_pci_dev_write_config (unsigned bus,
                                unsigned devfunc,
                                unsigned offset,
                                unsigned cnt,
                                size_t size,
                                void *bufptr)
```

This function writes a PCI configuration register (like `disp_pci_write_config()`, above), but to a specific bus and device (devfunc). Error return codes are documented in `pci_write_config()`.

**disp_pci_info()**

This function is a cover function for `pci_present()`. The prototype is:

```c
int disp_pci_info (unsigned *lastbus,
                   unsigned *version,
                   unsigned *hardware)
```

**Memory manager functions**

The memory-manager utilities set includes:

- `disp_mmap_device_memory()`
- `disp_mmap_device_io()`
- `disp_munmap_device_memory()`
- `disp_phys_addr()`
- `disp_getmem()`
- `disp_freemem()`
**disp mmap_device_memory()**

The prototype is:

```c
void *disp_mmap_device_memory (paddr_t base,
                                  size_t len,
                                  int prot,
                                  int flags)
```

It creates a virtual address space pointer to the physical address given in `base`, which is `len` bytes in length. The `prot` parameter is selected from one or more of the following bitmapped flags:

- DISP_PROT_READ
  - Allow read access.
- DISP_PROT_WRITE
  - Allow write access.
- DISP_PROT_NOCACHE
  - Don’t cache the memory (useful for register access, for example).

The `flags` parameter is selected from one or more of the following bitmapped flags:

- DISP_MAP.LAZY
  - Allows the CPU to delay writes and combine them into burst writes for performance. It’s ideal for mapping frame buffers (Intel calls it write combining, some other vendors call it write gathering). On CPUs that don’t support this feature, the flag is ignored.

**disp mmap_device io()**

This function creates either a virtual address space pointer (like `disp mmap_device_memory()`, above), or returns its argument `base`. The prototype is:
unsigned long disp_mmap_device_io (size_t  len,
        paddr_t  base)

On non-x86 architectures, this function returns a virtual address space
pointer (because these don’t have a separate I/O space), whereas for
x86 architectures it returns the argument base unmodified. Regardless
of the architecture, the return value can be used with functions such as
in8() and out8().

**disp_munmap_device_memory()**

This function releases any resources that were acquired by
disp_mmap_device_memory(). The prototype is:

void disp_munmap_device_memory (void *addr,
        size_t  len)

It invalidates (unmaps) the virtual address pointer in addr.

**disp_phys_addr()**

This function returns the physical address corresponding to the virtual
address passed in addr. The prototype is:

paddr_t disp_phys_addr (void *addr)

This call is useful with devices that use DMA (which must be
programmed with the physical address of the transfer area), under
architectures (e.g. x86) where device I/O and memory cycles aren’t
translated by the MMU.

Note that the paddr_t physical address is valid only for a maximum
of _PAGESIZE bytes (i.e. from the physical address corresponding to
the passed virtual address up to and including the end of the page
boundary), unless the memory is physically contiguous.

For example, if _PAGESIZE is 4096 (0x1000), and the virtual address
translated to a physical address of 0x7B000100, then only the
physical address range 0x7B000100 through to 0x7B000FFF
(inclusive) is valid.
**disp_getmem()**

This function allocates memory. The prototype is:

```c
void *disp_getmem (int size,
                   unsigned prot,
                   unsigned flags)
```

It allocates `size` bytes of memory that conform to the `prot` and `flags` parameters.

Allocations are rounded up to be a multiple of `_PAGESIZE`. For example, even if `size` is 1, `_PAGESIZE` bytes of system memory are used up.

The `size`, `prot` and `flags` arguments are the same as those passed to `disp_mmap_device_memory()` above, with the following additional bits for `flags`:

**DISP_MAP_PHYS**

Pages of memory allocated are physically contiguous. That is, the memory is a contiguous block, from both the device’s and CPU’s perspectives.

**DISP_MAP_BELOW_16M**

The memory allocated is located below the physical address `0x1000000`. This flag is useful for certain legacy x86 devices.

Note that you don’t supply a `base` parameter as with the other mapping function; instead, `disp_getmem()` finds a free block of memory (called anonymous memory) and allocates it.

This function returns a pointer (virtual address) to the memory, or NULL if the memory couldn’t be allocated.
**disp_freemem()**

This function invalidates the virtual address pointer in `addr` and deallocates the memory. The prototype is:

```c
void disp_freemem (void *addr,
                   int size)
```

**Video memory management functions**

The video memory management utilities set includes:

- `disp_vm_create_pool()`
- `disp_vm_destroy_pool()`
- `disp_vm_alloc_surface()`
- `disp_vm_free_surface()`
- `disp_vm_mem_avail()`

These functions are primarily intended to be called from your driver’s Video Memory Manager module. A simple Video Memory Manager is little more than a wrapper for these functions.

**disp_vm_create_pool()**

This function creates a new memory pool for the memory manager. The prototype is:

```c
disp_vm_pool_t *disp_vm_create_pool (   
    disp_adapter_t *adapter,
    disp_surface_t *surf,
    int bytealign)
```

Pass the `adapter` associated with this memory pool, a pointer to the surface in `surf`, and a byte alignment parameter, `bytealign`. The `bytealign` argument indicates the alignment for the memory manager — all chunks of memory returned by the memory manager for this pool are aligned to the number of bytes specified.
The `surf` argument describes the block of memory that the driver wants to have managed. The driver should set the `vidptr`, `offset` and `paddr` (if appropriate) members of the surface structure. When the libraries’ memory management routines are used to allocate a surface, the library uses these initial values to calculate the corresponding values for the allocated surface.

The `pixel_format` member of `surf` is typically set to `DISP_SURFACE_FORMAT_BYTES`, since the unallocated memory doesn’t have any particular pixel format. The `stride` and `height` members should be set such that `stride \times height` is equal to the size of the memory block. For example, the height could be set to 1 and the stride set to the size of the memory block.

Set the `flags` member to describe the characteristics of the video RAM.

A driver can create multiple pools, for example, if there are multiple regions of memory that have different characteristics. This is the case if not all of the devices RAM are addressable by the display controller. In this case, the driver might create a pool of CRTC-addressable memory, and a pool of non-CRTC-addressable memory. When the driver is asked to allocate some memory, it tries to obtain the memory from the non-CRTC-addressable pool first (assuming the framework hasn’t explicitly requested displayable memory via the `DISP_SURFACE_DISPLAYABLE` flag). If the allocation from the non-CRTC-addressable pool fails, the driver then tries to obtain the memory from the CRTC-addressable pool.

The return value is a `pool_handle_t` that’s passed to subsequent `disp_vm_*()` function calls.

**disp_vm_destroy_pool()**

This function deallocates all surfaces that were allocated from the pool and releases the resources associated with tracking the pool allocation. The prototype is:

```c
int disp_vm_destroy_pool (disp_adapter_t *adapter,
                        disp_vm_pool_t *pool)
```
**disp.vm_alloc_surface()**

This function allocates a surface from the memory pool specified by `pool`. The prototype is:

```c
disp_surface_t *disp_vm_alloc_surface (pool_handle_t *pool,
                   int width,
                   int height,
                   int stride,
                   unsigned format,
                   unsigned flags)
```

The `flags` parameter is selected from the set of manifest constants defined in the `flags` argument for the `disp_surface_t` data type. This specifies the characteristics of the memory that’s being requested. If there’s no memory available with the specified characteristics, the function should return NULL. Otherwise, this function returns a pointer to a `disp_surface_t` structure, which describes the allocated memory.

The `width`, `height` and `stride` parameters define the area and stride of the surface to be allocated.

The `pixel_format` member of the `disp_surface_t` structure is set to the value specified by the `pixel_format` argument.

The `format` parameter is selected from the set of manifest constants beginning with `DISP_SURFACE_FORMAT_*`. For more information, see “Pixel formats,” under the description for `devg_get_corefuncs()` in the Writing a Graphics Driver chapter.

**disp.vm_free_surface()**

This function releases the surface memory identified by `surf` back into the surface memory manager’s pool. It also frees the `surf` structure. The prototype is:

```c
int disp_vm_free_surface (disp_adapter_t *adapter,
                         disp_surface_t surf)
```

This function returns 0, or -1 if an error occurred.
disp_vm_mem_avail()

This function returns how much memory is available in the pool identified by pool, in bytes. The prototype is:

```c
unsigned long disp_vm_mem_avail ( 
    disp_vm_pool_t *pool, 
    unsigned sflags)
```

It should report only memory that has at least the characteristics specified by sflags.

Video BIOS services (x86 only)

The video BIOS services include:

- `vbios_register()`
- `vbios_unregister()`
- `vbios_int()`
- `vbios_call()`
- `vbios_dowarmboot()`
- `vbios_get_reaptr()`
- `vbios_context_t`

The `vbios_*` routines provide drivers with the ability to make calls to the VGA / VESA BIOS. These calls are often referred to as Int 10 calls, since the Video BIOS is generally accessed via vector 0x10 in the Real Mode Interrupt Vector table.

These services are available only on x86 platforms; this section isn’t relevant to non-x86 architectures or to drivers that are targeted toward machines that don’t have a Video BIOS.

Typically, graphics drivers that use the Video BIOS use only the VESA / VGA Modeswitching services, performing other functions, such as drawing, by directly programming the hardware. Often, using the BIOS can be the most reliable method of writing a modeswitcher,
since the BIOS on the video card is tailor-made for the device to which it is attached.

One of the main drawbacks of using the BIOS to switch modes is that the BIOS might not provide a method of selecting a video refresh rate. However, some vendors do provide a way to set a specific refresh rate using the BIOS. If the BIOS for a particular device doesn’t have refresh-rate support, you need to write a direct modeswitcher to program the hardware directly, in order to provide refresh rate support.

Video BIOSes are generally implemented with real 8086 code. Hence special support is required to allow execution of the BIOS under a memory-protected operating system.

Under QNX Neutrino, there’s special support in the kernel that allows the Video BIOS to be executed in the x86 processor’s Virtual 8086 mode.

vbios_register()

This function registers your driver with the VBIOS services. You should call this function before calling any of the other vbios_* routines. The prototype is:

```c
int vbios_register( disp_adapter_t *adp, 
                   unsigned flags );
```

The `adp` argument is a pointer to your driver’s `disp_adapter_t` structure. There are currently no flags defined; pass 0 for the `flags` argument.

If your driver uses the video BIOS functions as well as the `disp_*` functions, it must call `vbios_register()` before calling `disp_register_adapter()`.

This function stores a pointer to a `vbios_context_t` structure in the `vbios` member of the structure pointed to by `adp`. This pointer is to be used in subsequent calls to `vbios_*` functions.
This function initializes the `xfer_area_seg`, `xfer_area_off`, and `xfer_area_ptr` members of the `vbios_context_t` structure. These members point to a communications area that’s used to supply data to and retrieve data from the BIOS code. The `xfer_area_ptr` member points to the communications area in the driver’s address space. Together, `xfer_area_seg` and `xfer_area_off` specify the real-mode address that the BIOS code can use to address the communications area.

**vbios_unregister()**

You driver should call this function when it no longer requires VBIOS services. The prototype is:

```c
void vbios_unregister( vbios_context_t *ctx );
```

The `ctx` argument is a pointer to the `vbios_context_t` structure that `vbios_register()` stored in the `vbios` member of the driver’s `disp_adapter_t` structure.

**vbios_int()**

This function makes a BIOS call. The prototype is:

```c
int vbios_int( vbios_context_t *ctx,
               int inum,
               vbios_regs_t *regs,
               int xfer_size );
```

The arguments are:

- `ctx` A pointer to the `vbios_context_t` structure that `vbios_register()` stored in the `vbios` member of the driver’s `disp_adapter_t` structure.
- `inum` The interrupt vector number; to call the Video BIOS, specify `0x10`.
- `regs` The contents of the CPU’s registers on entry to the call.
**xfer_size** The number of bytes to transfer from the communications area. The specified number of bytes are copied from the driver’s address space to the real-mode communications area before the call, and the same number of bytes are copied back into the driver’s address space after the call.

The driver accesses the communications area data via the `xfer_area_ptr` member of `ctx`, whereas the BIOS code accesses it using the real-mode address specified by `xfer_area_seg` and `xfer_area_off`.

### `vbios_call()`

This function makes a real-mode call. The prototype is:

```c
int vbios_call( vbios_context_t *ctx,
                 int seg,
                 int offs,
                 vbios_regs_t *regs,
                 int xfer_size );
```

The `ctx` argument is a pointer to the `vbios_context_t` structure that `vbios_register()` stored in the `vbios` member of the driver’s `disp_adapter_t` structure.

The `seg` and `offs` arguments specify the real-mode address of the routine to call. This function is similar to `vbios_int()`, except that the address of the routine to execute is passed directly, rather than coming from the real-mode Interrupt Vector Table.

### `vbios_dowarmboot()`

The prototype is:

```c
int vbios_dowarmboot( vbios_context_t *ctx,
                       unsigned ax_val,
                       unsigned char *biosptr,
                       int bios_size );
```

In the event that there’s more than one video card in the system, it may be necessary to perform a warm boot of the non-primary cards.
At boot time, the system BIOS selects one of the adapters to be the BIOS primary. The system then calls the BIOS of this card in order to initialize the card and put it into text mode. Meanwhile, other graphics devices stay dormant.

Many drivers don’t duplicate the code that's present in the BIOS, and which performs this device initialization; they rely on the fact that the BIOS has already performed basic device initialization. In fact, it often isn’t possible to perform this initialization, since the driver may lack certain knowledge that’s necessary to initialize the device (e.g. what type of memory is present on the adapter).

However, on devices that are compliant to PCI 2.1 (or greater), it’s possible to perform the BIOS warm boot sequence for one of the nonprimary cards. This involves disabling the active VGA device and performing an initialization sequence similar to the sequence performed by the POST (Power On Self Test) on the primary VGA device, at boot time.

If your driver can’t initialize the graphics device from scratch, you should call this routine. It determines whether or not the device has already been warm booted, and performs the BIOS POST sequence, if necessary.

The arguments to `vbios_dowarmboot()` are:

- `ctx` A pointer to the `vbios_context_t` structure that `vbios_register()` stored in the `vbios` member of the driver’s `disp_adapter_t` structure.
- `ax_val` The value to be put into the AX register; see the PCI specification.
- `biosptr` A pointer to the BIOS image, which you can get by calling `disp_get_rom_image()`.
- `bios_size` The size of the BIOS image, in bytes.
vbios_get_realptr()

The prototype is:

```c
void *vbios_get_realptr( vbios_context_t *ctx,
    unsigned seg,
    unsigned off );
```

This function takes the real-mode address specified via the segment `seg` and the offset `off`, and returns a pointer to this memory within the driver’s address space.

The `ctx` argument is a pointer to the `vbios_context_t` structure that `vbios_register()` stored in the `vbios` member of the driver’s `disp_adapter_t` structure.

Since not all addresses within the real-mode address range (0x00 - 0x10ffff) are usable by the driver or by the BIOS, this function can return NULL. Generally, the address specified by `seg:off` should reside either within the first 4K of memory, or within the range 0xa0000 - 0x100fff (640K to 1Meg + 4K).

vbios_context_t

This structure stores the Video BIOS context information;
`vbios_register()` stores a pointer to this structure in the `vbios` member of the driver’s `disp_adapter_t` structure.

```c
typedef struct vbios_context {
    disp_adapter_t *adp;
    unsigned xfer_area_seg;
    unsigned xfer_area_off;
    unsigned char *xfer_area_ptr;
} vbios_context_t;
```

- `adp` A pointer to the device’s adapter structure.
- `xfer_area_seg` The real-mode address of the communications area.
- `xfer_area_ptr` The virtual address of the communications area.

vbios_get_realptr()
FFB library — 2D software fallback routines

The FFB (Flat Frame Buffer) library includes:

Core functions that you can fall back on in `disp_draw_corefuncs_t`:

- `ffb_core_blit1()`
- `ffb_core_blit2()`
- `ffb_draw_span_8()`, `ffb_draw_span_16()`, `ffb_draw_span_24()`, and `ffb_draw_span_32()`
- `ffb_draw_span_list_8()`, `ffb_draw_span_list_16()`, `ffb_draw_span_list_24()`, and `ffb_draw_span_list_32()`
- `ffb_draw_solid_rect_8()`, `ffb_draw_solid_rect_16()`, `ffb_draw_solid_rect_24()`, and `ffb_draw_solid_rect_32()`
- `ffb_draw_line_pat8x1_8()`, `ffb_draw_line_pat8x1_16()`, `ffb_draw_line_pat8x1_24()`, and `ffb_draw_line_pat8x1_32()`
- `ffb_draw_line_trans8x1_8()`, `ffb_draw_line_trans8x1_16()`, `ffb_draw_line_trans8x1_24()`, and `ffb_draw_line_trans8x1_32()`
- `ffb_draw_rect_pat8x8_8()`, `ffb_draw_rect_pat8x8_16()`, `ffb_draw_rect_pat8x8_24()`, and `ffb_draw_rect_pat8x8_32()`
- `ffb_draw_rect_trans8x8_8()`, `ffb_draw_rect_trans8x8_16()`, `ffb_draw_rect_trans8x8_24()`, and `ffb_draw_rect_trans8x8_32()`

Context functions that you can fall back on in `disp_draw_contextfuncs_t`:

- `ffb_ctx_draw_span()`
- `ffb_ctx_draw_span_list()`
- `ffb_ctx_draw_rect()`
- `ffb_ctx_blit()`
Draw state update notify functions:

- `ffb_update_draw_surface()`
- `ffb_update_pattern()`
- `ffb_ctx_update_general()`
- `ffb_ctx_update_color()`
- `ffb_ctx_update_rop3()`
- `ffb_ctx_update_chroma()`
- `ffb_ctx_update_alpha()`

Color space conversion utility:

- `ffb_color_translate()`

Miscellaneous:

- `ffb_hw_idle()`
- `ffb_wait_idle()`

Draw function retrieval routines:

- `ffb_get_corefuncs()`
- `ffb_get_contextfuncs()`

You can use these functions when you create your driver. For example, you may start out with a driver that doesn’t actually do very much, and instead relies upon the functionality of these routines to perform the work. As you progress in your development cycle, you can take over more and more functionality from these functions and do them in a card-specific manner (e.g. using the hardware acceleration).

In general, this can be done quite simply by taking the function table pointer that’s passed to you in your initialization function, and calling the appropriate function (`ffb_get_corefuncs()` or `ffb_get_contextfuncs()`).
to populate your function table array with the defaults from this library. Note, however, that all functions in the library are exposed; you don’t have to bind to them by way of the \texttt{ffb\_get\_*()} functions; you can just simply link against them.

The next step in the development cycle is to take over some of the functions, and follow the outlines discussed above for each of them. If you find that you’re able to support a given operation in a card-specific manner, demultiplex that case out of the function call and handle it, while relying on the library routines to perform functions that your hardware doesn’t support or that you don’t wish to write the code for right at that point. Since the supplied libraries are hardware-independent (i.e. everything is implemented in software), they’re likely to be much slower than your hardware-accelerated versions.

Another advantage of the way that the library and graphics framework are structured is that in case your driver becomes out-of-date (i.e. a newer version of the graphics framework has been released that has more functions), the shared library that’s supplied with the newer version will know how to handle the extra functions, without any additional intervention on your part. You may then release a new version of your driver that supports accelerated versions of the extra function(s) at your convenience.
There are four sets of functions for some of the core functions supplied, optimized based on the pixel depth. For example, instead of the expected single function `ffb_draw_span()`, there are in fact four of them:

1. `ffb_draw_span_8()`
2. `ffb_draw_span_16()`
3. `ffb_draw_span_24()`
4. `ffb_draw_span_32()`

Use the `pixel_format` argument passed to `ffb_get_corefuncs()` to determine which function to bind to the `draw_span` entry in the `disp_draw_corefuncs_t` function table.

The other functions (that aren’t listed as having 8/16/24/32 bit pixel-depth variants) support all pixel depths.

The impact on your driver is that you may choose to call the `ffb_get_corefuncs()` four times, (once for each color depth), and fill four separate arrays, or you may choose to call it whenever your `get_corefuncs()` call-in is called, so that you can dynamically bind the appropriate library routines. The `get_corefuncs()` call-in gets called very infrequently (only during initialization and modeswitch operations) so efficiency isn’t of paramount importance in this case.

### `ffb_get_corefuncs()`

This function populates the function table pointed to by `funcs` with the core functions from the Flat Frame Buffer library. The prototype is:

```c
int ffb_get_corefuncs (  
    disp_adapter_t *context,  
    unsigned pixel_format,  
    disp_draw_corefuncs_t *funcs,  
    int tabsize)
```

This function returns 0 on success, or -1 if an error occurred (e.g. the pixel format is invalid).
**ffb_get_contextfuncs()**

This function populates the function table pointed to by `funcs` with the context functions from the Flat Frame Buffer library. The prototype is:

```c
int ffb_get_contextfuncs (  
    disp_adapter_t *context,  
    disp_draw_contextfuncs_t *funcs,  
    int tabsize)
```

This function returns 0 on success, or -1 if an error occurred.

**ffb_color_translate()**

This function takes the `color` that corresponds to the surface type specified by `srcformat`, and returns a `disp_color_t` that corresponds to the same (or closest available) color in the surface type specified by `dstformat`. The prototype is:

```c
disp_color_t ffb_color_translate (  
    disp_draw_context_t *context,  
    int srcformat,  
    int dstformat,  
    disp_color_t color)
```

For more information about the surface types, see “Pixel formats” in the Writing a Graphics Driver chapter.

Note that this function doesn’t yet handle a palette-based format for `dstformat`. 

---

November 2, 2006 Chapter 5 • Libraries 197
AGP

Accelerated Graphics Port; a high-performance bus designed specifically for graphical devices.

alpha-blending

A technique of portraying transparency when drawing an object. It combines the color of an object to be drawn (the source) and the color of whatever the object is to be drawn on top of (the destination). The higher the blending factor of the source object, the more opaque the object looks.

Mathematically, a blending factor is a real number between 0 and 1, inclusive. A 32-bit color is made up of four 8-bit channels: alpha, red, green, and blue. These channels are represented as (A, R, G, B). When referring to the source, the channels are denoted as As, Rs, Gs, and Bs; for the destination, they’re Ad, Rd, Gd, and Bd.

BIOS

Basic Input/Output System; standard ROM services that are available on x86 platforms.

blitting

BLock Image Transfer; copying an image from one place on the screen to another.

chroma keying

A method of masking out pixel data during a rendering operation (blits, image rendering, etc.) based on a key color value.

CRTC

Cathode-Ray Tube Controller.

DPMS

Display Power Management System; a system that reduces the amount of energy consumed by a monitor when it’s idle.
falling back on software

Using software to perform operations that the graphics hardware doesn’t support.

FFB

Flat Frame Buffer

PCI

Peripheral Components Interface; a general-purpose system bus that’s found in most modern PCs.

pixel

A picture element, one of the dots that make up an image on the screen.

raster operations (ROP)

A per-pixel operation to be performed when rendering an object. A raster operation may involve up to three input parameters (source, pattern, and destination) that are combined in accordance with the raster operation that’s currently in effect. The resulting pixel value is written to the destination.

ROP3

A set of 256 raster operations, as defined by Microsoft.

stride

The number of bytes from the beginning of one scanline to the beginning of the next.

surface

A two-dimensional area.
ternary raster operation

A raster operation involving 3 input parameters.

VESPA

Video Electronics Standards Association.
Index

! 16

<display.h> 16

1

2D

capabilities, getting 60, 98
drawing functions 10, 13
FFB versions 10
getting 48
driver acceleration 65

3D driver acceleration 65

A

adapter, graphics hardware 63
AGP bus 64
alloc_layer_surface() 108
alloc_surface() 107
alpha blending
factors
destination 76, 79

source 76, 79
map
offset 77
size 76
modes 76, 78
pixel formats 27
updating 84
video viewport 157

B

background color 75
updating 83
BIOS services 187
bitmaps, drawing 91
blit() 82
blit1() 90
blit2() 90
blitting 82, 90
bus

AGP 64
ISA 64
PCI 64
unknown 64
Index

© 2006, QNX Software Systems GmbH & Co. KG.

VESA local 64

crttrap 13

C

capabilities
  information 100
chroma key 14
colors 18, 76
mode 76, 77
updating 84
circular patterns 23
clipping 26
close_channel() 167
disp_vcapfuncs_t 156
colors
  background and foreground 75
  updating 83
  chroma key 18, 84
  formats 18, 27
  palette, setting 95, 120
connectors 4
context
  2D drawing functions 16, 19, 81
  fallback routines 20, 193
  information 20, 74
  updating 20, 83
coordinates
  inclusiveness 19
  order of 19
  origin 18
core 2D drawing functions 16, 19, 85
  fallback routines 20, 193

CRT Controller (CRTC) settings 70

D

DDK_REVISION 136
DDK_VERSION_MAJOR 136
DDK_VERSION_MINOR 136
devg_get_contextfuncs() 16, 21, 46
devg_get_corefuncs() 16, 21, 48
devg_get_memfuncs() 16, 50
devg_get_miscfuncs() 16, 52
devg_get_modefuncs() 16, 54
devg_get_vcapfuncs() 16, 56
devg_get_vidfuncs() 16, 58
disable_hw_cursor() 97
disp_2d_caps_t 60
disp_adapter_t 17, 63
disp_crtc_settings_t 70
disp_draw_context_t 20, 74
disp_draw_contextfuncs_t 23, 81
  blit() 82
  draw_rect() 82
  draw_span() 81
  draw_span_list() 82
  fallback functions 20, 21, 193
  update_alpha() 20, 84
  update_chroma() 20, 84
  update_color() 20, 83
  update_general() 20, 83
  update_rop3() 20, 83
disp_draw_corefuncs_t 85
  blit1() 90
  blit2() 90
draw_bitmap() 91
Index

fini() 153
get_channel_caps() 154
get_channel_status() 155
init() 153
module_info() 154
set_channel_props() 155
disp_vga_state_t 176
disp_vid_alpha_t 157
disp_vid_channel_caps_t 159
disp_vid_channel_props_t 161
disp_vidfuncs_t 164
close_channel() 167
fini() 164
get_channel_caps() 165
init() 164
module_info() 165
next_frame() 166
set_channel_props() 165
DISP_2D_ACCEL_COMPLEX_ROPS 60
DISP_2D_ACCEL_DSTALPHA_BLEND_GLOBAL 61
DISP_2D_ACCEL_DSTCHROMA 61
DISP_2D_ACCEL_MONO_PAT 60
DISP_2D_ACCEL_OPAQUE_BLIT 60
DISP_2D_ACCEL_OPAQUE_FILL 60
DISP_2D_ACCEL_SIMPLE_ROPS 60
DISP_2D_ACCEL_SRCALPHA_BLEND_GLOBAL 61
DISP_2D_ACCEL_SRCALPHA_BLEND_MAP 61
DISP_2D_ACCEL_SRCALPHA_BLEND_PIXEL 61
DISP_2D_ACCEL_SRC_CHROMA 61
DISP_2D_ACCEL_TRANS_PAT 60
DISP_2D_SRC_DST_STRIDE_EQUAL 61
disp_acquire_vga_resources() 173
DISP_ADD_FUNC() 17, 67
DISP_ALPHA_M1_DST_PIXEL_ALPH A 78
DISP_ALPHA_M1_GLOBAL 78
DISP_ALPHA_M1_SRC_PIXEL_ALPH A 78
DISP_ALPHA_M2_DST_PIXEL_ALPH A 78
DISP_ALPHA_M2_GLOBAL 79
DISP_ALPHA_M2_SRC_PIXEL_ALPH A 78
DISP_BLEND_DST_FACTOR 79
DISP_BLEND_DST_M1_ALPHA 79
DISP_BLEND_DST_M2_ALPHA 79
DISP_BLEND_DST_ONE_MINUS_M1_ALPHA 79
DISP_BLEND_DST_ONE_MINUS_M2_ALPHA 79
DISP_BLEND_SRC_M1_ALPHA 79
DISP_BLEND_SRC_M2_ALPHA 79
DISP_BLEND_SRC_ONE_MINUS_M1_ALPHA 79
DISP_BLEND_SRC_ONE_MINUS_M2_ALPHA 79
DISP_BLEN D_DST_TYPE_AG P 64
DISP_BLEN D_DST_TYPE_J SA 64
DISP_BLEN D_DST_TYPE_P CI 64
DISP_BLEN D_DST_TYPE_U NKNOWN 64
DISP_BLEN D_DST_TYPE_V L 64
DISP_BYTES_PER_PIXEL() 29
DISP_bwritel bit 0 154
DISP_CAP_2D_ACCEL 65

208 Index November 2, 2006
DISP_CAP_3D_ACCEL 65
DISP_CAP_MULTI_MONITOR_SAFE 65, 173
DISP_CHROMA_OP_DRAW 76, 78
DISP_CHROMA_OP_DST_MATCH 76, 78
DISP_CHROMA_OP_NO_DRAW 76, 78
DISP_CHROMA_OP_SRC_MATCH 76, 77

disp_crtc_calc() 172
DISP_DRAW_FLAG_COMPLEX_ROP 75
DISP_DRAW_FLAG_MONO_PATTERN 75
DISP_DRAW_FLAG_SIMPLE_ROP 75
DISP_DRAW_FLAG_TRANS_PATTERN 75
DISP_DRAW_FLAG_USE_ALPHA 75

disp_freeemem() 184
disp_getmem() 183
disp_get_rom_image() 175
display offset setting 120

DISP_LAYER_FORMAT_ARGB1555 30
DISP_LAYER_FORMAT_ARGB8888 30
DISP_LAYER_FORMAT_PAL8 29
DISP_LAYER_FORMAT_RGB565 30
DISP_LAYER_FORMAT_RGB888 30
DISP_LAYER_FORMAT_UYVY 30
DISP_LAYER_FORMAT_V422 30
DISP_LAYER_FORMAT_YUV420 31
DISP_LAYER_FORMAT_YUV2 30
DISP_LAYER_FORMAT_YV12 31
DISP_LAYER_FORMAT_YVU9 31
DISP_LAYER_FORMAT_YVYU 30
DISP_MAP_BELOW_16M 183
DISP_MAP_LAZY 181
DISP_MAP_PHYS 183
DISP_MCAP_DPMS_SUPPORTED 113
DISP_MCAP_SET_DISPLAY_OFFSET 113
DISP_MCAP_VIRTUAL_PANNING 113
disp_mmap_device_io() 181
disp_mmap_device_memory() 181
DISP_MODE_GENERIC 112
DISP_MODE_LISTEND 117
DISP_MODE_NUM_REFRESH 113
DISP_MODE_TVOUT 112
DISP_MODE_TVOUT_NTSC 112
DISP_MODE_TVOUT_OR_MONITOR 112
DISP_MODE_TVOUT_OVERSCAN 112
DISP_MODE_TVOUT_PAL 112
DISP_MODE_TVOUT_SECAM 112
DISP_MODE_TVOUT_WITH_MONITOR 112
disp_munmap_device_memory() 182
DISP_PATTERN_FORMAT_MONO_8x1 76
DISP_PATTERN_FORMAT_MONO_8x8 76

DISP_PCI_ALL 178
DISP_PCI_BASE[0-5] 177
DISP_PCI_BASES 177
disp_pci_dev_find() 179
disp_pci_dev_read_config() 179
disp_pci_dev_write_config() 180
disp_pci_info() 180
disp_pci_init() 177
DISP_PCI_INIT_IRQ 177
disp_pci_read_config() 178
disp_pci_shutdown() 178
disp_pci_write_config() 179
disp_perror() 174
disp_phys_addr() 182

Index

November 2, 2006
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>disp_print()</code></td>
<td>174</td>
</tr>
<tr>
<td><code>DISP_PROT_NOCACHE</code></td>
<td>181</td>
</tr>
<tr>
<td><code>DISP_PROT_READ</code></td>
<td>181</td>
</tr>
<tr>
<td><code>DISP_PROT_WRITE</code></td>
<td>181</td>
</tr>
<tr>
<td><code>disp_register_adapter()</code></td>
<td>172</td>
</tr>
<tr>
<td><code>disp_release_vga_resources()</code></td>
<td>173</td>
</tr>
<tr>
<td><code>DISP_SURFACE_2D_READABLE</code></td>
<td>141</td>
</tr>
<tr>
<td><code>DISP_SURFACE_2D_TARGETABLE</code></td>
<td>141</td>
</tr>
<tr>
<td><code>DISP_SURFACE_3D_READABLE</code></td>
<td>141</td>
</tr>
<tr>
<td><code>DISP_SURFACE_3D_TARGETABLE</code></td>
<td>141</td>
</tr>
<tr>
<td><code>DISP_SURFACE_CPU_LINEAR_READABLE</code></td>
<td>140</td>
</tr>
<tr>
<td><code>DISP_SURFACE_CPU_LINEAR_WRITEABLE</code></td>
<td>140</td>
</tr>
<tr>
<td><code>DISP_SURFACE_DISPLAYABLE</code></td>
<td>140, 185</td>
</tr>
<tr>
<td><code>DISP_SURFACE_DMA_SAFE</code></td>
<td>141</td>
</tr>
<tr>
<td><code>DISP_SURFACE_FORMAT_ARGB1555</code></td>
<td>28</td>
</tr>
<tr>
<td><code>DISP_SURFACE_FORMAT_ARGB8888</code></td>
<td>28</td>
</tr>
<tr>
<td><code>DISP_SURFACE_FORMAT_BYTES</code></td>
<td>29, 184, 185</td>
</tr>
<tr>
<td><code>DISP_SURFACE_FORMAT_MONO</code></td>
<td>28</td>
</tr>
<tr>
<td><code>DISP_SURFACE_FORMAT_PACKEDYUV_UVYY</code></td>
<td>28</td>
</tr>
<tr>
<td><code>DISP_SURFACE_FORMAT_PACKEDYUV_V422</code></td>
<td>29</td>
</tr>
<tr>
<td><code>DISP_SURFACE_FORMAT_PACKEDYUV_YUY2</code></td>
<td>28</td>
</tr>
<tr>
<td><code>DISP_SURFACE_FORMAT_PACKEDYUV_YUVU</code></td>
<td>29</td>
</tr>
<tr>
<td><code>DISP_SURFACE_FORMAT_PAL8</code></td>
<td>28</td>
</tr>
<tr>
<td><code>DISP_SURFACE_FORMAT_RGB565</code></td>
<td>28</td>
</tr>
<tr>
<td><code>DISP_SURFACE_FORMAT_RGB888</code></td>
<td>28</td>
</tr>
<tr>
<td><code>DISP_SURFACE_FORMAT_UPLANE</code></td>
<td>29</td>
</tr>
<tr>
<td><code>DISP_SURFACE_FORMAT_VPLANE</code></td>
<td>29</td>
</tr>
<tr>
<td><code>DISP_SURFACE_FORMAT_YPLANE</code></td>
<td>29</td>
</tr>
<tr>
<td><code>DISP_SURFACE_NON_CACHEABLE</code></td>
<td>141</td>
</tr>
<tr>
<td><code>DISP_SURFACE_OTHER_ENDIAN</code></td>
<td>141</td>
</tr>
<tr>
<td><code>DISP_SURFACE_PAGE_ALIGNED</code></td>
<td>141</td>
</tr>
<tr>
<td><code>DISP_SURFACE_PHYS_CONTIG</code></td>
<td>141</td>
</tr>
<tr>
<td><code>DISP_SURFACE_SCALER_DISPLAYABLE</code></td>
<td>141</td>
</tr>
<tr>
<td><code>DISP_SURFACE_VMI_TARGETABLE</code></td>
<td>141</td>
</tr>
<tr>
<td><code>DISP_SYNC_POLARITY_H_POS</code></td>
<td>72</td>
</tr>
<tr>
<td><code>DISP_SYNC_POLARITY_NN</code></td>
<td>72</td>
</tr>
<tr>
<td><code>DISP_SYNC_POLARITY_NP</code></td>
<td>72</td>
</tr>
<tr>
<td><code>DISP_SYNC_POLARITY_PN</code></td>
<td>72</td>
</tr>
<tr>
<td><code>DISP_SYNC_POLARITY_PP</code></td>
<td>72</td>
</tr>
<tr>
<td><code>DISP_SYNC_POLARITY_V_POS</code></td>
<td>72</td>
</tr>
<tr>
<td><code>DISP_TVOUT_STANDARD()</code></td>
<td>113</td>
</tr>
<tr>
<td><code>DISP_TV_SIGNAL_*</code></td>
<td>144, 148</td>
</tr>
<tr>
<td><code>disp_unregister_adapter()</code></td>
<td>172</td>
</tr>
<tr>
<td><code>disp_usescspin()</code></td>
<td>175</td>
</tr>
<tr>
<td><code>disp_acquire_vga_resources()</code></td>
<td>173</td>
</tr>
<tr>
<td><code>disp_crtc_calc()</code></td>
<td>172</td>
</tr>
<tr>
<td><code>disp_fremem()</code></td>
<td>184</td>
</tr>
<tr>
<td><code>disp_getmem()</code></td>
<td>183</td>
</tr>
<tr>
<td><code>disp_get_rom_image()</code></td>
<td>175</td>
</tr>
<tr>
<td><code>disp_mmap_device_io()</code></td>
<td>181</td>
</tr>
<tr>
<td><code>disp_mmap_device_memory()</code></td>
<td>181</td>
</tr>
<tr>
<td><code>disp_munmap_device_memory()</code></td>
<td>182</td>
</tr>
<tr>
<td><code>disp_pci_dev_find()</code></td>
<td>179</td>
</tr>
<tr>
<td><code>disp_pci_dev_read_config()</code></td>
<td>179</td>
</tr>
<tr>
<td><code>disp_pci_dev_write_config()</code></td>
<td>180</td>
</tr>
<tr>
<td><code>disp_pci_info()</code></td>
<td>180</td>
</tr>
<tr>
<td><code>disp_pci_init()</code></td>
<td>177</td>
</tr>
<tr>
<td><code>disp_pci_read_config()</code></td>
<td>178</td>
</tr>
<tr>
<td><code>disp_pci_shutdown()</code></td>
<td>178</td>
</tr>
</tbody>
</table>
disp_pci_write_config() 179
disp_perr() 174
disp_phys_addr() 182
disp_printf() 174
disp_register_adapter() 172
disp_release_vga_resources() 173
disp_unregister_adapter() 172
disp_usecspin() 175
disp_vga_restore_state() 176
disp_vga_save_state() 176
disp_vm_alloc_surface() 186
disp_vm_create_pool() 184
disp_vm_destroy_pool() 185
disp_vm_free_surface() 186
disp_vm_mem_avail() 187
disp_wait_condition() 174
vbios_context_t 192
vbios_call() 190
vbios_dowarmboot() 190
vbios_get_realptr() 192
vbios_int() 189
vbios_register() 188
vbios_unregister() 189
DISP_VCAP_AUDIO_SOURCE_* 148
DISP_VCAP_CAP_* 143
DISP_VCAP_FLAG_* 147
DISP_VCAP_SRC_* 147
DISP_VCAP_STATUS_* 152
DISP_VCAP_UPDATE_* 150
disp_vga_restore_state() 176
disp_vga_save_state() 176
DISP_VID_CAP_BRIGHTNESS_ADJUST 160
DISP_VID_CAP_BUSTMASTER 159
DISP_VID_CAP_CONTRAST_ADJUST 160
DISP_VID_CAP_DOUBLE_BUFFER 159
DISP_VID_CAP_DST_CHROMA_KEY 159
DISP_VID_CAP_SRC_CHROMA_KEY 159
DISP_VID_CHROMA_FLAG_DST 162
DISP_VID_CHROMA_FLAG_SRC 162
DISP_VID_FLAG_CHROMA_ENABLE 162
DISP_VID_FLAG_DOUBLE_BUFFER 162, 166
DISP_VID_FLAG_ENABLE 161
disp_vm_alloc_surface() 186
disp_vm_create_pool() 184
disp_vm_destroy_pool() 185
disp_vm_free_surface() 186
disp_vm_mem_avail() 187
disp_wait_condition() 174
dlopen() 16
DPMS
mode, setting 119
support, indicating 113
draw stream interpreter 5
draw_bitmap() 91
drawing functions,
miscellaneous 94
draw_line_pat8x1() 87
draw_line_trans8x1() 88
draw_rect() 82
draw_rect_pat8x8() 88
draw_rect_trans8x8() 89
draw_solid_rect() 87
draw_span()
draw_span_list()

disp_draw_contextfuncs_t
 82
disp_draw_corefuncs_t
 86
drivers
dirty rectangle 98
samples 9

E

enable_hw_cursor() 97
enum-devices 13
enumeration, mode 10, 13

F

fallback functions
  fb_color_translate() 194, 197
  fb_core_blit1() 90, 193
  fb_core_blit2() 91, 193
  fb_ctx_blit() 83, 193
  fb_ctx_draw_rect() 82, 193
  fb_ctx_draw_span() 82, 193
  fb_ctx_draw_span_list() 82, 193
  fb_ctx_update_alpha() 84, 194
  fb_ctx_update_chroma() 84, 194
  fb_ctx_update_color() 83, 194
  fb_ctx_update_general() 83, 194
  fb_ctx_update_rop3() 83, 194
  fb_draw_bitmap_*() 92
  fb_draw_line_pat8x1_8() 88, 193
  fb_draw_line_trans8x1_8() 88, 193
  fb_draw_rect_pat8x8_8() 89, 193
  fb_draw_rect_trans8x8_8() 89, 193
  fb_draw_solid_rect_8() 87, 193
  fb_draw_span_8() 86, 193
  fb_draw_span_list_8() 87, 193
  fb_get_contextfuncs() 21, 194, 197
  fb_get_corefuncs() 21, 27, 194, 196
  fb_hw_idle() 86, 194
  fb_update_draw_surface() 92, 194
  fb_update_pattern() 93, 194
  fb_wait_idle() 86, 194
falling back on software 10, 20, 193

FFB 5, 9, 20, 193
  fb_color_translate() 194, 197
  fb_core_blit1() 90, 193
  fb_core_blit2() 91, 193
  fb_ctx_blit() 83, 193
  fb_ctx_draw_rect() 82, 193
  fb_ctx_draw_span() 82, 193
  fb_ctx_draw_span_list() 82, 193
  fb_ctx_update_alpha() 84, 194
  fb_ctx_update_chroma() 84, 194
  fb_ctx_update_color() 83, 194
  fb_ctx_update_general() 83, 194
  fb_ctx_update_rop3() 83, 194
  fb_draw_bitmap_8() 92
  fb_draw_line_pat8x1_8() 88, 193
**Index**

```plaintext
flat_draw_line_trans8x1*()  88, 193
flat_draw_rect_pat8x8*()  89, 193
flat_draw_rect_trans8x8*()  89, 193
flat_draw_solid_rect*()  87, 193
flat_draw_span*()  86, 193
flat_draw_span_list*()  87, 193
flat_get_contextfuncs()  21, 194, 197
flat_get_corefuncs()  21, 27, 194, 197
flat_hw_idle()  86, 194
flat_update_draw_surface()  92, 194
flat_update_pattern()  93, 194
flat_wait_idle()  86, 194

fini()

disp_draw_miscfuncs_t
  17, 95
disp_memfuncs_t  17, 106
disp_modefuncs_t  17, 116
disp_vcapfuncs_t  153
disp_vidfuncs_t  164

Flat Frame Buffer  5, 9, 20, 193

flushrect()  97

font manager  5, 12
foreground color  75
    updating  83

formats
    colors  18, 27
    pixel layers  29
    pixels  27

frame buffer functions  16
free_surface()  109

functions
    adding to a table  67
    calling sequence  17
    context 2D drawing  13, 16, 19
        fallback routines  20, 193
        getting  46
    core 2D drawing  13, 16, 19
    fallback routines  20, 193
    getting  48
    fallback
        flat_color_translate()  194, 197
        flat_core_blit1()  90, 193
        flat_core_blit2()  91, 193
        flat_ctx_blit()  83, 193
        flat_ctx_draw_rect()  82, 193
        flat_ctx_draw_span()  82, 193
        flat_ctx_draw_span_list()  82, 193
        flat_ctx_update_alpha()  84, 194
        flat_ctx_update_chroma()  84, 194
        flat_ctx_update_color()  83, 194
        flat_ctx_update_general()  83, 194
        flat_ctx_update_rop3()  83, 194
        flat_draw_bitmap*()  92
        flat_draw_line_pat8x1*()  88, 193
        flat_draw_line_trans8x1*()  88, 193
        flat_draw_rect_pat8x8*()  89, 193
        flat_draw_rect_trans8x8*()  89, 193
        flat_draw_solid_rect*()  87, 193
        flat_draw_span*()  86, 193
        flat_draw_span_list*()  87, 193
```

November 2, 2006
Index

ffb_get_contextfuncs() 21, 194, 197
ffb_get_corefuncs() 21, 27, 194, 196
ffb_hw_idle() 86, 194
ffb_update_draw_surface() 92, 194
ffb_update_pattern() 93, 194
ffb_wait_idle() 86, 194
frame buffer 16
memory management 16
miscellaneous 2D drawing 16
getting 52
modeswitcher 16, 115
getting 54
video capture 16
getting 56
video memory
management 184
getting 50
video overlay 16
getting 58
capabilities information 100
context 2D drawing
functions 81
context information 20, 74
coordinates 19
core 2D drawing functions 85
debug stream 174
devg_get_contextfuncs() 16, 46
devg_get_corefuncs() 16, 48
devg_get_memfuncs() 16, 50
devg_get_miscfuncs() 16, 52
devg_get_modefuncs() 16, 54
devg_get_vcapfuncs() 16, 56
devg_get_vidfuncs() 16, 58
fallback routines 10, 195
memory management
functions 106
miscellaneous drawing
functions 94
modeswitcher functions 115
surfaces, allocating and freeing 66
video capture functions 153
video overlay functions 164
graphics hardware adapter 63

G

get_2d_caps() 60, 98
get_channel_caps() 165
disp_vcapfuncs_t 154
get_channel_status() 155
get_contextfuncs_sw() 99
get_corefuncs_sw() 98
get_modeinfo() 117
get_modelist() 117
graphics framework
binding your driver to 16

H

hardware
idle
determining if 86
waiting until 85
initializing 95, 116
shutting down 95, 116
hardware cursor
attributes, setting 96
  disabling 97
  enabling 97
  position, setting 97
hotspot 96
hw_idle() 86

I
  init()

  disp_draw_mscfuncs_t
    17, 95
  disp_memfuncs_t
    17, 106
  disp_modelfuns_t
    17, 116
  disp_vcapfuncs_t
    153
  disp_vidfuncs_t
    164
io-graphics 3, 11
ISA bus 64

L
  layer_query 100
  layer capabilities, query 121
  layer format, select 122
  layer surfaces
    allocating 108
    layer, begin update 133
    layer, disable 123
    layer, enable 123
    layer, end update 133
    layer, reset 134
    layer, set blending 127
    layer, set brightness 129
    layer, set chromakey 128
  layer, set flags 131
  layer, set saturation 130, 131
  layer, set surface 124
  layer_disable() 123
  layer_enable() 123
  layer_query() 121
  layer_reset() 134
  layer_select_format() 122
  layer_set_blending() 127
  layer_set_brightness() 129
  layer_set_chromakey() 128
  layer_set_contrast() 131
  layer_set_dest_viewport() 126
  layer_set_flags() 131
  layer_set_saturation() 130
  layer_set_source_viewport() 125
  layer_set_surface() 124
  layer_update_begin() 133
  layer_update_end() 133
  lines, drawing
    context 81, 82
    patterned 87, 88
    solid 86

M
  mem_avail() 110
  memory management 10, 14
    functions 16, 50, 106
  miscellaneous 2D drawing
    functions 16
  modes
    DPMS
      setting 119
      information 111
Index

getting 117
setting 118
video, enumerating 13
modeswitcher 10, 13
functions 16
module_info()

disp_draw_miscfuncs_t 95
disp_memfuncs_t 107
disp_modefuncs_t 116
disp_vcapfuncs_t 154
disp_vidfuncs_t 165
modules 10, 16
information 136
information about, getting 95, 116
monochrome format 28

N

next_frame() 166

O

offscreen memory manager 10, 14
overlay, video 10, 14
functions 16, 58

P

palettes
8-bit colors 28

R

raster operations
current 76
updating 83
rasterizer 5
rectangles, drawing
context 82
patterned 88, 89
solid 87
render library 5, 12
reset() 107
retrace, vertical, waiting for 119
ROP3 76
ROP3, updating 83

S

scaler 10
scroll position, setting 121
services, BIOS 187

216 Index November 2, 2006
Index

set_channel.props() 165

disp.vcapfuncs_t 155

set_display_offset() 120

set_dpms_mode() 119

set_hw_cursor() 96

set_hw_cursor_pos() 97

set_mode() 17, 118

set_palette()

disp_draw_miscfuncs_t 95

disp_modefuncs_t 120

set_scroll_position() 121

shared objects, loading 11, 12

software, falling back on 10, 20, 193

stride 77

bitmap 91

cursor image 96

surfaces 61, 139, 185, 186

surfaces 2D capabilities, getting 98

allocating 50, 66, 107, 186

blitting 82, 90

channel properties 155, 165

color formats 18, 27

current target 77

description of 138

flushing to 97

freeing 66, 109, 185, 186

idle

determining if 86

waiting until 85

images, drawing 91

memory, managing 14, 184

mode, setting 118

pattern rotation 23

properties 110

stride 61, 139, 185, 186

updating 92

T

TV tuner 10

U

unknown bus 64

update_alpha() 20, 84

update_chroma() 20, 84

update_color() 20, 83

update_draw_surface() 20, 92

update_general() 20, 83

update_pattern() 20, 92

update_rop3() 20, 83

V

vbios_context_t 192

vbios_call() 190

vbios_dowarmboot() 190

vbios_get_realptr() 192

vbios_int() 189

vbios_register() 188

vbios_unregister() 189

vertical retrace, waiting for 119

VESA local bus 64

VGA

cards, working with other 65

registers, accessing 173
Index

video BIOS services   187
video capture   10
    capabilities   143
    function table   153
    functions   16, 56
    properties   146
    status   152
video memory management
    functions   50, 184
video modes, enumerating   13
video overlay   10, 14
    function table   164
    functions   16, 58
video scalers
    capabilities, general   159
    channels
        properties, configurable   161
viewport, set destination   126
viewport, set source   125

W

wait_idle()   85
wait_vsync()   119
write combining or gathering   181