QNX

Manual

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

© QNX Software Systems Ltd. 1999
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without the prior written permission of QNX Software Systems Ltd.
Although every precaution has been taken in the preparation of this book, we assume no responsibility for any errors or omissions, nor do we assume liability for damages resulting from the use of the information contained in this book.

Publishing history

November 1999 Pre-Alpha edition

QNX is a registered trademark of QNX Software Systems Ltd.
All other trademarks and registered trademarks belong to their respective owners.
Cover art by ????.
Printed in Canada.
Part Number: ????

Contents

Chapter 1

Introduction to Neutrino Device Drivers

In this chapter. . .

Understanding drivers in Neutrino
Typographical conventions

Here are the types of drivers we'll be discussing in this book:

- Audio driver
- Block I/O driver
- Character I/O driver
- Flash filesystem driver
- Graphics driver
- Input driver
- Media players
- Network driver
- PCI
- USB drivers

Audio driver

An audio driver serves to decouple a particular implementation of a sound card from the generic APIs for sound support. The audio driver operates in its own independent process and conforms to the API outlined in the ALSA ("Advanced Linux Sound Architecture") specification.

Block I/O driver

The block I/O driver is responsible for providing a CAM-compatible interface to a block-oriented storage medium. The driver is implemented as a DLL that gets bound in with the filesystem components at runtime.

Character I/O driver

Character I/O drivers are responsible for providing standard, POSIX-API compatible interfaces to devices that operate on a character-by-character basis (examples include serial ports, parallel ports, and pseudo-tty's). Neutrino ships with a character I/O library that performs much of the common functions, such as interpreting editing characters, maintaining input and output buffers, and so on. The part of the driver that you supply deals almost exclusively with the hardware or device.

Flash filesystem driver

Flash filesystems are responsible for organizing raw flash memory devices into a filesystem.

Graphics driver

Graphics drivers are responsible for providing a set of proprietary APIs for the various GUI products we offer for Neutrino.

Input driver

An input driver is the piece of software that goes between an input device, (keyboards, mice, etc.) and a piece of higher-level software, like the Photon GUI.

Media players

Neutrino allows you to write your own media plugin modules that the standard **`phplay`** command can use, or you can write your own player as well. This chapter shows you how to do both.

Network driver

Network drivers actually fall into the following classes:

- hardware interface
- **`qnet`** protocol stacks
- custom protocol stacks
- @@@ others? @@@

A network driver that provides a hardware interface is responsible for presenting an abstract view of the networking hardware so that other QSSL-supplied components (such as the TCP/IP stack, for example) can function. @@@ how much is provided by libs? @@@

For native Neutrino networking, the **`qnet`** native networking manager relies on protocol stacks that may or may not be based on TCP/IP. For example, you may have several machines connected via a proprietary backplane in a VLAN configuration, and you may need to write a customized protocol driver for the VLAN.

If you're providing a custom protocol stack that uses an existing driver, then you'll need to know about the hooks provided in Neutrino's networking framework for this purpose.

PCI

@@@ no idea @@@

USB drivers

The Universal Serial Bus (USB) drivers are responsible for providing sub-devices on the USB. Neutrino ships with a base USB driver that talks to the USB hardware on the bus; you may wish to provide support through that driver to USB devices on the USB bus. @@@ what do we provide? @@@

Understanding drivers in Neutrino

For each type of driver, we'll examine in detail:

- the overall duties of the driver
- how it fits into a Neutrino system
- what parts of the driver *you* have to provide and what parts are "standard"
- the interfaces between:
  - the driver and its clients
  - the part that you write and the standard libraries
- hints on how to make your driver faster, smaller, better
- how to debug your driver
- common pitfalls
- a complete driver, in source form, analyzed step-by-step.

Duties of a driver

At the highest level, a driver is something that provides a service. Some drivers may be standalone processes, while others may be integrated into other processes (via a DLL).

The driver is responsible for handling the details of a particular piece of hardware, a protocol, a filesystem, or some other abstract service.

The goal of a driver is to provide a consistent interface to these services, so that client programs can simply use the service without having to be intimately involved in the details of the service itself.

For example, an audio driver is responsible for the details of the audio card and presents a simple interface to the audio subsystem that clients can use. A client program wants only to *open()* an abstract audio device and *write()* audio data — the client doesn't want to worry about manipulating the hardware of the audio device, handling interrupts, dealing with DMA transfers, etc.

As another example, consider a graphics driver. Although the interface between the driver and the GUI may be more complicated than that provided by the audio driver, the principal is the same — clients want to be able to draw lines, polygons, filled areas, etc., without explicit knowledge of the underlying hardware implementation.

How a driver fits into the system

A client program can access the services of a driver through an API. In some cases, the API is defined by POSIX (for example, if you're using a serial port, then you'd use standard calls like *open()*, *devctl()*, *read()*, *write()*, etc.). In other cases, the API is a de facto standard (such as Linux's ALSA — "Advanced Linux Sound Architecture"), and in still other cases the API is proprietary (as in the case of Photon). Regardless of its nature, the API is usually implemented via message passing at some level. In this manual we assume you have a good understanding of the concepts of Neutrino's message-passing services; if not, take a look at the References appendix for some additional reading material.

Resource managers

If the driver you're designing is accessed by a standard POSIX API, then you'll also want to be familiar with Neutrino's "resource managers." A resource manager is a server that accepts certain, well-defined messages and handles them in certain, well-defined ways. Neutrino ships with a library that aids in the creation of resource managers. Again, the References appendix will be helpful here.

If you're already familiar with drivers under other operating systems, then you'll want to pay particular attention to the following points, which highlight some of the unique characteristics of Neutrino drivers:

- a driver is *not* bound into the kernel
- a driver operates in the context of a process
- a driver can be started and stopped on the fly
- a driver communicates via message passing,
- @@@ others @@@

@@@ Let's talk about these characteristics in more detail, eh?

Typographical conventions

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

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

Single-step instructions are formatted like this:

➤ To reboot your computer, press Ctrl – Alt – Shift – Delete.

Notes, cautions, and warnings are used to highlight important messages:

☞ Notes point out something important or useful.

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

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

Chapter 2

Audio Drivers

In this chapter. . .

Audio Drivers

Audio Drivers

This chapter describes the audio driver in detail.

Chapter 3

Block I/O Drivers

In this chapter. . .

Block I/O drivers

Block I/O drivers

This chapter describes the block I/O drivers in detail.

Chapter 4

Character I/O Drivers

In this chapter. . .

Character I/O drivers

Character I/O drivers

This chapter describes the character I/O drivers in detail.

Chapter 5

Flash Filesystem Drivers

In this chapter. . .

Flash Filesystem Drivers

Flash Filesystem Drivers

This chapter describes the flash filesystem driver in detail.

Chapter 6

Graphics Drivers

In this chapter. . .

Graphics drivers
Writing your own driver
Utility Functions
PETE – Photon 1.XX drivers
PETE – New API features

Graphics drivers

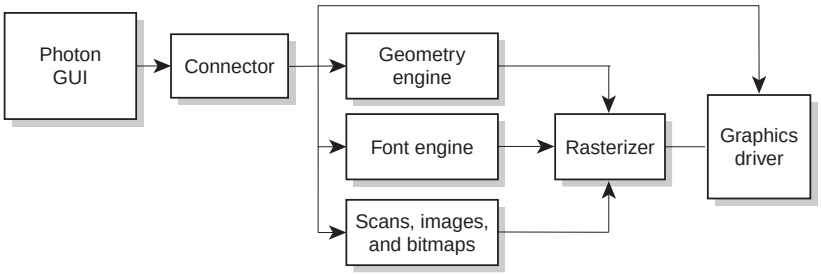
VERSION 0004141500

The graphics drivers are independent of Photon; the driver that you supply is implemented as one or more DLLs (your choice) that can be used by Photon or by any other graphical system.

You provide a set of well-defined entry points, and the appropriate graphics system will DLL-load your driver and call the entry points.

Graphics drivers in the Photon environment

By way of example, this is how your driver interacts with Photon:



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. The connector also contains the draw stream interpreter, which interprets Photon's *draw streams* and decodes the graphical commands. The interpreter converts the draw stream from whatever endian format it's in to native-endian format.

Geometry engine
: Converts complex shapes (like circles) into lower-level drawing primitives that the graphics driver can handle.

Font engine
: Converts textual information into bitmaps.

Scans, images, and bitmaps
: Deals with bitmap data.

Rasterizer
: Converts lower-level drawing primitives into a raster format.

H/W DLL
: Your graphics driver, supplied as one or more DLLs.

Note that your graphics driver may wish to take over some of the functionality from the various supplied components (e.g. your card can draw lines using hardware acceleration). Generally speaking, you'll only need to provide the functions that are unique to your card.

Writing your own driver

In this section, we'll look at how you write a driver for your own card.

We'll look at the following topics:

- the "Big Picture"
- binding your driver to the graphics framework
- conventions used in function calls

Although the primary purpose of this document is to provide a means of creating third-party Photon 2.0 drivers for Neutrino, the same DLLs will also work with the Photon 1.1x drivers.

For developers that wish to maintain a Neutrino version as well as a QNX 4 version, there's special logic in the **`Makefile`**s provided that will build a statically linked Photon 1.1x driver for QNX 4.

We provide a technote that's shipped with the toolkit that details how to use the **`Makefile`**s and various source files included so that you can build various versions of the graphics drivers. This technote is called **`README`** and is located in the root of the tree that you unpacked that contains the source.

The "Big Picture"

Before we look at the data structures and functions, it's important to understand the "big picture" for the Photon 2.0 Graphics Driver Development Kit (GDDK).

The purpose of the GDDK is to allow third parties to write accelerated drivers without requiring QSSL to become involved in doing the work.

Prerequisites

In this chapter, we assume that you have a basic familiarity with graphics cards, terminology, and concepts. We assume that you know what a pixel is, what a span is, blitting, alpha, chroma and raster operations (there are brief descriptions in the glossary appendix, however). We also assume that you have sufficient hardware documentation for your card in order to be able to program all the registers. A working knowledge of the C language is essential.

Examples

Two examples are provided with the GDDK:

1. a generic flat frame buffer driver, and
2. an accelerated driver based on the 3DFX Voodoo Banshee card.

We chose the Voodoo Banshee card as the basis for our example because the register level programming docs are available to anyone without needing an NDA.

The flat frame buffer example should be a good starting point for nearly any modern card. You should start with this driver, and implement accelerated versions for as many of the routines as possible.

The flat frame buffer driver mostly just calls routines in the FFB shared library. (The flat frame buffer driver is really quite small; it consists mainly of library callouts.) You should check the flags in the context argument to determine if your code can draw the specific type of object being asked for (e.g. does it ask for alpha blending?). If your code can perform the operation, then do it using the hardware, otherwise, fall back to the flat frame buffer routines as shown in the example source.

The modules

The Photon 2.0 GDDK is a set of DLLs that have been chosen because they expose groups of functionality in a modular fashion.

@@@ would this be a good place to mention the tier stuff, in an introductory manner? Then later we could tie it in with the functions…

The main feature of the Photon 2.0 driver architecture is the manner in which functional groups of accelerated routines are provided and accessed.

Future modules can be defined and accessed in the same way as we have defined the access methods for the current sets of functions.

The groups of routines currently defined are:

1. Mode switching and enumeration
2. 2-D Drawing
3. Offscreen memory manager
4. Video overlay control

Examples of routines that will be defined in future revisions of this GDDK are:

- 3-D drawing routines,
- TV tuner control routines,
- Video capture routines, and
- 2-D geometric primitives.

We've defined our GDDK using separate modules for each functional group to make it easier to package a complete driver solution. For example, most graphics cards are able to define a seperate "stride" for the source and destination surfaces when used in the various drawing routines, and for these cards, the offscreen memory management routines become routines that manage simple, linear chunks of memory. For these cards, the "standard" offscreen memory manager library routines can be used, and no card specific code needs to be written.

Another example of this is if the implementor wants to use the standard VESA routines for mode switching and enumeration. In that case, the standard VESA DLL could be used for that module.

On the other hand, if a vendor wishes to shrink the size of things down (for, say, a device with a fixed-size LCD screen), they could replace the generic VESA DLL that we supply with a very small, fast, customized direct mode switching module they wrote themselves.

Also of interest is the fact that we've engineered the function names such that you could provide all the modules in a single DLL, or in multiple DLLs, depending on your modularity and size tradeoff requirements.

The driver

With Photon 2.0, it's now possible (assuming the existence of the DLL routines described in this GDDK) to write a single "driver" that works for all cards. This driver is called **`io-graphics`** under Neutrino, and is responsible for:

- connecting to the appropriate Photon server,
- locating the correct set of DLLs to use for a particular user on a particular machine,
- and then loading those DLLs and using them to fulfill the instructions encoded into the Photon draw stream.

Although the current implementation of this driver is limited to driving one piece of hardware (i.e. one set of DLLs), it's our intention to make **`io-graphics`** eventually handle an arbitrary number of graphics devices, and also an arbitrary number of Photon servers, simultaneously.

The font engine 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. Good examples of these are circles and fonts.

Even if a graphics card *could* handle circles, it may draw them in a card-dependent way that would cause problems for users who expect consistent behaviour, so we need a way to handle them in a completely consistent way.

The **`io-graphics`** driver solves these problems by using the render library and the font manager to turn high-level entities into lower-level objects 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 which the driver draws as bitmaps or images, but we plan to eventually use it to return vector information that the driver could use directly.

The render library is used to "cook down" operations (other than fonts) which are defined in the Photon API, but which make little sense to implement in chipset-specific code. Circles are a good example, but also things like "thick dashed lines" are done by the render library.

The current implementation of the render library is designed to render its output directly into the frame buffer, 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.

Some of the planned changes mentioned above will be implemented using a "2-D geometry module" that will be added to this GDDK, but the main thing to remember is that you should only have to worry about implementing the routines described in this GDDK document.

Mode switching and enumeration

Mode switching and enumeration is the process of discovering what kind of video card you have, what its capabilities are, and putting the video card into one of its supported modes.

In the past, "trapping" for a particular graphics card was a potentially difficult and even dangerous operation. There was no *easy* way to determine if a particular card was present.

The overwhelming majority of video cards today are PCI or AGP devices, which makes the job of detecting video cards much, *much* easier.

In a "standard" Photon 2.0 environment, there's a list of PCI device IDs that are matched up with a text description of the set of DLLs required to drive that instance of a video card. This removes the need to call specific code in the DLL simply to find out if a given card is present and supported.

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 and returns information about that mode.

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

2-D drawing

2-D drawing routines are the functions that actually produce or manipulate an image.

Operations that fall into this category include:

- hardware cursor routines,
- filled rectangle routines,
- scanline operation routines, and
- BLT routines.

BLT routines include operations that draw 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.

There's no provision in the current GDDK to use Bresenham line hardware or polygon filling hardware. These operations will be addressed by a 2-D geometry DLL to be defined later.

Offscreen memory manager

Offscreen memory management routines are the code that allows the **`io-graphics`** driver to manage the process of using the accelerator to draw into various graphics objects, whether the objects are on the screen or not.

Offscreen memory is the most important new API feature in Photon 2.0, and is what allows applications to achieve much better performance than was possible in Photon 1.xx

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 such objects.

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 is a hardware feature that allows a rectangular area of the visible screen to be replaced by a scaled version of a different image. This process occurs without actually requiring the driver to

explicitly avoid drawing in the framebuffer “underneath” the overlaid region.
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.

Binding your driver to the graphics framework

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

Binding of the driver is performed by the graphics framework DLL-loading your driver, and then finding your entry point(s). The name of the entry point depends on which functional block(s) your DLL is providing; a single DLL can provide more than one functional block, hence the names are unique. The following table applies:

Functional block	Name of function
Core functions	*devg_get_corefuncs()*
Context functions	*devg_get_contextfuncs()*
Misc functions	*devg_get_miscfuncs()*
Modeswitcher	*devg_get_modefuncs()*
Memory manager / frame buffer	*devg_get_memfuncs()*
Video overlay	*devg_get_vidfuncs()*

☞ The three functions, *devg_get_miscfuncs()*, *devg_get_corefuncs()*, and *devg_get_contextfuncs() must* be supplied in the same DLL — all three of these functions constitute one "group."

Note that all functions in the table have a similar structure: they each get passed 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 parameters as appropriate).

The **`disp_adapter_t`** is the main "glue" that the graphics framework uses to hold everything together. We'll see this shortly.

The function pointers structure is what your function is expected to fill in 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 how many entries the function pointers structure holds. This is so that your initialization function doesn't overwrite the area provided. Note that there's a macro in **`display.h`** (called *DISP_ADD_FUNC()*) for stuffing function pointers into the table; it automagically checks the *tabsize* parameter.

Conventions

Before we look at the function descriptions, here are some conventions that you should be aware of.

Colour

@@@ **`pete`**: RGB/BGR whatever...

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 would be (0, 0) for the top left through to (1279, 1023), for the bottom right.

Coordinate ordering

Your driver will only be passed sorted coordinates. This means that if, for example, a "draw span" function gets called to draw a horizontal line from (*x1*, *y*) to (*x2*, *y*), we guarantee that *x1* ≤ *x2*; we will *never* pass *x1* **>** *x2*.

Coordinate inclusivity

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. Therefore, you'll want to be careful to avoid this common coding mistake:

```
...
    // WRONG!
    for (x = x1; x < x2; x++) {
...
```

and instead use:

```
...
    // CORRECT!
    for (x = x1; x <= x2; x++) {
...
```

Context

Every function is passed the **`disp_draw_context_t`** pointer as its first parameter. This gives the function access to the master context block.

If your functions modify any of the context blocks during their operation, they *must* restore them before they return. The graphics framework will modify the context blocks at will, and will then call the appropriate *update_*()* function to inform you which parts of the context data have been modified. Then, and only then, may one of your functions be called upon to do something with the hardware. We guarantee that we will *not* modify the context blocks *while* your function is running.

When a context function (i.e., a function that's in the **`disp_draw_contextfuncs_t`** group of callouts) is called, it's

expected to perform the following processing (whenever it can't do a particular operation or handle a particular mode, it should revert to the flat framebuffer version of the calls; this will perform the function in software, which will be slower):

1. Look at the *flag*; can we do this operation?
2. Do we recognize the pattern format?
3. Can we handle this particular *pixel_format*?
4. If a pattern is not involved, draw the plain version of the object.
5. If a pattern is involved, see if it's a transparent pattern or a fill pattern, and draw as appropriate.

@@@ **ddonohoe**, looks like the rectangle case doesn't handle all the cases (e.g., chroma, alpha, rop) — what else needs to be added here? Do we ship the source for the FFB *LIBRARY* for them to take a look at, or will the FFB *driver* be sufficient for a "see also" kind of thing?

As an optimization, your driver would perform these checks in its *update()* function (e.g. the **`disp_draw_contextfuncs_t`**'s *update_rop3()* function) and set a flag to itself (in its private context structure) that indicates whether or not it can do the appropriate function. This saves each and every context function from having to perform this work at runtime; it just checks the flag.

Patterns

Patterns are stored as a monochrome 8x8 array. Since many of the driver routines work with patterns, they're passed in 8-bit chunks (an **`unsigned char`**), with each bit representing one pixel. The most significant bit (MSB) represents the left-most pixel, through to the least significant bit (LSB) representing the right-most pixel. If a bit is on ("1") the pixel is considered "active," whereas if the bit is off ("0") the pixel is considered "inactive." The specific definitions of "active" and "inactive," however, depend on the context where the pattern is used.

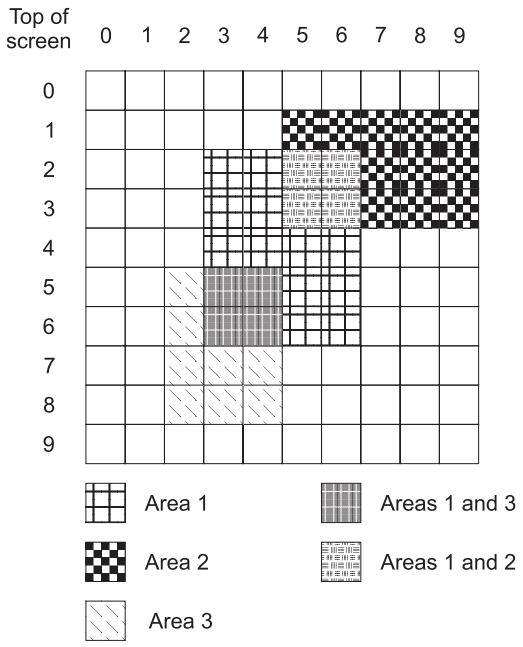
As an example, the binary pattern 11000001 (hex 0xC1) indicates three "active" pixels: the left-most, the second left-most, and the right-most.

Note that functions that have 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 a "circular" pattern, 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 was 11110000 and 15 bits of pattern were required, then the first eight bits would come from the pattern (i.e., 11110000) and then the next 7 bits would once again come from the beginning of the pattern (i.e., 1111000) for a total pattern of 111100001111000. 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, we start 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, we need to understand the relationships amongst 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.



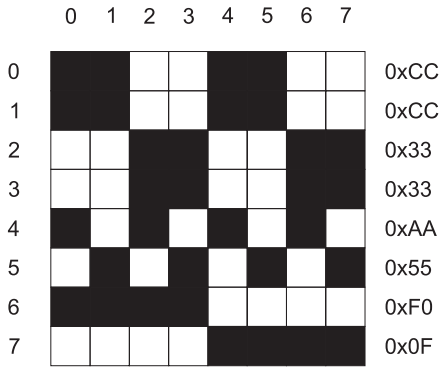
Three surfaces.

The diagram above shows three overlapping rectangles, representing three separate regions (for example, three **`pterm`**s); we'll focus our discussion on the middle one. If an application drew three rectangles within one Photon region, it would be up to the *application* to draw the three rectangles in the appropriate order — our discussion here about clipping only applies to separate regions managed by Photon.

If only the middle rectangle was present (i.e., there were no other rectangles obscuring it), your function to draw a rectangle with a pattern (e.g., *draw_rect_pat8x8()*), would be called once, with the following arguments:

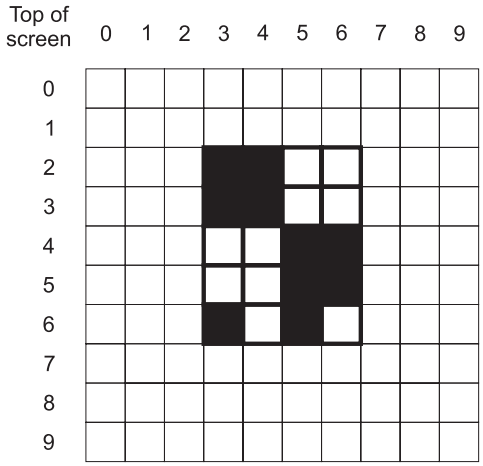
Note that the *x1*, *y1*, *x2* and *y2* parameters are passed to the function call itself, while the *pat_xoff* and *pat_yoff* parameters are part of a data structure that the function has access to. We'll just be listing the raw variables here instead of explicitly mentioning their locations.

The values for *x1*, *y1*, *x2* and *y2* are reasonably self-explanatory; draw a rectangle from (*x1, y1*) to (*x2, y2*). The *pat_xoff* and *pat_yoff* 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 our pattern looked like this:



Typical pattern.

Then the rectangle drawn would look like this:



Pattern filling a surface.

If we supplied a value of anything *other than* zero for the *pat_xoff* and *pat_yoff* parameters, (specifically, if we made those variables a function of the location of the rectangle) then the pattern would appear to "creep" along with the change of the location.

Let's now turn our attention to the case where the other two rectangles are partially obscuring our rectangle-of-interest.

When this needs to be drawn, the GUI may automatically transform 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 therefore implies that *draw_rect_pat8x8()* will be called three times:

x1	*y1*	*x2*	*y2*	*pat_xoff*	*pat_yoff*
3	2	4	3	0	0
3	4	6	4	0	2
5	5	6	6	2	2

Notice how the *pat_xoff* and *pat_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*):

```
x_index = (x + pat_xoff) % 8;
y_index = (y + pat_yoff) % 8;
```

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

devg_get_corefuncs()

This function is used by the graphics framework to get your driver's core functions:

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

The *pixel_format* parameter

The extra parameter, *pixel_format*, is defined below.

Note that you're *not* expected to be able to render into the formats tagged with an asterisk ("*") — these can only act as sources for operations, not as destinations.

@@@ **ddonohoe** can you recheck this list of non-renderable formats? We've added and removed a bunch, so I'm not convinced that this is up-to-date. Also, some of the names appear to be inconsistent, DISP_SURFACE_FORMAT versus DISP_PACKED and DISP_PLANAR, is this correct?

Therefore, these formats wouldn't be 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), you should call the flat-framebuffer helper functions (see below, under Graphics helper utilities) to perform the operation.

Also, the website **`www.webartz.com/fourcc`** contains an extensive list of FOURCC (for "Four Character Code") pixel formats, corresponding to the definitions used below, with excellent diagrams and explanations.

DISP_SURFACE_FORMAT_MONO (*)

pixel is 1 bit, and is monochrome.

DISP_SURFACE_FORMAT_PAL4 (*)

pixel is 4 bits, and is selected from a palette of 16 (4 bits) colours.

DISP_SURFACE_FORMAT_PAL8

pixel is 8 bits, and is selected from a palette of 256 (8 bits) colours.

DISP_SURFACE_FORMAT_ARGB1555

pixel is 16 bits, and the colour 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

pixel is 16 bits, and the colour components for red and blue are 5 bits each, while green is 6 bits.

DISP_SURFACE_FORMAT_RGB888

pixel is 24 bits, and the colour components for red, green, and blue are 8 bits each.

DISP_SURFACE_FORMAT_ARGB8888

pixel is 32 bits, and the colour 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_IYU1

12 bit format used in mode 2 of the IEEE 1394 Digital Camera 1.04 specification. The format is YUV (4:1:1) UYYVYY.

DISP_SURFACE_FORMAT_PACKEDYUV_IYU2

24 bit format used in mode 2 of the IEEE 1394 Digital Camera 1.04 specification. The format is YUV (4:4:4) UYVUYV.

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_PACKEDYUV_YVYU

Effectively 16 bits per pixel, organized as YVYU, two pixels packed per 32-bit quantity.

DISP_SURFACE_FORMAT_PACKEDYUV_V422

Same as YUY2, above.

DISP_SURFACE_FORMAT_PACKEDYUV_CLJR

Cirrus Logic's pixel format. Packs 4 pixel samples into a single 32-bit quantity by having the Y samples be 5 bits and the U and V samples be 6 bits each. Organization is YYYYUV.

DISP_SURFACE_FORMAT_YPLANE

@@@Organization?@@@

DISP_SURFACE_FORMAT_UPLANE

@@@Organization?@@@

DISP_SURFACE_FORMAT_VPLANE

@@@Organization?@@@

DISP_PACKED_YUV_FORMAT_IYU1

12 bits per pixel, layout is U2Y2Y2V2Y2Y2 (horizontal 1:4:4, vertical 1:1:1)

DISP_PACKED_YUV_FORMAT_IYU2

24 bits per pixel, layout is U4Y4V4U4Y4V4 (horizontal 1:1:1, vertical 1:1:1)

DISP_PACKED_YUV_FORMAT_UYVY

16 bits per pixel, layout is U8Y8V8Y8 (horizontal 1:2:2, vertical 1:1:1)

DISP_PACKED_YUV_FORMAT_YUY2

16 bits per pixel, layout is Y8U8Y8V8 (horizontal 1:2:2, vertical 1:1:1)

DISP_PACKED_YUV_FORMAT_YVYU

16 bits per pixel, layout is Y8V8Y8U8 (horizontal 1:2:2, vertical 1:1:1)

DISP_PACKED_YUV_FORMAT_V422

16 bits per pixel, layout is V8Y8U8Y8 (horizontal 1:2:2, vertical 1:1:1)

DISP_PACKED_YUV_FORMAT_CLJR

8 bits per pixel, layout is V6U6Y5Y5Y5Y5 (horizontal 1:3:3, vertical 1:1:1)

DISP_PLANAR_YUV_FORMAT_YVU9

9 bits per pixel, layout is YVU (horizontal 1:4:4, vertical 1:4:4)

DISP_PLANAR_YUV_FORMAT_YV12

12 bits per pixel, layout is YUV (horizontal 1:2:2, vertical 1:2:2)

DISP_PLANAR_YUV_FORMAT_I420

12 bits per pixel, layout is YVU (horizontal 1:2:2, vertical 1:2:2)

DISP_PLANAR_YUV_FORMAT_CLPL

Same as DISP_PLANAR_YUV_FORMAT_YV12 except that the U and V planes do not have to contiguously follow the Y plane. Also known as the “Cirrus Logic Planar format.”

DISP_PLANAR_YUV_FORMAT_VBPL

Same as DISP_PLANAR_YUV_FORMAT_YV12 except that the U and V planes do not have to contiguously follow the Y plane. Also known as the “VooDoo Banshee Planar format.”

DISP_SURFACE_FORMAT_BYTES

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

DISP_SURFACE_FORMAT_PAL

A flag that’s OR’d in to the surface type to indicate it’s a palette based format.

DISP_SURFACE_FORMAT_YUV

A flag that' OR'd in to the surface type to indicate a YUV colour format.

The pointer to function table is defined as follows (the parameters are listed in the function definitions section, below):

```
typedef struct disp_draw_corefuncs {
    void (*wait_idle) (...);

    void (*update_draw_surface) (...);
    void (*update_pattern) (...);

    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) (...);
} disp_draw_corefuncs_t;
```

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

void (*wait_idle) (disp_draw_context_t *context)

This function will wait for the hardware to become idle, and will then return. This implies that it's safe to directly access the frame buffer after this function returns.

void (*update_draw_surface) (disp_draw_context_t *context)

The surface has changed, examine the members pointed to by the *surface* structure pointer member of the *context*.

void (* *update_pattern* **) (disp_draw_context_t *** *context* **)**

The pattern has changed, examine the *context* members *pat*, *pat_xoff*, *pat_yoff*, and *pattern_format*.

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

Draw a plain, ordinary, opaque, horizontal line with the given colour from (*x1*, *y*) to (*x2*, *y*). Does *not* make use of any pattern information — the line is a single, solid colour.

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

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.

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

Draw a plain, ordinary, opaque rectangle with the given colour (in *color*), from (*x1*, *y1*) to (*x2*, *y2*). Does *not* make use of any pattern information — the rectangle is a single, solid colour.

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* **)**

Uses the passed *pattern* as described in the “Patterns” section of “Conventions,” above. An active bit is drawn with the *fgcolor* colour, and an inactive bit is drawn with the *bgcolor* colour.

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

Uses the passed *pattern* as described in the “Patterns” section of “Conventions,” above. An active bit is drawn with the *color* colour, and an inactive bit does not affect existing pixels.

void (*draw_rect_pat8x8**) (disp_draw_context_t ***context**, disp_color_t** fgcolor**, disp_color_t** bgcolor**, int** x1**, int** y1**, int** x2**, int** y2**)**

Uses the context structure’s members *pat*, *pat_xoff*, *pat_yoff*, (but not *pattern_format* as it’s already defined implicity by virtue of this function being called). The pattern is used as described in the “Patterns” section of “Conventions,” above. An active bit is drawn with the *fgcolor* colour, and an inactive bit is drawn with the *bgcolor* colour. See the section “Patterns,” above, for more information about patterns.

void (*draw_rect_trans8x8**) (disp_draw_context_t ***context**, disp_color_t** color**, int** x1**, int** y1**, int** x2**, int** y2**)**

Uses the context structure’s members *pat*, *pat_xoff*, *pat_yoff*, (but not *pattern_format* as it’s already defined implicity by virtue of this function being called). The pattern is used as described in the “Patterns” section of “Conventions,” above. An active bit is drawn with the *color* colour, and an inactive bit does not affect existing pixels. See the section “Patterns,” above, for more information about patterns.

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

Blits within the surface defined by the context structure’s *surface* 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 transferred to the same-sized area defined by the coordinates (*dx*, *dy*).

```
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)
```

Blits from the source surface specified 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* are transferred to the same-sized area defined by the coordinates (*dx*, *dy*). Note that the *src* and *dst* surfaces can be the same or different, whereas in *blit1()* (above), the operation takes place on the *same* surface (as implied by the lack of a destination surface parameter). Therefore, the driver should check the surface flags to see where the *src* and *dst* images are (either in system memory or video memory) before performing the operation.

devg_get_contextfuncs() This function is used by the graphics framework to get your driver's context functions:

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

The pointer to function table is defined as follows (the parameters are listed in the function definitions section, below):

```
typedef struct disp_draw_contextfuncs {
    void (*draw_span) (...);
    void (*draw_span_list) (...);
    void (*draw_rect) (...);

    void (*blit) (...);

    void (*update_general) (...);
    void (*update_fg_color) (...);
    void (*update_bg_color) (...);
    void (*update_rop3) (...);
    void (*update_chroma) (...);
    void (*update_alpha) (...);
} disp_draw_contextfuncs_t;
```

All functions in the context drawing structure must obey the members of the **disp_draw_context_t** structure (e.g., the current foreground colour); 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.

@@@ remind them of the *update_*()* funcs; here or in each applicable description?

void (*draw_span***) (disp_draw_context_t *** *context*, **int** *x1*, **int** *x2*, **int** *y*)

Called to draw a single, horizontal line from (*x1*, *y*) to (*x2*, *y*).

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

Called to draw *count* number of horizontal lines as given by the arrays *x1*, *x2*, and *y*.

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

Called to draw a rectangle from (*x1*, *y1*) to (*x2*, *y2*).

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*)

Called to perform a blit. The pixels from the source surface (*src*) specified by the rectangle beginning at (*sx*, *sy*) for the specified size (*length* and *height*) should be moved to the destination surface beginning with the rectangle at (*dx*, *dy*) for the same size.

void (update_general*) (disp_draw_context_t **context*)**

Re-read all members of the context; potentially, all of them could have changed.

void (update_fg_color*) (disp_draw_context_t **context*)**

Re-read only the foreground colour of the context. This is *fg_color*.

void (update_bg_color*) (disp_draw_context_t **context*)**

Re-read only the background colour of the context. This is *bg_color*.

void (update_rop3*) (disp_draw_context_t **context*)**

Re-read only the raster operation-related members of the context. Check *flags* to see if ROP3 functions are enabled or disabled. If enabled, look at *rop3*.

void (update_chroma*) (disp_draw_context_t **context*)**

Re-read only the chroma-related members of the context. Check *flags* to see if chroma functions are enabled or disabled. If enabled, look at *chroma_mode* and *chroma_color0*.

void (update_alpha*) (disp_draw_context_t **context*)**

Re-read only the alpha-related members of the context. Check *flags* to see if the alpha functions are enabled or disabled. If enabled, look at *alpha_mode*, *s_alpha*, *d_alpha*, *alpha_map_width*, *alpha_map_height*, *alpha_map_xoff*, *alpha_map_yoff*, and *alpha_map*.

devg_get_miscfuncs() This function is used by the graphics framework to get your driver's miscellaneous functions:

int

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

The pointer to function table is defined as follows (the parameters are listed in the function definitions section, below):

```
typedef struct disp_draw_miscfuncs {
    int  (*init) (...);
    void (*fini) (...);

    void (*set_palette) (...);

    int (*set_hw_cursor) (...);
    void (*enable_hw_cursor) (...);
    void (*disable_hw_cursor) (...);
    void (*set_hw_cursor_pos) (...);
} disp_draw_miscfuncs_t;
```


Note that if the driver does not support hardware cursors then it should set *all* of the hardware cursor entry points to NULL. If any one of the hardware cursor entry points is non-NULL then *all* hardware cursor entry points must be supplied.

`int (*`*init*`) (disp_adapter_t *`*adapter*`)`

Initialize the drawing hardware, allocate resources; whatever. Refer to the call chart below (in the description for the **`disp_adapter_t`**'s *init()* callout) for more information on where this initialization function "fits" into the general flow.

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

Un-initialize yourself by freeing resources, etc. See the call chart below (in the description for the **`disp_adapter_t`**'s *init()* callout) for more information on where this uninitialization function "fits" into the general flow.

void (* *set_palette* **) (disp_draw_context_t *** *ctx* **, int** *index* **, int** *count* **, disp_color_t *** *pal* **)**

This function is called to set the palette. Note, however, that if the modeswitcher version of this function (**disp_modefuncs ->** *set_palette*) is present, it will be called instead (i.e., the modeswitcher function overrides this function).

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* **)**

Set the attributes of the hardware cursor. Note that the term “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, etc.).

If the cursor cannot be displayed properly, this function should return a -1, which will cause the framework to show a software cursor instead. For example, if *sizex* or *sizey* is too big, this function should return -1.

The cursor image itself is defined by two bitmaps. The two colours, *color0* and *color1* apply respectively 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 colour given by *color0*. A 1 in *bmp1* means draw the corresponding pixel using the colour given by *color1*. If there’s a 1 in *both* bitmaps, then *color1* is to be used.

void (* *enable_hw_cursor* **) (disp_adapter_t *** *ctx* **)**

Make the cursor visible.

```
void (*disable_hw_cursor) (disp_adapter_t *ctx)
```

Make the cursor invisible.

```
void (*set_hw_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.

devg_get_modefuncs() This function is used by the graphics framework to get your driver's modeswitcher functions:

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

The pointer to function table is defined as follows (the parameters are listed in the function definitions section, below):

```
typedef struct disp_modefuncs {
    int  (*init) (...);
    void (*fini) (...);

    int  (*get_modeinfo) (...);
    void (*get_modelist) (...);
    int  (*set_mode) (...);

    int  (*disable_vga) (...);
    void (*reenable_vga) (...);

    void (*set_dpms_mode) (...);

    void (*set_display_offset) (...);

    void (*set_palette) (...);

    void (*get_current_crtc_settings (...);
} disp_modefuncs_t;
```

`int (*`*init*`) (disp_adapter_t *`*ctx*`)`

Initialize your hardware. The return value from this function is -1 to indicate an error, or a natural number to indicate the number of displays that this mode switcher controls. As an example, a display card could control both a flat-panel and a monitor simultaneously; in this case the return value would be 2.

The following call chart applies:

```
modeswitch -> init ();
    modeswitch -> setmode ();
    mem -> init ();
        misc -> init ();
            ...
            // graphics functions get called here
            ...
            // user requests a new mode; shut everything down
        misc -> fini ();
    mem -> fini ();
    // at this point no more graphics functions will be called

    modeswitch -> setmode ();
    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 ();
```

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

Return your hardware to the uninitialized state; deallocate resources, etc.

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

Populate the *info* structure with information pertaining to the mode specified in *mode* for the display number referenced by *dispno*. See the note about modes in *get_modelist()* below for more information.

void (*get_modelist***) (disp_adapter_t *****ctx***, int** *dispno*, **unsigned *****list***, int** *index*, **int** *size***)**

Returns a maximum of *size* modes into the array *list*, starting at the index *index*, for the display number 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 (therefore you should not return this as a valid mode!). The list of modes must be returned in the exact same order each time, but there is no requirement to "order" the list by any sorting criteria.

If you AND a mode number with the constant DISP_MODE_GENERIC you can tell whether the mode supports generic timings. This means that you must be careful about the mode numbers that you select, so that they correctly have the DISP_MODE_GENERIC bit set or unset as appropriate. See below in *set_mode()* for more information.

Note that 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 returned the following array:

```
list [0] = 0x1234;
list [1] = 0x070B;
list [2] = 0x8086;
list [3] = DISP_MODE_LISTEND; // terminate list
```

Then your *get_modeinfo()* and *set_mode()* functions would be called with, for example, 0x8086 and *not* the index 2.

int (*_set_mode**) (disp_adapter_t ***_ctx_**, int** _dispno_**, unsigned** _mode_**, disp_crtc_settings_t ***_settings_**, disp_surface_t ***_surf_**, unsigned** _flags_**)**

Set the hardware for the display referenced by *dispno* to the mode specified by *mode*. See the note about modes in *get_modelist()* above for more information.

The *settings* parameter is valid *only* if the mode number ANDed with DISP_MODE_GENERIC is non-zero, implying that you can pass an arbitrary X and Y resolution and refresh rate.

int (*_disable_vga_**) (disp_adapter_t ***_ctx_**)**

Disables the VGA registers, if possible. If not possible (i.e., the VGA card is on an ISA bus), the function returns -1 and sets *errno* to ENOSYS. If the VGA registers can be disabled, this function returns the previous state of the VGA registers (a 1 to indicate "enabled," or a 0 to indicate "disabled") so that the state can be restored by *reenable_vga()*, below.

void (*_reenable_vga_**) (disp_adapter_t ***_ctx_**)**

Enables the VGA registers. This function will not be called unless *disable_vga()* previously returned a 1 to indicate that the VGA had been enabled.

void (*_set_dpms_mode_**) (disp_adapter_t ***_ctx_**, int** _dispno_**, int** _mode_**)**

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

Mode	Meaning
0	On

continued...

Mode	Meaning
1	Standby
2	Suspend
4 (not a typo)	Off

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

Moves the video memory base for the display referenced by *dispno*. Note that the *offset* member must be a multiple of the *crtc_start_gran* member of the **disp_mode_info_t** structure.

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

Called to set the palette for the display referenced by *dispno*. 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 colour values, one per entry, to set.

Note that if this function is specifed (i.e., not NULL in the function pointer *set_palette*), then it will be called regardless of whether or not the *set_palette()* function in the miscellaneous callouts structure has been supplied:

```
if (disp_modefuncs -> set_palette) {
    (*disp_modefuncs -> set_palette) (…);
} else if (disp_draw_miscfuncs -> set_palette) {
    (*disp_draw_miscfuncs -> set_palette (…);
}
```

```
void (*get_current_crtc_settings) (disp_adapter_t *ctx, int
dispno, disp_crtc_settings_t *settings)
```

Fills the *settings* structure based on the current state of the display controller specified by *dispno*.

devg_get_vidfuncs() This function is used by the graphics framework to get the video overlay functions:

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

The pointer to function table is defined as follows (the parameters are listed in the function definitions section, below):

```
typedef struct disp_vidfuncs {
    int  (*init) (...);
    void (*fini) (...);
    void (*module_info) (...);
    int  (*get_channel_caps) (...);
    int  (*set_channel_props) (...);
    int  (*next_frame) (...);
    int  (*close_channel) (...);
} disp_vidfuncs_t;
```

@@@ Need a general, high-level overview/description kind of thing to put this into context...

General capabilities of a video scaler, for a given format:

```
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;
```

size Size of this struct.

reserved0, reserved

Reserved, do not examine or modify.

flags Flags beginning with the string DISP_VID_CAP, see below.

format The pixel format, see "The *pixel_format* parameter," above.

src_max_x, src_max_y

Maximum width and height of source frames.

max_mag_factor_x, max_mag_factor_y

Magnification — a -1 means cannot scale upwards.

max_shrink_factor_x, max_shrink_factor_y

1 means cannot scale downwards.

The following *flags* member bits are defined:

DISP_VID_CAP_SRC_CHROMA_KEY

Video viewport supports chroma-keying on frame data.

DISP_VID_CAP_DST_CHROMA_KEY

Video viewport supports chroma-keying on desktop data.

DISP_VID_CAP_BUSMASTER

Scaler device can bus-master the data from system ram.

DISP_VID_CAP_DOUBLE_BUFFER

Scaler channel can be double-buffered.

DISP_VID_CAP_DRIVER_CAN_COPY

The driver can perform the transfer of frame data.

DISP_VID_CAP_APP_CAN_COPY

The app can transfer the frame data.

DISP_VID_CAP_BRIGHTNESS_ADJUST

Brightness of video viewport can be adjusted.

DISP_VID_CAP_CONTRAST_ADJUST

Contrast of video viewport can be adjusted.

Configurable properties of a video scaler channel:

```
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;
```

And the fields are as follows:

size Size of this structure.

reserved0, reserved1, reserved

Reserved, do not examine or modify.

flags See below for details.

format Format of the frame data.

chroma_key0 Chroma-key colour.

chroma_flags Chroma-key comparison operation.

chroma_key_mask
Colours are masked with this before chroma comparison.

chroma_mode Type of chroma key match to perform, see below for details.

x1, y1 Top left corner of video viewport in display coords.

x2, y2 Bottom right corner of video viewport in display coords.

src_width, src_height
Dimensions of the video source data.

fmt_index Selects the format of the source frame data.

brightness Brightness adjust. 0x7fff = normal, 0 darkest, 0xffff brightest.

contrast Contrast adjust. 0x7fff = normal, 0 minimum, 0xffff maximum.

alpha Array of regions of the video viewport to be blended with desktop.

The *flags* member can be selected from the following:

DISP_VID_FLAG_ENABLE
Enable the video viewport.

DISP_VID_FLAG_CHROMA_ENABLE
Perform chroma-keying.

DISP_VID_FLAG_DOUBLE_BUFFER

Perform double-buffering.

DISP_VID_FLAG_DRIVER_DOES_COPY

Driver performs the copy of frame data in the *next_frame()* routine.

DISP_VID_FLAG_APP_DOES_COPY

Driver copies the frame data after calling the *next_frame()* routine.

The *chroma_mode* member can be selected from the following:

DISP_VID_CHROMA_FLAG_DST

Perform chroma test on desktop data.

DISP_VID_CHROMA_FLAG_SRC

Perform chroma test on video frame data.

And now, the entry points:

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

Returns the number of scalers available (functions similarly to the way the modeswitcher's *init()* function returns the number of display controllers available).

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

Frees resources, disables all scalers (makes them invisible). Must free any offscreen memory that was reserved for frame data.

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

Fills the **disp_module_info_t** structure (see below for contents).

int (*get_channel_caps***) (disp_adapter_t ****adapter***, int** *channel***, int** *fmt_index***, disp_vid_channel_caps_t ****caps***)**

Get the scaler capabilities for a given pixel format. We start with a *fmt_index* of 0, and keep calling with *fmt_index* being incremented, until the function returns -1. Thus, you can retrieve info on each format supported by the scaler denoted by *channel*. Channels are 0-based, i.e. if *init()* said there were 3 channels, then the valid channel numbers are 0 to 2, inclusive.

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***)**

Configure a scaler channel. Unless we set the DISP_VID_FLAG_APP_DOES_COPY flags, the **plane** parameters should be ignored. Otherwise, each plane parameter points to a surface descriptor. Only the *stride* and *paddr* members are defined. The *paddr* members are the physical address of the video frame data buffers. Unless we set the DISP_VID_FLAG_DOUBLE_BUFFER flag, the **plane2* parameters should be ignored. Unless the frame data format is planar (more than one surface needed) then the *uplane** and *vplane** parameters should be ignored by the driver.

int (*next_frame***)(disp_adapter_t ****adapter***, int** *channel***, disp_surface_t ****yplane***, disp_surface_t ****uplane***, disp_surface_t ****vplane***)**

If the driver is doing the copying, then *next_frame()* is called when a new frame is ready to copy. The **plane** pointers point to the surface data for the frame. If the data format is non-planar, then only the

yplane pointer is valid. (Note that the scaler may also support RGB or other non-YUV formats, in which case *yplane* points to the data). If the application is doing the copying, then the driver should ignore all plane pointers. In that case, *next_frame()* is called *before* the driver starts copying the next frame's data. This function will return the frame index (0 or 1) if double buffering, to specify whether the data should be copied to the **plane1* surface set, or the **plane2* surface set that was returned by *set_channel_props()*, above. In all other cases, this function should return 0.

For valid surface descriptors, only the *stride* and *vidptr* members are defined.

`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.

The `disp_module_info_t` structure

Here's the definition of the **`disp_module_info_t`** data type:

```
typedef struct disp_module_info {
    unsigned short  rev_major;
    unsigned short  rev_minor;
    char            *description;
} disp_module_info_t;
```

The *rev_major* and *rev_minor* members indicate the major and minor revision numbers, and the string *description* contains an ASCII description of the module. Here's an example:

```
disp_module_info_t info;

info.rev_major = 1;
info.rev_minor = 123;
info.description = "3dfx VooDoo Banshee / VooDoo3";
```

This is for a module called "3dfx VooDoo Banshee / VooDoo3" with a version of 1.123.

The big picture

Refer to the following include files for the details as we discuss the structures:

- **draw.h**
- **mode.h**
- **vmem.h**
- **display.h**

The master element is the **disp_adapter_t** structure, which is used as the "glue" that contains the individual pieces of the driver:

disp_adapter_t

The following is the **disp_adapter_t** structure:

@@@ **ddonohoe**, you left a comment about "I must add some pointers to function tables here", anything I should worry about? :-)

```
typedef struct disp_adapter {
    int             size;

    void            *gd_ctx;
    void            *ms_ctx;
    void            *mm_ctx;
    void            *vo_ctx;
    unsigned        reserved [8];

    int             irq;
    uintptr_t       rombase;
    uintptr_t       base [6];
    unsigned long   reserved1 [2];
    int             pci_handle;
    void            *pci_dev_handle;
    unsigned short  pci_vendor_id;
    unsigned short  pci_device_id;
    short           pci_index;

    unsigned        caps;
    FILE            *dbgfile;
    int             min_pixel_clock;
    int             max_pixel_clock;
    unsigned        intr_sources;
    char            *sysram_workspace;
    int             *sysram_workspace_size;
    unsigned        reserved2 [4];
} disp_adapter_t;
```

Each driver component has its own context block — these are identified in the structure as ending in *_ctx*. This area is for the use of the driver component; we don't define these areas. This can be particularly useful if, for example, you wish to supply multiple components, with the components calling functions within each other. Since the **`disp_adapter_t`** is available to each component, by placing function pointers within the context blocks this allows immediate access to the functions from different components.

The members of **`disp_adapter_t`** are defined as follows:

size	Size of this structure.
gd_ctx	Context block for graphics ("drawing") drivers.
ms_ctx	Context block for the modeswitch function group.
mm_ctx	Context block for the memory manager function group.
vo_ctx	Context block for the video overlay function group.
reserved, *reserved1*, and *reserved2*	Reserved, do not examine or modify.
irq	Interrupt vector used by graphics card (if card doesn't generate interrupts, contains the value -1).
rombase	Physical address of video ROM BIOS, if present, else NULL.
base	Array of up to six physical (PCI) base addresses.
pci_handle	Used internally by the display utilities; do not modify.
pci_dev_handle	Used internally by the display utilities; do not modify.

pci_vendor_id	Contains the PCI Vendor Identification number.
pci_device_id	Contains the PCI Device Identification number.
pci_index	Contains the PCI Index. Together, the three fields *pci_vendor_id*, *pci_device_id*, and *pci_index* uniquely identify a hardware device in the system.
caps	Capabilities; a bitmap of the following values: DISP_CAP_MULTI_MONITOR_SAFE (indicating card can work with other VGA cards in the same system), DISP_CAP_2D_ACCEL (indicating 2-D driver acceleration), and DISP_CAP_3D_ACCEL (indicating 3-D driver acceleration). The modeswitcher ORs in the multi-monitor safe flag, if appropriate, and the other components would OR in their own capability flags if supported.
dbgfile	A **`FILE *`** (or NULL) file pointer where debugging information gets written to. You'd use *disp_perror()* and *disp_printf()* within your driver, and those functions would take care of getting the debugging information into the file (if not NULL). Note that this will slow down your driver, so it's best if it's only used in extreme (or low-running) cases.
intr_sources	Indicates what can cause an interrupt, bit-wise OR of the following: DISP_INTR_SOURCE_VSYNC (vertical sync), DISP_INTR_SOURCE_2D_IDLE (2-D drawing engine idle), DISP_INTR_SOURCE_3D_IDLE (3-D drawing engine idle), and DISP_INTR_SOURCE_CAPTURED_FRAME (a video frame has been captured).
sysram_workspace	For use by the flat frame buffer or driver as a scratch area in system RAM.

sysram_workspace_size

Size of the scratch area in *sysram_workspace*.

disp_draw_context_t

The following is the **disp_draw_context_t** structure:

```
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  *surface
    unsigned        reserved [4];
} disp_draw_context_t;
```

The members are defined as follows:

size — Size of the structure.

adapter — Pointer back to the **disp_adapter_t** discussed above.

gd_ctx — Graphics driver's context.

cfuncs — A pointer to the core functions for rendering into the currently targetted draw surface. This surface

is of the type specified by the *surface* structure's *pixel_format* member.

flags — 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. These flags are used to indicate what kind of operations should be performed in all subsequent "context draw" functions.

fgcolor — The foreground colour that's to be used.

bgcolor — The background colour that's to be used.

image_palette — List of colours used to translate a palette index into a true colour.

image_palette_size — Number of entries in *image_palette* (above).

pat — Pattern buffer; see description in "Patterns" (above, in the "Conventions" section), as well as the context functions *draw_rect_pat8x8()*, and *draw_rect_trans8x8()*.

pat_xoff and *pat_yoff* — Used to specify an offset for the pattern to cause it to be shifted. See the section on "Patterns," above, for more information about patterns.

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 below, in "Chroma mode bits": 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. (i.e., SRC and DST are mutually exclusive, as are DRAW and NO_DRAW.)

chroma_color0 — Chroma key; indicates the colour to test on.

chroma_color1, *chroma_mask* — Reserved; do not examine or modify.

alpha_mode — Bitmask indicating alpha blending operations, see below, in "Alpha mode bits." For unrecognized alpha operations, call the supplied flat frame buffer functions.

s_alpha — Source alpha blending factor.

d_alpha — Destination alpha blending factor.

alpha_map_width, *alpha_map_height* — Width and height of the alpha map (below) in pixels.

alpha_map_xoff, *alpha_map_yoff* — X and Y offset of the alpha map (below). See the discussion above in "Patterns" for more information.

alpha_map — 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 NULL, means that

	there's no alpha map. The stride here is equal to the width, i.e., one byte per element.
surface	A pointer to a **`disp_surface_t`** structure that contains the definition of the currently targetted draw surface.
reserved	Reserved, do not examine or modify.

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

Chroma mode bits

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.

DISP_CHROMA_OP_NO_DRAW

If match, don't draw.

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

@@@ General bits (**ddonohoe** sez ask **drempel**):

DISP_ALPHA_OP_BLEND

This is an alpha blending operation

DISP_ALPHA_OP_DST_GLOBAL

Use the alpha in the *d_alpha* member (Ad=d_alpha)

DISP_ALPHA_OP_SRC_GLOBAL

Use the alpha in the *s_alpha* member (As=s_alpha)

DISP_ALPHA_OP_SRC_PACKED

src(x) is image source (stored as alpha component of the image data).

DISP_ALPHA_OP_SRC_MAP

src(x) is alpha map.

Alpha mode blending (source) bits

@@@ Alpha blending source factor (**ddonohoe** sez ask **drempel**):

DISP_BLEND_SRC_ZERO

(0,0,0,0)

DISP_BLEND_SRC_ONE

(1,1,1,1)

DISP_BLEND_SRC_DST_COLOR

(Ad,Rd,Gd,Bd)

DISP_BLEND_SRC_ONE_MINUS_DST

(1,1,1,1)-(Ad,Rd,Gd,Bd)

DISP_BLEND_SRC_SRC_ALPHA

(As,As,As,As)

DISP_BLEND_SRC_ONE_MINUS_SRC_ALPHA

(1,1,1,1)-(As,As,As,As)

DISP_BLEND_SRC_DST_ALPHA

(Ad,Ad,Ad,Ad)

DISP_BLEND_SRC_ONE_MINUS_DST_ALPHA

(1,1,1,1)-(Ad,Ad,Ad,Ad)

Alpha mode blending (destination) bits

@@@ Alpha blending destination factor (**ddonohoe** sez ask **`drempel`**):

DISP_BLEND_DST_ZERO

(0,0,0,0)

DISP_BLEND_DST_ONE

(1,1,1,1)

DISP_BLEND_DST_SRC_COLOR

(As,Rs,Gs,Bs)

DISP_BLEND_DST_ONE_MINUS_SRC

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

DISP_BLEND_DST_SRC_ALPHA

(As,As,As,As)

DISP_BLEND_DST_ONE_MINUS_SRC_ALPHA

(1,1,1,1)-(As,As,As,As)

DISP_BLEND_DST_DST_ALPHA

(Ad,Ad,Ad,Ad)

DISP_BLEND_DST_ONE_MINUS_DST_ALPHA

(1,1,1,1)-(Ad,Ad,Ad,Ad)

disp_surface_t The **disp_surface_t** structure is used as an argument to several functions, and is also used within other structures (such as **disp_draw_context_t**). Here is its definition:

```
typedef struct disp_surface {
    int             size;
    unsigned        pixel_format;
    unsigned        offset;
    unsigned char   *vidptr;
    unsigned        stride;
    unsigned        flags;
    int             height;
    int             width;
    disp_color_t    *pal_ptr;
    int             pal_valid_entries;
    unsigned        reserved [2];
} disp_surface_t;
```

The members are defined as follows:

size	Size of the structure.
pixel_format	Defined above.
offset	Device-dependent address.
vidptr	Virtual address.
stride	In bytes (see diagram below).
flags	Surface flags, defined below.
height	Height, in number of scan lines (see diagram below).
width	Width, in pixels (see diagram below).
pal_ptr	Pointer to the palette for this surface. If not a palette type, this pointer is NULL.

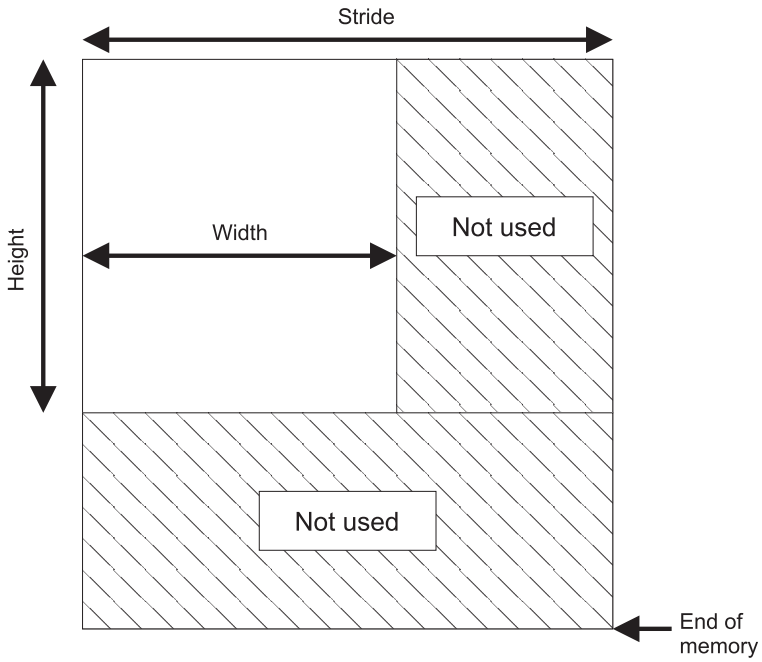
pal_valid_entries

Number of entries that are valid in the *pal_ptr* palette. This is used to limit the size of the palette table in case only a few colours are used.

reserved Reserved, must be zero.

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, the non-shaded portions represent the memory area that's

actually used for the surface. Note that it's important to *not* overwrite the "not used" areas.

flags

The *flags* member is a bitmap of the following values:

DISP_SURFACE_DISPLAYABLE

Surface can be displayed via CRT controller.

DISP_SURFACE_CPU_LINEAR_READABLE

CPU can read this surface directly.

DISP_SURFACE_CPU_LINEAR_WRITEABLE

CPU can write to this surface directly.

DISP_SURFACE_2D_TARGETABLE

2-D engine can render into surface.

DISP_SURFACE_2D_READABLE

Surface is read-accessible by 2-D engine.

DISP_SURFACE_3D_TARGETABLE

3-D engine can render into surface.

DISP_SURFACE_3D_READABLE

Surface is read-accessible by 3-D engine.

DISP_SURFACE_OPTIMIZED_CPU_ACCESS

Video memory is optimized for CPU access.

DISP_SURFACE_OPTIMIZED_ENGINE_ACCESS

Video memory is optimized for graphics engine access.

DISP_SURFACE_OPTIMIZED_UNBIASED

Video memory is equally accessible by the CPU and the graphics engine, or we don't know, or care.

DISP_SURFACE_SCALER_DISPLAYABLE

Surface can be displayed via video overlay scaler.

DISP_SURFACE_VMI_TARGETABLE

Video in hardware can write frames into surface.

DISP_SURFACE_DMA_SAFE

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

disp_mode_info_t The **disp_mode_info_t** structure is defined as follows:

```
typedef struct disp_mode_info {
   short     size;
   unsigned  mode;
   int       xres, yres;
   unsigned  pixel_format;
   unsigned  flags;
   uintptr_t fb_addr;
   unsigned  fb_stride;
   unsigned  fb_size;
   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;
          uint16_t  reserved0;
       } generic;
   } u;
   unsigned  reserved [6];
} disp_mode_info_t;
```

The members are defined as follows:

size Size of this structure.

mode	Mode number.
xres, yres	Display dimensions in pixels.
pixel_format	Frame buffer pixel format.
flags	See below.
fb_addr	Physical address of the frame buffer.
fb_stride	Stride of the frame buffer (in bytes).
fb_size	Size of the frame buffer (in bytes).
crtc_start_gran	Values passed in the *offset* parameter to the *devg_get_modefuncs()* function *set_display_offset()* must be a multiple of this value.
caps	List of available features, see below.
fixed.refresh	Array of possible refresh rates (in Hz) for this mode.
generic.min_vfreq, *generic.max_vfreq*, *generic.min_hfreq*, *generic.max_hfreq*	Monitor limits in Hz.
generic.min_pixel_clock, *generic.max_pixel_clock*	Pixel clock rates in kHz.
generic.h_granularity	Horizontal granularity; X resolution must be a multiple of this.
generic.v_granularity	Vertical granularity; Y resolution must be a multiple of this.
generic.reserved0, *reserved*	Reserved, do not examine or modify.

`disp_mode_info_t` *flags* member

The *flags* member is selected from the following:

DISP_MODE_TVOUT

Indicates that this mode drives a TV, and not a monitor.

DISP_MODE_TVOUT_WITH_MONITOR

Indicates that this mode can drive a TV and a monitor simultaneously.

DISP_MODE_TVOUT_OVERSCAN

Indicates that the overscan goes beyond the edge of the TV (i.e., there are no borders at the edges).

DISP_MODE_TVOUT_NTSC

Indicates that this mode generates NTSC format video signal.

DISP_MODE_TVOUT_PAL

Indicates that this mode generates PAL format video signal.

DISP_MODE_TVOUT_SECAM

Indicates that this mode generates SECAM format video signal.

Note that there's a macro, *DISP_TVOUT_STANDARD()* that's used to return just the type of output (PAL, NTSC, SECAM).

`disp_mode_info_t` *caps* member

And the *caps* member:

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 is not "hard-coded" meaning that it can perform double-buffering operations.

DISP_MCAP_DPMS_SUPPORTED

Display supports DPMS (if this bit set), else no support.

`disp_mode_info_t` *mode_num* member

And the *mode_num* member:

DISP_MODE_NUM_REFRESH

Returns the size of the *refresh* member (i.e., maximum number of refresh rates supported for a given mode).

`disp_crtc_settings_t`

The following is the definition for the **`disp_crtc_settings_t`** structure, which contains the CRT Controller (CRTC) settings:

```
typedef struct disp_crtc_settings {
    short       xres;
    short       yres;
    uint8_t     h_granularity;
    uint8_t     v_granularity;

    short       refresh;
    unsigned    pixel_clock;

    uint8_t     sync_polarity;

    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;

    unsigned    reserved [8];
} disp_crtc_settings_t;
```

With the members defined as follows (note that the *h_granularity*, *v_granularity*, *pixel_clock*, *sync_polarity*, *h_total*, *h_blank_start*, *h_blank_len*, *h_sync_start*, *h_sync_len*, *v_total*, *v_blank_start*, *v_blank_len*, *v_sync_start*, and *v_sync_len* members are used in conjunction with "generic" modes only (with the *refresh* member applicable to both generic and fixed modes); see the *get_modelist()* function in the section on *devg_get_modefuncs()*, above, for more information):

xres, *yres* — Horizontal and vertical resolution, respectively, in pixels.

h_granularity, *v_granularity* — Horizontal and vertical granularity; X and Y resolutions must be multiples of these (respectively).

refresh — Refresh rate (in Hz)

pixel_clock — Pixel clock rate (in kHz)

sync_polarity — See below.

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 pixels.

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.

reserved — Reserved, do not examine or modify.

The *sync_polarity* member

The values defined for *sync_polarity* consist of none, one, or both 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 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.

Utility Functions

The following sets of utility functions can be useful when writing graphics drivers:

- display driver utilities

- PCI configuration access utilities
- memory manager utilities
- video memory management utilities

These functions are provided in the **disputil** (display utilities) library.

Display driver utilities

The following functions are provided in the display driver utilities set:

- *disp_register_adapter()*
- *disp_unregister_adapter()*
- *disp_acquire_vga_resources()*
- *disp_release_vga_resources()*
- *disp_perror()*
- *disp_printf()*
- *disp_usecspin()*

int *disp_register_adapter* **(disp_adapter_t *** *adapter***)**

Registers with the display utilities libraries. This call performs things like the calibration of timers.

int *disp_unregister_adapter* **(disp_adapter_t *** *adapter***)**

Frees any resources allocated by the preceeding *disp_register_adapter()* function call, above.

int *disp_acquire_vga_resources* **(disp_adapter_t *** *adapter***)**

Acquires access to the VGA registers; you must call this *before* activating any of the VGA registers.

int *disp_release_vga_resources* **(disp_adapter_t *** *adapter***)**

Opposite of *disp_acquire_vga_resources()*, you would call this when you are done with the VGA registers. You must de-activate the card's response to VGA cycles before this function is called.

void *disp_perror* **(disp_adapter_t *** *adapter***, char *** *what***)**

Prints the string given by *what* along with the string interpretation of the global *errno* to the graphics framework's debug port (as given in the *adapter* member *dbgfile*). Functions similarly to the standard C library's *perror()* function.

void *disp_printf* **(disp_adapter_t *** *adapter***, const char *** *fmt***, ...)**

Prints the given string (starting with the *fmt* parameter and any additional parameters specified) to the graphics framework's debug port (as given in the *adapter* member *dbgfile*). Functions similarly to the standard C library's *printf()*/*fprintf()* functions.

void *disp_usecspin* **(unsigned** *usecs***)**

Busy waits for at least *usecs* µs. While polling is generally discouraged in a realtime operating system, sometimes the hardware demands that registers be accessed only after a certain (small) delay. Therefore, use this function only if *absolutely* necessary — since the graphics drivers usually run at a priority higher than "normal" user processes, this could have a direct, negative impact on the scheduling latency for normal user processes.

PCI configuration access utilities

The following functions are provided in the PCI configuration access utilities set:

- *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()*

int *disp_pci_init* **(disp_adapter_t *** *adapter*, **unsigned** *flags*)

Performs a *pci_attach_device()* using the *pci_vendor_id*, *pci_device_id*, and *pci_index* members of the *adapter* structure. For a description of the *flags* argument, see the *pci_attach_device()* manpage.

int *disp_pci_shutdown* **(disp_adapter_t *** *adapter*)

Effectively calls *pci_detach()* to release the resources from a previous *disp_pci_init()* function, above.

int *disp_pci_read_config* **(disp_adapter_t *** *adapter*, **unsigned** *offset*, **unsigned** *cnt*, **size_t** *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*. See the *pci_read_config()* function for details on the return values.

int *disp_pci_write_config* **(disp_adapter_t *** *adapter*, **unsigned** *offset*, **unsigned** *cnt*, **size_t** *size*, **void *** *bufptr*)

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. See the *pci_write_config()* function for details on the return values.

int *disp_pci_dev_find* **(unsigned** *devid*, **unsigned** *venid*, **unsigned** *index*, **unsigned *** *bus*, **unsigned *** *devfunc***)**

Similar to *pci_find_device()* — this function 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.

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()*.

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()*.

int *disp_pci_info* **(unsigned *** *lastbus*, **unsigned *** *version*, **unsigned *** *hardware***)**

Cover function for *pci_present()*.

Memory manager utilities

The following functions are provided in the memory manager utilities set:

- *disp_mmap_device_memory()*
- *disp_mmap_device_io()*

- *disp_munmap_device_memory()*
- *disp_phys_addr()*
- *disp_alloc_dmasafe()*
- *disp_free_dmasafe()*

void ***_disp_mmap_device_memory_ **(paddr_t *base*, **size_t** *len*, **int** *prot*, **int** *flags*)

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

Do not cache the memory (useful for register access, for example).

DISP_MAP_LAZY

Allows CPU to delay writes, and combine them into burst writes for performance. Ideal for mapping frame buffers (Intel calls it “write combining”). On CPUs that don’t support this feature, the flag is ignored.

The *flags* parameter is 0 or the constant DISP_MAP_BELOW16M (indicating that the memory must lie within the first 16 megabytes of physical address space).

unsigned long *disp_mmap_device_io* **(size_t** *len*, **paddr_t** *base*)

Creates either a virtual address space pointer (like *disp_mmap_device_memory()*, above, or returns its argument *base*. A virtual address space pointer is returned on non-x86 architectures (because these don't have a separate "I/O" space), whereas the argument *base* is returned unmodified on x86 architectures. Regardless of the architecture, the return value can be used with functions like *in8()*, *out8()*, etc.

void *disp_munmap_device_memory* **(void *** *addr*, **size_t** *len*)

Invalidates ("unmaps") the virtual address pointer in *addr*.

paddr_t *disp_phys_addr* **(void *** *addr*)

Returns the physical address corresponding to the virtual address passed in *addr*. This call is useful with devices that use DMA (which must be programmed with the physical address of the transfer area). Note that the **paddr_t** physical address is only valid 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). For example, if __PAGESIZE was 4096 (0x1000), and the virtual address translated to a physical address of 0x7B000100, then only the physical address range 0x7B000100 through to 0x7B000FFF (inclusive) would be valid.

void * *disp_alloc_dmasafe* **(int** *size*, **unsigned** *prot*, **unsigned** *flags*)

Creates a virtual address pointer to an area somewhere in memory that conforms to the *size*, *prot* and *flags* parameters that is guaranteed to be safe to use with a DMA controller on the particular architecture. This implies that the data area is physically contiguous, and is addressable by the DMA controller. The *size*, *prot* and *flags* parameters are the same as those passed to *disp_mmap_device_memory()* above.

Note that you don't supply a *base* parameter as with the other mapping function; instead, this function finds a free block of memory (called "anonymous" memory) and allocates it.

void *disp_free_dmasafe* **(void *** *addr*, **int** *size*)

Invalidates the virtual address pointer in *addr* and deallocates the memory.

Video memory management utilities

The following functions are provided in the video memory management utilities set:

- *disp_vm_alloc_surface()*
- *disp_vm_free_surface()*
- *disp_vm_surface_info()*
- *disp_vm_mem_avail()*
- *disp_vm_walk_surface_list()*
- *disp_vm_create_pool()*
- *disp_vm_destroy_pool()*

Your driver must supply these functions; at a bare minimum your driver's versions of these functions should simply call the provided library entry points.

disp_sid_t *disp_vm_alloc_surface* **(disp_vm_pool_t *** *pool*, **int** *width*, **int** *height*, **unsigned** *format*, **unsigned** *flags*, **void *** *user_info*)

Allocates a surface from the pool of surfaces. If successful, the surface memory returned conforms to the *flags* and *format* parameters. If a surface can't be found that matches those requirements, NULL is returned. Note that the surface memory is identified by a handle (the return parameter, of type **disp_sid_t**).

Use the *disp_vm_surface_info()* function (below) to get information about the surface. The *pool* parameter that you pass in is the return value from the *disp_vm_create_pool()* function (below) — this implies that you must first create the pool before using it.

The *flags* parameter is selected from the set of manifest constants defined in the *flags* argument for the **disp_surface_t** data type, above.

The *format* parameter is selected from the set of manifest constants beginning with DISP_SURFACE_FORMAT_* and is documented above, under the description for *devg_get_corefuncs()*.

int *disp_vm_free_surface* **(disp_adapter_t *** *adapter*, **disp_sid_t** *sid*)

Releases the surface memory identified by *sid* back to the surface memory manager's pool.

int *disp_vm_surface_info* **(disp_adapter_t *** *adapter*, **disp_sid_t** *id*, **disp_surface_t *** *surf*, **void **** *user_info*)

If *surf* is not NULL, returns information about the surface identified by *id* into the **disp_sid_t** pointed to by *surf*. If *user_info* is not NULL, this function puts whatever value was supplied (in the *user_info* parameter) when the surface was created (via *disp_vm_alloc_surface*</*()*) back into the *user_info*.

unsigned long *disp_vm_mem_avail* **(disp_vm_pool_t *** *pool*)

Returns how much memory is available in the pool identified by *pool*, in bytes.

int *disp_vm_walk_surface_list* **(disp_vm_pool_t *** *pool*, **int (*** *callback*) **(disp_adapter_t *, disp_sid_t))**

The function is used to iterate across the list of surfaces associated with *pool*. The function returns a -1 in case of an (internal) error, else 0 to indicate success.

The user-supplied callback function *callback()* will be invoked for each surface in *pool* with a **disp_adapter_t** pointer and a surface id. The return values from the user-supplied callback function are -1 to stop walking, and 0 otherwise (positive values are currently reserved).

disp_vm_pool_t * *disp_vm_create_pool* **(disp_adapter_t *** *adapter*, **disp_surface_t *** *surf*, **int** *bytealign*)

Used to create a new memory pool for the memory manager. You pass the *adapter* associated with this memory pool, a pointer to the surface in *surf*, and a byte alignment parameter, *bytealign*. The *bytealign* parameter indicates the alignment for the memory manager — all chunks of memory returned by the memory manager for this pool will be aligned to the number of bytes specified.

The return value is a **disp_vm_pool_t** pointer (effectively a "handle") which can be used with the other *disp_vm_*()* functions.

int *disp_vm_destroy_pool* **(disp_adapter_t *** *adapter*, **disp_vm_pool_t *** *pool*)

Reallocates all surfaces and releases the resources associated with tracking the pool allocation.

Graphics helper utilities

The following functions are provided in the graphics helper utilities set:

- Core functions:
  - *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:
  - *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_fg_color()*
  - *ffb_ctx_update_bg_color()*
  - *ffb_ctx_update_rop3()*
  - *ffb_ctx_update_chroma()*
  - *ffb_ctx_update_alpha()*
- Colour space conversion utility:
  - *ffb_color_translate()*
- Miscellaneous

  - *ffb_wait_idle()*
  - *ffb_set_draw_surface()*

- Draw function retrieval routines:
  - *ffb_get_miscfuncs()*
  - *ffb_get_corefuncs()*
  - *ffb_get_contextfuncs()*

The assumption with these functions is that you'll use them during the creation of 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'll most likely 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 (one of the three supplied functions *ffb_get_corefuncs()*, *ffb_get_contextfuncs()*, and *ffb_get_miscfuncs()*) to populate your function table array with the "defaults" from this library. Note, however, that *all* functions in the library are exposed; you do *not* have to bind to them by way of the *ffb_get_*()* functions; you can just simply link against them.

The next step in the development cycle would be 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, you'd 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'll be (from "somewhat" to "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 which 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.

☞ Note that 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()*

Which specific one gets bound to the *ffb_draw_span()* function pointer in the core functions array (**`disp_draw_corefuncs_t`** type) member *draw_span* depends on the *pixel_format* argument passed to *ffb_get_corefuncs()* (below).

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 colour 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 paramount in this case.

---

int *ffb_get_miscfuncs* **(disp_adapter_t *** *context*, **disp_draw_miscfuncs_t *** *funcs*, **int** *tabsize*)

This function is used to populate the passed *funcs* pointer to function pointer table with the miscellaneous functions from the flat frame buffer library.

int *ffb_get_corefuncs* **(disp_adapter_t *** *context*, **unsigned** *pixel_format*, **disp_draw_corefuncs_t *** *funcs*, **int** *tabsize*)

This function is used to populate the passed *funcs* pointer to function pointer table with the core functions from the flat frame buffer library.

int *ffb_get_contextfuncs* **(disp_adapter_t *** *context*, **disp_draw_contextfuncs_t *** *funcs*, **int** *tabsize*)

This function is used to populate the passed *funcs* pointer to function pointer table with the context functions from the flat frame buffer library.

disp_color_t *ffb_color_translate* **(disp_draw_context_t *** *context*, **int** *srcformat*, **int** *dstformat*, **disp_color_t** *color*)

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) colour in the surface type specified by *dstformat*.

Note that it's not always possible to get an exact match — for example, if the source surface was a 24 bits-per-pixel type, (e.g. DISP_SURFACE_FORMAT_RGB888) and the destination had less colours (e.g. DISP_SURFACE_PAL8), then the colour returned would be a "closest match" to that available on the destination surface.

PETE – Photon 1.XX drivers

Pete's gonna describe how the 1.XX drivers relate to the "new-and-improved" 2.00 driver structure described herein.

PETE – New API features

Pete's gonna describe stuff here that's new; like offscreen memory usage (linear vs rectangular).

Chapter 7

Input Devices

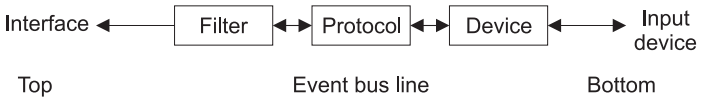
In this chapter. . .

Input drivers
Writing an input driver

Input drivers

This chapter provides an overview of writing input device drivers for Neutrino. Use this document along with the code in the sample directory.

The following is a brief overview of how the input driver framework functions. The input driver consists of two components, a group of input modules and a library used in manipulating these modules. At run time modules are linked together to form a data path used to gather data from an input device, process it, and output it to the system. There are three types of modules, device modules, protocol modules and filter modules. They are typically organized as follows:



Input chain.

When modules are linked together, they form an "event bus line." Data passes from an input device up the event bus line and out to the system. There are three different types of event bus lines:

- relative
- absolute
- keyboard

Types of event bus lines

The term "relative" simply means that the device provides position data that is relative to the last location it reported. This is typically the method that mouse-type pointing devices use.

An "absolute" bus line is used with devices that provide position data at absolute coordinates. An example of this is a touchscreen.

Finally, a “keyboard” type of bus line is one in which some sort of keypad device provides codes for every key press and release.

Modules

A device layer module is responsible for communicating with a hardware or software device. It typically has no knowledge of the format of the data from the device; it’s only responsible for getting data. A protocol layer module interprets the data it gets from a device module according to a specific protocol.

A filter module provides any further data manipulation common to a specific class of event bus.

Modules are linked together according to the command line parameters passed into the input driver. The command line has the following format:

devi-*driver_name* **[***options***] protocol [***protocol_options***] [device [***device_options***]]**

In this example:

```
devi-hirun ps2 kb -2 &
```

hirun	the hirunner input driver, which contains mouse and keyboard drivers used in most desktop systems.
ps2	specifies the PS/2 mouse protocol, a three byte protocol indicating mouse movement and button states.
kb	specifies the **kb** device module, which can communicate with a standard PC 8042-type keyboard controller.
-2	specifies an option to the **kb** module, telling it to set up communication to its second (or auxilliary) port, which is for a PS/2 mouse.

Specifying a filter module isn’t necessary because the three classes of event bus lines are represented by three modules, called **rel**, **abs**,

and **keyboard**. When the input driver parses the command line, it can tell from the **ps2** module that it needs to link in the **rel** filter-module. The only time you would specify a filter module on the command line is if you need to pass it optional command line parameters, for example:

```
devi-hirun ps2 kb -2 rel -G2
```

This tells the relative filter module to multiply *X* and *Y* coordinates by 2, effectively providing a gain factor (a faster-moving mouse).

Interface to the system

After data has passed from the input device up the event bus line to the filter module, it's passed to the system. There are currently two interfaces to the system:

Photon interface

> This requires that the Photon server is running. It passes data from the input to Photon via raw system events. Keyboard data is given by raw keyboard events, while relative and absolute data is given by raw pointer events. See the Photon docs to get more info on Photon events.

Resource manager interface

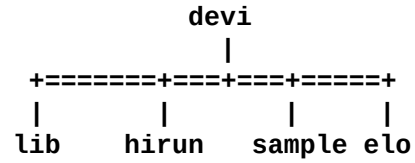
> This interface establishes a pathname under the **/dev** directory, which can be read by applications to get input data. For example, a relative event bus line would be represented by the file **/dev/mouse0**. Reading from **/dev/mouse0** would provide pointer packets, as defined in **<sys/dcmd_input.h>**. Multiple opens are allowed, and device files can be opened in blocking or non-blocking mode, with I/O notification (i.e. *select()*, *ionotify()*) supported.

The default interface started by the input system is the Photon interface. Unless you have a need to run input drivers without Photon, you'll never need to use the resource manager interface. The resource manager interface is started by passing the **-r** option to the **devi-***

driver. The Photon interface can be disabled by passing the **-P** option to the **devi-*** driver.

Source file organization for devi-*

The input (or **devi-*** source base is organized as follows:



The **lib** directory contains "glue" code used by all drivers. It contains the command line parsing code, the code used to manipulate modules and event bus lines, the code for the photon and resmgr interfaces, as well as the filter modules (**rel**, **abs**, and **keyboard**). In addition, the **lib** directory also contains functions used by modules to request services of the input system (e.g. for attaching interrupts and pulse handlers, mapping device I/O space, etc.)

It's recommended that you do not change anything in the **lib** directory. The source code is there simply to aid in understanding and debugging. The implementation of it could change internally at any time, although the interfaces used by the modules will not change.

The **hirun** directory is an example of an actual input driver, **devi-hirun**. In this directory, you'll find various device and protocol modules.

The **elo** directory contains source for the "ELO" touchscreen drivers.

When writing your own input driver, you would create your own directory and put your new input modules there.

Writing an input driver

To write an input driver, you must create your own input module. The **`sample`** directory contains a sample skeleton for creating a module. We recommend that you use this as a starting point.

A module is represented by a data type called **`input_module_t`**. It contains various data fields and function pointers representing its interface. Writing an input module consists of simply creating an **`input_module_t`** representing your module and filling in the relevant interface functions.

The code in the sample directory provides tons of comments detailing the steps required to initialize your module, and what to put in your module's functions.

The module **`samp_dev`** is an example of a device module. The module **`samp_proto`** is the MS mouse protocol code with lots of comments. The **`README`** file in the sample directory also talks about writing a combination device/protocol module. This case is very common when writing input drivers for embedded systems.

In addition, the **`README`** file also provides further background info on how the system processes data from **`keyboard`** and **`absolute`** devices.

Chapter 8

Media Players

In this chapter. . .

Media Players
Using the supplied plugins — writing your own player
Writing your own media plugin

Media Players

This chapter describes the media player plugins in detail.

A *media player plugin* (or just “plugin”) is either a separate process, or a DLL, that is responsible for handling a particular type of medium. By “handling” we mean performing a series of functions so that a high-level (perhaps GUI-based) program can simply do functions like “play a DVD movie,” or “play an audio track” from the media.

In this chapter, we’ll look at both how you’d use the existing QNX-supplied plugins, as well as how you’d write your own.

@@@ more stuff here about the general case…

Using the supplied plugins — writing your own player

We provide a number of plugins that you can write your own players for:

- DVD player
- MPEG audio player
- MPEG video player
- Audio player (for non-MPEG audio, e.g. **`.wav`**)
- CD audio player

While these plugins are all different, at the highest level they share the following characteristics:

- the player loads them as a shared object (DLL)
- an initialization function is provided

- several command processing functions are provided
- common data structures are used

Writing your own media plugin

In this section, we'll see the steps that you need to take to write your own plugin module. By implication, you'll be able to use this information to use an existing plugin with your own "player" program by calling the functions defined in the plugin, just like **`phplay`** does.

Binding to the player

Your plugin (in the simplest case) is a DLL that exports one visible symbol:

```
#include <sys/Mv.h>

MvInitF_t MvInit;
```

The **`MvInitF_t`** is a pointer to a function that has the following prototype:

```
int
MvInit (MvPluginCtrl_t *pctrl);
```

When the player loads your DLL, it will search for the symbol *MvInit*, and will then call it with the *pctrl* variable. This *pctrl* variable is effectively the "handle" that gets used for all communications between your DLL and the player.

MvPluginCtrl_t

The **`MvPluginCtrl_t`** structure is defined as follows:

```
typedef struct MvPluginCtrl
{
    // set by the plugin
    MvPluginData_t       *pdata;
    MvPluginFunctions_t  *calls;
```

```
    MvPluginFlags_t         pflags;
    unsigned                nhotkeys;
    MvPluginHotkey_t        *hotkeys;

    // set by the player
    MvSetup_t               setup;
    MvViewerCallbackF_t     *cb;
    void                    *dll_handle;
    char                    *name;
    unsigned                version;
    unsigned                APIversion;
} MvPluginCtrl_t;
```

The top set of elements is filled in by the plugin when its *MvInit()* function gets called, whereas the bottom set of elements is provided by the player.

The fields are defined as follows:

pdata — A pointer to a **MvPluginData_t** data type, see below.

calls — A pointer to a **MvPluginFunctions_t** data type, see below.

pflags

nhotkeys

hotkeys

setup

cb — A pointer to a **MvViewerCallbackF_t** function, see below.

dll_handle

name

version

APIversion

MvPluginData_t This data structure is supplied by the plugin itself, and is used to keep track of whatever context and state information the plugin wishes to use. There are no restrictions or definitions of its content.

MvPluginFunctions_t

```
typedef struct MvPluginFunctions
{
void            (*terminate) (MvPluginCtrl_t *pdata);
MvMediaInfo_t   *(*get_item) (MvPluginCtrl_t *pdata, MvMediaInfoFlag_t
int             (*command) (MvCommandData_t *cmdData);
} MvPluginFunctions_t;
```

Each plugin must supply the three functions listed in the table.

terminate()	Called by the player to terminate your plugin. Your plugin should clean up after itself (free up any resources it may have allocated, quiesce the hardware, etc.) Note that if your plugin is a DLL that's loaded into the player, then your plugin must *not* call *exit()* — this will take down the entire process, which of course includes the player! It's up to the player to unload your plugin DLL from itself, this is not something that you need worry about.
get_item()	Used to fetch an item from the plugin, based on the *which* parameter and the *index*, see below.
command()	This is the primary command interface to your plugin. By calling this function, with the *cmdData* parameter, the player is requesting that your plugin perform some kind of function. The function codes and their corresponding parameters are documented below.

command() function commands

The following commands are defined for the *command()* function that your plugin *can* handle — not all commands apply to all plugins, of course (e.g., CMD_PLUGIN_SELECT_SUBTITLE has no meaning to a pure audio plugin).

CMD_PLUGIN_OPEN_URLS

This is usually the first command given to a plugin, and gives it the URL of the item to act upon.

CMD_PLUGIN_CLOSE

Tells the plugin that the player is finished with the URL that was opened via CMD_PLUGIN_OPEN_URLS. The plugin should discontinue rendering the item.

CMD_PLUGIN_START

Begins rendering the item (e.g., for an audio wave file, this command will begin playing the audio file; for an MPEG video, this command will begin displaying the video and playing the associated audio track).

CMD_PLUGIN_PAUSE

Pauses the current rendering operation.

CMD_PLUGIN_STOP

Stops the current operation, but does not close the item.

CMD_PLUGIN_SEEK_TO

Moves the current position within the item to a new location.

CMD_PLUGIN_SEEK_RELATIVE

Selects a different item (for example, on an audio CD, this would be used to select a different track for playing).

CMD_PLUGIN_SET_PARAMETER

References **`MvPlaybackParams_t`** to indicate a parameter that should be adjusted.

CMD_PLUGIN_SET_WINDOW

CMD_PLUGIN_SET_STOP_TIME

Not implemented.

CMD_PLUGIN_DISPLAY_GUI

Tells plugin to update its GUI.

CMD_PLUGIN_GET_STATUS

Query plugin as to current position within item (e.g., on an audio item, this tells us the current position within a track)

CMD_PLUGIN_HOTKEY

Not implemented.

CMD_PLUGIN_EJECT_DISK

Bring on the dancing bears of “duh!”

CMD_PLUGIN_LOAD_DISK

Opposite of CMD_PLUGIN_EJECT_DISK (i.e., send the dancing bears of “duh” away).

CMD_PLUGIN_SELECT

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_SELECT_UP

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_SELECT_DOWN

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_SELECT_RIGHT

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_SELECT_LEFT

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_SELECT_AUDIO

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_SELECT_SUBTITLE

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_MENU

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_ANGLE

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_DIRECT_AUDIO

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_SET_SPEED

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_BOOKMARK_SET

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_BOOKMARK_GOTO

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_BOOKMARK_VIEW

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_KARAOKE_MIX

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_KARAOKE_RECORD

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_PLAY_ALL_FRAMES

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_PLAY_REALTIME

DVD player, corresponds to buttons (JBoucher).

CMD_PLUGIN_RESERVED_0 through CMD_PLUGIN_RESERVED_9
: Reserved for future expansion.

CMD_PLUGIN_USER_0 through CMD_PLUGIN_USER_9
: Reserved for custom commands for plugins; these will vary wildly between plugins. We do not enforce any particular meaning for these, nor will we ever invoke them from the standard **`phplay`** player. The idea here is that if you are writing both the player and the plugin, you can agree on some useful extensions.

MvViewerCallbackF_t()

The function that's supplied by the player is stored in the **`MvPluginCtrl_t`** data structure's *cb* member. When the player calls you with a command to perform (via your *command()* function pointer from the **`MvPluginFunctions_t`** data structure that you supplied), you are expected to perform the command, and then call the callback function with the status.

Here's the prototype for the callback function:

```
typedef void
MvViewerCallbackF_t (MvPluginCtrl_t *ctrl,
                     MvEventFlags_t change,
                     MvPluginStatus_t const *status);
```

The members are as defined below:

ctrl This is the handle that was passed to the plugin during the initialization phase. Simply pass this back, as it's used by the player to track this particular request.

change A bitmap of flags (see below) indicating which parameters in the **`MvPluginStatus_t`** parameter (the *status* argument) are indeed valid.

status A structure that contains a number of members, as defined below, that your plugin will fill in with the required information. The plugin will then set the flags in

the *change* member to indicate which of the fields within the *status* structure are valid as a result of the call.

The `MvViewerCallbackF_t` *change* flag

The following table indicates the correspondance between the flags (of type **`enum MvEventFlags`**) passed in the *change* argument and the various fields of the *status* structure. Note that the flags are individual bits, and can be OR'd together in case multiple fields are valid.

Flag	Member
MVS_PLUGIN_STATE	*state*
MVS_FLAGS	*flags*
MVS_MEDIA	*media_info*
MVS_POSITION	*position*
@@@	*duration*
MVS_VPSIZE	*vpsize*
MVS_ERRORMSG	*errormsg*
MVS_VIDEO_WND_TITLE	*videoWndTitle*
MVS_DISPLAY_INFO	*displayInfo*
MVS_AUDIO_LIST	*audioList*
MVS_SUBTITLE_LIST	*subtitleList*
MVS_UPDATE_GUI	@@@

Chapter 9

Network Drivers

In this chapter. . .

Network Drivers
Writing your own driver
The details

Network Drivers

REVISION 00 06 12 09 30

Neutrino's network subsystem consists of a process, called **io-net**, that loads a number of DLLs. Each DLL provides one or more of the following types of service:

up producer	produces data for a higher level (e.g., an ethernet driver provides data from the network card to a TCP/IP stack)
down producer	produces data for a lower level (e.g., the TCP/IP stack produces data for an ethernet driver)
up filter	a filter that sits between an up-producer and the bottom end of a convertor (e.g., a protocol sniffer)
down filter	a filter that sits between a down-producer and the top end of a convertor (e.g., NAT)
convertor	converts data from one format to another (e.g., between IP and Ethernet)

Note that these terms are relative to **io-net** and do not encompass any non-**io-net** interactions. For example, a network card driver (while forming an integral part of the communications flow) is viewed only as an "up producer" as far as **io-net** is concerned — it doesn't *produce* anything that **io-net** interacts with in the "down-going" direction, even though it actually transmits the data originated by an upper module to the hardware.

This chapter will focus on the creation of device drivers (e.g. for a network card). We'll take a look at:

- the big picture; what all the pieces are
- lifecycle of a packet

The big picture

From the command line, when you start **io-net**, you tell it which protocols to load:

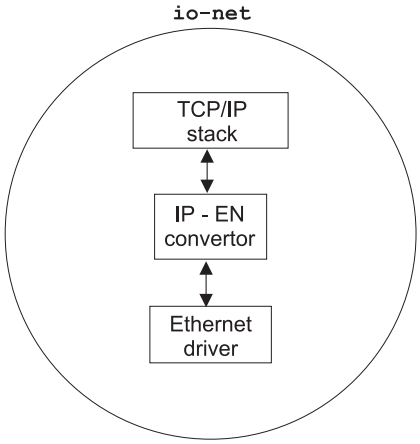
```
$ io-net -del900 verbose -pttcpip if=en0:11.2 &
```

This would cause **io-net** to load the **devn-el900** ethernet driver, and the tiny TCP/IP protocol stack. The "**verbose**" and "**if=en0:11.2**" options are "suboptions" passed to the individual components.

Alternatively, you can also use the **mount** and **umount** commands to start and stop modules dynamically. The previous example could be rewritten as:

```
$ io-net &
$ mount -Tio-net -overbose devn-el900.so
$ mount -Tio-net -oif=en0:11.2 @@@TCPIP@@@
```

Regardless of the way that you've started it, here's the "big picture" that results:



Big picture of io-net.

In the diagram above, we've shown **io-net** as the "largest" entity. This was done simply to indicate that **io-net** is responsible for

loading all the other modules (as DLLs), and that it's the one that "controls" the operation of the entire protocol stack.

The TCP/IP stack is at the top of the hierarchy, as it presents a user-accessible interface. A user would typically use the socket library function calls to access the exposed functionality. (The mechanism used by the TCP/IP stack to present its interface is not defined by **io-net** — it's a private interface that **io-net** has no knowledge of or control over.)

The TCP/IP stack, however, depends on an IP module that knows how to handle the IP protocol (in terms of converting IP to Ethernet and vice-versa, and routing information to/from an appropriate Ethernet module). The IP module sends packets down to the Ethernet driver (and receives packets from the Ethernet driver and gives them to the TCP/IP stack).

Finally, at the lowest level, we show an Ethernet driver that accepts Ethernet packets (generated by the IP module), and sends them out the hardware (and the reverse; it receives Ethernet packets from the hardware and gives them to the IP module).

As far as Neutrino's namespace is concerned, the following entries will exist:

/dev/io-net

The main device created by **io-net** itself.

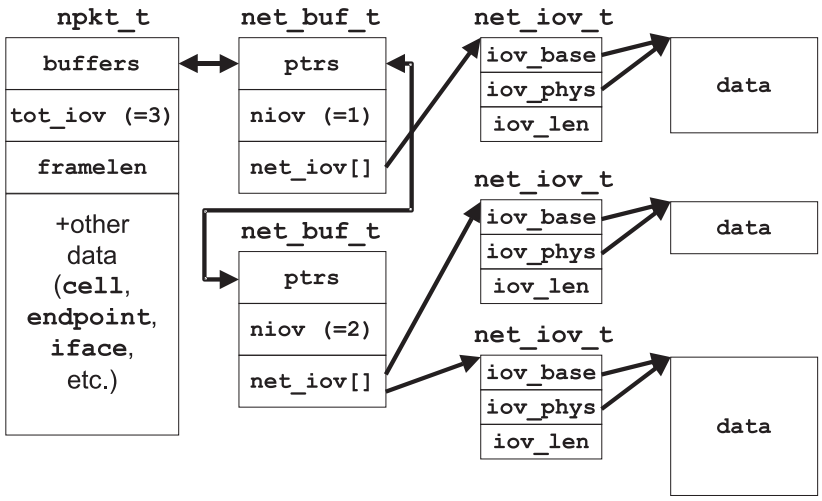
/dev/io-net/en*N*

The Ethernet device corresponding to LAN *N* (where *N* is 0 in our example).

At this point, you could *open()* **/dev/io-net/en0**, for example, and perform *devctl()* operations on it — this is how the **nicinfo** command gets the ethernet statistics from the driver.

The lifecycle of a packet

The next thing we need to look at is the lifecycle of a packet — how data gets from the hardware to the end user, and back to the hardware. The main data structure that holds all packet data is the **npkt_t** data type (see below). The buffers are managed via the *TAILQ()* macros from **<sys/queue.h>**, and form a doubly-linked list. Buffer data is stored in a **net_buf_t** data type (see below). This data type consists of a list of **net_iov_t**s (each containing a virtual address, physical address, and length) which are used to indicate one or more buffers:



net_iov_t relationship.

The *TAILQ()* macros allow you to iterate through the list of elements. The following code snippet illustrates:

```
// from <sys/queue.h>
#define TAILQ_FIRST(head) ((head) -> tqh_first)
#define TAILQ_NEXT(elm, field) ((elm) -> field.tqe_next)

net_buf_t *buf;
net_iov_t *iov;
int       i;

// walk all buffers
for (buf = TAILQ_FIRST (&npkt -> buffers); buf; buf = TAILQ_NEXT (buf, ptrs)) {
    for (i = 0, iov = buf -> net_iov; i < buf -> niov; i++, iov++) {
        // buffer is        :  iov -> iov_base
        // length is        :  iov -> iov_len
```

```
        // physical addr is :  iov -> iov_phys
    }
}
```

Going down

We'll start with the downward-going direction (from the end-user to the hardware). A message is sent from the end-user (via the socket library), and arrives at the TCP/IP stack. The TCP/IP stack does whatever error checking and formatting it needs to do on the data. When the TCP/IP stack is ready to send the data off to the IP module, it allocates a packet buffer. This packet buffer contains just the data from the TCP/IP stack — no provision is made for any of the other protocols' headers or encapsulation information (this is handled later, by each individual module).

Since the TCP/IP stack and the other modules aren't bound to each other, it's up to **`io-net`** to do the work of accepting the packet from the TCP/IP stack and giving it to the convertor module. The TCP/IP stack informs **`io-net`** that it has a packet that should be sent to "a lower level" by calling the *tx_down()* function within **`io-net`**. **`io-net`** looks at the various fields in the packet and the parameters passed to the function, and calls the *rx_down()* function in the IP module. Note that the *contents* of the packet aren't copied — since all these modules (e.g., the TCP/IP stack and the IP module) are DLL-loaded into **`io-net`**'s address space, all that needs to be transfered between modules are pointers to the data (and not the data itself).

Once the packet arrives in the IP module, a similar set of events occur as described above: the IP module converts the packet to an Ethernet packet, and sends it to the Ethernet module to be sent out to the hardware. Note that the IP module will need to add data in front of the packet in order to encapsulate the IP packet within an Ethernet packet. (The IP module may also do other "tricks" like fragmenting the packet.)

☞ **seanb** sez 000424 (*comments not integrated into above paragraphs yet -RK*):

This is not quite correct. The stack currently registers as a down producer of type "IP." Once it calls **`io-net`**'s *tx_down()*, it's already passing on a fully-formed IP packet (i.e., fragmentation has already occurred). What you're describing is something like:

```
 (TCP)    producer (down)  \
   |                         |
   v                         |
(TCP/IP)  convertor         |  These are currently
   |                         \ all done inside the
   v                         / stack.  It could be
 (IP)     producer (down)   |  done like this, but
   |                         |  currently isn't.
   v                         |
(IP EN)   convertor        /
   |
   v
 (EN)     producer (up)
```

Again, to avoid copying the packet data in order to insert the Ethernet encapsulation header in front of it, only the data pointers are moved. By inserting an element at the front of the IOV list (by moving the IOV entry list down by one entry and placing a new entry at the "hole"), the Ethernet header can be prepended to the data buffer without having to actually *copy* the data bytes themselves from the IP header — the only "data" that got moved was the address/length tuple(s).

Going up

In the upward-headed direction, a similar chain of events occurs. The ethernet driver receives data from its hardware, and allocates a packet into which it places the data. (For efficiency, it may use memory-mapping tricks to cause the hardware to directly place the packet into a pre-allocated area.) It then calls **`io-net`**'s *tx_up()* function, telling it that it has a packet that's ready to be given to a higher level. **`io-net`** figures out who to give it to, and calls their *rx_up()* function. In our example, this would be the IP-EN convertor module, as it now needs to look at the packet and get at just the IP

portion (the packet arrived from the hardware with Ethernet encapsulation).

Note that in an upward-headed packet, data is *never* added to the packet as it travels up to the various modules, so the list of **`net_buf_t`**s is not manipulated. For efficiency, there are two arguments to **`io-net`**'s *tx_up()* and correspondingly to a registered module's *rx_up()* function, namely *off* and *framelen_sub*. These are used to indicate how much of the data within the buffer is of interest to the level to which it's being delivered. For example, when an IP packet arrives over the Ethernet, there will be 14 bytes of Ethernet header at the beginning of the buffer. This Ethernet header is of no interest to the IP module — it's only relevant to the Ethernet module. Therefore, the *off* argument would be set to the value 14 to indicate to the next higher layer that it should ignore the first 14 bytes of the buffer. This saves the various levels in **`io-net`** from having to continually copy buffer data from one format to another. The *framelen_sub* operates in a similar manner, except that it refers to the tail end of the buffer — it specifies how many bytes should be ignored at the end of the buffer, and is used with protocols that place a tail-end encapsulation on the data.

The details

Now that we've seen the overall architecture, and how a packet travels through **`io-net`**, we'll look at the details of the various modules.

Producers

@@@ Describe producers; how much detail do we want here?

A producer can be an "up producer," a "down producer," or both. The "up" direction is from the hardware (the lowest level in the **`io-net`** hierarchy) towards the end-user, and "down" is the opposite direction.

When a module is an "up producer," this means that the module may pass packets on to modules above it. Whether the packet originated at up-producer$_A$, or up-producer$_A$ received it from up-producer$_B$ below it, from the next recipient's point of view the packet came from the up producer directly below it.

A producer may produce both types (up and down) of packets, as would be the case, for example, with the TCP/IP module.

Filters

@@@ Describe filters; how much detail do we want here?

Convertors

@@@ Describe convertors; how much detail do we want here?

On a "downward-headed packet," a convertor may add headers or trailers as part of its duties and may manipulate the list of **`net_buf_t`**s.

Writing your own driver

In this section, we'll look at the work that you must do to write a driver for your own hardware card. From **`io-net`**'s perspective, the card will be an "up producer" because it produces data that goes up into the **`io-net`** infrastructure. It's not a "down producer" because it does *not* produce any data that goes down in the **`io-net`** infrastructure — the down-going direction is strictly limited to the hardware and network interface of the card.

We'll look at the following topics:

- general overall structure
- binding your driver to **`io-net`**
- required functions

Our example will be a "null" driver that absorbs any data sent to it (it pretends it went out the hardware) and, once per second, generates incoming data (it pretends data arrived from the hardware).

Binding to io-net

The first thing that you must do in your driver is create a public symbol called *io_net_dll_entry* of type **io_net_dll_entry_t**. This is used by the **io-net** process when it loads your DLL:

```
// forward prototype
int
my_init (void *dll_hdl,
         dispatch_t *dpp,
         io_net_self_t *ion,
         char *options);

io_net_dll_entry_t io_net_dll_entry =
{
    2,
    my_init,
    NULL
};
```

Here we've simply defined it as containing a single function called *my_init()*. At a minimum, this function should:

1. store the passed handle (the *dll_hdl* argument) and function pointers array (the *ion* argument) for future reference
2. register with **io-net** to indicate what kind of driver this is
3. return EOK (or an error)

At this point, the first phase of initialization has been performed. In a "real" driver, the initialization function may perform additional functions:

1. parse additional command line arguments
2. detect and initialize devices
3. attach interrupts, map memory, and allocate any other required resources
4. create additional threads

When **io-net** calls *my_init()*, it'll pass 4 arguments. We'll ignore the **dispatch_t *** *dpp* and the **char *** *options*; we don't use them in

our trivial example here. The other two parameters we'll just stash into global variables for later use:

```
void          *null_dll_hdl;
io_net_self_t *null_ion;

int
my_init (void *dll_hdl, dispatch_t *dpp, io_net_self_t *ion, char *options)
{
    null_dll_hdl = dll_hdl;
    null_ion = ion;

    if (!null_register_device ()
    || (errno = pthread_create (NULL, NULL, null_rx_thread, NULL))) {
        return (-1);    // couldn't register, fail; errno says why
    }
    return (0);         // success
}
```

Notice how we've created a receiver thread (using *pthread_create()*). For our trivial example, this thread will simply sit in a do-forever loop, sleep for one second, and then pretend that data has arrived from somewhere, finally giving the data to **io-net** (we'll see the code for this shortly). In a real driver, the functionality would be similar; the thread would be waiting for some kind of indication from the hardware that data has arrived (perhaps via a hardware interrupt) and would then get the data from the hardware, process it, and give it to **io-net**.

Important! Since your driver is part of a DLL (and is not its own, seperate process), you'll have to be *very* careful about error checking, memory leaking, and such issues. For example, if you call *exit()* within your driver, you'll take down the *entire* **io-net** process! If your driver gets loaded and unloaded many times, and you have a memory leak, this will add up and eventually your system will run out of memory!

Telling `io-net` about our functions

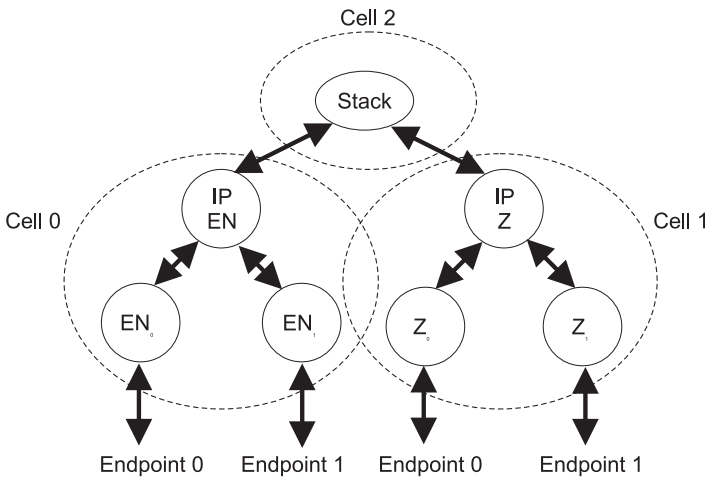
Now, to perform the second phase of our initialization, we need to tell **`io-net`** about our driver. Since we're going to be an "up-producer" and nothing else, this call is as follows:

```
// functions that we supply
io_net_registrant_funcs_t null_funcs =
{
    9,
    NULL,
    null_send_packets,
    null_receive_complete,
    null_shutdown1,
    null_shutdown2,
    null_advertise,
    null_devctl,
    null_flush,
    NULL
};

// a description of our driver
io_net_registrant_t null_entry =
{
    _REG_PRODUCER_UP,   // we're an "up" producer
    "devn-null",        // our name
    "en",               // our top type
    NULL,               // our bottom type (none)
    NULL,               // function handle (see note below)
    &null_funcs,        // pointer to our functions
    0                   // #dependencies
};

int         null_reg_hdl;
uint16_t    null_cell;
uint16_t    null_lan;

static int
null_register_device (void)
{
    if ((*null_ion -> reg)
        (null_dll_hdl,
         &null_entry,
         &null_reg_hdl,
         &null_cell,
         &null_lan) < 0) {

        return (0);     // failed
    }

    return (1);         // success
}
```

At this point, you've registered your device driver with **`io-net`**.

Note that for simplicity, we've used global variables (e.g., *null_cell*) — in a real driver, you'd most likely allocate a structure, and pass a pointer to that structure around. This helps your driver support multiple cards, as each card's context information (or "handle") can be passed individually. The **`io-net`** infrastructure allows you to associate your own handle with the binding (in the *func_hdl* member of **`io_net_registrant_t`**, identified with the comment "function handle," in the example above — we've passed a NULL).

Here's a "big picture" to illustrate:



Cells and endpoints.

As you can see, there are three levels in this hierarchy. At the topmost level, we have the TCP/IP stack — it provides an interface for programs to use. For our example, the stack will only be a down producer (it won't produce or pass on anything for modules above it.)

☞ In reality, the stack would probably register as both an up *and* down producer. This is permitted by **`io-net`** to facilitate stacking of protocols.

When the TCP/IP stack started it told **`io-net`** that it produces packets in the downward-going direction of type "IP" — there's no other binding between the stack and its drivers. We've labelled this top-level entity as "cell 2," which is the identifier used by **`io-net`**.

Joining the stack (down producer) to the drivers (up producers), we have two "convertor" modules. Take the convertor module labelled "IP_{EN}," as an example. When this module registered as type _REG_CONVERTOR, it told **`io-net`** that it takes packets of type "IP" on top and packets of type "EN" on the bottom.

Again, this is the only binding between the IP stack and its lower level "drivers." The IP_{EN} portion, along with its ethernet drivers, is called "cell 0" and the IP_Z portion, along with its Z-protocol drivers is called "cell 1" as far as **`io-net`** is concerned.

The purpose of the intermediate convertors is twofold:

1. It allows for increased flexibility when adding future protocols or drivers (one simply has to write a new convertor module to connect the two), and
2. it allows for filter modules to be inserted either above or below the convertor.

Finally, on the bottom-most level of the hierarchy, we have two different ethernet drivers and two different Z-protocol drivers. These are "up producers" from **`io-net`**'s perspective, because they only generate data in the upward-going direction. These drivers are responsible for the low-level hardware details. As with the other components mentioned above, these components advertise themselves to **`io-net`** indicating the name of the service that they're providing, and that's what's used by **`io-net`** to "hook" all the pieces together.

Since all seven pieces are independent DLLs, loaded by **io-net** when it starts up (or later, via the **mount** command), it's important to realize that the interface names are the key to the interconnection of all the pieces, and that the loading order isn't important — **io-net** figures all this out at runtime.

Advertising the driver's capabilities to io-net

Continuing with our discussion, the next thing to do is "advertise" the driver's capabilities. This is done via the *null_advertise()* function which you call whenever you detect a card. In our simple example, we'll assume that the **devn-null** device has always detected exactly one card, so we'll simply call the *null_advertise()* function ourselves, once.

Note that **io-net** will call in to your *null_advertise()* function some time later as well. This happens whenever some *other* driver is mounted above yours, so that it too can be informed of your driver's capabilities. This ties in with our discussion (above) about the dynamic nature of the loading of the modules.

Here's the code for our *null_advertise()* function (the numbers in the comments correspond to the notes just after the code sample):

```
// macros for function pointers
#define ion_alloc       null_ion -> alloc
#define ion_alloc_npkt  null_ion -> alloc_up_npkt
#define ion_add_done    null_ion -> reg_tx_done
#define ion_free        null_ion -> free
#define ion_rx_packets  null_ion -> tx_up
#define ion_tx_complete null_ion -> tx_done

#define MTUSIZE         1514

int
null_advertise (int reg_hdl, void *func_hdl)
{
    npkt_t                  *npkt;
    net_buf_t               *nb;
    net_iov_t               *iov;
    io_net_msg_dl_advert_t  *ap;

    // 1) allocate a packet; we'll use this for communications with io-net
    if ((npkt = ion_alloc_npkt (sizeof (*nb) + sizeof (*iov), (void **) &nb)) == NULL) {
        return (0);
    }

    // 2) allocate room for the advertisement message
    if ((ap = ion_alloc (sizeof (*ap), 0)) == NULL) {
        ion_free (npkt);
```

```
        return (0);
    }

    // 3) set up the packet into the queue
    TAILQ_INSERT_HEAD (&npkt -> buffers, nb, ptrs);

    iov = (net_iov_t *) (nb + 1);

    nb -> niov = 1;
    nb -> net_iov = iov;
    iov -> iov_base = ap;
    iov -> iov_len = sizeof (*ap);

    // 4) generate the info into the advertisement message
    memset (ap, 0x00, sizeof (*ap));
    ap -> type          = _IO_NET_MSG_DL_ADVERT;
    ap -> iflags        = (IFF_SIMPLEX | IFF_BROADCAST | IFF_MULTICAST | IFF_RUNNING);
    ap -> mtu_min       = 0;
    ap -> mtu_max       = MTUSIZE;
    ap -> mtu_preferred = MTUSIZE;
    sprintf (ap -> up_type, "en%d", null_lan);
    strcpy (ap -> dl.sdl_data, ap -> up_type);

    ap -> dl.sdl_len = sizeof (struct sockaddr_dl);
    ap -> dl.sdl_family = AF_LINK;
    ap -> dl.sdl_index  = null_lan;
    ap -> dl.sdl_type = IFT_ETHER;
    ap -> dl.sdl_nlen = strlen (ap -> dl.sdl_data); // not terminated
    ap -> dl.sdl_alen = 6;
    memcpy (ap -> dl.sdl_data + ap -> dl.sdl_nlen, "\x12\x34\x56\x78\x9a\xbc", 6);

    // 5) bind the advertisement message to the packet; note the use of
    //    the _NPKT_MSG flag to indicate to the upper modules that this
    //    is a message intended for them, rather than a "regular" packet
    npkt -> org_data = ap;
    npkt -> flags |= _NPKT_MSG;
    npkt -> iface = 0;
    npkt -> framelen = sizeof (*ap);

    if (ion_add_done (null_reg_hdl, npkt, NULL) == -1) {
        ion_free (ap);
        ion_free (npkt);
        return (0);
    }

    // 6) complete the transaction
    ion_rx_packets (null_reg_hdl, npkt, 0, 0, null_cell, null_lan, 0);
    ion_tx_complete (null_reg_hdl, npkt);
    return (0);
}
```

In the code sample above, the following steps are taken: @@@ MORE INFO, this is just an outline @@@

1 Allocate a packet for the communication. The function *ion_alloc_npkt()* is a macro expansion for the *alloc_up_npkt()*

function (defined below), which is responsible for allocating an upgoing packet. Here we've created the initial packet that we're going to send to the upper layer (**`io-net`** itself).

2 Allocate room for the advertisement message. The function *ion_alloc()* is a macro expansion for the *alloc()* function (defined below) and is used to create room for @@@.

3 Set up the packet into the queue. Here we bind the pointers to the buffers into the **`net_iov_t`** data type that we allocated above.

4 Generate the info into the advertisement message. We create the advertisement message ourselves here by filling the various members of the *ap* structure (of type **`io_net_msg_dl_advert_t`**). @@@ EXPL flags, like _IO_NET_MSG_DL_ADVERT, IFF_* family.

5 Bind the advertisement message to the packet. Finally, we perform pointer manipulations to attach the data (the advertisement message) into the packet.

6 Complete the transaction. To complete the transaction, we call *ion_rx_packets()* (a macro that expands to *tx_up()*, defined below) which sends a packet to the layer above you. Then, we call *ion_tx_complete()* (a macro that expands to *tx_done()*, defined below) which indicates that the packet has been consumed (i.e., we're now done with the packet). In our case, this will @@@?do what?@@@

At this point any modules which are attached to you from above know the characteristics of your driver.

The next two things to look at are how your driver receives data from the higher levels (destined for transmission via the hardware) and how it tells the higher levels that data has arrived (from the hardware).

Receiving data and giving it to a higher level

In our sample **devn-null** driver, recall that we created a thread to perform the "receive data from hardware" function:

```
...
    pthread_create (NULL, NULL, null_rx_thread, NULL);
...
```

Let's now look at this function:

```
void *
null_rx_thread (void *not_used)
{
    @@@     data;  // what type should this be?
    npkt_t  *pkt;

    while (1) {
        // 1) wait for hardware
        sleep (1);
        // 2) pretend data has arrived

        // 3) allocate upward-headed packet
        pkt = (*null_ion -> alloc_up_npkt) (sizeof (net_buf_t) + sizeof (net_iov_t) + MTUSIZE, &data);
        // set _NPKT_NO_RES flag to ensure your tx_done called before tx_done returns

        // 4) fill data at MTU location in up packet
        // 5) call reg_tx_done()
        // 6) call tx_up()
        // 7) call null_ion -> tx_done() (outcalls to your tx_done() function)
    }
}
```

@@@ Step-by-step work in progress. . .

1. Wait for hardware. In our simple driver, we simply called *sleep()* to wait for one second, to simulate some form of delay as might be encountered while waiting for data from a network. Depending on the complexity of your actual hardware, the *sleep()* call might just be replaced with something equally simple, like an *InterruptWait()*. This really depends on the hardware architecture, however.

2. Pretend data has arrived. In our simple driver, we assume that data is available at this point (i.e., we'll create some). Obviously, this will be one of the key, hardware-specific portions of your driver, as you'll have to get the data from the hardware.

3. Allocate upward-headed packet. At this step, we need to allocate a packet that we can place the data into. Note that we set the _NPKT_NO_RES flag, even though it's less efficient (but easier to understand). It means that before the *tx_done()* that's called in the *null_rx_thread()* returns, *your* outcall function *tx_done()* has been called, implying that the packet has been freed. This effectively *prevents* you from reserving the packet memory and being able to reuse it. It's easier to understand because the entire lifecycle of the packet is presented, rather than having to discuss data buffer management issues **:-)**, although we'll get to those later.

4. Fill data at MTU location in up packet. At this point, we stuff data into the up-going packet. Notice that we're just stuffing a constant message for our simple example here.

5. Call *reg_tx_done()* function. This binds a *tx_done()* handler to the packet. When the reference count goes to zero, the bound function will be called, and it's up to it to release the storage for the packet.

6. Call *tx_up()* function. This sends the packet up to the next higher layer.

7. Call *tx_done()* function. This does what?@@@ I'm guessing: indicates to **io-net** that the packet can be freed, and that it (**io-net**) can call the chain of *tx_done()*s that are bound to the packet to free it?

Transmitting data to the hardware

@@@ TODO need more

When a higher level sends data to a lower level for processing, one of the "tricky" things to watch out for is the fact that the data may be presented as a number of buffers (rather than just one single buffer as it is in the up-going direction). This is because of the way that the higher levels prepend and append encapsulation data onto the packet.

The details

Now that we've seen an outline of a sample driver, we'll take a look in detail at the definitions for the functions that we used.

Binding your driver to `io-net`

You must include the file **`<sys/io-net.h>`** which contains structures that you'll use to bind your driver to **`io-net`**.

Binding of the driver is performed by **`io-net`** DLL-loading your driver, and looking for a specific symbol: *io_net_dll_entry*. This symbol is of type **`io_net_dll_entry_t`**, and contains the following members:

```
typedef struct _io_net_dll_entry {
    int nfuncs;
    int (*init) (void *dll_hdl, dispatch_t *dpp, io_net_self_t *ion, char *options)
    int (*shutdown) (void *dll_hdl);
} io_net_dll_entry_t;
```

The members are defined as follows:

nfuncs — The number of functions in the **`io_net_dll_entry_t`** structure. In the structure above, this would be the constant 2 as there are two functions, *init()* and *shutdown()*.

init — A pointer to your initialization function. This will be the first function called by **`io-net`** in your driver. You should initialize your driver in this function. This function is mandatory.

shutdown — An (optional) pointer to your "master" shutdown function. This is called just before **`io-net`** finally closes your driver DLL. A particular DLL can register multiple times as multiple different things (e.g., as an up-producer and as a convertor). When a particular registration instance (a "registrant") is shut down, its *shutdown1()* and *shutdown2()* functions (from the

io_net_registrant_t structure's **io_net_registrant_funcs_t** function pointer array) are called. When *all* of the DLL's registrants are closed, then *this shutdown()* function is called. If you don't wish to supply this function, place a NULL in this member.

The *init()* function that you supply is then responsible for the following:

- processing of sub-options passed in the *options* argument.
- detection and configuration of all cards (one or more, can be "auto-detect" or can be based on the sub-options in the *options* argument).
- binding to **io-net**.

Arguments

The *init()* function that you supply gets passed the following arguments:

void * *dll_hdl*

An internal handle used by **io-net** — you'll need to hold onto this handle for future calls into the **io-net** framework.

dispatch_t * *dpp*

Dispatch handle.

io_net_self_t * *ion*

A big honkin' structure, see below.

char * *options*

Command line sub-options related to your driver.

The `io_net_self_t` structure

The **`io_net_self_t`** pointer points to a structure that contains **`io-net`**'s functions that are accessible to you. You should cache this pointer (passed to you in your *init()* function) so that you have access to those functions later.

The structure is defined as follows (the arguments are shown in the individual function descriptions below):

```
typedef struct _io_net_self {
    u_int   nfuncs;
    void    *(*alloc) (...);
    npkt_t  *(*alloc_down_npkt) (...);
    npkt_t  *(*alloc_up_npkt) (...);
    int     (*free) (...);
    paddr_t (*mphys) (...);
    int     (*reg) (...);
    int     (*dereg) (...);
    int     (*tx_up) (...);
    int     (*tx_down) (...);
    int     (*tx_done) (...);
    int     (*reg_tx_done) (...);
    int     (*reg_byte_pat) (...);
    int     (*dereg_byte_pat) (...);
    int     (*devctl) (...);
    int     (*tx_up_start) (...);
    int     (*memcpy_from_npkt) (...);
    int     (*raw_devctl) (...);
} io_net_self_t;
```

The *nfuncs* member indicates how many functions are provided in the table; it's filled automatically by **`io-net`**.

`void *(*`*alloc***`) (size_t`** *size*`, int`** *flags***`)`**

Allocates a buffer that's safe to pass to any other module.

`npkt_t *(*`*alloc_down_npkt***`) (int`** *registrant_hdl*`, size_t`** *size*`, void **`***data***`)`**

Allocates an **`npkt_t`** and initializes its internal members to values required for downward travel. The required number of *tx_done* array

elements (slots) immediately implicitly following the **npkt_t** is allocated in order to successfully reach any endpoint registrant this driver is currently connected to.

npkt_t *(*alloc_up_npkt***) (size_t** *size*, **void ****data***)*

Allocates an **npkt_t** and initializes its internal members to values required for upward travel, as follows:

num_complete	the value 1 to indicate that it has room for one *tx_done* (the originator's) array element immediately implicitly following the structure.
req_complete	the value 0 to indicate the single slot has not been used yet.
ref_cnt	the value 1 as the reference count (only in use by one module at this point).
flags	the bits _NPKT_UP and _NPKT_EXCLUSIVE are on, indicating it's an upward-bound packet and your module has exclusive access to it.
buffers	initialized to an empty **TAILQ** queue structure.

On successful completion, *data* points to a buffer of *size* bytes in length.

int (*free***) (void ***ptr***)**

Frees a buffer allocated by any of the above 3 methods (*alloc()*, *alloc_down_npkt()*, and *alloc_up_npkt()*).

paddr_t (*mphys***) (void ***ptr***)**

Quick lookup of physical address of buffer allocated by any of the above 3 methods (*alloc()*, *alloc_down_npkt()*, and *alloc_up_npkt()*).

int (*_reg_) (void *_dll_hdl_, io_net_registrant_t *_registrant_, int *_reg_hdlp_, uint16_t *_cell_, uint16_t *_endpoint_)

This call binds your driver to **io-net**. The *dll_hdl* is what you got called with in your *init()* function (from the **io_net_dll_entry_t** data type that you provided). The *registrant* parameter is a pointer to an **io_net_registrant_t**, which is defined below. The registrant describes what they are registering as. On success, *reg_hdlp* is filled, and should be used as the *registrant_hdl* parameter to subsequent calls into **io_net**, with the *cell* and *endpoint* indicating the registrant's place to other registrants (see *tx_up()*, below).

int (*_dereg_) (int _registrant_hdl_)

Deregister from **io-net**. Note that if a DLL has registered multiple times, its main DLL shutdown function (above) is not called until *after* all registrants have deregistered.

int (*_tx_up_) (int _registrant_hdl_, npkt_t *_npkt_, int _off_, int _framlen_sub_, uint16_t _cell_, uint16_t _endpoint_, uint16_t _iface_)

Send a packet to the layer above you. The parameter *off* indicates to the layer above at which offset into the packet the type your layer presents starts. The *framelen_sub* parameter indicates how many bytes on the end of the packet are not your type. These two parameters allow a packet to be "decapsulated" without the need to perform a copy operation. Finally, *cell*, *endpoint*, and *iface* indicate to the layers above who this packet came from. The *cell* and *endpoint* are supplied by **io-net** when you registered above. The *iface* is for internal use and allows a single registrant to present multiple interfaces of the same type to upper modules. It should start at 0 and increase sequentially. In the case of a driver talking to hardware (a simple up producer with no modules below it), it's actually more flexible to register multiple times if multiple interfaces are present (once for each interface). In this case, the *iface* parameter is always 0.

int (*tx_down**) (int** registrant_hdl**, npkt_t ***npkt**)**

Send a packet down to the layer below you. The destination that you're trying to reach is stored in the *cell*, *endpoint*, and *iface* members of *npkt*.

int (*tx_done**) (int** registrant_hdl**, npkt_t ***npkt**)**

For downward-headed packets, this function is called once by the module that consumes the packet. This causes the chain of *tx_done()*s stored in *npkt* to be called in LIFO order. For upward-headed packets, this function is called by each module (including the originator) when finished with the packet. The single *tx_done()* stored in the packet is called when the *ref_cnt* member goes to zero.

int (*reg_tx_done**) (int** registrant_hdl**, npkt_t ***npkt**, void ***done_hdl**)**

This function is used to store a *tx_done()* callback in the packet *npkt*. It's called once by the originator for upward-headed packets, and called by every module that adds to the *npkt* buffer chain for downward-headed packets. You must call this function rather than stuffing the value directly because **io-net** keeps track of how many *tx_done()*s a module has outstanding (used for unmounting the module).

int (*reg_byte_pat**) (int** registrant_hdl**, unsigned** off**, unsigned** len**, unsigned char ***pat**, unsigned** flags**)**

Before a module will receive any upward-headed packets, it must register with **io-net** to indicate what subtype it wants. This is in place to allow packet filtering, so that the module isn't getting packets that it will not be dealing with. The module already specified its bottom type when it registered (e.g. **"en"**, this would specify Ethernet subtypes 0x0800 and 0x0806 (for arp), or the IP protocol type PROT_QNET for **qnet**). If a module wants to get *all* subtypes, it would use the constant _BYTE_PAT_ALL in the *flags* parameter.

Important! Your module *must* register for *some* kind of byte pattern, otherwise it will *not* get any up-headed packets.

int (*dereg_byte_pat***) (int** *registrant_hdl*, **unsigned** *off*, **unsigned** *len*, **unsigned char *** *pat*, **unsigned** *flags***)**

Deregisters a byte pattern from **io-net**.

int (*devctl***) (int** *registrant_hdl*, **int** *dcmd*, **void *** *data*, **size_t** *size*, **int *** *ret***)**

Send a *devctl()* to **io-net**.

npkt_t *(*tx_up_start***) (int** *reg_hdl*, **nptk_t *** *npkt*, **int** *off*, **int** *framelen_sub*, **uint16_t** *cell*, **uint16_t** *endpoint*, **uint16_t** *iface*, **void *** *done_hdl***)**

A utility function for use by originators of up-headed packets. Unlike the rest of the functions provided by **io-net**, the *npkt* parameter can be a linked list of packets rather than a single entity. It efficiently combines **io-net**'s *reg_tx_done()*, *tx_up()* and *tx_done()* functions (3 common operations for originators of up-packets) as follows:

```
reg_tx_done (reg_hdl, npkt, done_hdl);
tx_up       (reg_hdl, npkt, off, framelen_sub, cell, endpoint, iface);
tx_done     (reg_hdl, nptk);
```

This processing is done for all *npkt*s in the linked list. This function returns a linked list of *npkt*s that had errors, or NULL if all succeeded.

int (*memcpy_from_npkt***) (const iov_t *** *dst*, **int** *dparts*, **int** *doff*, **const npkt_t *** *snpkt*, **int** *soff*, **int** *smax_len***)**

Utility function that's generally useful for copying data from packets. Similar to *memcpyv()*. The return value is the number of bytes copied.

```
int (*raw_devctl) (resmgr_context_t *ctp, io_devctl_t
*m, io_net_iofunc_attr_t *attr)
```

The idea here is to allow someone to write a class of _FILTER_ABOVE which sits above, say, all Ethernet registrants and is passed all *open()*s so they could then handle all *read()*s, *write()*s, etc., to that device. The only I/O message that **io-net** is concerned about is the message corresponding to the *devctl()* function call, so if they got a *devctl()* they didn't handle the default would be to call this function.

The io_net_registrant_t structure

Here's the definition for the **io_net_registrant_t** structure (a member of **io_net_self_t**, above):

```
typedef struct _io_net_registrant {
    uint32_t  flags;
    char      *name;
    char      *top_type;
    char      *bot_type;
    void      *func_hdl;
    io_net_registrant_funcs_t *funcs;
    int       ndependencies;
} io_net_registrant_t;
```

This structure is assumed to be followed by a variable number of **io_net_dependency_t** elements, as specified by the *ndependencies* member:

```
typedef struct _io_net_dependency {
    char      *dep;
    uint32_t  flags;
} io_net_dependency_t;
```

The members (for both structures) are defined as follows:

flags (from **io_net_registrant_t**)

Indicates the type of driver being registered, see below.

name A pointer to the ASCII name of the driver, for example, **"devn-speedo"**.

top_type A pointer to the ASCII name of the top type binding, for example, **"en"**.

bot_type A pointer to the ASCII name of the bottom type binding, for example, **"ip"**.

func_hdl A handle that you define, which will get passed to your functions when they get called. It's a convenient way of binding a data structure to this particular registration instance (because your module can register multiple times as different things).

funcs A pointer to a function table, see below.

ndependencies

The number of elements in the **io_net_dependency_t** table that's assumed to implicitly follow the **io_net_registrant_t** table.

dep @@@

flags (from **io_net_dependency_t**)

@@@

The *flags* member of **io_net_registrant_t** is a bit field, selected from the following:

_REG_FILTER_ABOVE

A filter that sits above an up producer and below the bottom end of a convertor.

_REG_FILTER_BELOW

A filter that sits below a down producer and above the top end of a convertor.

_REG_CONVERTOR

A convertor.

_REG_PRODUCER_UP

A producer in the “up” direction.

_REG_PRODUCER_DOWN

A producer in the “down” direction.

The *funcs* member of **`io_net_registrant_t`** is a pointer to a function table that you supply, as per the following (function parameters given below):

```
typedef struct _io_net_registrant_funcs {
    int nfuncs;
    int (*rx_up) (...);
    int (*rx_down) (...);
    int (*tx_done) (...);
    int (*shutdown1) (...);
    int (*shutdown2) (...);
    int (*dl_advert) (...);
    int (*devctl) (...);
    int (*flush) (...);
    int (*raw_open) (...);
} io_net_registrant_funcs_t;
```

The members are defined as follows:

`int` *nfuncs*`;` The number of function pointers in the structure. For the structure as given above, this should be the constant 9.

`int (*`*rx_up***`) (npkt_t *`***npkt***`, void *`***func_hdl***`, int `***off***`, int `***framlen_sub***`, uint16_t `***cell***`, uint16_t `***endpoint***`, uint16_t `***iface***`);**

This function is called when your module receives an up-headed packet from a module below you. The *cell*, *endpoint*, and *iface* parameters describe which module the packet is coming from.

`int (*`*rx_down***`) (npkt_t *`***npkt***`, void *`***func_hdl***`);**

This function is called when your module receives a down-headed packet from a module above you. The

cell, *endpoint*, and *iface* members of the *npkt* structure describe the destination of the packet, and the *buffers* member contains the packet data.

int (*tx_done***) (npkt_t *** *npkt*, **void *** *done_hdl*, **void *** *func_hdl*);

Called when a packet that you're responsible for has its reference count go to zero; effectively indicating that it has been consumed and may now be "recycled" (or disposed of).

int (*shutdown1***) (int** *registrant_hdl*, **void *** *func_hdl*);

This is the first "shutdown" function that gets called when your driver is asked to shutdown. You can prevent the driver from being shut down by returning an error indication (for example, EBUSY to indicate that there are active transmissions occuring; it would be up to the higher level to retry later) or an EOK to allow the shutdown to occur. The implication here is that one *cannot* force a shutdown of a driver that returns an error indication. If you proceed with the shutdown, it's your last chance to flush out buffers using the thread that called *shutdown1()*.

int (*shutdown2***) (int** *registrant_hdl*, **void *** *func_hdl*);

At this point, everything in **io-net** has detached from your driver, and you *must* shutdown. This is the call that you'd use to kill off any of your worker threads, for example. Generally speaking, *shutdown2()* does most of the "work" associated with shutting down the driver.

int (*dl_advert***) (int** *registrant_hdl*, **void *** *func_hdl*);

Called by **io-net** to cause your driver to advertise itself. Generally, this will be called whenever a new higher-level driver starts up, as it will need to be made aware of the capabilities of all drivers at

levels below it, so that it can determine what capabilities exist underneath it.

int (*devctl***) (void *****hdl*, **int** *dcmd*, **void *****data*, **size_t** *size*, **int *****ret*);**

This is the callin to your driver to perform a *devctl()* function. This would get invoked when someone does a *devctl()* on your driver's **/dev/io-net/***driver* pathname. Currently, only the "nic info" *devctl()* is defined, which is used to fetch statistics from a driver. You don't have to support a *devctl()* handler. @@@ to document nicinfo structure and devctl @@@

int (*flush***) (int** *registrant_hdl*, **void *****func_hdl*);

@@@

int (*raw_open***) (resmgr_context_t *****ctp*, **io_open_t *****msg*, **io_net_iofunc_attr_t *****attr*, **void *****extra*);

@@@

Command line processing

The *init()* function that you supply in your *io_net_dll_entry* structure gets passed the suboptions string from **io-net**. You can use the *getsubopt()* function to parse command line arguments passed to your driver.

Detection and configuration of cards

Once you have processed any (optional) command line parameters for your driver, you should then detect any and all cards that your driver is responsible for. (Depending on your hardware, the PCI calls may come in handy — see the PCI chapter as well as the library reference.) Note that, depending on your implementation, you may wish to detect only cards that have been explicitly given on the command line, or you may wish to detect all cards, or only one specific card — it's up to you as the driver writer. The "standard" behaviour, though, in the absence of any command line options to the contrary, is to detect and install all cards, but if command line options are specified indicating a particular card, then only that card should be detected and installed. Generic command line options (like **verbose**, for example) should have no effect on the card-scanning functionality.

Once you've detected your card(s), you'll need to perform whatever setup is appropriate at the hardware level and the software level (e.g., initialization of control ports, hardware interrupt allocation and binding, creation of data structures, etc.).

Binding to `io-net`

Once the device is configured, you'll want to bind it into the **`io-net`** hierarchy. This is done via the *reg()* function, from the table of function pointers that is passed in the **`io_net_registrant_funcs_t`** (described above).

Once bound in, you'll receive callouts from **`io-net`** into the functions that you specified. Your hardware will most likely generate interrupts (or inform you in some other way that data has arrived); you'll then use the callins to **`io-net`** to inform it that data has arrived (after suitable processing on your end).

@@@ Is something like this at all useful? It's a work-in-progress that I used to get some initial understanding. We could probably use this for a "big picture" with suitable massaging…

```
io_net_registrant_funcs_t    speedo_funcs = {
    8,
    NULL,                       /* rx_up  - I'm a driver */
    &speedo_send_packets,       /* rx_down() */
    &speedo_receive_complete,   /* tx_done() */
    &speedo_shutdown1,          /* shutdown() */
    &speedo_shutdown2,          /* shutdown() */
    &speedo_advertise,          /* advertise_ifaces ??? */
    &speedo_devctl,             /* devctl() */
    &speedo_flush,              /* flush() */
    NULL                        /* RAW open() ??? - nraw */
};

io_net_registrant_t speedo_entry = {
    _REG_PRODUCER_UP,           /* flags */
    "devn-speedo",              /* name */
    "en",                       /* top_type */
    NULL,                       /* bot_type */
    NULL,                       /* rx_down_hdl - load with Nic on each register */
    &speedo_funcs,              /* funcs */
    0                           /* dependencies */
};

in_net_dll_entry_t:
    init = speedo_init;
    speedo_init() {
        speedo_detect() {
            nic_create_device();
            speedo_scan() {
```

```
            speedo_register_device(){
                speedo_config() {
                }
                speedo_initialize() {
                }
                #define ion_register        ext->ion->reg
                ion_register(dll_hdl, &speedo_entry);
            }
            speedo_advertise(){
            }
        }
    }
}
```

The `npkt_t` data type

The **`npkt_t`** structure is defined as follows:

```
typedef struct _npkt {
    TAILQ_HEAD(, _net_buf) buffers;
    npkt_t        *next;
    void          *io_net0;
    void          *org_data;
    uint32_t      flags;
    uint32_t      framelen;
    uint32_t      tot_iov;
    uint32_t      io_net1;
    uint32_t      ref_cnt;
    uint16_t      num_complete;
    uint16_t      req_complete;
    uint16_t      cell;
    uint16_t      endpoint;
    uint16_t      iface;
    uint16_t      skip;
    union {
        void      *p;
        unsigned char c [16];
    } inter_module;

// this field follows the structure implicitly:
    npkt_done_t   c [];
} npkt_t;
```

The fields are defined as:

buffers — A queue of buffers, managed using the *TAILQ*()* macros from **`<sys/queue.h>`**.

next — Pointer to next **`npkt_t`**.

io_net0, io_net1, ref_cnt

Internal to **`io-net`**, do not examine or modify.

org_data — For the exclusive use of the originator of this **`npkt_t`**. No one else should touch this member.

flags — Status of buffer, see below.

framelen — Total length of the entire packet.

tot_iov — Total number of **`iov`**s in the packet.

num_complete — @@@number complete? Indicates number of elements in the **`npkt_done_t`** array which implicitly immediately follows this structure?

req_complete — Required number of **`npkt_done_t`** elements this downward-headed packet required before it reached its final destination. Only for information purposes (read-only) in originator's *tx_done()* function if originator isn't using **`io-net`**'s *alloc_down_pkt()* function.

cell — Cell **`npkt_t`** is headed to / from.

endpoint — Endpoint within cell.

iface — Interface within endpoint.

skip — @@@**`seanb`** sez that this is reserved for the future. Here's a work in progress: For use by _REG_FILTER_BELOW types of modules. The idea is that they could receive a packet from above, modify it somehow, stuff their *reg_hdl* in the *skip* member, and return TX_DOWN_AGAIN (from **`<sys/io-net.h>`**) from their *rx_down()* function. The down producer would see TX_DOWN_AGAIN and resend it down (after re-checksuming, re-routing, etc.), but this time, the _REG_FILTER_BELOW would be skipped by

io-net. This hasn't really been tried yet. The code is in **io-net**, but no filters have been written yet and the stacks don't check for TX_DOWN_AGAIN yet...

inter_module.p, *inter_module.c*

A data area that can be used by any module to pass information to the module above or below it.

c (implied) (array, implicitly immediately after this structure) On a downward-headed transmission, this array is *num_complete* elements long, whereas on an upward-headed transmission, it's always 1 element long. Note that this array isn't part of the structure proper, but implicitly immediately follows the structure.

And the *flags* parameter is selected from the following:

_NPKT_EXCLUSIVE

You have exclusive access to this upward bound **npkt_t**.

_NPKT_NO_RES

Up producer wants its buffer back right away.

_NPKT_UP **npkt_t** is headed in the up direction; down if this bit is not set.

_NPKT_MSG Indicates that this message is intended for a different layer, rather than actually containing packet data.

_NPKT_MSG_DYING

When a driver is unmounted, **io-net** synthesizes a _NPKT_MSG|_NPKT_MSG_DYING *npkt* and sends it up as though it came from the driver. It has no data in it. It means this *cell*, *endpoint*, and all *iface*s are gone.

_NPKT_BCAST @@@Broadcast?

_NPKT_MCAST

@@@Multicast?

_NPKT_INTERNAL

Internal to **`io-net`**.

Chapter 10

PCI Drivers

In this chapter. . .

PCI drivers

PCI drivers

This chapter describes the PCI drivers in detail.

Chapter 11

USB Drivers

In this chapter. . .

USB drivers
USB Driver Library reference
USB Skeleton Driver

USB drivers

REVISION 00 06 12 09 30

This chapter describes:

- the architecture of the USB stack and driver library,
- the functions available to writers of class drivers,
- the data structures used by the stack, and
- a sample “skeleton” driver which can be used as the basis for your own class driver.

Overview

USB (Universal Serial Bus) is a hardware and protocol specification for the interconnection of various devices to a host controller. We supply a USB stack that implements the protocol, and allows user-written class drivers to communicate with these devices. We also supply a USBD (USB Driver) library that class drivers use to communicate with the USB stack. (Note that the class driver can be considered to be a “client” of the USB stack.)

The stack is implemented as a stand-alone process that registers the pathname of **`/dev/usb`** (by default). The stack (currently) contains the hub class driver within it.

Data buffers are implemented via a shared memory interface that’s managed by the USB stack and the client library. That is to say, the client library provides functions to allocate data buffers in the shared memory region, and the stack manages these data buffers and gives the client library access to them. This means that all data transfers must use the provided buffers. The one limitation that this imposes is that a class driver *must* be on the same node as the USB stack. The *clients* of the class driver, however, *can* be network distributed. The advantage of this approach is that no additional memory copy occurs between the time that the data is received by the USB stack and the time that it’s delivered to the class driver (and vice versa).

@@@ talk about USB enumerators here... Here's a random, unverified paragraph for your amusement/discussion (from **rk**):

A USB enumerator is supplied with Neutrino. The enumerator attaches to the USB stack, and waits for device insertions. When a device insertion is detected, the enumerator looks in the configuration manager's database to see which class driver it should start. The driver is then started, and provides the appropriate services for that class of device — for example, a USB Ethernet class driver would register with **io-net** and bring the interface up. For small, deeply-embedded systems, the enumerator is not required; the class drivers can be started individually, and they'll wait around for their particular devices to be detected by the stack. After that, they'll provide the appropriate services for that class of device, just as if they'd been started by the enumerator. When a device is removed, the enumerator will shut down the class driver.

USB Driver Library reference

This section describes the USBD API calls available and the data structures that are commonly used. Generally, a class driver will perform the following operations (see the "USB Skeleton Driver," below, for implementation details):

1 connect to the USB stack (via the *usbd_connect()* function), and provide two callbacks; one for insertion and one for removal.

2 in the insertion callback:

- **2a** connect to the USB device (via the *usbd_attach()* function),
- **2b** select the configuration (*usbd_select_config()*) and interface (*usbd_select_interface()*), and
- **2c** set up communications pipes to the appropriate endpoint (*usbd_open_pipe()*).

3. in the removal callback, detach from the USB device (via the *usbd_detach()* function)
4. all data communications (e.g., reading and writing data, sending and receiving control information) are set up via the *usbd_setup_*()* functions and initiated via the *usbd_io()* function (with completion callbacks if required).

☞ Note that the term "pipe" is a USB-specific term, and has *nothing* to do with standard POSIX "pipes" (as used, for example, in the command line **`ls | more`**). In USB terminology, a "pipe" is simply a handle; something that identifies a connection to an endpoint.

Functions by category

The following function categories are provided:

- Connection
  - *usbd_connect()*
  - *usbd_disconnect()*
  - *usbd_attach()*
  - *usbd_detach()*
- Memory management
  - *usbd_alloc()*
  - *usbd_free()*
  - *usbd_alloc_dev()*
  - *usbd_free_dev()*
  - *usbd_alloc_urb()*
  - *usbd_free_urb()*
- Data transfer
  - *usbd_setup_bulk()*
  - *usbd_setup_feature()*

  - *usbd_setup_intr()*
  - *usbd_setup_isoch()*
  - *usbd_setup_vendor()*
  - *usbd_io()*

- Pipe management
  - *usbd_open_pipe()*
  - *usbd_reset_pipe()*
  - *usbd_abort()*
  - *usbd_close_pipe()*
- Configuration / interface management
  - *usbd_select_config()*
  - *usbd_select_interface()*
- Miscellaneous (@@@ should these be subdivided or moved?)
  - *usbd_get_desc()*
  - *usbd_hcd_info()*
  - *usbd_status()*

Alphabetical listing of functions and structures

This section contains the detailed function and structure definition descriptions, presented alphabetically.

Functions

The following functions are available to class drivers:

`usbd_abort (@@@TBD@@@)`

This routine aborts all requests on a pipe. This function can be used during an error condition (for example, to abort a pending operation), or during normal operation (for example, to halt an isochronous transfer).

void *usbd_alloc (int *flags*, **size_t** *size*)

Allocates a memory area that can then be used for message transfer. You *must* use the memory area allocated by this function, because it's allocated in shared memory (shared between the class driver, via its library, and the USB stack).

To free the memory, use *usbd_free()* below.

usbd_device_t *usbd_alloc_dev (size_t *****ext_size)

@@@ Does stuff, eh?

To free the memory, use *usbd_free_dev()* below.

urb_t *usbd_alloc_urb (int *flags*, **size_t** *extra*)

Allocates an URB for subsequent URB-based operations. The *extra* parameter indicates how much extra data area (in bytes) should be allocated immediately after the URB. This data area is for your own use and isn't used by the stack in any way. @@@ need a function to get at it, as an URB is (supposedly) an opaque data type @@@

To free the memory, use *usbd_free_urb()* below.

usbd_attach (@@@TBD@@@)

This routine allows a class driver to attach to a specific device. The *flags* argument can be any one of the following:

- USBD_ATTACH_RDWR
- USBD_ATTACH_RDONLY
- USBD_ATTACH_EXCL
- USBD_ATTACH_SHARE

@@@ Prolly meant to say (RDWR or RDONLY) and (optionally) (EXCL or SHARE), right? @@@ What about WRONLY (for a printer, for example)?

Use *usbd_detach()* to detach from the device.

You'd generally call this function in your insertion callback (as passed to the *usbd_connect()* function) to attach to the newly-inserted device.

`usbd_close_pipe (usbd_pipe_t *`*pipe***`)`**

This function closes a pipe (passed via the *pipe* argument) previously opened by the *usbd_open_pipe()* function.

`usbd_connect (@@@TBD@@@)`

This function creates a connection to the USB stack. It provides notification of insertion / removal of devices through the **`usbd_entry_t *`***entry* parameter's callouts. Notification can be limited to specific devices by using the *class*, *sclass* (subclass), *protocol*, *did* (device ID), and *vid* (vendor ID) parameters. Any number of those parameters can be the wildcard constant USBD_CONNECT_WILDCARD. The *path* parameter is used to specify which USB stack to connect to. The default (recommended) stack can be specified by using the manifest constant USBD_DFLT_STACK.

Important! This function creates a thread on your behalf, which is used by the library to monitor the USB stack for device insertion or removal. The implication is that your insertion and removal callback functions are called by this new thread; therefore you'll have to ensure that any common resources used between that thread and any other thread(s) in your class driver are properly protected (e.g., via a mutex).

Use *usbd_disconnect()* to destroy the connection after you're done with it.

usbd_detach (@@@TBD@@@)

This routine releases the ownership of the device back to the stack. The device must have been attached using *usbd_attach()*.

usbd_disconnect (@@@TBD@@@)

This function disconnects from the USB stack. The *connection* parameter is the one previously obtained from the *usbd_connect()* function.

int usbd_free (void *_p_)

Frees memory allocated by *usbd_alloc()* above.

int usbd_free_dev (usbd_device_t *_dev_)

Frees memory allocated by *usbd_alloc_dev()* above.

int usbd_free_urb (urb_t *_urb_)

Frees memory allocated by *usbd_alloc_urb()* above.

int usbd_get_desc (usbd_device_hdlt_t *_dhdl_, _uint32 _type_, _uint32 _index_, _uint16 _langid_, usbd_addr_t _addr_, int _len_)

Retrieves descriptors defined by the *type* parameter; one of the following:

- USB_DESC_CONFIGURATION
- USB_DESC_DEVICE
- USB_DESC_ENDPOINT
- USB_DESC_INTERFACE

- USB_DESC_STRING

The descriptor is returned into the data area pointed to by *addr* and is limited to *len* bytes. In cases where more than one descriptor can be returned for a given request, the *index* argument is used to indicate which one is to be returned. (Use the value 0 to indicate the first or only one.) The following descriptor type definitions are available (conforming to the layout mandated by the USB specification) in the include-file **<@@@sys???/usb.h>** and correspond to the descriptor *type* manifests (above):

- **usb_configuration_descriptor_t**
- **usb_device_descriptor_t**
- **usb_endpoint_descriptor_t**
- **usb_interface_descriptor_t**
- **usb_string_descriptor_t**

The data structure definitions for the individual descriptor types are detailed below, under "Structures."

usbd_hcd_info (@@@TBD@@@)

@@@ Does stuff, eh? HCD == Host Controller Driver

Refer to the structure definition section, under **usbd_hcd_info_t** for information on the returned data.

usbd_io (@@@TBD@@@)

This routine submits a previously-setup URB (from one of the functions *usbd_setup_bulk()*, *usbd_setup_feature()*, *usbd_setup_intr()*, *usbd_setup_isoch()*, or *usbd_setup_vendor()*, below) to the USB stack.

This function is the one that *actually* makes the data transfer happen; the setup functions simply set up the URB for the data transfer.

usbd_open_pipe (@@@TBD@@@)

This function initializes the pipe described by the **usb_endpoint_descriptor_t *** *desc* parameter. The pipe handle is returned through the pointer to *pipe*. The pipe may be closed via *usbd_close_pipe()*.

usbd_reset_pipe (usbd_pipe_t * *pipe* **)**

Clears a stall condition on an endpoint identified by the pipe *pipe*.

usbd_select_config (@@@TBD@@@)

@@@ Does stuff, eh?

usbd_select_interface (@@@TBD@@@)

@@@ Does stuff, eh?

usbd_setup_bulk (@@@TBD@@@)

Sets up an URB for a bulk transfer operation, which can be triggered by a subsequent call to *usbd_io()*. @@@ describe parameters

usbd_setup_feature (@@@TBD@@@)

Sets up an URB for a feature transfer operation, which can be triggered by a subsequent call to *usbd_io()*. @@@ describe parameters

usbd_setup_intr (@@@TBD@@@)

Sets up an URB for an interrupt transfer operation, which can be triggered by a subsequent call to *usbd_io()*. @@@ describe parameters

usbd_setup_isoch (@@@TBD@@@)

Sets up an URB for an isochronous transfer operation, which can be triggered by a subsequent call to *usbd_io()*. @@@ describe parameters

usbd_setup_vendor (@@@TBD@@@)

Sets up an URB for a vendor transfer operation, which can be triggered by a subsequent call to *usbd_io()*. @@@ describe parameters

usbd_status (@@@TBD@@@)

Returns status information on an URB.

Structures

The following structure definitions are used in conjunction with the functions listed above:

urb_t

The structure is formally defined as:

```
typedef struct _urb {
// @@@ is this guy opaque or not?
// @@@ if not, where do I find it?
} urb_t;
```

The members are defined as follows:

@@@member

    @@@definition

usb_configuration_descriptor_t

The structure is formally defined as:

```
typedef struct _usb_configuration_descriptor {
    _uint8   bLength;
    _uint8   bDescriptorType;
    _uint16  wTotalLength;
    _uint8   bNumInterfaces;
    _uint8   bConfigurationValue;
    _uint8   iConfiguration;
    _uint8   bmAttributes;
    _uint8   MaxPower;
} usb_configuration_descriptor_t;
```

The members are defined as follows:

bLength Size of this structure.

bDescriptorType
Contains the value @@@.

wTotalLength @@@?

bNumInterfaces The number of interfaces present.

bConfigurationValue
@@@?

iConfiguration @@@?

bmAttributes @@@?

MaxPower @@@maximum power in @@@ (units, mA?)

usb_device_descriptor_t

The structure is formally defined as:

```
typedef struct _usb_device_descriptor {
    _uint8   bLength;
    _uint8   bDescriptorType;
    _uint16  bcdUSB;
    _uint8   bDeviceClass;
    _uint8   bDeviceSubClass;
    _uint8   bDeviceProtocol;
    _uint8   bMaxPacketSize0;
    _uint16  idVendor;
    _uint16  idProduct;
    _uint16  bcdDevice;
    _uint8   iManufacturer;
    _uint8   iProduct;
    _uint8   iSerialNumber;
    _uint8   bNumConfigurations;
} usb_device_descriptor_t;
```

The members are defined as follows:

bLength Size of this structure.

bDescriptorType
Contains the value @@@.

bcdUSB @@@?

bDeviceClass The device class.

bDeviceSubClass
The device subclass.

bDeviceProtocol
@@@?

bMaxPacketSize0
Maximum packet size for endpoint 0 (the control endpoint).

idVendor Vendor ID.

idProduct Product ID.

bcdDevice @@@?

iManufacturer @@@?

iProduct @@@?

iSerialNumber @@@?

bNumConfigurations
@@@?

usb_endpoint_descriptor_t

The structure is formally defined as:

```
typedef struct _usb_endpoint_descriptor {
    _uint8  bLength;
    _uint8  bDescriptorType;
    _uint8  bEndpointAddress;
    _uint8  bmAttributes;
    _uint16 wMaxPacketSize;
    _uint8  bInterval;
} usb_endpoint_descriptor_t;
```

The members are defined as follows:

bLength Size of this structure.

bDescriptorType
Contains one of (@@@ at least):
USB_ISOCHRONOUS_ENDPOINT,
USB_BULK_ENDPOINT, or
USB_INTERRUPT_ENDPOINT.

bEndpointAddress
@@@?

bmAttributes @@@?

wMaxPacketSize

Maximum packet size for this endpoint.

bInterval @@@ speed? transfer rate? in what units?

usb_interface_descriptor_t

The structure is formally defined as:

```
typedef struct _usb_interface_descriptor {
    _uint8  bLength;
    _uint8  bDescriptorType;
    _uint8  bInterfaceNumber;
    _uint8  bAlternateSetting;
    _uint8  bNumEndpoints;
    _uint8  bInterfaceClass;
    _uint8  bInterfaceSubClass;
    _uint8  bInterfaceProtocol;
    _uint8  iInterface;
} usb_interface_descriptor_t;
```

The members are defined as follows:

bLength Size of this structure.

bDescriptorType

Contains the value @@@.

bInterfaceNumber

@@@?

bAlternateSetting

@@@?

bNumEndpoints

The number of endpoints.

bInterfaceClass

@@@?

bInterfaceSubClass

@@@?

bInterfaceProtocol

@@@?

iInterface @@@?

usb_string_descriptor_t

The structure is formally defined as:

```
typedef struct _usb_string_descriptor {
    _uint8  bLength;
    _uint8  bDescriptorType;
    _uint16 bString [1];
} usb_string_descriptor_t;
```

The members are defined as follows:

bLength Size of this structure.

bDescriptorType

Contains the value @@@.

bString The returned string, in 16 bits-per-character encoding as per the specified language (@@@izit UTF-something-or-other?@@@).

usbd_connection_t

The **usbd_connection_t** structure is a member of both **struct _usbd_device_ctrl** and **usbd_device_hdl_t**, and represents @@@

The structure is formally defined as:

```
typedef struct _usbd_connection {
    int                                 fd;
    TAILQ_HEAD (,_usbd_device_handle)   dlist;
    usbd_entry_t                       *entry;
    _uint32                             flags;
    _uint32                             class;
    _uint32                             sub_class;
    _uint32                             protocol;
    _uint32                             device_id;
    _uint32                             vendor_id;
    _uint32                             rsvd [8];
} usbd_connection_t;
```

The members are defined as follows:

fd	@@@?
dlist	@@@?
entry	@@@?
flags	@@@?
class	Device class.
sub_class	Device subclass.
protocol	@@@?
device_id	Device ID.
vendor_id	Vendor ID.
rsvd	Reserved, do not examine or modify.

`struct _usbd_device_ctrl`

The **`struct _usbd_device_ctrl`** structure is used for @@@

The structure is formally defined as follows:

```
struct _usbd_device_ctrl {
    TAILQ_HEAD (, _usbd_device) dlist;
    usbd_connection_t           *connection;
    usbd_entry_t                *entry;
};
```

The members are defined as follows:

dlist @@@?

connection @@@?

entry @@@?

usbd_device_hdl_t

The **usbd_device_hdl_t** structure is a member of **usbd_device_t** and represents @@@

The structure is formally defined as:

```
typedef struct _usbd_device_handle {
    usbd_connection_t                   *uhdl;
    TAILQ_ENTRY (_usbd_device_handle)    dlink;
    _uint32                              dev_path;
    _uint32                              dev_addr;
    _uint32                              dev_cfg;
    _uint32                              dev_iface;
    TAILQ_HEAD (,_usbd_pipe)             plist;
    _uint32                              rsvd [8];
} usbd_device_hdl_t;
```

The members are defined as follows:

uhdl @@@?

dlink @@@?

dev_path @@@?

dev_addr @@@?

dev_cfg @@@?

dev_iface @@@?

plist @@@?

rsvd Reserved, do not examine or modify.

usbd_device_t

The **usbd_device_t** structure is used for @@@.

The structure is formally defined as:

```
typedef struct _usbd_device {
    TAILQ_ENTRY (_usbd_device)  dlink;
    void                       *dext;
    usbd_device_hdl_t          *dhdl;
    uint32_t                    dstatus;
    uint32_t                    dflags;
    uint32_t                    dev_addr;
    uint32_t                    dev_conf;
    uint32_t                    dev_iface;
    uint32_t                    verbosity;
} usbd_device_t;
```

The members are defined as follows:

dlink @@@?

dext @@@?

dhdl @@@?

dstatus @@@?

dflags @@@?

dev_addr Device address.

dev_conf Device configuration.

dev_iface Device interface.

verbosity @@@?

usbd_entry_t

The **usbd_entry_t** structure is passed to *usbd_connect()* to provide callback functions.

The structure is formally defined as:

```
typedef struct _usbd_entry {
    _uint32 nentries;

    void    (*insertion)
                (_uint32 rsvd,
                 _uint32 dev_addr,
                 _uint32 class,
                 _uint32 sclass,
                 _uint32 proto,
                 _uint32 did,
                 _uint32 vid);

    void    (*removal)
                (_uint32 dev_addr,
                 usbd_device_t *devptr);
} usbd_entry_t;
```

The members are defined as follows:

nentries The number of entries in the structure. For the structure as shown above, this would be the value 2.

insertion Callback function to call when an insertion of a USB device is detected. Filtering of the particular type of USB device is achieved via the *usbd_connect()* function's parameters. The callback function is called

with parameters describing the device that was detected: *class*, *sclass*, *proto*, *did*, and *vid* contain the class, subclass, protocol, device ID, and vendor ID (respectively). The parameter *dev_addr* contains the USB address of the device.

removal An optional callback function to call when a USB device is removed. If you don't wish to supply a *removal()* function, specify the constant NULL.

Note that the insertion and removal functions are called from a thread that's created by the *usbd_connect()* function, so you must take care to ensure that any resources shared between that thread and any other thread(s) are properly protected (e.g. via a mutex).

usbd_hcd_info_t

The **usbd_hcd_info_t** structure is filled by *usbd_hcd_info()*, above, and contains information about the Host Controller Driver (HCD).

The structure is formally defined as:

```
typedef struct _usbd_hcd_info {
    _uint32     version;
    _uint32     capabilities;
    _uint32     bandwidth;
    _uint32     rsvd [12];
    cfg_info_t  cfg; // @@@where's cfg_info_t defined?
} usbd_hcd_info_t;
```

The members are defined as follows:

version Top 16 bits represent the USB version, and the bottom 16 bits represent the stack version.

capabilities	Capabilities; one or more of the following bit values: CAP_CNTL, CAP_BULK, CAP_INTR, CAP_ISOCH, CAP_LOW_SPEED, and CAP_HIGH_SPEED.
bandwidth	The currently allocated bandwidth in @@@ (units).
rsvd	Reserved, do not examine or modify.
cfg	@@@?

usbd_pipe_t

The **usbd_pipe_t** structure is the handle used to identify a pipe.

The structure is formally defined as:

```
typedef struct _usbd_pipe {
    usbd_device_hdl_t        *dev;
    TAILQ_ENTRY (_usbd_pipe)  plink;
    _uint32                   type;
    _uint32                   endpoint;
    _uint32                   interval;
    _uint32                   packet_size;
    _uint32                   reserved [4];
} usbd_pipe_t;
```

The members are defined as follows:

dev	@@@?
plink	@@@?
type	@@@?
endpoint	@@@?
interval	@@@?
packet_size	@@@?
reserved	Reserved, do not examine or modify.

USB Skeleton Driver

The following annotated code sample shows the skeletal outline of a typical USB class driver, which conforms to the general outline presented in the USB Driver Library reference above.

```
// work in progress...
// Example class driver.

#include <stdio.h>
#include <errno.h>
#include <stddef.h>
#include <signal.h>
#include <pthread.h>
#include <sys/usbdi.h>

#include "skel.h"

void skel_write_complete( urb_t *urb, void *chdl )
{
    chdl = chdl;

    // notify io-blk, io-net of status
    pthread_sleepon_lock( );
    pthread_sleepon_signal( urb );
    pthread_sleepon_unlock( );
}

int skel_write( void *ihdl, void *dptr, _uint32 len )
{
    _uint32         urb_status;
    _uint32         usb_status;
    _uint32         residual;
    skel_ext_t      *ext;
    usb_device_t    *sdev;

    sdev = (usb_device_t *)ihdl;
    ext  = (skel_ext_t *)sdev->ext;

    if( ( urb = usbd_alloc_urb( 0, 0 ) ) == NULL ) {
        return( ENOMEM );
    }

    if( ( uptr = usbd_alloc( 0, len ) ) == NULL ) {
        usbd_free_urb( urb );
        return( ENOMEM );
    }

    memcpy( uptr, dptr, len );

    usbd_setup_bulk( urb, NULL, USBD_DIR_OUT, uptr, len );

    if( usbd_io( urb, ext->ep_bout, skel_write_complete, sdev, USBD_TIME_DEFAULT ) ) {
        usbd_free_urb( urb );
        usbd_free( uptr );
        return( EIO );
```

```
    }

    pthread_sleepon_lock( );
    while( usbd_status( urb, &urb_status, &usb_status, &residual ) ) {
        pthread_sleepon_wait( urb );
    }
    pthread_sleepon_unlock( );

    usbd_status( urb, &urb_status, &usb_status, &residual );

    usbd_free_urb( urb );
    usbd_free( uptr );
    return( ( urb_status == USBD_REQ_CMP ) ? EOK : EIO );
}

int skel_setup_pipes( usb_device_t *sdev )
{
    _uint32                     ep;
    _uint32                     scan;
    _uint32                     found;
    usb_device_descriptor_t     ddesc;
    usb_endpoint_descriptor_t   edesc;
    skel_ext_t                 *ext;

    ext   = (skel_ext_t *)sdev->ext;
    scan  = SKEL_CONTROL_EP | SKEL_BLKIN_EP | SKEL_BLKOUT_EP;
    found = 0;

    if( usbd_get_desc( sdev->dhdl, USB_DESC_DEVICE, 0, 0,
                (usbd_addr_t *)&ddesc, sizeof( ddesc ) ) == EOK ) {
        if( usbd_open_pipe( sdev->dhdl, &ddesc, &ext->ep_cntl ) == EOK ) {
            found |= SKEL_CONTROL_EP;
        }
    }

    for( ep = 0; ep < USB_MAX_ENDPOINT; ep++ ) {
        if( usbd_get_desc( sdev->dhdl, USB_DESC_ENDPOINT, ep, 0,
                        (usbd_addr_t *)&edesc, sizeof( edesc ) ) ) {
            continue;
        }
        switch( edesc.bDescriptorType ) {
            case USB_ISOCHRONOUS_ENDPOINT:
                break;

            case USB_BULK_ENDPOINT:
                switch( edesc.bEndpointAddress & USB_EP_DIR ) {
                    case USB_EP_DIR_OUT:
                        if( usbd_open_pipe( sdev->dhdl, &edesc,
                                &ext->ep_bout ) == EOK ) {
                            found |= SKEL_BULKOUT_EP;
                        }
                        break;

                    case USB_EP_DIR_IN:
                        if( usbd_open_pipe( sdev->dhdl, &edesc,
                                &ext->ep_bin ) == EOK ) {
                            found |= SKEL_BULKIN_EP;
                        }
                        break;
                }
```

```
                break;

            case USB_INTERRUPT_ENDPOINT:
                break;
        }
    }
    return( ( found == scan ) ? SUCCESS : ERROR );
}

void skel_removal( _uint32 dev_addr, usbd_device_t *hdl )
{
    dev_addr = dev_addr;

    // free resources
    detach from io-blk, io-net, etc...
    TAILQ_REMOVE( &SkelCtrl.dlist, hdl, dlink );
    usbd_detach( hdl );
    usbd_free_dev( hdl );
}

void skel_insertion( _uint32 rsvd, _uint32 dev_addr, _uint32 class,
                     _uint32 sclass, _uint32 proto, _uint32 did, _uint32 vid )
{
    usbd_device_t                   *sdev;
    usb_device_descriptor_t         ddesc;
    usb_interface_descriptor_t      idesc;
    usb_configuration_descriptor_t  cdesc;

    if( ( sdev = usbd_alloc_dev( sizeof( skel_ext_t ) ) ) == NULL ) {
        perror( "usbd_alloc_dev:  " );
        return;
    }

    if( usbd_attach( SkelCntl.connection, dev_addr, USBD_ATTACH_RDWR, &sdev->dhdl ) ) {
        perror( "usbd_attach:  " );
        usbd_free_dev( sdev );
        return;
    }

    if( usbd_get_desc( sdev->dhdl, USB_DESC_DEVICE, 0, 0,
                       (usbd_addr_t *)&ddesc, sizeof( ddesc ) ) ) {
        perror( "usbd_get_desc (device):  " );
        usbd_detach( sdev->dhdl );
        usbd_free_dev( sdev );
        return;
    }

    if( usbd_get_desc( sdev->dhdl, USB_DESC_CONFIGURATION, 0, 0,
                       (usbd_addr_t *)&cdesc, sizeof( cdesc ) ) ) {
        perror( "usbd_get_desc (configuration):  " );
        usbd_detach( sdev->dhdl );
        usbd_free_dev( sdev );
        return;
    }

    if( usbd_get_desc( sdev->dhdl, USB_DESC_INTERFACE, 0, 0,
                       (usbd_addr_t *)&idesc, sizeof( idesc ) ) ) {
        perror( "usbd_get_desc (interface):  " );
        usbd_detach( sdev->dhdl );
        usbd_free_dev( sdev );
```

```
        return;
    }

    // selecting the configuration/interface will depend on your device
    if( usbd_select_config( sdev->dhdl, config ) ) {
        perror( "usbd_select_config:  " );
        usbd_detach( sdev->dhdl );
        usbd_free_dev( sdev );
        return;
    }

    if( usbd_select_interface( sdev->dhdl, config, interface, alternate ) ) {
        perror( "usbd_select_interface:  " );
        usbd_detach( sdev->dhdl );
        usbd_free_dev( sdev );
        return;
    }

    if( skel_setup_pipes( sdev ) ) {
        fprintf( stderr, "skel_setup_pipes:  \n" );
        usbd_detach( sdev->dhdl );
        usbd_free_dev( sdev );
        return;
    }

    // attach to io-blk, io-net, etc...

    // add sdev to device list
    TAILQ_INSERT_TAIL( &SkelCtrl.dlist, sdev, dlink );
}

int main( int argc, char *argv[] )
{
    sigset_t        set;
    siginfo_t       info;

    SkelCntl.entry.nentries   = 2;
    SkelCntl.entry.removal    = skel_removal;
    SkelCntl.entry.insertion  = skel_insertion;

    if( usbd_connect( USBD_DFLT_STACK, 0, &SkelCtrl.entry, USBD_CONNECT_WILDCARD,
        USBD_CONNECT_WILDCARD, USBD_CONNECT_WILDCARD, SKEL_DEVICE_ID,
        SKEL_VENDOR_ID, &SkelCntl.connection ) ) {
        perror( "usbd_connect:  " );
        exit( EXIT_FAILURE );
    }

    // become a resource manager at this point, or whatever...
    // in this example, we just wait for a termination signal
    sigfillset( &set );
    sigdelset( &set, SIGTERM );
    pthread_sigmask( SIG_BLOCK, &set, NULL );

    sigemptyset( &set );
    sigaddset( &set, SIGTERM );
    while( SignalWaitinfo( &set, &info ) == -1 )
        ;

    // free resources
```

```
    if( usbd_disconnect( SkelCntl.connection ) ) {
        perror( "usbd_disconnect:  " );
        exit( EXIT_FAILURE );
    }
    exit( EXIT_SUCCESS );
}
```

Appendix A
References

In this appendix. . .

References

References

The following publications are useful for gaining a good understanding of the QNX Neutrino operating system:

- Building Embedded Systems (QSSL)
- C Library Reference (QSSL)
- Programmer's Guide (QSSL)
- System Architecture Guide (QSSL)

Audio driver references

The audio driver APIs are based on the Linux ALSA ("Advanced Linux Sound Architecture") audio standard. For more information, visit **`www.alsa-project.org`** on the web.

Block I/O driver references

CAM spec? Various manufacturer docs, linux drivers...

Character I/O driver references

POSIX specs?

Graphics driver references

Various manufacturer docs, linux drivers...

Network driver references

Various manufacturer docs, linux drivers...

PCI driver references

Various manufacturer docs, linux drivers...

USB driver references

Various manufacturer docs, linux drivers... Also, **`www.usb.org`**.

Glossary

Alpha (graphics) Alpha blending is 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 portion of source color, the more opaque the object looks.

ALSA Advanced Linux Sound Architecture; an industry standard for audio devices for the Linux community. Our sound drivers are based on the **API** presented in the specification.

Anonymous Memory A chunk of memory that's not identified by a name; for example, you may require a chunk of memory in which to perform a DMA transfer, but you don't need to identify that memory to other processes.

API Application Program Interface; the interface through which applications access services. This is the "published" interface that applications should use when communicating with a driver.

BLT (graphics) BLock Transfer (sometimes "BLIT" for BLock Image Transfer); generally refers to the ability to move a rectangular array of pixels from one location to another. The source and destination may be on the video card or the system RAM, depending on the configuration.

Block (driver) A driver for a device that is accessed in a "block" manner — that is, accessed as a collection of bytes, rather than on an individual byte-by-byte basis. Contrast with **Character (driver)**.

CAM Common Access Method; a specification for @@@

Character (driver) A driver for a device that is accessed on a character-by-character basis. Contrast with **Block (driver)**.

Chroma keying (graphics) Chroma operations are a method of masking out pixel data during a rendering operation (copies, image rendering, rectangles, etc.) based on a chroma color value. The four basic modes of operation are: masking on the source key color, masking on the destination key

color, masking on everything but the source key color, and masking on everything but the destination key color.

Colour The correct spelling for "color." `:-)` See also "labour," "neighbour," and "judgement."

Configuration Descriptor (USB) @@@ See also Endpoint Descriptor (USB) and Device Descriptor (USB).

Device Descriptor (USB) @@@ See also Endpoint Descriptor (USB) and Configuration Descriptor (USB).

DMA Direct Memory Access; a technique used by hardware peripherals to transfer data between the memory subsystem and the peripheral without involving the CPU in the data transfer itself.

DLL Also known as a shared object; an object module that can be loaded at runtime to augment the process that it's loaded into.

Double Buffer (graphics) @@@ a technique that makes use of two buffers; one is the "current" buffer that's displayed on the device, and the second is the "drawing" buffer that's being updated. When the drawing buffer is updated, the graphics card is told to use that buffer as the current buffer, and the previously-current buffer becomes the drawing buffer. Allows updates to an image to occur without any intermediate drawing operations being visible; useful for animation.

DPMS (graphics) Display Power Management System; a method of putting the monitor into one of several low-power modes defined by VESA.

Endpoint Descriptor (USB) @@@ See also Device Descriptor (USB) and Configuration Descriptor (USB).

Endpoint (USB) @@@

FIFO First In First Out — a queueing order in which the oldest entry added to a queue is the first entry that's removed from the queue, then the next-oldest entry, and so on. Contrast with LIFO. Also refers to a hardware component that implements this queueing behaviour by typically using a memory component for the storage of fixed-size entries.

Frame Buffer (graphics) The memory area that's currently being used for display.

Isochronous (USB) A transfer mode characterized by continuous, realtime requirements. For example, a video stream must arrive at a defined rate, and hence must reserve a certain amount of bandwidth.

LIFO Last In First Out — a queueing order in which the most-recent entry added to a queue is the first entry that's removed from the queue, then the next-most-recent, and so on. Contrast with FIFO.

Message Passing Neutrino's primary inter-process communications scheme. Messages are sent from client to server, with the client blocking until the server replies. The server can block, waiting for messages to arrive.

NTSC (graphics) @@@ North American Television Standard for Colour (?) or Never The Same Colour; a standard defining the electrical signal used to represent video. Contrast with PAL and SECAM.

PAL (graphics) Phase Alternate Line; a European standard defining the electrical signal used to represent video. Contrast with NTSC and SECAM. Also, to confuse the issue, the abbreviation "PAL" is used for palette.

Palette (graphics) A (usually small) number of colours represented by an index as opposed to "directly" by red, green, and blue components. Used to save on the amount of memory space provided on a graphics card, at the expense of providing a full spectrum of colours.

PCI Peripheral Component Interconnect; a hardware bus present on many types of systems that allows peripherals to be interfaced to the CPU.

Physical Address An address that corresponds directly to the signals emitted on the bus (ISA, PCI, etc.) Generally used with DMA devices. Contrast with Virtual address.

Pipe (USB) In the context of USB, a pipe is a connection between a client program and an endpoint (and does not refer to a traditional UNIX pipe).

POSIX Portable Operating System Interface; a specification defining various commands and APIs for a conforming system.

QSSL QNX Software Systems Limited; the company that manufactures the QNX family of operating systems, of which Neutrino is the latest member.

Resource Manager (or "resmgr") A device driver for Neutrino that handles certain well-defined messages from clients. These messages correspond to various file-descriptor based functions that the client of the driver calls.

RGB format (graphics) A format for storing pixel colours where the "R" component represents the intensity of red, "G" for green, and "B" for blue. Contrast with YUV format. Also used to refer to the signals present on a connector, whereby the red, green, and blue components of the colour are presented on separate pins.

ROPS (graphics) @@@ Raster Operations.

SECAM (graphics> @@@???@@@ Contrast with NTSC and PAL.

Span (graphics) Another term for "horizontal line."

Stride (graphics) The number of bytes that must be added to a given memory offset to get to the next pixel below the given one. This is a function of the memory organization of the graphics card.

Surface (graphics) @@@

USB Universal Serial Bus; an interconnect bus targetted for peripherals.

VESA (graphics) Video Electronics Standards Association.

Virtual Address An address that does *not* (necessarily) correspond with the signals emitted onto the hardware bus. Such an address is local to a process; this means that two different processes running on the same processor may in fact both have the same virtual address, but each process will have the virtual address translated to different physical addresses by the MMU (Memory Management Unit). Contrast with Physical Address.

VLAN Very Local Area Network; used to denote an (often) proprietary network architecture often used for high-availability systems.

YUV format (graphics) A format for storing pixel colours where the “Y” component represents @@@, the “U” component represents @@@, and the “V” component represents @@@. Contrast with RGB format.