Graphics Driver for External Use

Sean Williams, Kevin Williams, Fernando Urbina

Apple Computer, Inc.

Version 1.0.1

July 17, 1995

Overview

Graphics Driver for External Use (GDX) is a template for a native graphics driver which can be quickly adapted to new hardware implementations. This driver fully conforms to all the requirements for a native driver, as described in *Designing PCI Cards and Drivers for Power Macintosh Computers*.

This template has been extensively tested, since it is the basis of all the graphics drivers used in the initial round of PCI based Power Macintoshes. Third party developers need only create a single file describing their hardware implementation.

The code is structured into three modules:

- Core
  For graphics drivers, there is a core set of software which is invariant across hardware implementations or OS services. This core handles the majority of the control and status calls received. Third parties will not have to modify the Core[1].

- Hardware Abstraction Layer (HAL)
  This section is responsible for performing operations on the underlying hardware, and reporting the hardware's capabilities to the Core. A HAL will have to be provided for each implementation of graphics hardware. Two HAL templates have been provided which can be easily adapted for third party hardware.

- Operating System Services (OSS)
  This section handles OS services, such as how to register and service interrupts, receive parameter blocks, etc. Third parties will not have to modify the OSS.

Although the majority of third parties will be developing graphics drivers for PCI devices, GDX is not PCI centric. Rather, it will run under any operating system that supports Slot Manager Independent (SMI) graphics drivers. For example, PCI, PDS, NuBus, PCMCIA, or direct attach frame buffer controllers can be supported.

1. See the section "Changing the Core" on page 30 for a list of exceptions.

How Did We Do?
Abstraction and modularization look good on paper, but carrying the design forward into implementation is sometimes more challenging. Here is a self-assessment of GDX's implementation:

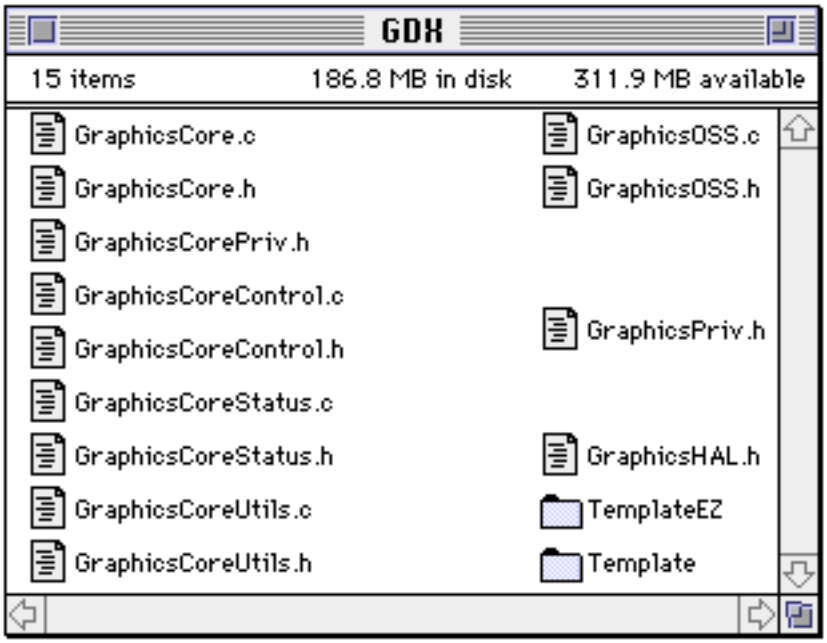
Core: A+
The Core came into place quite nicely. Every nuance of a graphics control or status call is handled, or passed off to the HAL when appropriate. The process of developing five GDX based drivers assured that there were no hardware dependencies in the core.

HAL: A+
The interface between the HAL and the Core is extremely robust. Moreover, that interface has proved flexible enough to support wildly different hardware implementations.

OSS: C
As it became apparent that Copland and System 7.5.2 were converging in their use of the Name Registry, Expansion Manager, Interrupt Manager, etc., less effort was applied toward developing a robust OSS layer. For example, it would not be possible to simply write a new OSS layer for Windows, and have the Core and HAL remain unchanged. However, the existing OSS is a nice start down the path toward that ideal.

Design Layout

The files in GDX are organized as follows:



All Modules
All three modules need to include the following:

GraphicsPriv.h
This file has declarations, error codes, and constants that are used throughout GDX. All GDX files will need to include this. However, no items outside of GDX will need access to this file, hence the 'Priv' postfix.

Core
The Core is composed of multiple files and constitutes the majority of the brains of GDX. Although it is not necessary to make changes here, inspecting this code can be quite useful for understanding the relationship between a graphics driver and its clients. Additionally, this would also be a useful reference if you were developing or maintaining a graphics driver from another source base.

GraphicsCore.c
This files contains the majority of the code which handles the driver commands `kInitializeCommand`, `kReplaceCommand`, `kOpenCommand`, `kCloseCommand`, `kControlCommand`, `kStatusCommand`, `kSupersededCommand`, and `kFinalizeCommand`.

Additionally, it has the routines to initialize and kill the private data that the Core uses to maintain its state information.

GraphicsCore.h
This merely has the function declarations of the Core routines that have external scope to other GDX files.

GraphicsCorePriv.h
This has declarations that are strictly private to the items that comprise the Core. Neither the OSS or HAL will need to include this file.

GraphicsCoreControl.c
This file implements the core portion of the various Control calls.

GraphicsCoreControl.h
This has the function declarations of the Control routines that have external scope to other GDX files.

GraphicsCoreStatus.c
This file implements the core portion of Status calls.

GraphicsCoreStatus.h
This has the function declarations of the Status routines that have external scope to other GDX files.

GraphicsCoreUtils.c
This has some basic utility functions that an item in the Core or HAL might want to make use of. In particular, a HAL which has standard sense codes will probably want to make

use of the `GraphicsUtilMapSenseCodesToDisplayCode`(), which will uniquely map a raw sense code / extended sense code pair to a `DisplayCode`.

GraphicsCoreUtils.h
This has the function declarations of the utility routines that have external scope to other GDX files.

HAL
This consists only of a header file and a C file. The HAL is responsible for accessing the hardware and reporting its abilities. To ease development efforts, two templates have been provide which can be used as a basis for a real HAL. Choose the template which best matches your needs, or combine certain aspects of them to produce a new template.

GraphicsHAL.h
This contains the function declarations that a HAL must implement. These functions have external scope to other GDX files.

GraphicsHALTemplateEZ.c
This can be used as a template for implementing a HAL. This template is referred to as "EZ" since the hardware has a simple register model. For example, to establish a proper raster for a 640 x 480 display at 67 Hz, only a single register needs to be accessed in the frame buffer controller and the CLUT.

Additionally, it can be considered "EZ" for the following reasons:

Always has enough VRAM available to support all of its resolutions.
No hardware cursor.
No special lower power modes.

GraphicsHALTemplate.c
This can be used as a template for implementing a HAL. This template has a more complex internal model than the "EZ" template for the following reasons:

Support for hardware cursor.
Special low power modes.
Complex register model.
Different resolutions available depending on amount of VRAM present.

OSS
The OSS provides a thin layer of abstraction for operating system services. Essentially, it can be considered as a set of utility functions for OS services that a graphic driver uses often.

GraphicsOSS.c
This file implements all of the OSS functionality. Essentially, the OSS provides a means for saving, retrieving, and deleting properties from the Name Registry, saving and retrieving the Core's and HAL's preferences, and dealing with interrupts.

GraphicsOSS.h
This has the function declarations of the OSS routines that have external scope to other GDX files.

Creating a HAL from a Template

The templates provided are extensively documented in the code. If you are familiar with your hardware specifics, they can be adapted in a short period of time. Here are some hints to get off to a quick start.

Global Search and Replace

The templates use a strict naming convention for their hardware specific variables, so a major portion of the adaptation can be accomplished via a global *case sensitive* search and replace in the template file.

For example, in TemplateEZ, perform the following:

TABLE 1. Search and Replace for TemplateEZ

Search For…	Replace With…
TemplateEZ	YourArchitecture
templateEZ	yourArchitecture
Cosmo	YourFrameBufferController
cosmo	yourFrameBufferController
Irazu	YourCLUT
irazu	yourCLUT

Similarly, the process of converting Template can be started by performing the following:

TABLE 2. Search and Replace for Template

Search For…	Replace With…
Template	YourArchitecture
template	yourArchitecture
Toynbee	YourFrameBufferController
toynbee	yourFrameBufferController
Spur	YourCLUT
spur	yourCLUT
MrSanAntonio	YourTimingGeneratorGoesHere
mrSanAntonio	yourTimingGeneratorGoesHere

Create Your Register Models

The register models used in the templates were designed to be generic, so they will need to be updated to reflect your hardware.

Start at the Top, Work to the Bottom
Start at the first function in the template, and try and adapt it to your hardware. If uncertain what to do, defer the decision and go on to the next function.

Don't Try to Map a HAL Routine Directly to a Control or Status Call
The HAL routines are quite primitive. In most cases, they don't correspond directly to a Control or Status call. Instead, they provide the Core with information it needs to respond to a Control or Status call. Think of the HAL routines as the simple items that they are, and the Core will deal with patching everything together.

GraphicsHALxxx vs. Templatexxx
In the templates, some functions will be prefixed with 'GraphicsHAL' and others will be prefixed by 'Template.'

The routines which start with 'GraphicsHAL' must be implemented by all HALs. The routines starting with 'Template' are strictly private to the HAL, and are completely implementation dependent.

Use PopupFuncs
This is a really cool utility routine will add a popup menu listing all the source file's functions to the title bar. It can be installed into any file editor, such as MPW Shell, Metrowerks, or Think.

This makes navigating through unfamiliar source files a breeze. The demo installer has been provided.

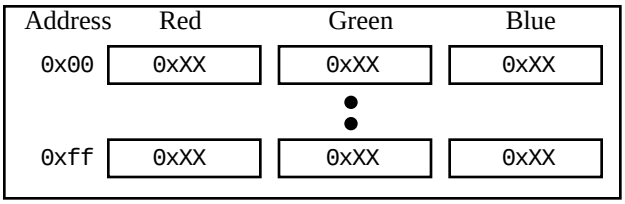
MPW Make File
For those of the MPW persuasion, a make file has been provided that builds both template drivers. Install the headers and libraries from the PCI DDK into you MPW folder, set the directory to GDX, and Build!

Metrowerks Projects
A Metrowerks project has been provided for building each template. Again, install the headers and libraries from the PCI DDK into Metrowerks and build. Don't forget to either recompile the headers, or don't use precompiled headers.

CLUTs & You (A Primer)

Color Lookup Tables / Digital to Analog Convertors (CLUT/DACs, hereafter referred to simply as CLUTs) provide hardware that converts pixel values stored in the frame buffer to some actual RGB video value. For indexed color modes (1-8 bits per pixel), the operation of the CLUT is fairly intuitive. For direct color modes (16 or 32 bits per pixel), the operation is somewhat obtuse. Consider a generic triple 256x8 CLUT which supports 1 - 32 bpp as shown below:

FIGURE 1. Generic Triple 256x8 CLUT

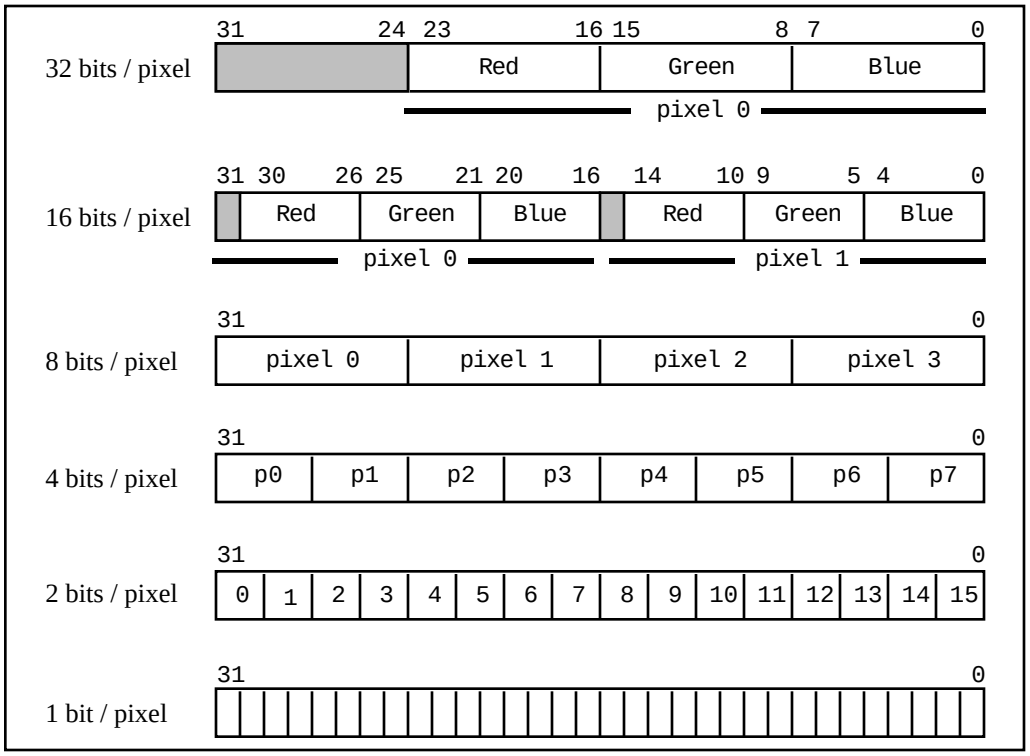


This CLUT is referred to as a “triple 256x8” CLUT for the following reasons:

- triple The CLUT has three channels: red, green and blue.
- 256 The CLUT has 256 physical addresses.
- 8 For each channel at a given address, an 8-bit value can be stored.

To help explain the differences between indexed and direct modes, a generic representation of how pixels are represented in a frame buffer is detailed below. Please notice that for 16 and 32 bpp, each channel has a number of bits dedicated to it, whereas that is not the case for 1 - 8 bpp.

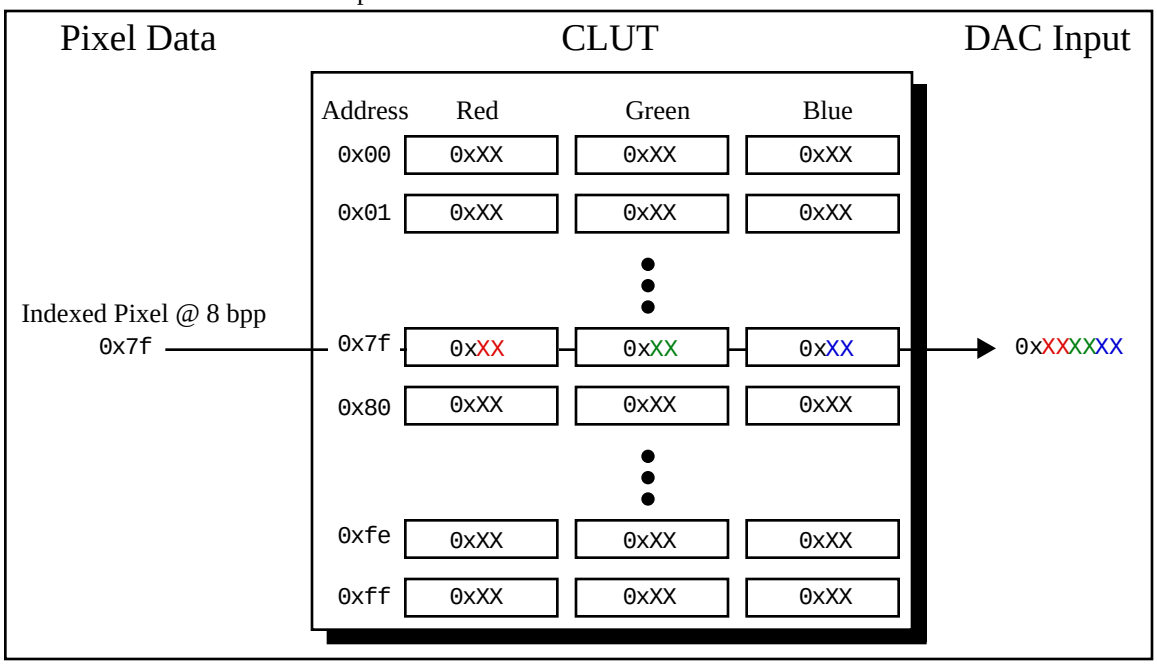
FIGURE 2. Generic Pixel Data Format in Frame Buffer



Indexed Color & CLUT Operation

For 1 - 8 bpp, the pixel data in the frame buffer represents the logical address of a CLUT entry. For each pixel, a single lookup takes place, and three 8-bit values (red, green, blue) are extracted and used as inputs for their respective channel's DAC. Figure 3 illustrates this:

FIGURE 3. Indexed Color Example



From this example, it should be clear that the pixel data (`0x7f`) is used as the logical CLUT address. From that address, three 8-bit values are extracted, and then used as inputs to the DAC to provide the correct analog video signal.

The range of logical CLUT addresses is based solely on the number of bits per pixel:

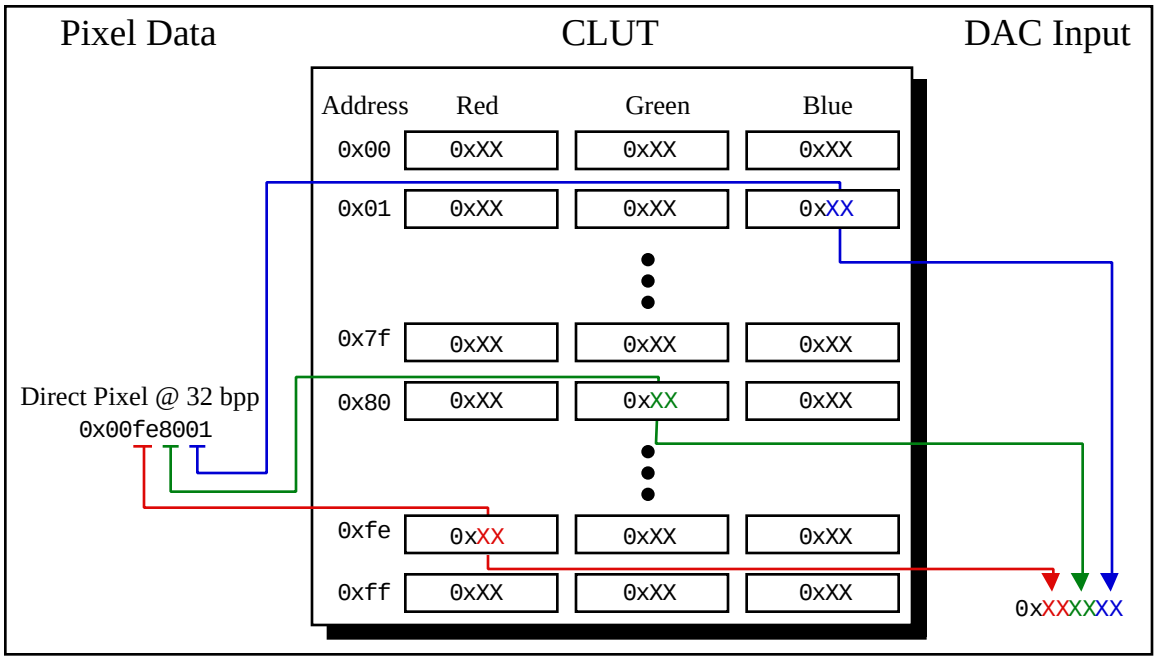
TABLE 3. Logical Address Range (Indexed Color)

Bits Per Pixel	Logical Address Range (2^{bpp}-1)
1	0 - 1
2	0 - 3
4	0 - 15
8	0 - 255

Direct Color & CLUT Operation

For 16 or 32 bpp, the CLUT is used solely for gamma correction. Since each pixel has bits dedicated to each channel (red, green, and blue), a lookup for each channel occurs. For each lookup, the logical CLUT address is derived from that channel's pixel data. The gamma corrected value is then extracted from the appropriate channel's lookup table.

FIGURE 4. Direct Color Examples



As shown above, three lookups occur. The red channel bits (`0xfe`) indicate the logical CLUT address to access. At the location, the 8-bits of red information is extracted, and used as input to the red channel's DAC. This process is repeated for the green and blue channels.

For direct color, the range of logical addresses is based on the number of bits per *channel*:

TABLE 4. Logical Address Range (Direct Color)

Bits Per Pixel	Bits Per Channel	Logical Address Range (2^{bpc}-1)
16	5	0 - 31
32	8	0-255

Graphics Core Routines

The routines that live in the Core are largely defined by the client API to the graphics driver. Since the client API is documented in *Designing PCI Cards and Drivers for Power Macintosh Computers*, it will not be repeated here.

However, in addition to the Control and Status routines in the Core, there are several utility routines, one of which a HAL might want to take advantage of:

GraphicsUtilMapSenseCodesToDisplayCode()
This routine will map `RawSenseCode` / `ExtendedSenseCode` pairs to their corresponding `DisplayCode` for frame buffer controllers which either have ‘standard’ sense code hardware or can coerce their raw / extended sense codes to appear standard.

This functionality is provided as a utility routine in the Core, because a large number of frame buffer controllers have support for standard sensing.

```
GDXErr GraphicsUtilMapSenseCodesToDisplayCode(RawSenseCode rawSenseCode,
            ExtendedSenseCode extendedSenseCode,
            DisplayCode *displayCode)
```

-> `rawSenseCode`
Result from reading sense lines when none are being actively driven.

-> `extendedSenseCode`
Result from applying extended sense algorithm to sense lines.

<- `displayCode`
`DisplayCode` which the `RawSenseCode` / `ExtendedSenseCode` pair maps to.

Graphics OSS Routines

The majority of the OSS routines are used by the Core, but some might be utilized by the HAL. The routines that the HAL might want to access involve accessing the NameRegistry, accessing preferences in NVRAM, and enabling/disabling interrupts.

GraphicsOSSSaveProperty()
The OSS calls the Name Registry to save information. The OSS doesn't care about the content.

```
GDXErr GraphicsOSSSaveProperty(const RegEntryID *regEntryID,
            const char *propertyName, const void *propertyValue,
            ByteCount propertySize,
            OSSPropertyStorage ossPropertyStorage)
```

-> `regEntryID`
The node in which the property should be saved.

-> `propertyName`
C string property name.

-> `propertyValue`
Opaque pointer to the buffer containing the data to be stored.

-> `propertySize`
Size of the data, in bytes.

-> `ossPropertyStorage`
GDX internal flags that describe how the property should get saved:

`kOSSPropertyAvailableAtDisk`	Available when disk is up (saved across boots).
`kOSSPropertyVolatile`	Property not saved across boots.

GraphicsOSSGetProperty()
The OSS calls the NameRegistry to get information. The OSS doesn't care about the content. If the property doesn't exist, that's an error.

```
GDXErr GraphicsOSSGetProperty(const RegEntryID *regEntryID,
            const char *propertyName, void *propertyValue,
            ByteCount propertySize);
```

-> `regEntryID`
The node from which the property should be retrieved.

-> `propertyName`
C string property name.

-> `propertyValue`
Opaque pointer to the buffer to contain the retrieved data.

-> `propertySize`
The expected size of the data, in bytes.

GraphicsOSSDeleteProperty()
The OSS calls the NameRegistry to delete properties. It doesn't care what property is deleted.

```
GDXErr GraphicsOSSDeleteProperty(const RegEntryID *regEntryID,
            const char *propertyName);
```

-> `regEntryID`
The node from which the property should be deleted.

-> `propertyName`
C string name of the property to delete.

GraphicsOSSSetHALPref()
Graphics drivers get 8 bytes of nonvolatile RAM (NVRAM) allocated to them. They can use these bytes to store preferences, so that the desired state can be retrieved during boot time.

The Core uses 4 of those bytes to maintain the state information it needs (`DisplayModeID`, `DepthMode`, and `DisplayCode`), leaving the other 4 bytes for use by the HAL. The HAL can use those bytes for any data it chooses.

As with the routines listed above, the OSS uses the NameRegistry to store this information.

```
GDXErr GraphicsOSSSetHALPref(const RegEntryID *regEntryID,
            UInt32 halData)
```

-> `regEntryID`
The node for which the preferences should be set.

-> `halData`
The 4 bytes the HAL wishes to save in NVRAM.

GraphicsOSSGetHALPref()
This call allows the HAL to retrieve the 4 bytes it previously set.

```
GDXErr GraphicsOSSGetHALPref(const RegEntryID *regEntryID,
            UInt32 *halData)
```

-> `regEntryID`
The node from which the preferences should be retrieved.

<- `halData`
The 4 bytes retrieved from NVRAM for the HAL.

GraphicsOSSSetVBLInterrupt()
This routine is used to allow/prevent VBL interrupts from propagating to the processor. The exact behavior of this function depends on how the HAL responded to the `GraphicsHALGetVBLInterruptRoutines()` call, as detailed on page 16.

```
Boolean GraphicsOSSSetVBLInterrupt(Boolean enableInterrupts)
```

-> `enableInterrupts`
`true` if interrupts should be enabled, `false` otherwise.

<- `Boolean`
If disabling interrupts, then this is `true` if interrupts were previously enabled, `false` otherwise.

It is undefined when enabling interrupts.

GraphicsOSSVBLDefaultEnabler()
Sometimes, the HAL might wish to enable the motherboard interrupt source, regardless of what it returned for the `GraphicsHALGetVBLInterruptRoutines()`. This routine will do so, leaving the state of the internal interrupt source on the card unchanged.

```
void GraphicsOSSVBLDefaultEnabler(void)
```

GraphicsOSSVBLDefaultDisabler()
This is the counterpart to the above routine. Should the HAL wish to disable the motherboard interrupt source, it can call this function. As above, the state of the internal interrupt source on the card will be unchanged.

```
Boolean GraphicsOSSVBLDefaultDisabler(void)
```

<- `Boolean`
`true` if motherboard interrupts were previously enabled, `false` otherwise.

GraphicsOSSDoVSLInterruptService()
This routine calls the Video Service Library (VSL) to service the VBL tasks associated with this graphics device.

Normally, a HAL implementation would *never* have to call this routine. However, in the rare event that a HAL's hardware does not support true hardware interrupts, then the HAL should call this during its *simulated VBL* routine to allow the OS to service items in its VBL task queue.

```
void GraphicsOSSDoVSLInterruptService(void)
```

Graphics HAL Routines

The following is a description of the routines which comprise the Graphics HAL. All HALs must implement these routines.

GraphicsHALInitPrivateData()
This routine is called after the Core has received an `kInitializeCommand` or a `kReplaceCommand`. The HAL should allocate whatever private storage it requires, perform the necessary operations to determine its hardware's addresses, and initialize internal state information.

```
GDXErr GraphicsHALInitPrivateData(const RegEntryID *regEntryID,
            Boolean *replacingDriver)
```

`->` `regEntryID`
This is the `RegEntryID` for the driver. It should be copied via the `RegistryEntryIDCopy()` function, in the event that the NameRegistry is queried later.

`<->` `replacingDriver`
On input, this indicates whether the Core got a `kInitializeCommand` or a `kReplaceCommand`. These commands are similar, but with subtle differences. A `kInitializeCommand` is issued if no version of this driver has been previously loaded, whereas a `kReplaceCommand` is issued if a previous version of the driver has been loaded, but subsequently superseded.

If `false`, then a `kInitializeCommand` had been received by the Core, and the HAL should do a full hardware initialization.

If `true`, then a `kReplaceCommand` had been received by the Core, and the HAL can attempt to configure itself to its state prior to it being superceded.

On output, this allows the HAL to override the Core's default behavior if it chooses to do so.

If `false`, then the HAL is signaling the Core that it is unable to reconfigure itself to its state prior to being superseded, and the Core will continue as if a `kInitializeCommand` had occurred.

If `true`, then the HAL was able to re-configure itself in the event of being replaced, and the Core will proceeded accordingly.

GraphicsHALOpen()
It is possible for the driver to be opened and closed many times. This routine should perform the required initialization of the hardware in order to determine the amount of VRAM that is in the system, and any other hardware specific items that the HAL cares about. Additionally, the HAL might initialize internal state information or retrieve its preferences via the `GraphicsOSSGetHALPref()` routine.

No programming to set up a raster for a given `DisplayModeID` or `DepthMode` is necessary at this point.

```
GDXErr GraphicsHALOpen(const AddressSpaceID spaceID,
            Boolean replacingDriver)
```

-> `spaceID`
This is the `AddressSpaceID` for the HAL's hardware.

-> `replacingDriver`
`true` if the HAL should behave as if the driver is being replaced, `false` otherwise.

GraphicsHALClose()
Upon close, there are no major requirements, since the majority of the work will be handled elsewhere.

```
GDXErr GraphicsHALClose(const AddressSpaceID spaceID)
```

-> `spaceID`
This is the `AddressSpaceID` the HAL's hardware.

GraphicsHALTerminate()
This routine is called after the Core has received an `kFinallizeCommand` or a `kSupersededCommand`.

```
GDXErr GraphicsHALTerminate(Boolean superseded)
```

-> `superseded`
`true` if the current driver is going to be superseded by another driver, `false` otherwise.

If `true`, the current driver can choose to save any state that the replacement driver may need, and can choose whether or not to keep the raster going.

If `false`, no driver is going to replace it. In that event, it should stop the raster and leave the hardware in a polite state.

GraphicsHALKillPrivateData()
This routine is called when it is time for the HAL to dispose of its private data. For example, any memory it allocated should be returned, and any `RegEntryID`s that were copied should be disposed.

```
void GraphicsHALKillPrivateData(void)
```

GraphicsHALDetermineDisplayCode()
This routine is called whenever it is necessary to determine the type of display that is connected to the hardware. When this routine is called, the following actions should occur:

- Perform required steps to determine what display is connected (e.g., read sense lines).
- Update the HAL's state information regarding the type of display connected, if the HAL maintains that state information.

```
GDXErr GraphicsHALDetermineDisplayCode(DisplayCode *displayCode)
```

`<-` `displayCode`
`DisplayCode` for the attached display.

In the event that the HAL is does not recognize the specific type of display attached, it should set `*displayCode = kDisplayCodeUnknown.`

GraphicsHALDrawHardwareCursor()
This routine sets the hardware cursor's X and Y coordinates and its visible state. If the cursor was set successfully by a previous call to `GraphicsHALSetHardwareCursor()`, then the HAL must program the hardware with the given X, Y and visible state. If the previous call to `GraphicsHALSetHardwareCursor()` failed, then an error should be returned.

```
GDXErr GraphicsHALDrawHardwareCursor(SInt32 x, SInt32 y,
            Boolean visible)
```

`->` `x`
X coordinate.

`->` `y`
Y coordinate.

`->` `visible`
`true` if the cursor must be visible, `false` if it should be hidden.

GraphicsHALGetBaseAddress()
This returns the base address of a specified page in the current mode. This allows graphics pages to be written to even when not displayed.

```
GDXErr GraphicsHALGetBaseAddress(SInt16 page, char **baseAddress)
```

-> `page`
(0 based) Page number for which the base address is desired.

<- `baseAddress`
Base address of desired page.

GraphicsHALGetCLUT()
This routine will fill out the specified array of `ColorSpecs` with the contents of the CLUT.

The `RGBColor` structure in each `ColorSpec` uses 16-bits for each channel (red, green, and blue), whereas most CLUTs only use 8-bits. Therefore, when filling in the `RGBColor` structure, the most significant byte for each channel should be filled with the 8-bits extracted from its respective channel in the CLUT. Moreover, to maintain the same behavior as the previous drivers, the 8-bits from the CLUT should also be written to the least significant byte for each `RGBColor`.

It is important to note that the positions of the entries refer to logical positions, not physical ones. At 4 bpp, for example, the entry positions could range from 0, 1, 2,…, 15, even though the physical positions may not have this number sequence.

No range checking is required, because the Core has already done so.

```
GDIErr GraphicsHALGetCLUT(ColorSpec *csTable, SInt16 startPosition,
            SInt16 numberOfEntries, Boolean sequential,
            DepthMode depthMode)
```

<-> `csTable`
This is a pointer to the array of `ColorSpecs` provided by the caller to be filled with the contents of the CLUT.

-> `startPosition`
(0 based) Starting point in the array to fill.

-> `numberOfEntries`
(0 based) The number of entries to get.

-> `sequential`
If `false`, then the `value` field of the `ColorSpec` should be inspected to see what logical position should be retrieved. If `true`, then the array index indicates what logical position should be read.

-> `depthMode`
The relative bit depth. This is provided so that the HAL can decide how to map the logical entry positions to the physical entry positions.

GraphicsHALGetHardwareCursorDrawState()
This routine is used to determine the state of the hardware cursor. After HAL initialization the cursor's visible state and set state should be `false`. After a mode change the cursor should be made invisible but the set state should remain unchanged.

```
GDXErr GraphicsHALGetHardwareCursorDrawState(SInt32 *cursorX,
            SInt32 *cursorY, UInt32 *cursorVisible, UInt32 *cursorSet)
```

<- `cursorX`
X coordinate from last `GraphicsHALDrawHardwareCursor()` call.

<- `cursorY`
Y coordinate from last `GraphicsHALDrawHardwareCursor()` call.

<- `cursorVisible`
`true` if the cursor is visible, `false` otherwise.

<- `cursorSet`
`true` if last `GraphicsHALDrawHardwareCursor()` call was successful, `false` otherwise.

GraphicsHALGetDefaultDisplayModeID()
This routine is used to get the default `DisplayModeID` and `DepthMode` for a display. This routine gets called when a new display is connected to the computer. The HAL knows how much VRAM is available and which `DisplayModeID`s it supports, so this call is used to determine the best settings for a particular display.

```
GDXErr GraphicsHALGetDefaultDisplayModeID(DisplayCode displayCode,
            DisplayModeID *displayModeID, DepthMode *depthMode)
```

-> `displayCode`
The connected display.

<- `displayModeID`
The default `DisplayModeID` for the connected display.

<- `depthMode`
The default `DepthMode`.

GraphicsHALGetMaxDepthMode()

This takes a `DisplayModeID` and returns the maximum `DepthMode` that is supported by the hardware. No check is made to determine if the `DisplayModeID` is valid for the connected display. The HAL should return an error if the `DisplayModeID` is not supported or there is not enough VRAM to support the `DisplayModeID`.

```
GDXErr GraphicsHALGetMaxDepthMode(DisplayModeID displayModeID,
            DepthMode *maxDepthMode);
```

-> `displayModeID`
Get the information for this `DisplayModeID`.

<- `maxDepthMode`
Maximum relative bit depth for the `DisplayModeID`.

GraphicsHALGetModeTiming()

This is used to gather scan timing information for a specific `DisplayModeID`.

```
GDXErr GraphicsHALGetModeTiming(DisplayModeID displayModeID,
            UInt32 *timingFormat, UInt32 *timingFlags)
```

-> `displayModeID`
The `DisplayModeID` for which the information is desired.

<- `timingFormat`
Currently, `kDeclROMtables` is the only valid response for this field.

<- `timingFlags`
This bit field indicates whether the specified `DisplayModeID` is valid, safe, and/or the default for the connected display. The bits are defined as follows:

`kModeValid`	Set if HAL believes the connected display can support the specified `DisplayModeID`.
`kModeSafe`	Set if HAL is 100% certain the connected display can support the specified `DisplayModeID`.
`kModeDefault`	Set if the specified `DisplayModeID` is the default for the connected display.

If the HAL doesn't believe the specified `DisplayModeID` is applicable to the connected display, it should set `timingFlags` to 0, and the Display Manger will subsequently attempt to query any Display Modules present in the system.

GraphicsHALGetNextResolution()
This routine is used to iterate over the `DisplayModeID`s the HAL supports. The Core has taken care of most of this work, so the HAL simply has to return the next `DisplayModeID` supported. It is important to note that all `DisplayModeID`s should be reported, regardless of what display is physically connected.

```
GDXErr GraphicsHALGetNextResolution(DisplayModeID previousDisplayModeID,
            DisplayModeID *displayModeID, DepthMode *maxDepthMode)
```

-> `previousDisplayModeID`
If `previousDiplayModeID = kDisplayModeIDFindFirstResolution`, get the first supported resolution by the hardware.

Otherwise, `previousDiplayModeID` contains the `DisplayModeID` from the previous call, so report the subsequent `DisplayModeID`.

<- `displayModeID`
`DisplayModeID` of the next display mode following `previousDisplayModeID`. Set this to `kDisplayModeIDNoMoreResolutions` once all supported `DisplayModeID`s have been reported.

<- `maxDepthMode`
Maximum relative bit depth for the `displayModeID`.

GraphicsHALGetPages()
This routine reports the number of graphics pages supported for the specified `DisplayModeID` at the specified `DepthMode`.

No attempt should be made to determine whether or not a display capable of being driven with a raster of type `DisplayModeID` is physically connected.

```
GDXErr GraphicsHALGetPages(DisplayModeID displayModeID,
            DepthMode depthMode, SInt16 *pageCount)
```

-> `displaymodeID`
The `DisplayModeID` for which the page count is desired.

-> `depthMode`
The `DepthMode` for which the page count is desired.

<- `pageCount`
# of pages supported at the specified `DisplayModeID` and `DepthMode`. In the event of an error, `pageCount` is undefined. This is a counting number, so it is not zero based.

GraphicsHALGetPowerState()

The graphics hardware might have the ability to go into some kind of power saving mode. This call is used to determine the current power state. If the hardware does not support changing power states, then it can return `kGDXErrUnsupportedFunctionality`.

```
GDXErr GraphicsHALGetPowerState(VDPowerStateRec *vdPowerState)
```

For this routine, the relevant fields indicated by `vdPowerState` are:

<- `powerState`
The current power mode: `kAVPowerOff`, `kAVPowerStandby`, `kAVPowerSuspend`, or `kAVPowerOn`.

<- `powerFlags`
Bit field for conveying additional information. Currently the following bits are defined:

`kPowerStateNeedsRefresh`	Set if VRAM needs to be refreshed after coming out of the designated power state.

GraphicsHALGetSenseCodes()

This routine is called whenever the state of the sense codes need to be reported. This should only report the sense code information. No attempt should be made to determine what type of display is attached here. Moreover, the sense codes should be determined *every time* this call is made, and not make use of any previously saved values.

```
GDXErr GraphicsHALGetSenseCodes(RawSenseCode *rawSenseCode,
            ExtendedSenseCode *extendedSenseCode,
            Boolean *standardInterpretation)
```

<- `rawSenseCode`
For standard sense code hardware, this value is found by instructing the hardware not to actively drive any of the monitor sense lines, and then reading the state of the monitor sense lines 2, 1, and 0. (2 is the MSB, 0 the LSB)

<- `extendedSenseCode`
For standard sense code hardware, the extended sense code algorithm is as follows: (Note: as described here, sense line ‘A’ corresponds to ‘2’, ‘B’ to ‘1’, and ‘C’ to ‘0’)

- Drive sense line ‘A’ low and read the values of ‘B’ and ‘C’.
- Drive sense line ‘B’ low and read the values of ‘A’ and ‘C’.
- Drive sense line ‘C’ low and read the values of ‘A’ and ‘B’.

In this way, a six-bit number of the form BC/AC/AB is generated.

<- `standardInterpretation`
If standard sense code hardware is implemented (or the values are coerced to appear standard) then set this to `true`. Otherwise, set it to `false`, and the interpretation for `rawSenseCode` and `extendedSenseCode` will be considered private.

GraphicsHALGetSync()

This routine is called to determine the frame buffer controller's abilities for handling the various syncing signals, and also to determine the current status of the syncs. If the connected display supported the Video Electronics Standards Association (VESA) Device Power Management Standard (DPMS), it would respond to the horizontal and vertical syncs in the following manner:

TABLE 5. DPMS Interpretation & Sync Bits

DPMS State	Vertical Sync	Horizontal Sync	Display State	kDisableVerticalSyncBit	kDisableHorizontalSyncBit
Active	Pulses	Pulses	Active	0	0
Standby	Pulses	No Pulses	Blanked	0	1
Idle	No Pulses	Pulses	Blanked	1	0
Off	No Pulses	No Pulses	Blanked	1	1

```
GDXErr GraphicsHALGetSync(Boolean getHardwareSyncCapability,
            VDSyncInfoRec *sync)
```

-> `getHardwareSyncCapability`
If `true`, then report the capability of the hardware. If `false`, then report the current state of the sync lines and which channel (if any) that the hardware is syncing on.

For this routine, the relevant fields of the `VDSyncInfoRec` structure are as follows:

<- `csMode`
If `getHardwareSyncCapability = true`, then report the capability of the hardware. When reporting the capability of the hardware, set the appropriate bits of `csMode`:

`kDisableHorizontalSyncBit`	Set if HW can disable Horizontal Sync.
`kDisableVerticalSyncBit`	Set if HW can disable Vertical Sync.
`kDisableCompositeSyncBit`	Set if HW can disable Composite Sync.
`kSyncOnRedEnableBit`	Set if HW can sync on red.
`kSyncOnGreenEnableBit`	Set if HW can sync on green.
`kSyncOnBlueEnableBit`	Set if HW can sync on blue.
`kNoSeparateSyncControlBit`	Set if HW cannot enable/disable H, V, C sync independently. Means that HW only supports the 'Off' or 'Active' state.

If `getHardwareSyncCapability = false`, then report the current state of sync lines and if the hardware is syncing on red, green, or blue. Reporting the 'current state of the sync lines' effectively means 'report the state of the display.'

To report if the hardware is syncing on red, green or blue, set the following bits accordingly:

`kSyncOnRedEnableBit`	Set if HW is syncing on red.
`kSyncOnGreenEnableBit`	Set if HW is syncing on green.
`kSyncOnBlueEnableBit`	Set if HW is syncing on blue.

GraphicsHALGetVBLInterruptRoutines()

The OSS encapsulates how interrupts are handled by the system. This routine supplies the OSS with the HAL's interrupt routines that follow the OSS conventions. Hopefully, if the OS changes, only the OSS will need to change.

```
GDXErr GraphicsHALGetVBLInterruptRoutines(Boolean *installVBLInterrupts,
            Boolean *chainDefault, VBLHandler **halVBLHandler,
            VBLEnabler **halVBLEnabler, VBLDisabler **halVBLDisabler,
            void **vblRefCon);
```

<- `installVBLInterrupts`
`true` if the HAL's interrupt scheme can match the OSS's scheme. i.e. the HAL lets the OSS handle most of the interrupt functions.

`false` if the HAL's interrupt scheme is radically different than the OSS's scheme. The HAL is responsible for knowing how the OS handles interrupts. (Obviously, this is the escape mechanism for a poor OSS design.)

If this is `false`, all other parameters are ignored.

<- `chainDefault`
If `halVBLEnabler` or `halVBLDisabler` is `NULL`, this is ignored by the OSS for the respective function since the default enabler/disabler supplied by the OS is used.

If `chainDefault = true`, and if the `halVBLEnabler` or `halVBLDisabler` is not `NULL`, the OSS will call the default OS enabler/disabler after the HAL's enabler/disabler is called

If `chainDefault = false`, and if the `halVBLEnabler` or `halVBLDisabler` is not `NULL`, the OSS will *not* call the default OS enabler/disabler after the HAL's enabler /disabler is called. The HAL assumes the responsibility for enabling/disabling the interrupt source on the motherboard. (Dangerous!)

<- `halVBLHandler`
The HAL's VBL handler which should clear and reprime the internal interrupt source.

<- `halVBLEnabler`
If `halVBLEnabler = NULL`, the default OS enabler will be called and the HAL can ignore things.

If `halVBLEnabler != NULL` and `chainDefault = true`, the HAL needs to enable the internal interrupt source, and the OSS will call the default OS enable routine to enable motherboard interrupts.

If `halVBLEnabler != NULL` and `chainDefault = false`, the HAL needs to enable the internal and motherboard interrupt source. (Dangerous!)

<- `halVBLDisabler`
If `halVBLDisabler = NULL`, the default OS enabler will be called and the HAL can ignore things.

If `halVBLDisabler!=NULL` and `chainDefault=true`, the HAL can choose to disable the internal interrupt source, and the OSS will call the default OS disable routine to disable motherboard interrupts.

If `halVBLDisabler!=NULL` and `chainDefault=false`, the HAL can choose to disable the internal and must disable the external interrupt source. (Dangerous!)

<- `vblRefCon`
If the HAL needs some data for the interrupt routines, then this opaque pointer can be used to reference it. The OSS will not attempt to interpret this in any manner.

GraphicsHALGetVideoParams()
This routine is used to obtain the `rowbytes` for a specified `DisplayModeID` and `DepthMode`. As a courtesy to the caller, the relative bit depth is also returned.

```
GDXErr GraphicsHALGetVideoParams(DisplayModeID displayModeID,
            DepthMode depthMode, UInt32 *bitsPerPixel,
            SInt16 *rowBytes)
```

-> `displayModeID`
The `DisplayModeID` for which the information is desired.

-> `depthMode`
The relative bit depth for which the information is desired.

<- `bitsPerPixel`
Absolute bit depth for the specified `DepthMode`.

<-> `rowBytes`
On input, `rowbytes` contains the horizontal pixels for the specified `DisplayModeID`.

On output, `rowbytes` should have the number of bytes between successive rows of video memory for the specified `DisplayModeID` and `DepthMode`.

GraphicsHALGrayCLUT()
This routine sets all the CLUT entries to 50% gray. This is useful so that the pixel depth can be subsequently changed without introducing screen anomalies, since 50% gray has the same representations at all bit depths. The 50% gray value will be obtained by using the midpoint value of the supplied gamma table.

```
GDIErr GraphicsHALGrayCLUT(const GammaTbl *gamma)
```

-> `gamma`
This is a pointer to a gamma table. An acceptable 50% gray value can be obtained by using the midpoint of each channel's correction data. It is the responsibility of the Core to make sure the gamma table is valid, so the HAL does not have to perform any error checking.

GraphicsHALMapDepthModeToBPP()
This routine is used to map a relative pixel depth (`DepthMode`) to an absolute pixel depth (bits per pixel).

```
GDXErr GraphicsHALMapDepthModeToBPP(DepthMode depthMode,
            UInt32 *bitsPerPixel)
```

-> `depthMode`
The relative pixel depth

<- `bitsPerPixel`
Corresponding absolute pixel depth.

GraphicsHALModePossible()

This routine checks to see if the hardware is capable of driving the given `DisplayModeID` at the indicated `DepthMode` and `page`. This does not check to see that the `DisplayModeID` is valid for the display type that is physically connected.

Important Note: The `GDXErr` return value does not indicate whether the mode is possible or not. It only signifies whether or not the value returned in `modePossible` was correctly determined. In the event of an error, `modePossible` does not contain valid information.

```
GDXErr GraphicsHALModePossible(DisplayModeID displayModeID,
            DepthMode depthMode, SInt16 page, Boolean *modePossible)
```

-> `displaymodeID`
The `DisplayModeID` for which the information is desired.

-> `depthMode`
The `DepthMode` for which the information is desired.

-> `page`
The page for which the information is desired.

<- `modePossible`
This will be `true` if the frame buffer can support the desired items, `false` otherwise. In the event of an error, `modePossible` is undefined.

GraphicsHALProgramHardware()

This routine attempts to program the graphics hardware to the desired `DisplayModeID`, `DepthMode`, and `page`. The HAL is not required to specifically check to see if the inputs are valid, since it can assume that the checking has been done elsewhere.

```
GDXErr GraphicsHALProgramHardware(DisplayModeID displayModeID,
            DepthMode depthMode, SInt16 page, Boolean *directColor,
            char **baseAddress)
```

-> `displayModeID`
The desired DisplayModeID.

-> `depthMode`
The desired relative bit depth.

-> `page`
The desired page.

<- `directColor`
`true` if the desired `DepthMode` results in the hardware being in a direct color mode, otherwise it is `false`. In the event on an error, it is undefined.

<- `baseAddress`
The resulting base address of the frame buffer's VRAM. In the event of an error, it is undefined.

GraphicsHALSetCLUT()

This routine will program the CLUT with the specified array of `ColorSpec`s. Two such arrays are provided, the original, and a second that has been luminance mapped (if appropriate) and gamma corrected. It is up to the HAL implementation to decide which array should be applied to the hardware. Most hardware will use the corrected version.

It is important to note that the positions of the entries refers to logical positions, not physical ones. At 4 bpp, for example, the entry positions could range from 0, 1, 2,…, 15, even though the physical positions may not have this number sequence.

No range checking is required, because the caller has already done so.

```
GDIErr GraphicsHALSetCLUT(const ColorSpec *originalCSTable,
            ColorSpec *correctedCSTable, SInt16 startPosition,
            SInt16 numberOfEntries, Boolean sequential,
            DepthMode depthMode)
```

-> `originalCSTable`
This is a pointer to the array of `ColorSpec`s provided by the caller. This is only provided in the event that the hardware should not use the `correctedCSTable`. If any adjustments need to made to it, then they should be done to a copy. Don't throw away the `const`!

-> `correctedCSTable`
This is essentially a copy of `originalCSTable`, except that it has been luminance mapped (if appropriate) and gamma corrected. Most hardware will use this information to set the CLUT. Though it is unlikely that this information will need to be changed, it is not marked as `const` in case it is used to build a special version from the `originalCSTable`. In that event, the array can be altered as necessary.

During the gamma correction process, the 16-bit representation of each channel in the `RGBColor` structure was mapped to an 8-bit (or less) representation.

For example, if prior to correction, an `RGBColor` was represented as follows:

```
rgbColor.red      = 0xAAAA;
rgbColor.green    = 0xBBBB;
rgbColor.blue     = 0xCCCC;
```

After gamma correction it might appear as:

```
rgbColor.red      = 0x00A9;
rgbColor.green    = 0x00B6;
rgbColor.blue     = 0x00C4;
```

Additionally, regardless of the size of the `originalCSTable` array, `correctedCSTable` points to an array of `ColorSpec`s with 256 entries.

-> `startPosition` (0 based) Starting point in the array.

-> `numberOfEntries` (0 based) This is the number of entries to be set.

-> `sequential`
If `false`, then the `value` field of the ColorSpec should be inspected to see what logical position should be set. If `true`, then the array index indicates what logical position should be set.

-> `depthMode`
The relative bit depth. This is provided so that the HAL can decide how to map the logical entry positions to the physical entry positions.

GraphicsHALSetHardwareCursor()

This routine is called to setup the hardware cursor and determine if whether the hardware can support it. The HAL should remember whether this call was successful for subsequent GetHardwareCursorDrawState() or DrawHardwareCursor() calls, but should NOT change the cursor's X or Y coordinates, nor its visible state.

```
GDXErr GraphicsHALSetHardwareCursor(const GammaTbl *gamma,
            Boolean luminanceMapping, void *cursorRef)
```

-> `gamma`
Current gamma table to correct cursor colors with, if the HAL can apply gamma correction.

-> `luminanceMapping`
This will be `true` if the Core had luminance mapping enabled and it was in an indexed color mode. If `true`, the HAL should luminance map the cursor CLUT *even* if the hardware cursor is a super-duper cursor capable of direct color. This is because the hardware cursor should look like the software cursor it is replacing.

-> `cursorRef`
Opaque data to be handed to `VSLPrepareCursorForHardwareCursor()`.

GraphicsHALSetPowerState()

The graphics hardware might have the ability to go into some kind of power saving mode. This call is used to change the current power state. If the hardware does not support changing power states, then it can return `kGDXErrUnsupportedFunctionality`.

```
GDXErr GraphicsHALGetPowerState(VDPowerStateRec *vdPowerState)
```

For this routine, the relevant fields indicated by `vdPowerState` is:

-> `powerState`
The desired power mode: `kAVPowerOff`, `kAVPowerStandby`, `kAVPowerSuspend`, or `kAVPowerOn`.

<- `powerFlags`
Bit field for reporting special conditions.

`kPowerStateNeedsRefresh`	Set if VRAM needs to be refreshed after coming out of the designated power state.

GraphicsHALSetSync()

This routine is used set the state of the hardware's sync lines. If the connected display conformed to DPMS, then it would respond as shown in Table 5, "DPMS Interpretation & Sync Bits," on page 22.

```
GDXErr GraphicsHALSetSync(UInt8 syncBitField, UInt8 syncBitFieldValid)
```

-> `syncBitField`
Bit field indicating which of the sync bits need to be disabled or enabled:

`kDisableHorizontalSyncBit`	Set if HW should disable horizontal sync.
`kDisableVerticalSyncBit`	Set if HW should disable vertical sync.
`kDisableCompositeSyncBit`	Set if HW should disable composite sync.
`kSyncOnRedEnableBit`	Set if HW should sync on red.
`kSyncOnGreenEnableBit`	Set if HW should sync on green.
`kSyncOnBlueEnableBit`	Set if HW should sync on blue.

-> `syncBitFieldValid`
This is a mask of the bits in `syncBitField` which are valid.

GraphicsHALSupportsHardwareCursor()

This call is used to determine if the HAL supports a hardware cursor.

```
GDXErr GraphicsHALSupportsHardwareCursor(Boolean *supportsHardwareCursor)
```

<- `supportsHardwareCursor`
`true` if HAL supports a hardware cursor, `false` otherwise.

GraphicsHALPrivateControl()

This routine accepts private control calls, or control calls which the Core does not process. If the HAL knows what to do with the `privateControlCode`, it should deal with it accordingly.

```
OSErr GraphicsHALPrivateControl(void *genericPtr,
            SInt16 privateControlCode)
```

-> `genericPtr`
Points to the data structure that the HAL needs for this control call. Should be cast to appropriate data type if internal routine is invoked.

-> `privateControlCode`
The private `csCode` that the HAL might know what to do with.

GraphicsHALPrivateStatus()

This routine accepts private status calls, or status calls which the Core does not process. If the HAL knows what to do with the `privateStatusCode`, it should deal with it accordingly.

```
OSErr GraphicsHALPrivateStatus(void *genericPtr,
            SInt16 privateStatusCode)
```

-> `genericPtr`
Points to the data structure that the HAL needs for this control call. Should be cast to appropriate data type if internal routine is invoked.

-> `privateStatusCode`
The private `csCode` that the HAL might know what to do with.

Changing the Core

Though GDX is designed so that no changes to the Core or OSS are required when developing a HAL, here are some instances in which changes might be required:

Gamma Tables with More than 8 Bits
The Core only support gamma tables which have 8 bits or less of correction data per entry. It does not support 16 or 12 bit gamma tables. In the event that support for more than 8 bits is desired, the following changes will be required:

GraphicsCoreControl.c

GraphicsCoreSetGamma()
Remove the check for 8 bits or less.

Change the calculation of the tableSize from:

```
tableSize = sizeof(GammaTbl)                        // fixed size header
+ clientGamma->gFormulaSize                         // add formula size
+ clientGamma->gChanCnt * clientGamma->gDataCnt    // assume 1 byte/entry
- 2;                                          // correct gFormulaData[0] counted twice
```

to:

```
dataSize = (clientGamma->gDataWidth + 7) DIV 8;

tableSize = sizeof(GammaTbl)                        // fixed size header
+ clientGamma->gFormulaSize                         // add formula size
+ clientGamma->gChanCnt * clientGamma->gDataCnt * dataSize
- 2;                                  // correct gFormulaData[0] counted twice
```

Change the copying of the correction data from:

```
for (entryLoop = 0 ; entryLoop < gammaTable->gDataCnt ; entryLoop++)
       *newData++ = *clientData++;
```

to:

```
for (entryLoop = 0 ; entryLoop < (gammaTable->gDataCnt * dataSize) ; entryLoop++)
       *newData++ = *clientData++;
```

GraphicsCoreUtils.c

GraphicsUtilSetEntries()
This is where gamma correction is applied to the values about to be written to the CLUT. The changes required here are not quite as straight forward as above, but can be stated succinctly: allow 1 or 2 bytes of correction data per entry.

HAL Interface
In `GraphicsHALSetCLUT()`, which accepts a gamma corrected `ColorSpec` table, a parameter will have to be added to specify how many bits where used for correction. This will allow the HAL to chose the correct bits from each 16 bit `RGBColor`.

Adding New DisplayModeIDs

If the existing `DisplayModeID`s defined in `GraphicsPriv.h` are not sufficient to describe the rasters your hardware can produce, then you will need to define additional ones.

GraphicsPriv.h

Add the new `DisplayModeID` to the end of the existing enumeration.

GraphicsCore.c

GraphicsCoreInitPrivateData()
Add the new `DisplayModeID` to the `localTable`, describing its resolution and scan rate.

GraphicsCoreStatus.c

GraphicsCoreGetModeTiming()
Add the new `DisplayModeID` to the timingModeTable, describing its timing data.

Supporting Less Than 256 Colors

As computers get faster and faster, the burden of supporting high pixel depths has lessened. However, from a programing standpoint, the effort to support lower pixel depths requires as much programing (and testing) as supporting higher pixel depths. GDX only supports 8, 16, and 32 bits per pixel. However, the modification to support lesser depths is quite simple:

GraphicsCoreStatus.c

GraphicsCoreGetVideoParams()
Add the appropriate information in the switch statement for 1, 2, and 4 bits per pixel.

Release Notes

This is the change history of GDX.

Version 1.0
May 23, 1995. Initial release.

Version 1.0.1
July 17, 1995. This is an incremental release incorporating minor updates.

New Features and Enhancements

The following items are incorporated into GDX 1.0.1:

Default Gamma Applied at Start-up
Previously, a linear gamma table was applied to the hardware during the early stages of booting. Now, a default gamma table is applied, based on the type of display connected.

Better Support for Hardware Which Doesn't Generate Interrupts
A small number of graphics devices may not actually generate hardware VBL interrupts. If so, the HAL can simulate a VBL by using an interrupt timer[2]. When the timer goes off, the HAL can call `GraphicsOSSDoVSLInterruptService()` to allow the OS to service items in its VBL task queue.

Files Changed

This is the list of the items that were changed for the 1.0.1 releases:

GraphicsCore.c
In `GraphicsOpen()`, a default gamma table is applied instead of a linear one. Also, some slight changes where made to provide better support for replacing drivers that don't fully support the `kReplace/kSuperseded` commands.

GraphicsCoreControl.c
In `GraphicsCoreSetSync()`, reflect the fact that `syncBitFieldValid` is now an input only when calling `GraphicsHALSetSync()`.

GraphicsCoreStatus.c
In `GraphicsCoreGetConnection()`, now reporting new constants for fixed frequency 16", 19" and 21" color displays.

GraphicsCoreUtilities.c & GraphicsCoreUtilities.h
Added a new function `GraphicsUtilGetDefaultGammaTableID()`.

GraphicsOSS.c & GraphicsOSS.h
Added a new function `GraphicsOSSDoVSLInterruptService()`.

2. *Designing PCI Cards and Drivers for Power Macintosh Computers*, page 272.

GraphicsHALTemplate.c
In `GraphicsHALInitPrivateData()`, added a detailed comment about the nuances of replacing a driver.

In `GraphicsHALGetModeTiming()`, now setting `timingFormat = kDeclROMTables` prior to any error checking.

In `GraphicsHALSetSync()`, update to reflect that `syncBitFieldValid` is now only an input.

GraphicsHALTemplateEZ.c
Throughout this file, the types `long`, `unsigned long`, `short`, `unsigned short`, etc. were replaced with `SInt32`, `UInt32`, `SInt16`, `UInt16`, etc.

In `GraphicsHALInitPrivateData()`, added a detailed comment about the nuances of replacing a driver.

In `GraphicsHALGetModeTiming()`, now setting `timingFormat = kDeclROMTables` prior to any error checking.

In `GraphicsHALSetSync()`, update to reflect that `syncBitFieldValid` is now only an input. Also, `syncBitFieldValid` is examined more closely to check for error conditions.

In GraphicsHALGetVideoParams(), fixed a typo for the case of 32 bits per pixel in which ‘`<< 1′` was typed instead of ‘`<< 2`’.