White Paper

Ragav Gopalan

Platform Applications Engineer

Intel Corporation

Inside Linux* graphics

Understanding the components, upgrading drivers, and advanced use cases

April 2011

Executive Summary

The Linux* graphics software stack is a complex system that comprises several open source components. Like any open source project, there is a tight dependency between these components which results in version requirements. For instance, changes to the Xorg server may necessitate a change to the Mesa 3D driver. This introduces a dependency between the two components. Users new to Linux are often perplexed by the complexity of building and upgrading the drivers. It is important to appreciate the complexity of the system and process as a whole to get a better perspective on the ease of dealing with Linux graphics.

Users new to Linux are often perplexed by the complexity of building and upgrading the drivers. It is important to appreciate the complexity of the system and process as a whole to get a better perspective on the ease of dealing with Linux graphics

This paper includes a brief introduction to the various components in the Linux graphics stack and an understanding of how they interact. This paper also talks about some use cases that are relevant in the embedded space.

The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. www.intel.com/embedded/edc.

Contents

Figures

Open source driver components

X server

X server (hence forth referred to simply as X) is a software system that provides services for creating graphical user interfaces (GUI). A client is any program, running on the network or locally, that needs rendering services. The X server was designed when shared libraries did not exist on UNIX systems. To reduce the memory footprint of multiple clients using the same library, a lot of graphics code was put in to the X server. X was also designed to serve clients over the network. For these reasons, much of the communication protocols within X are based on network client-server technology. X provides ways to manage windows, displays and input devices for multiple clients. X provides the basic framework for building GUIs. X does not impose the look and feel of a desktop. That is the function of desktop managers. Desktop managers can be thought of as sophisticated X clients that may use other X libraries and clients. It is for this reason that Linux based installations from different distributions like Fedora and Ubuntu vary in look and feel. X has served its purpose well for 2D applications. But X is not really efficient in handling 3D clients. 3D clients typically pass a lot of graphics modeling information to the hardware and X offered a latency ridden path. Direct Rendering Infrastructure (DRI) was introduced to circumvent X and provide a faster access path for applications to get to the hardware. This fast path is not based on networking protocol. DRI is not a single piece of software, but rather a specification implemented by a collection of modules.

3D driver

OpenGL is the API of choice for developing 3D applications on Linux* systems. OpenGL is a specification maintained by the Khronos group. Mesa is an open source project that provides an implementation of the OpenGL specification. Hardware vendors may choose to provide their own implementation of OpenGL. Mesa, like many other open source projects, is a community developed project. Intel, as well as many other companies and individuals contribute to this project. The OpenGL implementation in the Mesa project is often referred to as libGL. On Linux systems, libGL is built as a shared library - libGL.so. libGL provides an API interface that is completely compatible with the OpenGL specification. libGL is agnostic of the actual hardware for the most part. libGL can be thought of as the front end for 3D

acceleration. The device specific driver converts libGL calls in to hardware specific commands and can be thought of as the Mesa back end for 3D acceleration. This driver is typically referred to as the DRI driver (DRI drivers are considered an integral part of the DRI architecture). The Mesa project maintains source code for many hardware specific driver implementations like Intel and ATI*. The Intel device specific driver gets built as i965_dri.so. Notice that libGL.so and i965_dri.so are user level shared libraries. The bulk of the graphics software stack resides in the user space. Critical functionalities like GPU memory management and DMA are pushed in to the kernel space. The Mesa project also provides a software implementation of 3D rendering. This is useful for platforms that do not have GPU devices or for debugging DRI drivers. This also provides a software based fall back mechanism for features missing in hardware.

Media driver

The Intel graphics driver supports hardware offloading of AVC and MPEG2 decode starting from the Westmere generation of processors. Architecturally, a video driver is no different than a DRI driver. It passes the encoded slices of bit streams to the GPU for decoding. It has direct access to the hardware just like the DRI drivers. But the video driver is fairly disjointed from the X server. For this reason, the video driver is portable to a non X based system. Typically a video player application allocates memory for storing decoded frames. This piece of memory is where the GPU stores the decoded frames. Subsequently, the application may choose to pass this buffer to the X subsystem for various operations not performed by the hardware like rotation. The application may also choose to render the decoded frame on an X surface. Alternatively, the video application may also choose to render to an OpenGL surface and apply texturing. Just like libGL is a high level API that exposes graphics functions, libVA (VA API) is an Intel backed standard API that exposes hardware offloading of media processing to the GPU. VA API is an open source project. There are several media players that utilize VA API – mplayer and RealPlayer to name a few. VA API has also found support in several media frameworks like FFMPEG and Gstreamer. Gstreamer provides a VA API plugin (gst-vaapi) that directly interfaces with VA API. Other GPU vendors like ATI and Nvidia have their proprietary standards for video processing (XVBA and VDPAU respectively). Interestingly, VA API provides backend implementations for XVBA and VDPAU. This enables an application written in VA API to run on an ATI or Nvidia GPU as well as an Intel GPU. At runtime VA API determines which backend to use. Intel video driver is typically part of the libva package. This

package produces the libva shared library (libva.so) and the Intel video driver (i965_video_drv.so).

Graphics Execution Manager – GEM & Libdrm

Graphics Execution Manager (GEM) is a kernel level functionality that manages GPU memory. The device specific driver implements the kernel level part of GEM and exposes APIs in the form of ioctls to user level API. That user level API is called libDRM. libDRM simply provides for a way for clients to request for buffers for rendering. Clients rarely manage their own buffers. The buffers are requested on behalf of clients by the X server.

2D driver

In modern Linux* graphics subsystems, the 2D driver can be thought of as a primarily a blitting engine. Once clients have rendered content ready for display, the 2D driver blits it for display. Historically 2D drivers also performed mode setting. But mode setting has since moved in to the kernel space. The 2D driver is also responsible for 2D rendering. The 2D driver is also known as the DDX – Device Specific X. Most of the X server system is hardware agnostic. There are few parts of X that are completely aware of the hardware. The DDX is one such module.

In Linux systems, 2D functionality is exposed via Xlib. Xlib hides the X specific protocols and exposes a simple interface for clients. But applications are rarely written directly to use the Xlib interface. Higher level toolkits such as Gtk+, Qt and Cairo provide the ability to draw objects and sit on top of Xlib. X encodes the rendering requests from Xlib and passes it to the 2D driver. The 2D driver takes generic 2D commands and sends them as device specific commands. On newer Intel platforms, this 2D driver is built as intel_drv.so. Note that the 2D driver resides in the user space just like the 3D driver. DDX drivers are typically part of xf86-video-<vendor> package, where <vendor> is typically the company that makes the hardware. In the case of Intel, this package is called xf86-video-intel.

Direct Rendering Infrastructure

3D clients typically pass a lot of vertex and texture information to the hardware. The X server was a bottleneck to achieving a low latency path from the client to the hardware because of the various protocols clients had to follow to talk to X server. The original DRI architecture allowed 3D clients to directly render on the frame buffer.

This architecture still required a lot of co-ordination between the 3D driver, X server and the 2D driver (also known as DDX driver). DRI2 was introduced that simplified the rendering scheme where clients could render to their private buffers and submit the buffers for presentation to the X server. DRI is obsolete. X server can be told via the Xorg config file to load either DRI or DRI2. In the absence of a config file, newer X servers always use DRI2 architecture. The inner details on the working of DRI/DRI2 are beyond the scope of this paper.

Kernel modules

There are several kernel level modules of interest to the graphics stack. The DRM module is a device independent kernel level module that implements several helper functions for the device dependent drivers. The device dependent driver is a kernel module that directly talks with the hardware and can be thought of as the final frontier before the commands are dispatched to the hardware. This driver is hardware specific. This driver is part of the kernel source tree and is built by default on Intel systems as i915.ko. The i915.ko driver also implements the GEM functionality at the kernel level. Libdrm implements the user level GEM interface as ioctls. This introduces a tight dependency between libDRM and i915.ko.

Driver distribution model

Linux* graphics is a fairly complex subsystem that is composed of several user and kernel level modules. Each module resides in its own repository. Developers up stream their code in to these repositories. At the end of every quarter, a recommendation is made on a collection of stable graphics components from the upstream repositories. This is the driver release package, as shown in Figure 1.

Figure 1: Driver release package
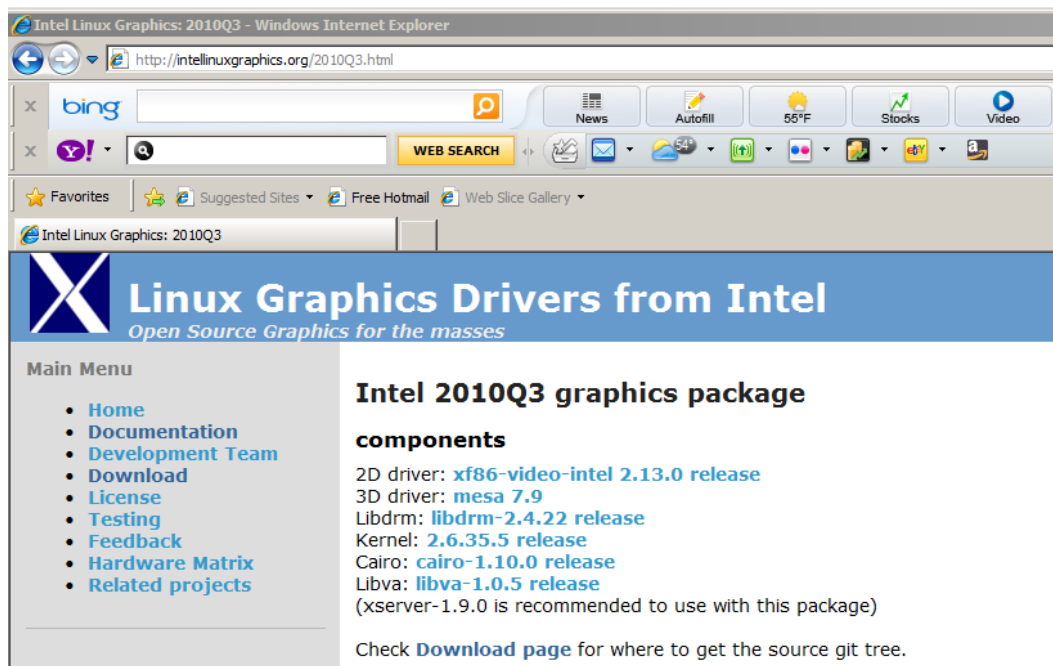
In order to upgrade the graphics driver, one has to upgrade all packages to the recommended versions. This may not be a problem for individual users. But upgrading components on an enterprise class distribution like Novel SUSE or Redhat is simply not an option.

Operating System Vendors (OSVs) make their own determination as to which driver package to use. Figure 2 shows component versions of the Intel, Fedora and Ubuntu distributions. As can be seen, Ubuntu deviated from the Intel suggested versions on its 10.04 release, but Fedora stuck with the recommendation.

Figure 2: Component versions of Intel, Fedora and Ubuntu distributions
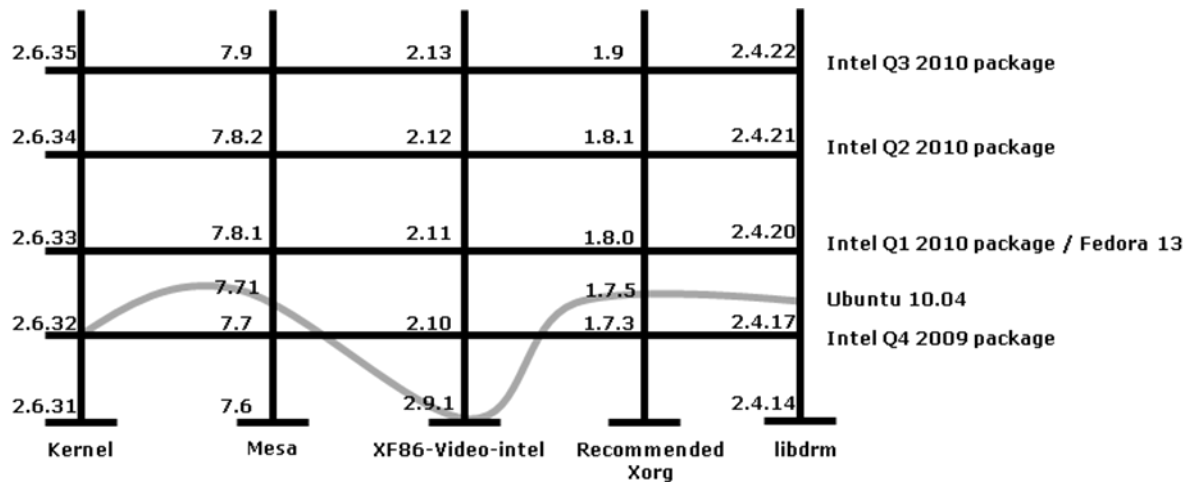
New hardware compatibility

Graphics support for a newer chipset/processor on an older distribution is a frequent source of confusion for customers. Let's say you locked down the production systems to run Fedora 13. Let's assume that support for your processor was introduced with the Intel Q3 2010 package. You are faced with the prospect of upgrading the kernel to 2.6.35. This may or may not break your other drivers and certainly introduces the need for additional validation. Upgrading user level components such as Mesa, XF86 and Xorg are not as risky as upgrading kernel. The other option is to back port the changes in kernel (if any) and device specific kernel modules to 2.6.33. If the distribution happens to be an enterprise version, a kernel upgrade could invalidate the support agreement. The choice in this case is to request for the OSV to back port the kernel and other components.

Tools for finding current versions of driver components

Determining what graphics features are available in a certain distribution is not straight forward. One option is to refer to the distribution's release notes. The other option is to examine the version number of the various graphics components and correlate that information to an Intel driver release package. Let's say we are using a Fedora* 12 system. We need to get version information on the kernel, Mesa, 2D driver, X Server and libdrm. The following commands are useful.

Kernel version – "uname –a"

```
[root@localhost Downloads]# uname -a
Linux localhost 2.6.31.5-127.fc12.i686.PAE #1 SMP Sat Nov 7 21:25:57 EST
2009 i686 i686 i386 GNU/Linux
```

libGL – Run glxinfo and look for "Mesa" version string

```
OpenGL vendor string: Mesa Project
OpenGL renderer string: Software Rasterizer
OpenGL version string: 2.1 Mesa 7.7-devel
OpenGL shading language version string: 1.20
OpenGL extensions:
```

2D driver – Check Xorg log at /var/log/Xorg.0.log messages from loading of "intel" module

```
(II) LoadModule: "intel"
(II) Loading /usr/lib/xorg/modules/drivers/intel_drv.so
(II) Module intel: vendor="X.Org Foundation"
        compiled for 1.7.0, module version = 2.9.1
        Module class: X.Org Video Driver
        ABI class: X.Org Video Driver, version 6.0
```

X server – Look for "X.org X Server" in /var/log/Xorg.0.log

```
X.Org X Server 1.7.1
Release Date: 2009-10-23
X Protocol Version 11, Revision 0
Build Operating System:  2.6.18-164.2.1.el5
```

Libdrm – pkg-config –modversion libdrm

```
[root@localhost ~]# pkg-config --modversion libdrm
2.4.15
```

Summary of Fedora 12 graphics components

- Kernel - 2.6.31.5

- Mesa - 7.7

- 2D driver – 2.9.1

- Xorg server – 1.7.1

- Libdrm – 2.4.15

Based on the information in the download section of www.intellinuxgraphics.org, the Fedora 12 driver is fairly close to the 2009Q3 driver release from Intel. The release notes has information on the new features that went in to this release. This is a handy way of figuring out if a distribution will suit your development needs.

Precompiled binary location

On a typical free distribution such as Fedora or Ubuntu – the DRI drivers are located under /usr/lib/dri/. It is also important to realize that binary locations are typically determined by the distribution. Under this directory you will also find DRI drivers for other hardware vendors such as ATI* and Nvidia*. The Intel DRI driver is typically called i965_dri.so.

The 2D drivers (also sometimes referred to as Xorg or DDX drivers) are located under /usr/lib/xorg/modules/drivers.

Libdrm is located under /usr/lib/ as libdrm.so. You will also notice device specific DRM implementations such as libdrm_intel.so, libdrm_nouveau.so and libdrm_radeon.so. Recall that the device specific kernel module implements the GEM interface.

The Intel kernel module is part of the kernel source tree and is built by default as a module called i915 in /lib/modules/<uname>.

libGL is located under /usr/lib/ as libGL.so

Closed source drivers

Closed source drivers have proprietary implementation of libraries and drivers. The proprietary 3D library implements the OpenGL spec just like Mesa does. Since the proprietary driver implements the OpenGL specification, it gets built as libGL.so – a shared library that is identical in name with the Mesa library. This is a source of confusion and problems, since the open source and proprietary GL implementations have the same name. Most proprietary drivers will replace the previous installations of libGL with their own versions. Most proprietary drivers will also reinstall the libGL from Mesa during un-installation. 2D drivers do not have this predicament as they are built

with distinct names. Besides, the xorg.conf file can explicitly invoke the loading of a 2D driver by name.

Embedded use cases

Hybrid multi monitor

Hybrid Multi Monitor (HMM) is a form of multi-head display technique that involves operating integrated and discrete graphics cards simultaneously. The implications are that 2D and 3D rendering from both integrated and discrete graphics devices will have to be managed. For the purposes of this discussion let us assume that we are dealing with a platform that has an Intel® Core™ i5 processor with integrated graphics (Code named Arrandale or Clarksdale) and an ATI* 5440 discrete PCIe x16 graphics card on a PEG port. Assume we are running the 32 bit version of Ubuntu* 10.04. It would be a good idea to get the run level to "3" for easier debugging.

BIOS changes

The BIOS on certain Intel CRBs will disable the integrated graphics device when the PEG port is made the primary display in BIOS settings. It is possible to see this behavior on OEM boards as well, depending on the BIOS implementation. In this case, either the BIOS needs to be modified or the integrated graphics device needs to be set as the primary display device.

Using open source drivers

The simplest way of enabling Hybrid Multi Monitors on Linux is to use open source libraries and drivers. Mesa provides an implementation of OpenGL and device specific DRI drivers for common hardware such as Intel, ATI* and Nvidia*. There are two choices for bringing up the desktop in this scenario. In the first method, a single X session is started with multiple device and display definitions in the Xorg config file.

```
startx – config <multi_monitor_config> &
```

Running startx in background makes it easy to kill X sessions from the same virtual terminal. This is very handy when bringing up multi monitor session on a trial and error basis. If the session does not behave in the expected manner, it is easy to switch to the virtual console from where X was started and kill it with a "killall Xorg".

Here is a sample multi_monitor_config:

```
Section "Module"
    #Load your desired modules
EndSection

Section "InputDevice"
    # generated from default
    Identifier          "Keyboard0"
    Driver              "keyboard"
    # The generic X keyboard driver is loaded
EndSection

Section "InputDevice"
    # generated from default
    Identifier          "Mouse0"
    Driver              "mouse"
    Option              "Protocol" "auto"
    Option              "Device" "/dev/psaux"
    Option              "Emulate3Buttons" "no"
    Option              "ZAxisMapping" "4 5"
    # The generic X mouse driver is loaded
EndSection

Section "Monitor"
    Identifier          "Integrated_gfx_Monitor_1"
EndSection

Section "Monitor"
    Identifier          "Integrated_gfx_Monitor_2"
EndSection

Section "Monitor"
    Identifier          "Discrete_gfx_Monitor_1"
EndSection

Section "Monitor"
    Identifier          "Discrete_gfx_Monitor_2"
EndSection

Section "Device"
    Identifier          "Video Device1"
    Driver              "intel"
    BusID               "PCI:0:2:0"
    # Run lspci on your system to get the device IDs
EndSection

Section "Device"
    Identifier          "Video Device2"
    Driver              "radeon"
    BusID               "PCI:1:0:0"
    # Run lspci on your system to get the device IDs
EndSection

Section "Screen"
    Identifier          "Screen1"
    Device              "Video Device1"
```

```
    Monitor          "Integrated_gfx_Monitor_1"
EndSection

Section "Screen"
    Identifier       "Screen2"
    Device           "Video Device1"
    Monitor          "Integrated_gfx_Monitor_2"
EndSection

Section "Screen"
    Identifier       "Screen3"
    Device           "Video Device2"
    Monitor          "Discrete_gfx_Monitor_1"
EndSection

Section "Screen"
    Identifier       "Screen4"
    Device           "Video Device2"
    Monitor          "Discrete_gfx_Monitor_2"
EndSection

Section "ServerLayout"
    Identifier       "Default Layout"
    Screen           "Screen1" 0 0
    Screen           "Screen2" RightOf "Screen1"
    Screen           "Screen3" Below "Screen1"
    Screen           "Screen4" RightOf "Screen3"
    InputDevice      "Keyboard0" "CoreKeyboard"
    InputDevice      "Mouse0" "CorePointer"
EndSection
```

This should bring up two desktops – one each on the integrated and the discrete devices. The keyboard and mouse will be shared between the two monitors in this case. Each display adapter can be configured to display cloned or independent displays from the GNU display menu. Generically, startx is the script that is used to initialize a desktop. startx calls xinit. If a .xinitrc is present, then xinit launches the client application mentioned in the .xinitrc file. In the absence of a .xinitrcfile, the GNU desktop manager is launched. In use cases where 3D acceleration is desired on the integrated and discrete devices simultaneously, there can be performance implications for Mesa. This is quite evident when multiple instances of Glxgears are run on both adapters. Reducing the number of instances of Glxgears on one adapter will give a better performance on the other adapter.

Isolating 3D applications to the better performing adapter is a good performance balancing strategy.

Using proprietary drivers

In some cases, proprietary drivers for the discrete cards offer better performance than open source drivers. It is possible to have the proprietary discrete driver and open source integrated driver co-exist. The trick is to isolate the runtime libraries for these devices. In many cases, the proprietary discrete driver will wipe out the pre installed GL libraries. This means that the Mesa libraries that shipped with the system are no longer available. This implies that 3D acceleration is not possible on the integrated device. If this is a desirable outcome, a single X session can be initiated. The screens on the integrated device will fail to accelerate 3D, but 2D applications will continue to work. Alternatively, two different X sessions can be started at boot time – one for the integrated device and one for the discrete device.

Alternatively, the driver stack for the integrated device can be isolated to a non standard location such as /opt/my_integrated_graphics. This location will house all the necessary libraries: GL, Mesa, Xorg drivers and X itself. It is then trivial to start a separate X session as follows

/opt/my_integrated_graphics/<bin_path>/X –config <my_integrated_config>

In this scheme, 3D acceleration is possible on both cards. Although these are not tested methods, it is reasonable to expect a working model after some initial debug.

Challenges

Using open source drivers for both devices is the least challenging and most predictable way of bringing up hybrid multi monitors. This is because both drivers were compiled by the OSV using the same version of libraries.

Using a combination of open source and closed source drivers is challenging. The problem specifically is with using the supplied kernel module. An incompatibility with the running kernel would force the proprietary driver installer to prompt a manual recompile of the kernel module. In many cases, this manual recompile could fail, forcing users to change to a kernel that is known to work with the proprietary driver. This upgrade could make it unstable for the integrated device. Linux users are often frustrated by problems arising from incompatible kernel and library versions and versioning in general. Users should conduct a thorough analysis of the pros and cons of using a certain kernel and make an informed decision on the kernel choice.

Conclusion

The open source Linux graphics driver is a collection of several open source projects.

xf86-video-intel is the Device Dependent X (DDX) driver (intel_drv.so) that is responsible for 2D rendering. The DDX driver is mostly a blitter engine. Historically DDX drivers performed mode setting, but mode setting has been moved into the kernel.

The Mesa project includes the open source implementation of OpenGL specification (libGL.so) and device specific DRI driver (i965_dri.so).

Libdrm is the user interface to Graphics Execution Manager (GEM).

Libva project includes the open source implementation of VA API (libva.so) and device specific video driver (i965_video_drv.so).

i915.ko is the device specific kernel level module for Intel graphics devices.

The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. http://intel.com/embedded/edc.

Author

Ragav Gopalan is a Platform Applications Engineer with Intel's Embedded and Communications Group.

---

§