Developing Drivers and Extensions for XFree86-4.x

Dirk Hohndel
SuSE Linux AG
Nürnberg, Germany
hohndel@suse.de
Robin Cutshaw
Intercore
Atlanta, GA, USA
robin@intercore.com

Abstract

This paper gives an introduction to driver development for XFree86-4.x. After a quick analysis of the existing problems in the previous XFree86 design, it describes the module loading architecture and the key interfaces that a graphics hardware driver for XFree86 must support.

1 Introduction

XFree86 has been the standard implementation of the X Windows System [1] for PC Unix systems for quite a while now; The XFree86 Project [2] was started in 1992 and incorporated in 1994. From the very beginning XFree86 was an open source project (even though back then the term "open source" wasn't widely used), still the first supported platforms were proprietary commercial Unix implementations. Open source simply came as a natural consequence of extending the X Window System. And with the success of Linux, XFree86 became one of the most often used implementations of the X Window System and to some extend the new driving force behind the development of X11.

Having started soon after the official release of X11R5 in 1991, XFree86 was in its design and source layout based on the original X386 work by Thomas Rll which was donated into X11R5 by SGCS.

This had many implications for the fundamental assumptions upon which this design was based. One of the more important (and more devastating) ones was the fact that there was no real design document. While Jim Gettys et al. in [3] describe the fundamental design of X11, and while Elias Israel and Erik Fortune in [4] explain the fundamental X Server design, there is no document that defines the design behind X386 and, subsequently, behind XFree86 versions 1 through 3.

The device dependent X (ddx) code that handles the programming of the hardware was always an area where companies tried to maintain some intellectual property outside the (open source) sample implementation of X11. This was true both for companies like SGCS as well as for the traditional Unix vendors. Therefore, little progress in the ddx code became part of the sample implementation

While X11R6 changed some of the source layout and some of the structure of the device independent X (dix) code, the design of the ddx was largely untouched. It was based on late-80s / early-90s design decisions that, to make matters worse, were not documented.

So in oder to develop drivers for the older versions of XFree86 one had to sit down and read the code. And we are talking about quite a large amount of code. The xfree86 ddx of XFree86-3.2, for example, contains about 1000 source files and roughly 350000 lines of code. For version 3.3.6 this increased to about 1400 source files with about 520000 lines of code.

Early on David Wexelblat, one of the founders of XFree86 and a member of the XFree86 Core Team provided a sceleton driver [5], but even using that it was still necessary to understand very subtle interdependencies and many assumptions that the overall code made about the type of hardware used, in order to successfully write new code for XFree86.

One of the more involved assumptions was that the graphics board in a PC would be VGA compatible. The code loading color maps, for example, relied on the fact that the default way to do so on a VGA board would work. This and other similar assumptions of course turned out to be problematic as new more advanced architectures for graphics card started to become available.

But aside from the technical details of the driver implementation, there were some logistical consequences to the design as well. The hardware drivers were part of the X server binary. This monolithic architecture of XFree86 before version 4 forced (in most cases, there were exceptions for some types of extensions and for Xinput drivers) the release of a complete server binary if a new driver (or an updated version of a driver) was to be released. At first glance this doesn't sound so bad. But the logistical problem stems from the fact that XFree86 supports more than a dozen OSs, including Linux, FreeBSD, NetBSD, OpenBSD, Solaris, SCO, QNX and even OS/2, to name a few. Several of these are supported on multiple hardware architectures.

Releasing even a simple driver therefore implied to get hold of all these platforms (or at least the more popular ones), to create server binaries and installation instructions, and to provide those to the users. And of course there's the added complexity if more that one group is working on improvements (so if you want the new extension from group A, e.g., VMware's new DGA version, and the new driver from group B, e.g. SuSE's new Rage128 driver, then you had to rely on these two groups talking to each other and importing each other's new features).

2 Module Loading Architecture

With the advent of XFree86 version 4 some fundamental assumptions about extending its functionality thankfully have changed. Metro Link has donated a module loading architecture that XFree86 has enhanced and extended so that now there exists a portable architecture to load object files (and even ar libraries) at run time into the address space of a process.

Several diffent object formats are supported in this architecture. While a `dlopen()` based loader is included, implementation issues with unresolved symbols in modules make it very hard to use. The obvious advantage of a `dlopen()` based loader (full support in system debuggers) contrasts the down side of implementation limitations and the non- portability of the resulting modules.

The most often used file format are ELF objects as created, e.g., on a Linux system. Additionally, a.out and COFF are supported as well. The host operating system and its preferred object file format have no influence on the selection, the loader code is fully selfcontained and needs no further support from the host OS beyond standard libc interfaces.

The server binary (which obviously has to be in the native executable file format of the host OS) can simply load modules at run time by calling

```
LoadModule(ModuleName, Path, SubDirs, Pattern,
           OptionList, ModReq, &errmaj, &errmin)
```

This allows the server to load the given module, finding it according to the rules given in Pattern in the SubDirs of Path. The options given in the OptionList are passed to the setup function of the module when the module is first initialized. The loader framework can ensure that the required ABI versions given in ModReq are met.

Which modules are loaded at run time is defined through compiled in defaults (e.g., a bitmap font renderer is mandatory to be loaded), through the XF86Config file and through dependencies among the modules. The deferred symbol resolution strategy allows to load modules that have dangling references to other modules that will be loaded at a later point in time. Additionally, modules can reference symbols that the main server binary exports to them.

When a module is first loaded, before all symbol references are fulfilled, the loader code searches for a data element named ¡ModuleName¿ModuleData that contains the module version information (including ABI versions that the module supports) and references to a setup and teardown function. In the next step the setup function is called with the options provided during the call to `LoadModule()`.

At this phase only the symbols that are exported from the main server binary and all in-module references are resolved. The setup function therefore needs to avoid calling functions that are provided through other modules.

Once all modules are loaded, the remaining symbols in the modules are resolved and the main program can match addresses to symbols by looking them up with FunctionPtr = LoaderSymbol(FunctionName).

3 Logistical considerations

This modular loading architecture allows to provide just the modified/new driver (or extension) module instead of the full X server. The server can load this new module at run time and utilize its new features. The architecture provides an additional simplification. Since the loader code does not rely on the host OS in order to read and interpret object files, all the OSs on the same hardware architecture can share the same type of modules as well. Releasing a driver for a new card now is as easy as releasing a single ".o" file for the x86 architecture, and maybe the same for other hardware platforms that the device can be used on.

The installation of this new driver implies simply to copy it to a well known location (e.g., /usr/X11R6/lib/modules/drivers for graphics hardware drivers) and, if this is a new driver, updating the XF86Config file accordingly.

The logistical problem now boils down to issues of ABI versions in the loader code, the need for exported symbols from the main server binary or maybe the host system libc implementation, and of course the issue of authentication of modules. Let's quickly go through these one by one.

ABI version The application binary interface that defines the loader architecture in XFree86 has a versioning scheme that allows (through major.minor numbering) compatible and incompatible changes to different parts of the ABI. By defining the ABI version in the XF86ModuleVersionInfo of a module the server can determine if it is able to load and execute the module.

Since different classes of modules have different ABI classes it is possible to change the ABI for one subset of the server without having to modify unrelated modules. For example, video drivers and font renderers have different ABI classes.

exported symbols The XFree86 ANSI C Emulation ABI defines a set of entry points and variables from the main server binary that can be referenced from the module. This includes many of the ANSI libc functions that are provided to the modules by means of special wrapper functions that allow portable access to host libc routines. This is necessary to make driver modules independent from special versions of libc and to make modules protable between different OSs for the same architecture.

authentication of modules Modules contain information on the manufacturer and the version of XFree86 against which they were built. This is additional information for the end user only, the relevant ABI versioning does not use this information. In order to be able to determine whether a module has been tampered with, authentication code that gives a cryptographic signature of the module (which can then be compared against a vendor-provided list of valid signatures) will be added.

At this point the code necessary to do this has not been fleshed out in detail.

4 Writing a Driver

What remains is the technical problem of developing drivers and extensions for XFree86. This still is a very dificult task, but that as well has become easier. After unpacking the XFree86 sources a detailed design document can be found at `xc/programs/Xserver/hw/xfree86/doc/DESIGN` This document covers the important details of the new server design and provides step by step analysis of the flow of control in the server, the mandatory interfaces that a driver needs to provide and the optional driver functions. Furthermore the data structures used in the drivers are defined and explained, handling of bus resources and the helper functions that the server provides for commonly needed tasks in a driver.

While all the details are in this design document, a quick overview of the necessary parts shall be given here.

- a ModuleData variable as described above that contains the necessary information to load and initialize the module

```
static MODULESETUPPROTO(zzzSetup);

XF86ModuleData zzzModuleData =
{ &zzzVersRec, zzzSetup, NULL };
```

a sample version info would look like this

```
static XF86ModuleVersionInfo zzzVersRec =
{
 "zzz",
 MODULEVENDORSTRING,
 MODINFOSTRING1,
 MODINFOSTRING2,
 XF86_VERSION_CURRENT,
 ZZZ_MAJOR_VERSION,
 ZZZ_MINOR_VERSION,
 ZZZ_PATCHLEVEL,
 ABI_CLASS_VIDEODRV,
 ABI_VIDEODRV_VERSION,
 MOD_CLASS_VIDEODRV,
 {0,0,0,0}
};
```

The MODULEVENDORSTRING would usually be i "The XFree86 Project" or the corresponding information for the vendor creating this module. The two MODINFOSTRINGs are magical values that identify allow to find the VersionInfo in a module (for example for use in a signing tool). Next comes the version of XFree86 that this module was built again and the version of the module itself. Then the ABI class and version as well as the module class. The final four integers are intended to hold the digital signature of the module.

- a `Setup()` function is needed to integrate the module into the XFree86 loader architecture. This function follows the ABI specification for loadable modules and makes the module functions and data available to the server.

```
static pointer
zzzSetup(pointer module, pointer opts,
         int *errmaj, int *errmin)
```

The `Setup()` function needs to call `xf86AddDriver()` in order to register the module as a driver.

- an `Identify()` function that prints out some identifying message is mandatory.

```
static void
ZZZIdentify(int flags)
{
 xf86PrintChipsets(ZZZ_NAME,
     "driver for ZZZ Tech chipsets",
     ZZZChipsets);
}
```

- a `Probe()` function is needed that locates the hardware that this driver supports (normally some type of graphics device) and registers this driver as driver for the resource. The Probe should be as unintrusive as possible. It must not modify any settings in the hardware and should not touch any hardware except the devices that are supported by this driver.

```
static Bool
ZZZProbe(DriverPtr drv, int flags)
```

for most current hardware the `Probe()` function can use information from the PCI data to verify if a supported card is available. A helper function `xf86MatchPciInstances()` is available to make matching PCI devices easier.

- a `PreInit()` function collects all the information that is necessary to determine the configuration of the hardware and to prepare the device for being used, without making any changes to the device at this stage. Again this function should be as unintrusive as possible.

```
static Bool
ZZZPreInit(ScrnInfoPtr pScrn, int flags)
```

Information that is filled in here includes things obtained from the XF86Config file (e.g., the Monitor info, the amount of video memory, if it was given in the config file), defaults for color depth and bits per pixel (bpp), visual, gamma correction, etc.

Additional, non-intrusive probing of hardware is done for that type of information that has not been given in the config file (e.g., exact type of chipset, amount of video memory, etc.).

Next the video modes given in the config file are validated against the restrictions that the hardware puts on video modes.

Finally, this function loads the submodules necessary for driving the hardware (e.g., the vgahw module for VGA compatible cards) and the framebuffer code.

- `Save()` and `Restore()` functions provide a means to save and restore the complete video state of the device. While these functions are not mandatory they are a very useful abstraction of a commonly needed task.

  ```
  static void
  ZZZSave(ScrnInfoPtr pScrn)

  static void
  ZZZRestore(ScrnInfoPtr pScrn)
  ```

- a `ModeInit()` function is used to setup a new video mode. This is again a recommended function but not part of the mandatory interfaces.

  ```
  static Bool
  ZZZModeInit(ScrnInfoPtr pScrn,
              DisplayModePtr mode)
  ```

- a `ScreenInit()` function is needed to hook the driver into the server and to setup the initial video mode (using `ModeInit()`). This is (in the flow of control) the first function that should modify the device.

  ```
  static Bool
  ZZZScreenInit(int scrnIndex,
                ScreenPtr pScreen,
                int argc,
                char **argv)
  ```

  This functions need to initialize the device independent parts of the X server as well. This includes informing setting up the visual type (`miSetVisualTypes()`), initializing the framebuffer (by calling the framebuffer `ScreenInit()` function, e.g. `cfbScreenInit()`), creating the initial color map, etc.

- `EnterVT()` and `LeaveVT()` functions are needed to allow switching to and from the X server.

  ```
  static Bool
  ZZZEnterVT(int scrnIndex, int flags)

  static void
  ZZZLeaveVT(int scrnIndex, int flags)
  ```

- a `SaveScreen()` function is mandatory to blank the screen.

  ```
  static Bool
  ZZZSaveScreen(ScreenPtr pScreen,
                Bool unblank)
  ```

- a `CloseScreen()` function is used to reset the device to its original state.

  ```
  static Bool
  ZZZCloseScreen(int scrnIndex,
                 ScreenPtr pScreen)
  ```

Once these functons are implemented and hooked together, a non-accelerated driver for the device is available and can be tested. Adding accelleration to this driver (using the XFree86 Acceleration Architecture XAA) is relatively straight forward and again well documented in the XAA documentation which can be found at `xc/programs/Xserver/hw/xfree86/xaa/XAA.HOWTO`

Summary and Conclusion

In summary, on the one hand, the documentation of the new ddx (devive dependent X) design in XFree86 has significantly improved over the previously available material. On the other hand, the structure and layering of the server itself has become much more straight forward. Many of the previously existing SVGA-isms have been removed, many other implicit assumptions about the design of the bus and the existence of one single graphics card are gone as well.

Following the design document it is fairly straight forward to implement a multi head capable driver for almost any type of graphics device.

The DDK

The next major step forward will be the advent of a real DDK, a real driver development kit. As of this writing, this is not finished, since we need to release 4.0 before we can focus on the DDK. But the design of XFree86 4.0 is finished, the initial release will happen before this paper is going to print, and in summer 2000 a release of a full DDK is planned.

So while this paper will have to skip the description of the DDK, I should be able to talk about this during my presentation at the conference.

5 Availability

You can get the XFree86 sources from our web site `http://www.XFree86.Org`.

References

[1] `http://www.x.org`

[2] `http://www.xfree86.org`

[3] Jim Gettys, Philip L. Karlton, Scott McGregor, *The X Window System Version 11*, DEC CRL 90/08.

[4] Elias Israel, Erik Fortune, *The X Window System Server*, Digital Press.

[5] *in the XFree86-3.x sources:* `xc/programs/Xserver/hw/xfree86/VGADriverDoc`