A Comparative Study of Device Driver APIs Towards a Uniform Linux Approach

Wadih Zaatar and Iyad Ouaiss
Lebanese American University
Byblos, Lebanon
iyad.ouaiss@lau.edu.lb

Abstract

Linux Application Program Interfaces (APIs) lack stability and standardization. There is a need for a standard API for Linux device drivers that allow backward compatibility while easing the development of new drivers. The advantage of standardizing the API is to make the kernel core more robust and the development of new drivers easier; however the main challenge is performance-based. This work starts by carefully studying the available APIs for Linux as well as for other platforms. Current solutions studied include the Uniform Driver Interface (UDI), the Intelligent I/O architecture (I2O), WinDriver, and APIs implemented in Solaris, and Windows XP. By listing the strengths and weaknesses of available APIs, a proposal for a new Linux API is constructed that defines a standard interface, provides backward compatibility, ensures kernel security, and handles errors, uniform block sizes, buffering, etc.

1 Introduction

Device driver implementations have always been an important field of study in the world of operating systems from proprietary models of companies such as Sun and Microsoft to providers of open source technologies such as RedHat, Mandrake, Suse, essentially targeted towards the Linux platform or NewBus toward the Unix platform [OpSys] [LinDriv] [NTDriv] [DrivDes]. Other solutions proposed by some companies such as WinDriver, would allow you to get a shareware Graphical User Interface (GUI) to build your set of drivers with prewritten code to get you started. Recently, an interesting approach with a mixed hardware and software solution named I2O comes with another innovative concept that would be explained later in Section 3.1. Therefore, we can see that there exists a multitude of different solutions for the problem, each with its own set of advantages and disadvantages.

This paper does not pretend to give a final solution to the problem; it simply tries to classify all the currently available models into categories and proposes a draft of a work matrix for an improved device driver interface. But proposing a solution requires a small introduction on how device drivers operate in general and Linux in particular:

A device driver is essentially a kernel component, but is developed independently form the rest of the kernel. Therefore, there should be some kind of interfacing service between the driver and the host operating system. A standard implementation would have two interfaces: The first one would communicate with the hardware itself, namely the Driver-Hardware interface and the second one would take care of communicating with the operating system and of course the user, called the Driver-Kernel interface. This is where the problem really resides: Due to several causes (architecture, OS and hardware differences), it is really rare to find a piece of software to be binary portable between two different machines, and thus nearly impossible to have some device driver (which is after all just a small specialized program) to run on different computers, running different operating systems. This is where layering and abstraction come handy: by analyzing all common components of devices, one could come up with a standard, platform-independent API that would group all repetitive system calls. This solution has two main advantages: unify device driver implementation, which will lead to faster development and better code reuse, but also would guarantee on the long run platform independence; and, in the case of API upgrade would allow old device drivers to benefit from the new API.

The following sections describe the currently available API implementations, starting with proprietary API with Microsoft and Sun, independent implementations with WinDriver and UDI, and finally, a new approach towards solving the problem with the I2O architecture that encompasses both hardware and software components. The final section will recapitulate all advantages and disadvantages of every API implementation, giving a skeleton for a work matrix and proposing a primary set of steps that will guide the development of a Linux device driver API that meets the requirements set forth in this paper.

2 Review of Software Approaches

The following section describes the different API implementations for the most widely used operating systems, namely Microsoft Windows and Linux/Unix main implementations alongside some third-party commercial and public license developers.

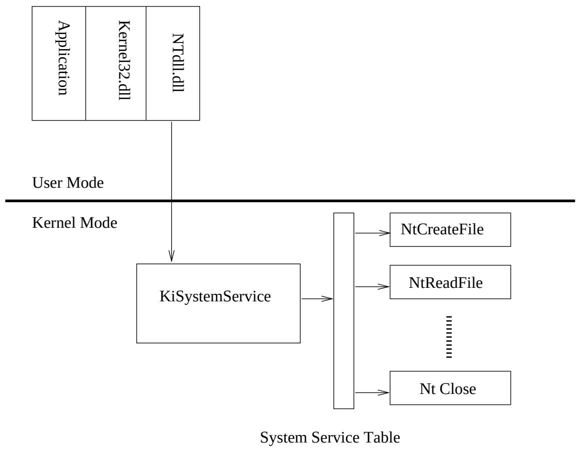

Figure 1: Windows API Model

2.1 Microsoft Windows API

Microsoft's device driver API [MSModel1] [MSModel2] were gradually enhanced with every new OS release, reaching a good level of stability due to two important factors: the first being software maturity starting from Windows 3.0 until the latest release of Windows XP based on NT technology, and second due to Microsoft's device driver compliance program that would take every device driver for any new hardware, run it and make sure that it is stable enough to be released with a "Certificate of Compliance" for it to be properly integrated with the latest OS release. This significantly reduced erratic OS crashes and restarts that made Windows platforms untrustworthy amongst the IT community. This model is depicted in Figure 1.

To be Microsoft-certified, a device driver has to have the following features:

- Handle I/O requests in standard format.

- Be object-based following the Windows model.
- Allows plug and play devices to be dynamically added or removed.
- Allow power management.
- Be configurable in terms of resources.
- Be multiprocessor code reentrant.
- Be portable across all Windows platforms.

From these major points, we can draw the following advantages:

- Windows drivers are portable between all platforms, as part of their requirements.
- Customizable as they are object based.
- Support new technology such as PnP and Power Management.

Are these really applied in reality? According to Microsoft themselves, their device model suffers the following:

- System instability: since they are run in kernel mode, and thus not isolated from one another or from the operating system, a failure in any device driver would result in system instability or a blue-screen.
- Little abstraction: the device driver interface is very low level and as such, there is little abstraction of the inner workings of the exported functions. This means that the device driver developer has to understand more about the workings to the interface than probably necessary.
- Plug and Play implementation: the PnP implementation is entirely set on the programmer's shoulders, requiring additional synchronizations and thus extra overhead.

2.2 Sun Solaris API

In System V Release 4 (SVR4), the interface between device drivers and the rest of the UNIX kernel has been standardized and completely documented [SunModel1] [SunModel2]. These interfaces are divided into the following subdivisions:

- The Device Driver Interface/Driver Kernel Interface (DDI/DKI) that includes architecture-independent interfaces supported on all implementations of System V Release 4 (SVR4).
- The Solaris DDI that includes architecture-independent interfaces specific to Solaris.
- The Solaris SPARC DDI that includes SPARC Instruction Set Architecture (ISA) interfaces specific to Solaris.
- The Solaris x86 DDI that includes x86 Instruction Set Architecture (ISA) interfaces specific to Solaris.
- The Device Kernel Interface (DKI) that includes DKI-only architecture-independent interfaces specific to SVR4. These interfaces may not be supported in future releases of System V.

The Solaris 2.x DDI/DKI allows platform-independent device drivers to be written for SunOS 5.x based machines. These drivers would allow third-party hardware and software to be more easily integrated into the OS. Furthermore, it is designed to be architecture independent and allow the same driver to work across a diverse set of machine architectures. The following main areas are addressed:

- Interrupt handling.

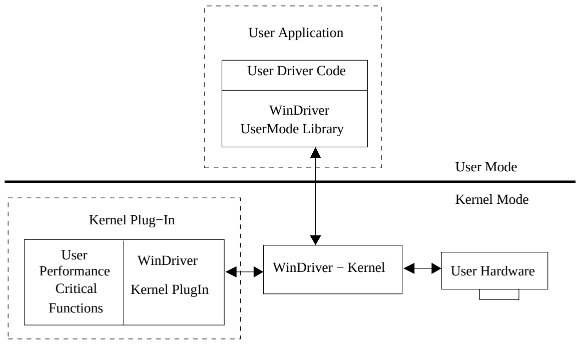

Figure 2: WinDriver Model

- Accessing the device space from the kernel or a user process (register mapping and memory mapping).
- Accessing kernel or user process space from the device (DMA services).
- Managing device properties.

2.3 WinDriver

The WinDriver API is a commercial, OS independent approach toward a common device driver API [WDModel]. WinDriver has the following important features:

- Source code compatibility between all supported operating systems: Windows, Linux, Solaris and VxWorks.
- Binary code compatibility between all Windows flavors (95, 98, Me, NT4, 2000, XP).
- Supports numerous architectures (PCI, E/ISA, and USB).
- Rapid device driver programming through the availability of many wizards. These wizards allow the device driver programmer, through a series of GUI-oriented steps to build the skeleton of the driver source code with initial procedure definition, global variables and program entry-points and basic calls.

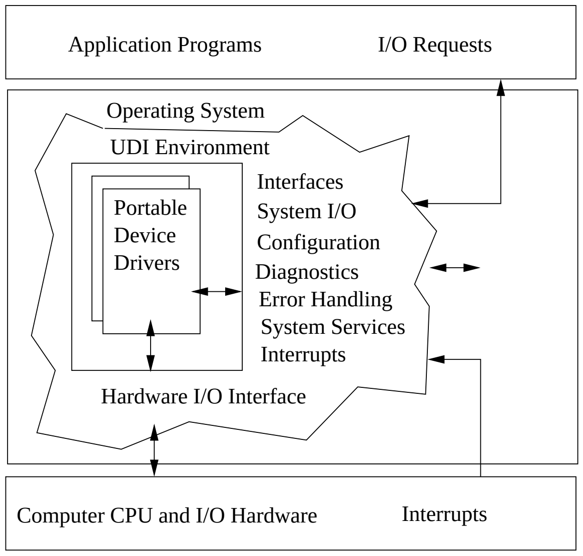

Figure 3: Uniform Device Driver Interface Model

From these features, we can deduce the following list of advantages:

- **C**ross platform compatibility and code reuse: No need for re-writing new drivers for the same hardware when porting to different architectures.
- **M**inimal performance hit: WinDriver offers a plug-in that would run performance critical parts right into the kernel, thereby achieving kernel mode performance.
- **E**asy drivers programming: WinDriver supports generic code generation in several programming languages, namely C/C++ and Delphi.

2.4 Uniform Device Driver Interface

The Uniform Device Driver Interface (UDI) is another initiative to create an architecture, platform, and OS independent solution for device drivers [UDIModel]; it is depicted in Figure 3. Similar to WinDriver, UDI has the following features:

- Platform neutrality, it abstracts all PIOs, DMAs, and interrupt handling through a set of interfaces that hide all architectures' variations.
- Drivers are written in ISO standard C and do not use any compiler specific extensions.
- UDI imposes shared memory restrictions, allowing system isolation of the driver code from the remainder of the OS, improving reliability and debuggability.
- Strict versioning allows evolution of the interfaces while preserving full binary compatibility of existing drivers.

The UDI device driver implementation is very much similar to WinDriver; therefore one could expect the following advantages:

- Cross platform compatibility, noting that UDI supports a narrower range of operating systems.
- Performance is comparable to native I/O drivers due to the fact that the code executes in kernel space, allowing minimal performance degradation.

3 Review of Mixed Approaches

The following section presents a new approach towards solving the device drivers' incompatibilities: By proposing a software and hardware components instead of a unique software approach.

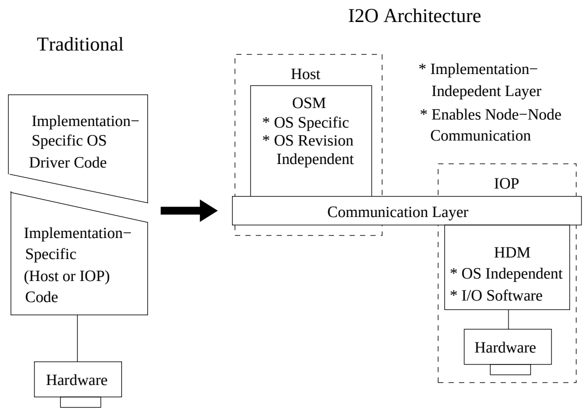
I2O Architecture
Traditional
Implementation−Specific OS Driver Code
Implementation−Specific (Host or IOP) Code
Hardware
Host
OSM
* OS Specific
* OS Revision Independent
* Implementation−Indepedent Layer
* Enables Node−Node Communication
IOP
Communication Layer
HDM
* OS Independent
* I/O Software
Hardware

Figure 4: I2O Architecture Model

3.1 The I2O Architecture

The I2O Architecture model is depicted in Figure 4.

Contrary to the standard implementation that relies on the host CPU to process all interrupt requests, the I2O eliminates processing bottlenecks by alleviating these tasks from the processor and taking care of them directly [I2OModel].

The I2O defines three software layers:

- The OS Services Module (OSM), that handles the communication between the host CPU operating system and device class.
- The I2O Messaging Layer that handles communication between the OSM and HDM in as standard way (see Figure). The standard I2O messaging layer does away the current requirement for OEMs to develop multiple drivers for the same device.
- The Hardware Device Module (HDM) handles the communication between the peripheral device and the I2O messaging

layer. The HDM is unique for each device. However, unlike a traditional device driver, only one HDM is required because it is independent of the host CPU operating system.

The I2O implementation offers the following improvements over traditional I/O processing:

- Standardized extensible architecture that is OS independent.
- Increased reliability and fault isolation to improve stability.
- Provides optimized I/O and system performance because the I/O process can be managed directly.

4 Analysis

Based on the previous sections, we can draw comparative measures that encompass all currently available solutions. The work matrix is shown in Table 1.

The following criteria were selected to distinguish between all available solutions:

- **Performance:** It is an essential component in any comparative study; however the measurement factor was much debated during the analysis of every solution. The number of interrupts per second that could be processed was finally selected. But due to the difficult nature of the task, this work is still taking an important amount of time and still not completed. One important note: The I2O group ceased operation and it was not possible to get access to any I2O based chips for testing.
- **Security:** This parameter describes the various security-related issues in every solution, memory-protection, and recoverability after a driver crash, etc. All four software solutions specify in their implementation the need for memory protection.
- **Compatibility:** Despite the Microsoft requirement that drivers should be upward compatible. Some tests performed on Microsoft Windows 2000 based drivers would erratically crash the system if installed on Windows XP, which would prove that these drivers still have compatibility issues. WinDriver and UDI drivers are still under test. The I2O alternative seems excellent on paper but for the same reason as described above, no tests were performed.
- **Portability:** WinDriver is the most versatile as it would run under most operating systems and is fully source compatible, a simple recompilation being enough. This would be ideal for multi-OS based solutions. UDI follows the lead with fewer but still promising implementations. The Windows and Solaris API models are proprietary and as such are not portable.

5 Conclusion

This paper represents the results of a data collection operation performed in order to analyze available device driver implementations for several operating systems. Many parameters still have to be exploited and carefully investigated. Most importantly, the performance factor that would gear towards choosing one solution over another is still being debated. With a better understanding of device driver APIs, the qualities and importance of each feature in the API and its role in the operating system can be analyzed. The next step in this research effort is to select a set of features that can be bundled together in order to form a compact, robust, and efficient Linux device driver API.

	Security	Compatibility	Portability	Notes
Windows API	Microsoft	Problems	Windows, Binary	Generic
Solaris API	Sun	Yes	SunOS Binary	Generic
Win Driver	To be tested	Yes	All, Source	Commercial
UDI	To be tested	Yes	Most, Source	Freeware
I2O	Hardware	Hardware	All, Independent	Discontinued

Table 1: Comparison Matrix

6 Acknowledgments

The authors would like to thank the students who participated in this work: Jamal Maalouf, Fady Matar, Naji Charbel, Francois Nader, and Elie Hajj.

References

[OpSys] Andrew S. Tanenbaum, *Modern Operating Systems*, 2nd Edition, Prentice Hall. (2001).

[LinDriv] Alessandro Rubim and Jonathan Corbet, *Linux Device Drivers*, 2nd Edition, OReilly. (2001).

[NTDriv] Edward N. Dekker and Joseph M. Newcomer, *Developing Windows NT 4.0 Device Drivers*, Addison Wesley Longman. (1998).

[DrivDes] *Introduction to Device Driver Design*. `http://www.itpapers.com/cgi/PSummaryIT.pl?paperid=9103&scid=264`

[MSModel1] *Windows Driver Model: Compatible Drivers for Microsoft Windows Operating Systems*. `http://www.itpapers.com/cgi/PSummaryIT.pl?paperid=20024&scid=273`

[MSModel2] *Windows Device Drivers Architecture*. `http://www.itpapers.com/cgi/PSummaryIT.pl?paperid=13052&scid=273`

[SunModel1] *Porting to Solaris 2.X, Sun Whitepaper*. `http://sunsite.nstu.nsk.su/sun/inform/whitepapers.html`

[SunModel2] *An Engineering Tutorial on Porting your Device Driver to Solaris 2.0, Sun Whitepaper*. `http://sunsite.nstu.nsk.su/sun/inform/whitepapers.html`

[WDModel] *Jungo Ltd.* `http://www.jungo.com/windriver.html`

[UDIModel] *Uniform Driver Interface, Official Specification Documents* `http://www.project-UDI.org/specs.html`

[I2OModel] Pauline Shulman, *Overview of the I2O Architecture*. PC Developer Conference. (1998).

Proceedings of the Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada