Modelling of Device Driver Software by Reflection of the Device Hardware Structure

Thomas Lehmann

HAW Hamburg, Berliner Tor 7, 20099 Hamburg, Germany
thomas.lehmann@haw-hamburg.de

Abstract. Embedded systems are highly optimised to operate in the physical world they are embedded to. Hence, dedicated peripheral devices are designed which need support by a device driver to raise the level of abstraction for the application programmer. Even with methods of hardware/software co-design, devices and drivers are still designed by two designer groups. This paper depicts a systematic approach to design the coarse grained structure of the device driver by reflection and mapping of the internal structure of the device hardware. Even though common operating systems are programmed in a functional programming language, means of object-oriented programming languages and design pattern are applied.

1   Introduction

Device drivers are adaptors between an application and peripheral devices integrated in an operating system (OS). They have to provide dedicated services towards the application (see Fig. 1) and hides away which services are performed in hardware and which in software, either by the driver or by helper functions of the operating system (see Fig. 2).

Embedded systems are highly optimised systems. Special hardware is designed to adapt to the physical world the device is embedded to. Hardware near programming is a complex task, because of various reasons. Hardware is parallel, software is sequential. Software design paradigms (such as layering[1]) are different from hardware designs paradigms (such as a hierarchy of components[2]). Low-level access to the device is operating systems and target dependent. The developer of a device driver struggles with various problems, next to the problem to find a straight-forward model for the inside of the device driver.
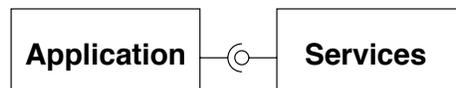
Fig. 1. Expected perspective of the application.
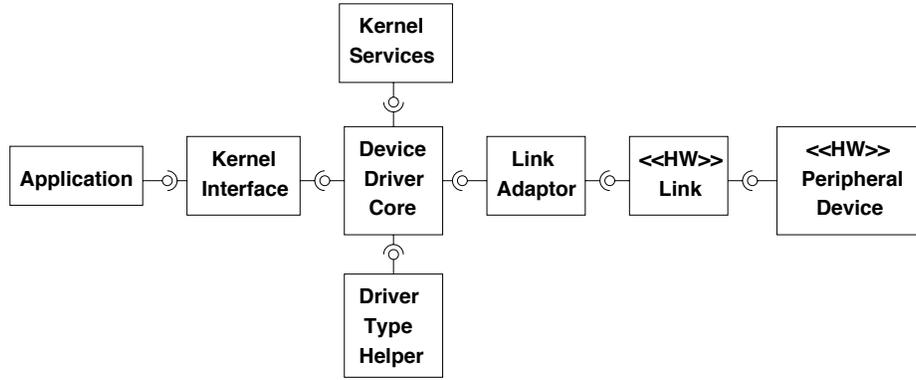
Fig. 2. Relation of application, device, driver, and OS-services.

A key question for the software side is, how to structure or to model the device driver to rapidly create a driver prototype? Following the idea of agile software development, this prototype doesn't need to have high performance. It has to demonstrate the combined features at an early stage of development.

A nowadays approach is to build the device driver as part of a hardware/software co-design. Here a model of the whole service functionality is the starting point and device and driver are generated. But still devices are designed by a hardware group and a software group needs to integrate the hardware by providing the device driver for integration. For instance, there are many devices available as IP-Core[3], but there is no corresponding device driver available. Especially, if the controlled system is a medical device, you will never find a standard device driver.

In this paper, an approach for modelling the device driver software based on an analysis of peripheral device structures is described. Hardware structures are identified and the corresponding software models are described. The paper starts with a discussion of other approaches. The approach shown in this paper starts with a description of the main idea and than shows how to reflect and map hardware interfaces and components to corresponding software interfaces and components. The modelling ends with an outline of other useful software structures in this field. The paper ends with a summary and an outlook.

2   Other Approaches

Even though device drivers are 'normal' software, less guidance can be found to model software in the field of device drivers.

Books on this topic deal on the one hand side with a very general view on device drivers. They cover operating systems in general (such as [1]) and hence can give only a rough idea how common device drivers, such as file system drivers, are designed. The second category of books deals with drivers in the environment

of dedicated operating systems like Linux[4] or Microsoft Windows[5]. These books focus on the interface (function calls) towards the application, the services the operating system provides for device drivers, and how to coarse grained structure the driver to fit into the operating system kernel. In both cases, the device developer can get a rough idea on how to structure the inside of the device driver. But still no guideline on how to analyse the device and how to structure the inside of a device driver based on the analysis is given.

Research papers focus on operating systems and special features like distribution, configuration or real-time scheduling. Some with separation into User Mode Driver and Kernel Mode Drivers[6] and the refactoring[7] towards such a separation. Other publications deal with a design flow support for device driver[8] or with the synthesis of interface between components[9], following the idea of hardware/software co-design. One group has derived a domain specific language (DSL) for a dedicated class of device drivers by analysis of a set of existing device drivers[10]. So for that class of devices a device driver can be generated, but on an extension of the features, the DSL will lack statements to support that new feature, because it has been constructed by reverse engineering.

In summary, most literature on device drivers is on how it is and only few hidden hints or guidelines are given on how to model device driver software for a dedicated device.

3   Approach of Structural Reflection

Nowadays operating systems, such as Linux, Microsoft Windows, or VxWorks, follow a layering paradigm as internal structure and use a non-object-oriented programming language like C. Other non-common operating systems already make use of the object-oriented paradigm, such as BeOS[11], DReAMs[12] or JNode[13].

Along with the object-oriented paradigm come ideas like Model Driven Design[14], Design Pattern[15], Aspect Oriented Programming[16], and Domain Specific Languages[10] based on the object-oriented concepts. The object-oriented paradigm is a powerful paradigm which is established in application programming and only on the rising in the field of operating systems, due to some drawbacks. Nevertheless, a modelling or structuring approach for device driver software should follow the object-oriented paradigm. An object-oriented model with appropriate restrictions can translated to a non-object-oriented paradigm for the use in common operating systems.

Object-oriented software is a structure of interacting components or objects with a strong separation of concerns. On the hardware side, digital devices are a graph of interacting digital components which are hierarchically organised. Usually, no layering approach is used to structure hardware whereas it is a common architectural pattern in software.

The approach to structure the internal software of the device driver is to map structures and interface identified in hardware to corresponding structures and interfaces in software, which follow the object-oriented paradigm. The graph of

interacting hardware components has to be identified and mapped to a graph of interacting objects on the software side. Interfaces between hardware components have to be mapped to appropriate interfaces in software.
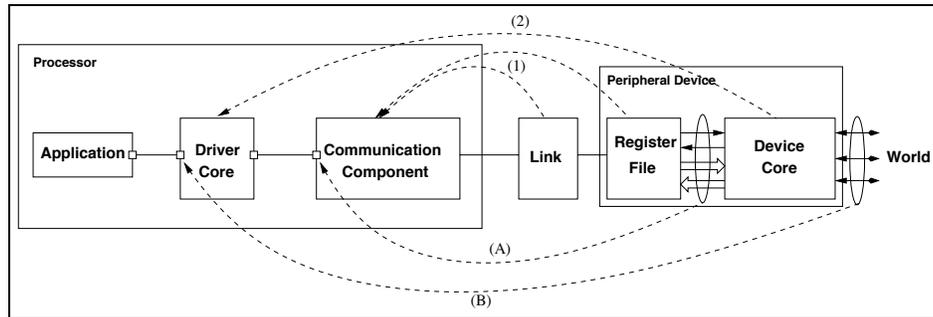
Fig. 3. Reflection of interfaces (A and B) and components (1 and 2).

So on a coarse grained level,

- register file and link maps to communication component (see Fig. 3,1),
- interface between register file and device core maps to interface between driver core and communication component (see Fig. 3,A),
- device core to driver core (see Fig. 3,2), and
- interface to the outside world maps to API of driver, extended by whole service functionality (see Fig. 3,B).

The next subsection will describe this mapping in more detail.

3.1 Register File and Communication Component

The register file is the interface between the device core and the communication link to the processor. A corresponding part on the software side, the communication component, has to hide away this interface and provide the driver core a transparent access to the device core. The objective of the communication component is to cover the communication topology between processor and device even if other devices are involved in the communication, such as serial links like serial peripheral interface (SPI).

The communication component needs to cover the transport via the communication link of various technology (e.g. bus or serial link) and topology (e.g. bus line or star with hubs). If the register file is accessible in a memory space manner, direct communication channels can be provided here. If other devices are involved on the path to the register file, the querries for the register file need to be encoded/decoded for the whole path. Thus the communication component works as an adaptor to device drivers which is responsible to handle the communication via the communication link to the device register file. Along with the

access means, management components needs to be integrated. In case the communication link has the ability for hot-plugging, the communication link needs to be monitored and in the case of an interruption the deriver core needs to be informed on the event.

The communication component is either realised by a single object or a set of objects, whereas one object is a facade class[15] with an appropriate interface. Depending on information exchange characteristics between the register file and the device core, additional elements need to be added, as depict in the next section.

3.2 Interface of the Communication Component

The communication component has to provide an interface towards the driver core and adapt to the interface of the communication link.

On the hardware side, via the register file signals to the device core can be set, states of signals can be read, or data is transferred. This interface is reflected to the interface between the communication component and the driver core (see Fig. 3). The provided functionality via this interface is to manipulate the signals and to query the signal states.

In comparison to simple signals, the interfaces for the transfer of data sets must support the kind of data flow demanded by the hardware. The communication pardims used in the hardware must be handled here. Buffers with random access, such as shared memory, have a different data flow than FIFO-buffers. In both cases, a synchronisation means must be provided. For instance, the FIFO buffer can only be read, if data is available. For shared memory, access control means must be provided to avoid collisions. Hence, the communication component needs a control component for each buffer type to provid seamless data transfer. The flow control is handled by dedicated classes, which are part of the communication component facade class (as depict in Fig. 4).
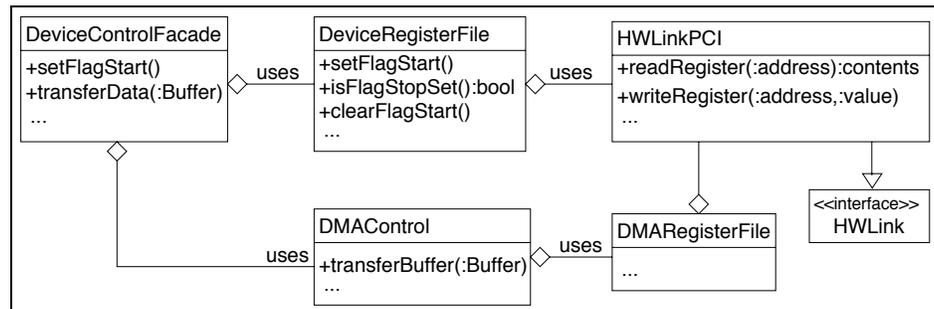

Fig. 4. Example for the class hierarchy for a device driver.

From the perspective of object-orientation, the communication component is an adaptor that hides away the communication via the registers and the com-

munication path between processor and register file. The adaptor can be seen as a delegator as well, because it forwards service requests to the device core. Furthermore, the level of abstraction can be lifted from programming on bit-mask level to query methods, such as *isFlagStopSet()*. The driver core now uses this interface to communicate with the device core transparently.

3.3 Device Driver Core

The device driver has to provide dedicated services with the support of the hardware device. The driver core, that corresponds to the device core, communicates via a transparent communication component with the device core. On the other side, it provides an application programmer interface and uses services of the operating system (see Fig. 3). The driver core has multiple objectives: It has to add missing functionality, it has to cover multiple functionality, it has to restrict functionality, and it has to adapt functionality, like filtering.

Here only a few hints can be derived from the hardware structure, because functionality is not realised by structure only, more by the interaction of the implemented functionality inside the components. Nevertheless, the software part needs to communicate with a dedicated component of the hardware via other components to provide a total functionality. So as reflection, on the software side a chain of software components needs to be created, which cover the (inverse) communication of their associated hardware component, to have a transparent communication to the destination.

Another typical pattern are parallel buffers which in combination with a consuming component. For instance, the message buffers of a CAN-controller. A buffer is selected for consuming depending on a dedicated algorithm. As inverses of the selection component in hardware, a dispatcher to select the message buffer needs to be provided on the software side.

3.4 Device Driver API

The driver core needs to be designed to provide in combination with the device core a dedicated service. If the service does not communicate with the physical world outside the device, the interface needs to provide the access means which are required by components implemented in the driver core.

If communication to the outside world is involved, these services can be put into two classes which have different effect on the interface design:

– End-points of the device shall be controlled by the driver, so access near to the physical level shall be provided. An API towards the application can be directly derived, because driver and device have to provide a transparent access in cooperation. As a simple example, the logical level of a hardware pin of a parallel interface shall be controlled. The API is simply to set the level. The driver must transparently configure the hardware to gain this access to the pin level.

– The other class of API is an abstraction of the device end-point to a higher level. As an example, a Full-CAN controller provides message boxes as interface an has an internal selection policy for the messages. In contradiction to the first class, here not the level on the send lines shall be controlled, but a service which allows messages to send. The driver core has to provide an API for sending messages. Still it is partially reflection of the hardware, because the structures look similar, functionality is different and at a higher level of abstraction.

For some devices, a mixture of abstraction levels is provided by the application interface, depending on the knowledge towards the outside world.

4 Additional Design Patterns

In the following sections, additional patterns are depict or discussed in more detail. Design Patterns[15] are an easy to understand descriptions for solutions in software. In the area of device driver design, some of these patterns can be either applied or used to describe parts of the design, not only for the driver but for parts outside the device driver system as well.

IRQ-Handling The publisher subscriber pattern[15] in combination with the chain of responsibility pattern[15] fits well to the handling of interrupt requests (IRQ). Interrupt requests are often collected from their hardware source by a chain of OR-elements to a single destination inside the processor (see for instance PowerPC-architecutre[17,18]). The processor branches to the main interrupt handler, which identifies the source and calls the interrupt service routine. The inverse pattern for the collection structure is a dispatcher pattern, for which code can be automatically generated[18]. Destination needs to be registered and assigned to a certain event they are interested in (Publish-Subscriber-Pattern). In some cases, the origin cannot be determined by the publisher itself, so sub-components (subscriber) need to gather more specific information them self. All subscriber are informed in a chain of responsibility and they check the assigned interrupt source.

Device and Driver Management Devices and driver management is usually part of the operating system. Device types are identified, corressponding drivers are load, configured and used. This management can be modelled with the multiton and the factory pattern[15]. The factory creates the driver object structure, configures the driver objects including linking to the corresponding device. Depending on the number of devices of the same class, only that number of corrsponding driver objects are created. The management features of the factory can include means of device identification and creation of the correct version of driver for the identified sub-class or sub-version of the device. Driver versions are implemented by inheritance of a base version and means of polymorphism are applied.

5 Conclusion and Future Work

This paper has discussed how to derive a first software model for a device driver for a new hardware device. The software structure and the interfaces has been derived from the interfaces in hardware and the coarse grained structure of the interacting components by reflection. Design Patterns have been applied to ease the software design. The resulting driver will not have high performance, but can be use for first integration in the sense of rapid prototyping.

As future work, a more fine grained reflection of the hardware functionality shall be explored. This requires an appropriate description of the hardware as well as the required total functionality.

References

1. Tanenbaum, A.S.: Moderne Betriebssysteme. 3rd edn. Prentice Hall (2009)
2. Reichardt, J., Schwarz, B.: VHDL-Synthese. 4th edn. Oldenbourg-Verlag (2009)
3. Open cores. Internet www.opencores.org (2009)
4. Jonathan Corbet, A.R., Kroah-Hartman, G.: Linux Device Drivers. 3rd edn. O'Reilly (2005)
5. Oney, W.: Programming the Microsoft Windows Driver Model. 2nd edn. Microsoft Press Books (2002)
6. Purohit, A., Wright, C.P., Spadavecchia, J., Zadok, E.: Cosy: Develop in user-land, run in kernel-mode. In: HotOS. (2003) 109–114
7. Ganapathy, V., Balakrishnan, A., Swift, M.M., Jha, S.: Microdrivers: A new architecture for device drivers. In: HotOS'07: Proceedings of the 11th Workshop on Hot Topics in Operating Systems, San Diego, California, USA, USENIX Association (May 2007) 85–90
8. Lehmann, T.: Towards Device Driver Synthesis. PhD thesis, University of Paderborn (2003)
9. Ihmor, S.: Modeling and automated synthesis of reconfigurable interfaces. Doktorarbeit, University of Paderborn (2006)
10. Thibault, S., Marlet, R., Consel, C.: A Domain-Specific Language for Video Device Drivers: from Design to Implementation. Technical report, Institut National de Recherche en Informatique et en Automatique (1997)
11. Team, T.B.D.: BeOS Advanced Topics-The Official Documentation for the BeOS. O'Reilly (1998)
12. Ditze, C.: Towards Operating System Synthesis. PhD thesis, University of Paderborn (2000)
13. Jnode - Java new operating system design effort. Internet www.jnode.de (2009)
14. Object management group - omg. Internet www.omg.org (2009)
15. Gamma, E.: Entwurfsmuster. Addison-Wesley (1996)
16. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: ECOOP '97 - Object-Oriented Programming, Berlin, Springer-Verlag (1997) 220ff.
17. Shanley, T.: PowerPC System Architecture. MindShare Inc., Addison-Wesley Publishing Company, Reading, Massachusetts (1995)
18. Lehmann, T., Zanella, M.: Modeling and software synthesis of interrupt systems. In: GI/ITG/GMM Workshop: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, Tübingen, GI/ITG/GMM (Feb 2002)