# Towards Device Driver Synthesis

## Dissertation

A thesis submitted to the
**Department of Mathematics and Computer Science**
of the
**University of Paderborn**
in partial fulfillment
of the requirements for the degree of *Dr. rer. nat.*

by

**Thomas Lehmann**

Paderborn

*Supervisors:*

Prof. Dr. Franz Josef Rammig, University of Paderborn
Prof. Dr. Hans-Ulrich Heiß, Technische Universität Berlin

*Date of public examination:* 15. Nov. 2002

## Acknowledgements

My dissertation is the result of different research projects, student projects, and lecturing carried out during my time in the working group of ”Design of Parallel Systems”at the Heinz-Nixdorf Institute (HNI). The HNI is an interdisziplinary centre for research and technology of the University of Paderborn, Germany.

First of all, I would like to express my gratitude to Prof. Rammig for his support of this work. He gave me the possibility to work in this field and he supported me in realising this project. Within his working group, I had the opportunity to work on different research projects, to supervice student projects, and to advice students on diploma thesises. All of this, including my involevement in his lectures, has given to me a very detailed insight into the wide field of embedded systems.

I like to thank my former colleagues who helped me in different ways to realise this final project and gave me a good time at the institute. Due to their wide range of interests, knowledge, and research projects they all brought in different aspects to my work.

I was lucky enough to work together with Mauro Zanella, Achim Rettberg, Christophe Bobda, and Stefan Ihmor on different projects. The work on the RABBIT system has been the most interesting. Mauro Zanella has established and steered this project and has brought in the perspective of a control engineer. Achim Rettberg brought in his knowledge on bit-serial architectures and FPGA programming. The project has been very interesting, because it has covered all aspects of the development of embedded systems. All members have been very good ”sparing partners”during the discussions on hardware, software, and embedded systems. Nevertheless, Achim Rettberg and Mauro Zanella were good readers on the technical content of this work.

In the same way, I have to thank all readers of my thesis who helped me not only to fix technical part. Special thanks here to Megan Starke, who reviewed the complete document very carefully although she is not a computer scientist.

Arthur Lochstampfer has to be mentioned for his critical opinions and his view on system security. He has been a very clever discussion partner on system architecture.

During my time in the HNI, the staff of the Paderborn Center for Parallel Computing ($PC^2$) gave me a very good support with special hardware and Linux system administration. Special thanks to Andreas Krawinkel for his help with the Debian Linux administration.

The most important person for this thesis has been my girl friend, Dr. Eva Starke, who kept me in a good mood and had all the time a very good understanding in what I was going through. Now I have got my revenge!

Hamburg, December 2002

# Contents

# Chapter 1

# Introduction

Device drivers are fundamental parts of operating systems. Drivers are a set of software components inside the operating system which are linked to hardware entities in the hardware architecture. They bridge the ”[..] indescribable boundary between the hardware and the software; the place where the physical meets the logical” [39]. To the user they provide an abstraction and adaption of the interactions with the hardware peripheral devices and isolate the I/O interactions from the rest of the system. The aim is to ”keep it simple, stupid” [89] for the application programmer. However, device drivers are exceedingly complex, highly specialised, optimised and have to be highly reliable.

The notorious complexity of driver grows along with the computer system intricacy. More and more intelligent peripheral devices surround the main CPU. Hierarchies of bus systems form a high performance in-system network. The devices in the network are running partially autonomous and also in parallel to the tasks on the main CPU. The sequential software has to interact with device hardware, where the internal functions are also carried out to a great extent in parallel. Furthermore, the driver uses services of a complex operating system, which may lead to interference with other system components.

The task of designing a device driver is difficult, due to the gap between hardware and software. The devices are mostly documented in natural languages. They contain ambiguous and sometimes unrelated information. On the other hand the requirements by the application are somehow vague and not finally determined. The implementation is done in low-level programming languages with the risk of getting entangled in the bit-operations. A domain specific design methodology is not available at present.

Testing and debugging of a driver is the critical discipline, because a failure often leads to a system crash, and the reason is hard to track. This is not an isolated problem, because the relationships between the hardware components and the software components have to be taken into consideration.

The design of device drivers is an interdisciplinary task of highly skilled experts. They need a broad knowledge on computer architecture, hardware design, operating systems, programming languages and impacts by the compiler, software design methods, bus systems and other interconnection networks, and low-level protocols, etc. The driver developer sits in between the application programmer and the device designer and has to understand both sides.

In the field of Embedded System design, there is a high demand for a systematic design methodology due to the great variety and the growing complexity of the systems. The pressure of fast time-to-market on the one side, and the critical interaction with hardware on the other side, demands synthesis methods or analysis techniques in order to accelerate the understanding and exploration of the hardware behaviour.

## 1.1 Aim of this Thesis

The aim of this thesis is to emphasise on different aspects of device driver design. Current approaches show that a driver cannot be completely synthesised at the moment. Therefore, a major topic is the systematic analysis of the interaction of device hardware and the software driver.

The device hardware is analysed under the concerns of structure and behaviour. On the lowest-level, the exchanged information between device and driver are classified. The idea is to reconstruct the purpose of some signals in the interaction of the device automata and the driver software, which can also be interpreted as an automaton. The classification gives hints for the position and the order of appearance of the signal exchange in the driver control-flow. This communication semantics re-construction is a result of the analysis of the device automaton structure.

A further problem in device behaviour analysis, is the determination of the required sequence of commands to obtain the dedicated reaction of the device. An automated analysis of the parallel interaction inside the hardware from the view point of sequential access, leads to instruction sequences which brings the device into a dedicated state. In this thesis, the use of counter examples of a model checker validation run is evaluated. The communication interface between driver and device is modelled in such a way, that the counter example reports the required access sequence to obtain a dedicated behaviour by bringing the device automata into a certain state.

On a higher level, the device can be structured into interacting components. The internal device structure leads to a component structure inside the driver. In this thesis the method of structural reflection will be presented, which derives the structure of driver components and their interactions from the hardware topology. The component interrelationships on board-level lead to a coarse grained structuring of corresponding driver components. This driver structure can be used as a starting point by the device driver designer for a refinement towards the final driver structure. With a refinement of the device components behaviour to an unambiguous semantics, the direct synthesis of software components in the driver becomes feasible. In a case study, the method of structural reflection and a reversal of the components semantics in software is applied for the synthesis of dispatcher software for interrupt architectures.

These identified patterns have to be explored and combined for the driver design by an experienced programmer. Thus the different high-level software development techniques are explored, whether they are appropriate for driver design or not. Or vice versa, the low- level constructs in hardware-near programming are shifted to an higher level to fit into the abstract view. It has to be distinguished between the software specification models, their specification methods, and the implementation techniques of these models. The automatic translation of the specifications into target code is sometimes still inappropriate in the field of hardware-near programming. Thus in this thesis the modelling of concepts is done in the Unified Modelling Language (UML) [13], still considering that the model is refined for the final implementation depending on the target system.

The device is embedded into a larger computer architecture with interconnection links by means of system busses. The device behaviour and the driver behaviour is almost independent from this communication network, if an adaption layer provides transparent access to the device. In the different approaches for driver synthesis, functions which cover the network up to the device are used. With a detailed specification of the interconnection architecture and the behaviour of each node, this communication layer can be synthesised. In this thesis a method is presented, which determines the software entry point, parameterises the channel, and compensates remaining side-effects. For specification, the behaviour descriptions on protocol level by attributed grammars is used. With the code generator methods from compiler construction, the communication layer can be synthesised. The behaviour specification of each node in the network is independent, so the system can easily be adapted to a new topology by a modified structure description of the communication network between device and driver.

The different design and synthesis methods presented in this thesis have to be integrated into a design flow for device driver design. In one chapter of this thesis, an integration of the methods into a tool set is proposed. The tool set supports the driver design from the analysis phase, over the design phase up to th target code generation.

Due to the use of a synthesis method for the communication software, the information on the semantic of a communication with the device can be preserved during target code generation. The communication is not tangled in memory access instructions in the driver code. Thus device specific optimisations on the target code can be done. In the outlook, the appliance of compiler techniques in the field of device driver is discussed. This includes the topic of dead hardware elimination, analogous to dead code elimination. In this section, the advanced topic of system security in relation to device driver is also elaborated.

The focus in this work is set on UNIX-like operating systems and contemporary desktop PC systems. Nevertheless, the results can be applied in other operating systems and computer architectures. The driver design places emphasis on maintainability and the rapid establishment of a reliable first driver prototype. Together with the analysis method, this prototype belongs to the exploration of the system.

Hence in this thesis, next to the synthesis of a hardware abstraction layer depending on the computer topology, the systematic analysis approaches, and the impacts on the high-level driver design, will be discussed.

## 1.2 Organisation of the Work

This thesis is organised as follows:

**Chapter 2** describes the anatomy of a device driver and the environment it is settled in. The interactions with the other operating system components are elaborated. This chapter discusses the different perspectives of a device driver, and describes its objectives within the collection of operating systems services. A special topic is the interaction with its counterpart, the device hardware. The different methods of exchanging messages are discussed. The last section treats the development process of the driver.

**Chapter 3** gives an overview on the different approaches which are related to the field of device driver design. It shows the orientation of the actual available course literature on operating systems and driver programming in the environment of the different operating systems. A major topic is the overview on the synthesis approaches for driver, and the frameworks designed for portable driver design. The last section contains a discussion on the programming languages used in this field, as well as high-level software design methods.

**Chapter 4** presents the developed approach of deriving the structure of the driver components from the communication relationship of the corresponding hardware components. Here the separation of a driver component into a communication proxy and a driver kernel is introduced. This decouples the behaviour adaption from the communication concern. The following two chapters emphasis on the refinement of these two components.

**Chapter 5** refers to the refinement of communication channel synthesis for software to device hardware communication. The behaviour of the communication nodes in the computer system topology is described by means of attributed grammars. The total behaviour of the communication

channel can be now evaluated, and adaption software to guarantee a transparent access to the device can be synthesised. The communication proxy covers all influences of the nodes on the route to the device.

**Chapter 6** handles the design of the device kernel which sits on top of the communication proxy. The analysis of the middle grained hardware structure leads to suggestions on the structures inside the driver kernel. Furthermore, fragments of the control code can be derived. With a refinement on the structure and the semantics of the component, the direct synthesis of software becomes feasible. Here, a case study for the synthesis of the dispatcher software for interrupt systems is presented.

The second part places emphasis on the systematic analysis of the device hardware under the particular conditions of the asynchronous communication between driver and device. The low-level interrelationships are elaborated, and a classification scheme for the exchanged signals is acquired. The scheme provides suggestions for the point and order of appearance in the driver code. The last section deals with the analysis of the contrast of sequential software and the parallel hardware. From the perspective of the driver, the interaction with the hardware has to be done in a sequential manner. Here, an unorthodox use of Model Checking as an analysis method is presented.

**Chapter 7** briefly discusses the integration of the methods presented into a design flow for device driver design. It includes the concept on integration of the design flow in the tool chain of a UML based design environment.

**Chapter 8** specialises on two topics which are out of the major scope of this thesis. The first one discusses the application of compiler techniques in the special field of device drivers. Different known compiler methods are evaluated as to which part of driver code they can be applied on. The second topic discusses the influence on system security by device drivers.

**Chapter 9** summarises the presented work. It concludes this thesis and gives a brief outlook on future work in the field of device driver design.

The next chapter starts with an overview on the internals, the interactions, and the objectives of a device driver within an computer system.

# Chapter 2

# Anatomy of a Device Driver

Device drivers play only a secondary role in operating system course literature (for example [19, 106, 84, 64, 102, 70]). The literature mostly focuses on processes, scheduling, and memory management. An upcoming topic is security and distributed computing. The access to ”external” devices is described in literature only from a very high level. The device drivers are just seen as modules which provide access to special hardware resources for high-level functions of the operating system or the user application. The objectives of a driver module are to provide an implementation of the device-class specific interface to the operating system kernel, do resource management, control the interactions with the device, and to perform safety checks.

In this chapter the objectives and the interaction with other modules in the kernel are depicted from different perspectives. Due to the inclusion of literature describing implemented operating systems and computer hardware, it goes beyond the scope of the operating system literature listed above.

## 2.1 The Driver Inside the Operating System

Device drivers are components in the Operating System (OS) which interact with other parts of the OS and hardware resources. From an abstract point of view, a driver is a brick in the operating system chart (see figure 2.1). Most of the time the term ”kernel” is used as a synonym for the operating system. In this thesis, only the set of components in the operating system which are not belonging to the group of device drivers are named *kernel*.

The driver implements an interface to the kernel for its device and provides a channel to this device. Therefore it requires a lower-level communication channel to the device hardware. It works as an adaptor from the kernel interface to the hardware interface. As a horizontal form of communication it requires kernel services or it can even so provide services to other kernel components.

The separation of the driver interface into three different interfaces (compare figure 2.2) is used by driver wrappers as an approach to design frameworks for portable device drivers. Examples of these architectures are the Uniform Driver Interface (UDI) [116, 117] or the portable driver approach in [93].

This leads to the following first step of defining perspectives for a device driver as an adaptor between components:

**Perspective 1** *A driver is a software component which adapts the communication between (operating) system components. Some of those components (the device hardware) may be implemented in hardware.*

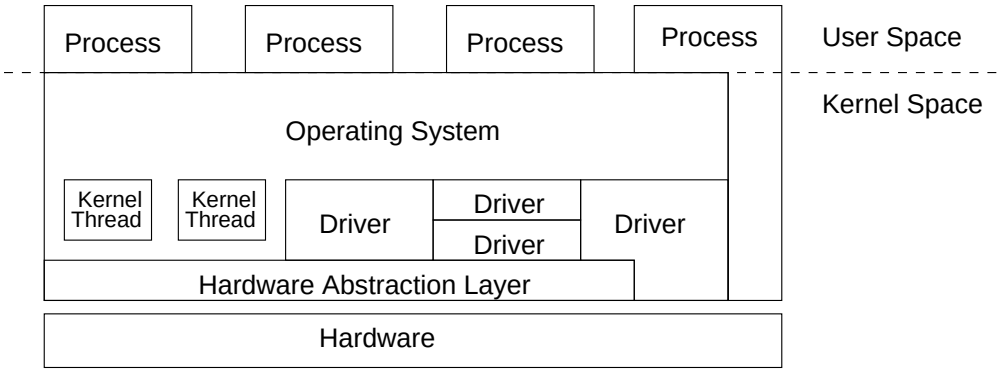



Figure 2.1: Device driver as a low-level module in the operating system structure.

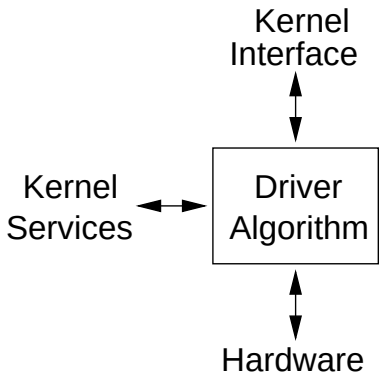



Figure 2.2:
Driver interacts with three interfaces: kernel, user, hardware.

In the case of direct interaction with a hardware resource, the driver is interacting with the device controller which is attached to the system bus. The device controller communicates with the external device [70, 64]. These days the devices have evolved from simple I/O interfaces to computers on their own [103] and are no longer external. For example, harddisks are integrated into the same housing as the main computer system (desktop PC) and have their own Integrated Device Electronic (IDE) with CPU and memory. So here the term *device* is used for specialised hardware resources in the computer system architecture, which are not the CPU or the main memory.

**Perspective 2** *A device is a specialised hardware resource for a dedicated task. The device is embedded in the computer system architecture and can interact with the CPU and other hardware resources in the system via a system bus.*

”The device driver is the part of the OS that directly controls the operation of a device.”[84]. The communication channel to the device can be provided to the driver by lower-level device drivers. On the software side, they provide channels to the device hardware for the drivers using their interfaces. This leads to stacked device drivers or ”cascaded drivers”[112]. The modules are more concrete towards the device and more abstract towards the kernel device interface. The communication is similar to the ISO/OSI reference model (for example compare [85]) for communication. The lower-level drivers provide a transparent communication to the device interface. An example is the device stack for an external CD-writer connected to the parallel printer port of a PC (see figure 2.3). Inside the external device, the inverse stack is implemented with is an adaptor from the printer port to the internal processing logic. This unpacks the communication to the Small Computer System Interconnect (SCSI) or IDE layer. Hence inside the external housing, the same standard SCSI or IDE drives can be used as inside the desktop PC housing.

For the top level driver in the stack, the external CD-writer behaves like a CD-writer in an SCSI-system. So the writer in combination with its drivers pretends to be a different device.

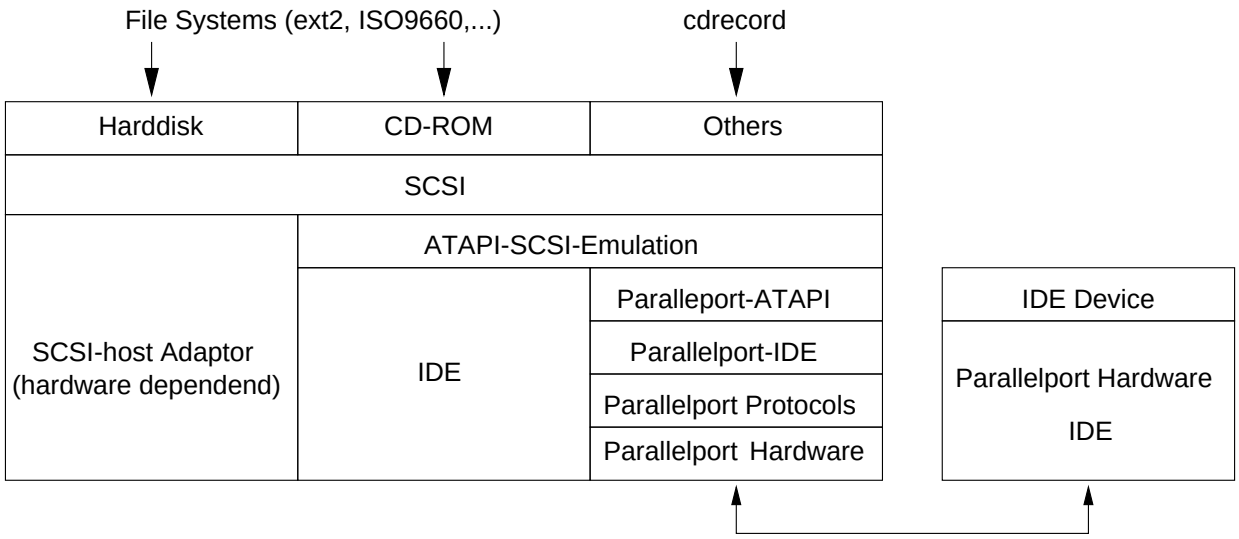



Figure 2.3: Device driver stack for CD-writer with a slot for external CD-writer on the printer port (from [119] extended by external device).

Other examples for emulation of a different device are RAM-disks where memory behaves like a harddisk, or a FAX-modem that behaves like a printer device towards the kernel. In some cases no specialised drivers for each behaviour are required; a driver module can provide and implement different interfaces (for example [52, 93]) for the kernel device interface, and for other kernel modules. The lower-level drivers in the driver stack do not implement a kernel interface. They provide only interfaces for other kernel modules or drivers to allow transparent access to a device.

**Perspective 3** *A driver is a software component which provides special services to other software components, in most cases with the help of a hardware component (the device).*

or

**Perspective 4** *A driver wraps a hardware component to fulfill a demanded total behaviour. The driver itself can be seen as an abstract device. The demanded behaviour can be emulated and no real hardware is required.*

In the operating system literature (see above), the device driver uses a Hardware Abstraction Layer (HAL) (for example the HAL of NT in [102]) which adapts to the low level behaviour of the computer system architecture, for instance: the correct bitwidth of an access[1]. With perspective 4, a device driver can be seen as a high-level hardware abstraction layer, up to the kernel interface. At this point the driver implements an abstraction layer for a *class of devices*.

Towards the application the driver abstracts the device. It generalises the dedicated hardware regarding the functionality of a *device class* and hides all direct interactions to the device hardware [106]. ”While the abstraction simplifies the way the application programmer controls the hardware, it also limits the flexibility by which specific hardware can be manipulated.”[70]. Thus it must be decided for the design of the driver, between a high abstraction level for the interface to the user application, and the accessibility of special features through the abstract interface.

In the rest of this thesis it is assumed, that a driver directly interacts with a hardware resource. The driver implements the interface to the kernel for this device-class, or provides implementations

[1]Many high-level programming languages do not provide a portable definition of the bitwidth of data types like char, integer, etc.

of special services to other kernel components or drivers. Adaption components between two software interfaces can be generated with special synthesis techniques discussed in ”State of the Art”in section 3 or with the design methods on component oriented software [40]. In the next sections the objectives of the device driver are depicted in more detail.

## 2.2 Resource Management

One of the device drivers task is resource management. The device driver is developed as an adapter from a special hardware implementation, an instance of a device as specialisation of a device class, to the kernel interface of that device-class. Hence the driver has to identify all hardware components in the system that it can operate with, and gather the identifiers to distinguish between different instances of a device type.

On the other side the driver has to provide and handle channels to the user of the devices and the driver services. Users are in this case applications on top of the operating system or other components from the inside of the operating system. The communication between the user and the driver is handled by a *user channel*. The relationship between the user and the driver is one-to-many or many-to-one. In most cases a mapping from $n$ user-channels to $m$ hardware resources must be managed by the driver (see figure 2.4).

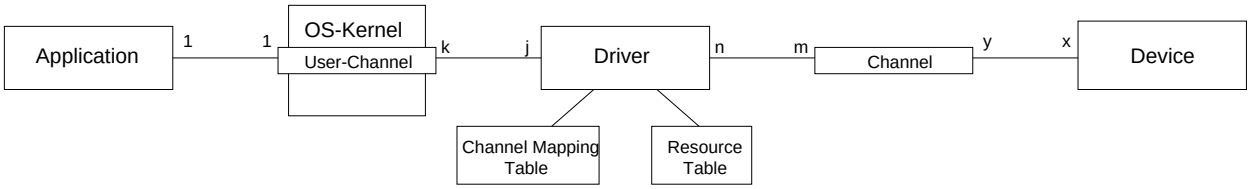



Figure 2.4: Device driver as mediator between user-channels and the resources.

### 2.2.1 Device Identification

Hardware identification is rather difficult [95]. With the use of kernel services the computer system must be scanned for compatible devices in the system architecture. The identification is difficult because most hardware resources do not provide exact identification services.

The system architecture of a computer system can be described by a graph of interacting components. The central element is the CPU that the driver and the operating system are running on. With the use of reflection methods of the system components, for instance the Basic Input Output System (BIOS), the surrounding architecture is explored and the graph of components with the CPU as root can be constructed. Basic requirement is that all components provide information on their type. The identification algorithm must decide whether to dive further into the architecture of a sub-system, for instance if the component has a bridging function, or to stop at this level. The identification process can then be handed over to the driver of this sub-system and to finalise the architecture identification.

In most systems a driver is responsible for internally creation of a hierarchy of components. The driver must follow the hierarchy to identify its sub-resources. The identification hierarchy can be defined as a rule system like in [110]. The rule system leads to an if-then-else tree of characteristics which have to be checked to identify a driver-corresponding component in the system. The descent into the if-then-else structure gropes along the hierarchy of the computer system components.

So each component must provide information on its type. Associated with this information is the device class and knowledge whether components beyond this one may exist or not. In some cases, the type information is sufficient to determine the succeeding structure and it does not has to be explored in further detail.

For example, to explore the devices on a Peripheral Component Interconnect (PCI) bus, the identifying task first has to evaluate the existence of a PCI-bus in the computer system architecture. With the help of the BIOS functions the different resources in the bus slots can be accessed. Resources on the PCI-bus can provide information [95, 83] by means of a device class ID, type ID, the vendor ID, and the device ID given by the vendor. In the case of an IEEE1394 [43] (FireWire(TM)) network host-adaptor, the identification of components in the network and the topology re-construction is left to the driver for this type of network.

Together with the type information, a unique identifier of the device has to be determined to unambiguously address the physical device identity. Similar to the objects in object oriented programming (OOP) the device has a unique identity (ID). In OOP the unique ID is only used internally and the design uses other mechanisms to resolve the reference to the object, for example the memory address, where the object is located. Analogously the hardware systems must provide a mechanism to identify the different instances of device hardware of the same type, as well as methods to separately address them. The unique identifier can be part of the component and be gathered the same way as the type information. Otherwise system internal enumeration schemes must be used, for example the slot number.

During device identification the driver sets up a resource table with ”cookies”for each identified device. ”A cookie is a device-specific piece of information which can be used to track device status information.”[107] The cookie can store the static information to directly address a device, instead of querying this information each time by means of system services. The device cookie can be further used to mirror settings of the device and other channel parameters to access the device.

The process of system identification in combination with the construction of a cookie table, can be interpreted as a flattening or reduction of the system graph. The leaves and nodes of interest are identified and a flat table is established in the driver, whereas the cookies provide direct links to the assigned devices. The communication channel abstraction has to support the perspective of direct communication. The driver provides an entry port through which all nodes are directly visible as if they were directly linked to this port.

For example, the topology re-construction of the FireWire-network can be combined with the determination of the node type and the node identifier (ID). The IEEE1394 standard [43] includes a coding scheme for the node position in the network, and a mapping to a memory address scheme (see figure 2.5). The cookie table includes for each node the ID and the address in the network. The address can be used as base-address for access to memory locations in a node, instead of looking up the node in the internal topology graph.

After identification of compatible devices, the driver sets up a mapping table between the hardware resources and the user-channels of the kernel interface.

### 2.2.2 Communication Channel Mapping

The objective of communication channel mapping is the binding of a user/app-lication interface to a dedicated device. The user application selects or allocates a device by an `open()` call [104, 70]. The kernel sets up a user-channel for communication. These user-channels identify an application which uses the driver during the use of driver services.

From the driver designer’s point of view, it has to be decided if management infrastructure has to be implemented for 0,1, or $N$ devices in combination with 0,1, or $M$ channels. Management

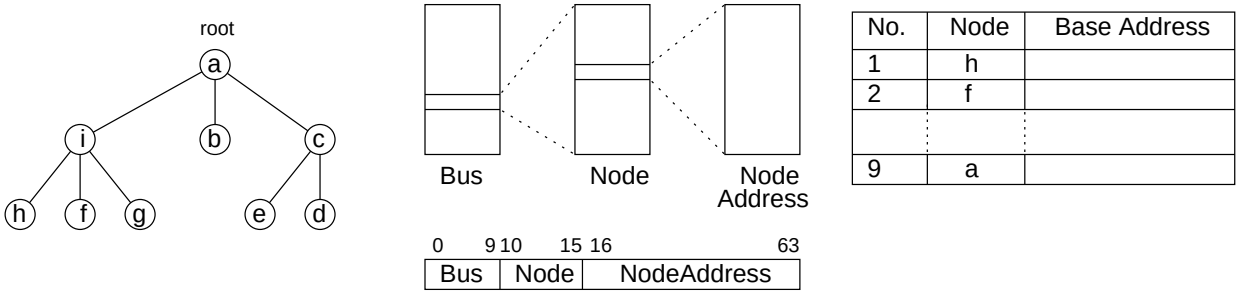



Figure 2.5: Possible topology of a FireWire (IEEE1394) network and a flat view to the nodes by a linear 64 bit address scheme (compare [2]). The cookie table enumerates the nodes and stores the unique identifier and the base address of each device in the network.

structures change dramatically if 0,1 or a limited number larger than 1 have to be managed [51].

It depends on the device class requirements if locking mechanisms or management functions for mutual exclusion have to be implemented in the driver. A different topic is whether the driver is able at its level of abstraction to decide if the access policy is used or not. Both cases can not be discussed in general, and depend on the device class and the desired use.

On the user side, an application must be able to select a dedicated device and the services which it likes to use. Under Linux, or other UNIX-like operating systems, all devices for file I/O or data stream I/O are listed in the `/dev` directory of the file system. In the list each abstract device name is assigned a *major* and *minor* number, access rights and a data-type class. The data-type class distinguishes between *character* devices which can handle data exchange of arbitrary length, instead of *block* devices which use a fixed size of data. Furthermore, the data-type class determines the interface type and the set of possible services of the abstract device.

A device driver registers itself at the kernel at load-time with a system unique major number. The correspondence between the major number in the `/dev` directory and the driver must be set up by the system administrator[2].

The user/application uses the device name of the `/dev` directory to identify the device and allocates it by an `open` system call. According to the *major* number assigned to the device name, the kernel identifies the driver for that device. The kernel instantiates a management structure called *file pointer* and assigns it to a user-channel number which is returned as the result of the open call. The user channel number is an identifier/shortcut for the complex internal resource identification structure. Further method calls by the user application use the channel number as reference for the file pointer structure. This structure is passed together with the system call parameters to the driver methods. On the other side, the file pointer structure is used by the driver as the identifier of the channel to the user application.

The distinction between different hardware resources covered by the same driver must be handled by a driver internal management system. On the user side each resource appears as an individual abstract device. The distinction is made by the assignment of a *minor* number to the abstract device name in the `/dev` directory. The driver can distinguish between the different resources, because this *minor* number is stored in the file pointer structure [95, 8]. This mapping is depicted in figure 2.6, whereas the different resources can even so be seen as a set of different services.

Providing a user communication channel depending on system calls, does not only mean an implementation of a part of the kernel interface. In some cases the driver can provide, with the

[2] In [95] a method for dynamic selection and reassignment in the `/dev` directory under Linux is shown.

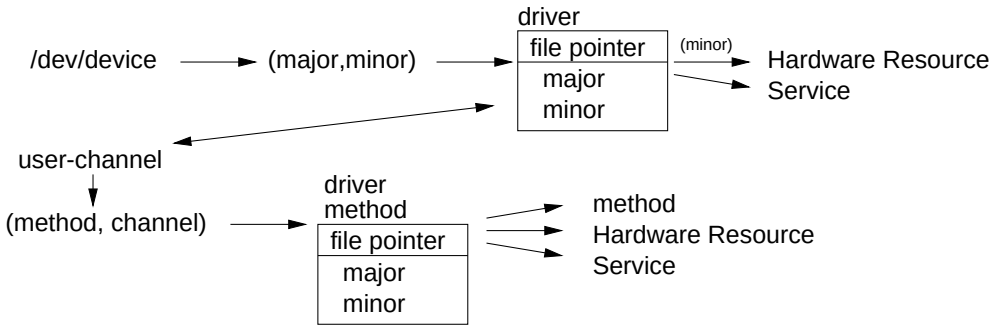



Figure 2.6: Mapping of a device to a user-channel under UNIX-like operating systems.

help of other kernel services, a different channel type which is a shortcut in the user-channel. For example, the UNIX-like operating systems allow the mapping of I/O memory space to the application (process) memory space. The application can request a direct access to the peripheral address space. The driver grants the direct access by reconfiguration of the memory management of the kernel (see [8]). The device is accessed from the application by pointer de-referencing instead of system calls. The avoidance of system and method calls leads to speed up of the application. The application has to be secure to get this access, as the driver is bypassed and is neither informed on the exchanged data, nor on the event of an exchange. Similar to code hoisting in compiler construction [66], unnecessary parameter checks by the driver are avoided and have to be performed on application level. This type of channel should only be provided by the driver if all potential users of the interface are secure. A general policy or mechanism for the negotiation is not provided by the kernel, and must be implemented individually in the driver.

A different strategy for device identification and allocation for applications is used in the object oriented operating system BeOS [9]. Devices are identified by a system-unique name. This requires that the possible names of the devices and the number of devices is known in advance, which is the case for the Joystick port, and the ADU and DAU channels of the BeBox[3]. Instead of system calls, an object for the device class is instantiated and the open call is a method of the object which binds the object to the device by the device name. Access to the device is given by means of method calls to that device object [107]. In the case of a dynamic number of devices, for example the serial computer interfaces, the application asks the object by means of reflection methods (compare for example [40]) which devices (names) are available. One is selected and the object is bound to that device by an open call [107]. So here the advantages of object oriented implementation are used. The management object can be a static part of the class which works as proxy for the device class. Each instantiated device object shares the management object, which keeps track of the available devices of that device class, and does the housekeeping in the system along with the device identification.

## 2.3 Control-Flow

The interaction of a device driver with other components and the kernel can be described by means of the Unified Modeling Language (UML) and design patterns. In the book on *design patterns* [36] different patterns for object oriented systems such as *facade*, *command*, *proxy*, and *observer* are described. In this section the behaviour of the driver and kernel as well as the interaction is described by means of these patterns.

[3]Computer system developed together with the BeOS.

### 2.3.1 User Process and Driver Interaction

The device driver implements the interface to the kernel by providing callback functions [95, 112]. The interface acts as a facade for the driver, its sub components, and the wrapped hardware. The communication of the user application with the kernel can be done by message passing, or by a software trap. Both systems are wrapped by library functions which cover the system calls. The kernel invokes the callback functions of the driver as a result of a system call. In a previous step the kernel has performed the work of a dispatcher.

By message passing, the message with the request of kernel services is sent to the kernel. The kernel processes the request and sends the answer back.

In the method of system traps, the system call stub stores the message parameter in a defined location, and a software interrupt is raised. The processor switches to super user mode and the request is processed. The answer is stored in a special location and the control-flow branches back to the user process [106].

In Linux the system call is done by a software interrupt (trap) with two parameters, the call code and the arguments, which are stored in defined processor registers. The system switches into superuser mode (kernel mode) and dispatches according to the call code, to a callback function inside the kernel [8].

In case of a driver, the kernel resolves the command that is included in the system call arguments, identifies the device class according to the used channel, and selects the driver. Depending on the method code, the corresponding callback function of the driver is selected and invoked. So the kernel acts like a dispatcher for commands, whereas the dispatcher is structured by the levels device-class, device driver, and abstract function of the device class. The demanded behaviour must be implemented or it has to be emulated in the callback function. Inside the function, a further dispatcher can interpret the given parameter sets as an encapsulated structure of commands and parameter structures. Hence, the kernel and the top level of a driver act as a dispatcher for commands and data-structures. This behaviour is similar to the ”Chain of Responsibility” pattern (compare [36] for the description of the pattern), though the receiver chain is not iterated, more the request is dispatched in a branch tree.

The communication between the user application and the driver is initiated by the user process. The driver is in the position of a server in a client-server relationship. The control-flow of the user process branches at the positions of the system calls to the kernel, and then to a device driver routine. The communication with driver is hidden by the call of functions from the operating system libraries.

### 2.3.2 Device and Driver Interaction

The relationship between the driver and the device hardware is in many cases not a master-servant relation. Both work as parallel systems on distributed parallel hardware. The hardware itself is massively parallel. The software is quasi-parallel in the sense of multi-threading or multi-processing[4]. Due to the different types of parallelism and the underlying communication hardware, the software and the hardware parts have to synchronise each other by the exchange of messages.

The CPU can send a message to the device which recognises the message by an access to the device register file. In [70, 64] the device (controller) has four types of registers embedded in the system memory map to exchange messages: data in, data out, command, and status. The registers can be read and written by the CPU.

The device can only change values in its registers or write values directly to the main memory

[4]Real parallel device drivers, for instance on Symmetric Multiprocessor Systems, are not assumed here.

by a Direct Memory Access (DMA). The driver software recognises the change only by polling or by an explicit notification by an Interrupt ReQuest (IRQ). The request raises a software exception[5] [102, 24]. The processor branches as a result of this exception to a special routine for hardware exception handling, the Interrupt Dispatching Routine (IDR). This routine has to identify the interrupt source and the reason for the signal, because the information content of the interrupt signal (in hardware) at the CPU, is in most cases only one bit. After source identification, the routine branches to the interrupt service routine (ISR) of the driver which is assigned to the interrupt source.

The control-flow, with respect to interrupt locking mechanisms of the processor, can change at any time to the ISR (see figure 2.7). This leads to a quasi parallel behaviour of the ISR and the rest of the software. So the driver/kernel/application and ISR work as parallel processes/threads.

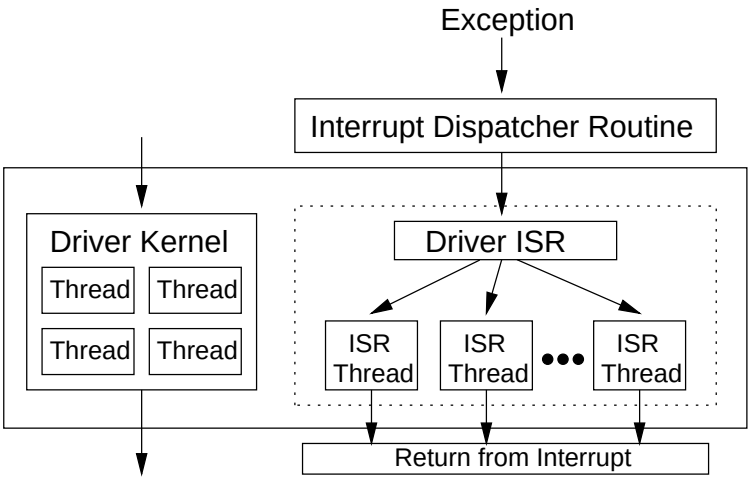



Figure 2.7: Quasi-parallel processing of driver threads and the ISR threads.

Hence the ISR of a driver can be modelled as working in parallel to the driver. The complete driver can be seen as a multi-threaded process with two or more threads: the driver code and the ISRs (see figure 2.7). Because they are working in the same memory space, the internal communication is performed by shared memory. Hence, synchronisation mechanisms, like semaphores, must be used to avoid race conditions [95].

The communication of the driver and the device can be done by polling (programmed I/O), or as interrupt driven I/O with and without DMA [103]. In interrupt driven I/O, next to the flag for signaling the new state, the device sends an interrupt request signal. The ISR of the driver dispatches the request according to the flag to a dedicated sub-routine. This sub-section is the same as in the case of active polling (compare figure 2.8).

In some operating systems this sub-section is implemented as a further thread, called the ”bottom half” [95]. The operation system scheduler provides in this case a further scheduler for such threads. The threads have a higher priority than the other system threads. The ISR only does the minimum of required interactions with the device, for instance the acknowledgment of the interrupt, and schedules the bottom half as a runnable thread of the driver (see figure 2.9) in the dedicated scheduler. The probability of a second interrupt request before termination of the ISR is reduced because the ISR just schedules its bottom half and returns. The bottom halves can be preempted by other ISRs, thus the time where the processor is in the state of servicing an interrupt request is reduced. Nevertheless, if the gap between consecutive requests is unknown, the ISR

[5] sometimes the term *trap* is used

must block recursive requests, which can lead to a loss of requests. The rate and the gap between consecutive requests is an important design parameter which is hard to determine most of the time.

From the driver designer's point of view the bottom halves are the parts in the control flow after the busy-waiting for notification from the device. The notification and the busy-wait is only redirected to an IDR and its assigned ISR. The busy-wait is suppressed by the operating system scheduler. This leads to multi-threaded drivers in the sense of the main driver kernel and the ISRs with its button half.

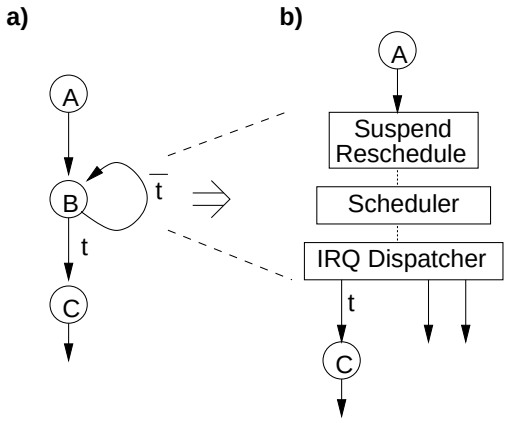



Figure 2.8: Transformation from busy-wait by polling (state $B$) in (a) to a multi-threading system with interrupts in (b). Part $A$ sets up the system and $C$ is the respondence of the driver by the device message $t$. The polling is now hidden in the scheduler which wakes the IRQ Dispatcher on the event.

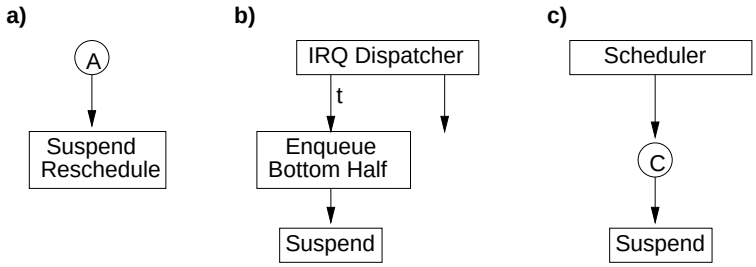



Figure 2.9: Split of the interrupt service routine (a) into a part for scheduling the bottom half (b) and the bottom half itself as message handler (c) activated by the scheduler.

The unknown branch in the control-flow to another thread, requires a mutual exclusion of critical sections and a design which allows re-entrantability. The mutual exclusion can be established with the semaphore services of the kernel. The avoidance of race conditions between the threads requires a more detailed analysis of the interrelationships of the threads.

In some cases, the user process next to the the driver has to also react to the interrupt driven messages from the device. The user process is informed under UNIX by a *signal* [104]. The kernel scheduler branches in the user process to the dedicated routine, the signal handler of the user process. It acts like a parallel thread to the user process, similar to the ISR to the driver. The signal handler must register itself as a listener to a signal from the driver (see diagram in figure

2.10) similar to the observer design pattern described in [36]. The listener management must be implemented in the driver with the help of kernel services [95].

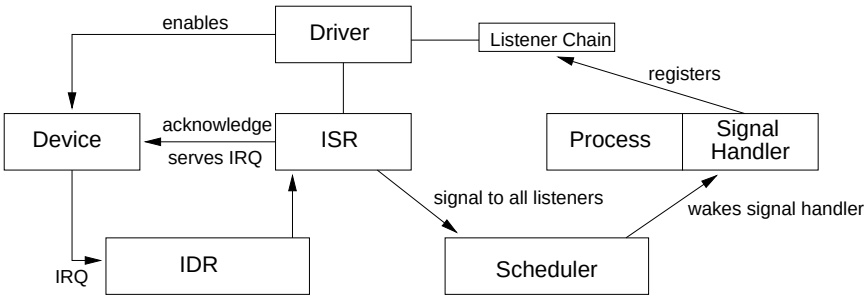



Figure 2.10: Forwarding of an interrupt as a signal to the signal handler of a user process.

The information exchange between the device and the software can be done by a Direct Memory Access (DMA) transfer. The data can be exchanged between the device and the main memory without the help of the CPU (just initialisation) [106, 102, 70, 62, 95]. A dedicated DMA-controller executes the data transfer on the busses. This controller can be part of the computer architecture or of the device itself.

The nature of the exchange of messages from the device to the software memory by a DMA transfer is different from the transport of data to the device registers by the CPU. The software side recognises the transfer to its memory only if it has initiated the transfer. The access to a device register can be recognised by the device by an extension of the interface between the register file and the system bus. Together with the signal for storing information in the register, an event of that access can be produced as further information for the device. If the bus protocol has a setup and closing section, a pre- and post-event (see figure 2.11) can be generated, too. This is similar to a function call in software. Not only the parameters are passed to the function, even the event of the function call can be used as a signal.

Though data exchange by means of a DMA transfer under the control of the device, the driver message buffers can be modified without recognition by the software. Hence, the device has to explicitly notify the driver of a change by means of an interrupt ”[...]interrupts are a hardware assisted mechanism for synchronising the processor with the (asynchronous) events”[64]. This must be expanded to the point of view that interrupts are part of the message exchange mechanism between driver and device.

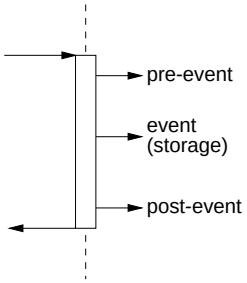



Figure 2.11: Sequence chart of the different events in the register file during an access from CPU to a register over the system bus. The events are the result of the recognition of the bus protocol sections.

Depending on the parallel nature of hardware, the extra events from the register file do not decrease the system performance. In the case of interrupt requests, each exception leads to a context switch in the processor with the drawback of a decrease in performance. Thus the notification messages of a value is not sent on every change. The driver designer can define on which change the driver is informed of, by notification settings in the device. As summary the exchange of messages

with the device can lead to a branch, depending on the events in the device at each access, whereas the device can only change the control-flow of the driver by the result of polling or by an interrupt request.

## 2.4 Access to the Device Hardware

As stated above, the driver algorithms can be demarcated to the kernel by a facade which is an implementation of the kernel interface. On the other side the driver requires access to the hardware device. Most drivers are designed for a von-Neumann computer architecture with memory mapped peripheral devices [70]. The classical devices are only registers embedded in the memory space which act as a memory cell. The external device hardware is directly attached to those registers. An anachronism is the parallel printer port of contemporary desktop PCs. It still acts as a simple register file embedded in the I/O memory space of the Intel(TM) `x86`-processors. The register content is directly routed to the connector pins [95, 62]. Later in the evolution, the peripheral devices have been enhanced by their own ”intelligence”, for instance the harddisks with Integrated Device Electronic (IDE) [99]. The peripheral devices have their own processor and memory, and are computers on their own [103]. Still the communication with the main CPU is performed by registers embedded in the system memory map.

Modern computer architectures have a hierarchy of bus systems or have access to distributed peripherals by network architectures. An example of the hierarchy is the system of a CPU memory bus, PCI bus, and ISA bus in contemporary desktop PCs. The bus systems are connected by bus bridges for exchange of data. Distributed peripherals are for example mass storages like CD-Writer on FireWire [29, 98], or sensors and actuators on fieldbus systems (compare for example [12]).

The different bus systems and the bus bridges have an impact on the communication between the device and the driver algorithm. For example, the endian-mode can change, the chronological order is changed, or a special protocol is required to get the correct manipulation of the device registers. Thus, a lower access layer in the driver must provide a transparent access to the device registers. This is similar to the ISO/OSI reference model of communication. The driver algorithm and the device are in the same level of abstraction. The lower levels in the driver and the register file in the device must in combination provide a transparent communication. This communication system will be called in this thesis ”Hardware Abstraction Layer” (HAL).

The bus hierarchy and the modern processor architecture lead to the effect, that the communication between CPU and device is asynchronous. The internal CPU speed is much higher than the speed for external data exchange. Next to the device registers ”[...] memory is always ready to furnish a datum whose address is designated by the processor. It does it so within nearly constant time, the memory-access time,[...]”[64]. The processor waits until the datum returns. Buffers in the bus bridges lead to a communication with a soft timing behaviour. Due to the variety of architectures, an automatic generation of the communication layer is not easy. In the Devil approach [61], abstract constructs named *ports* cover the communication channel from the driver software to the device registers. In other approaches, for instance [75], platform dependent library functions cover the channel. But they are not efficient, as optimisation can only be done within the library functions but not across. The compiler is not able to handle the problem, because it lacks knowledge about devices. In the different program languages, driver register access, is reduced to memory access and the compiler handles these locations only as volatile memory and not as device registers.

## 2.5 Development Process and Lifeline of a Device Driver

The development process of device drivers is somehow similar to the development of application software. It starts with the specification of the driver requirements. A problem for the programmer is to analyse and to understand the behaviour of the hardware. Similar to the use of a library set, the programmer must evaluate the behaviour of the hardware system and select the required components. After specification and evaluation, the driver is coded, tested and debugged.

Debugging in device driver programming and hardware-near programming is a very difficult task, because of the software/hardware interactions. ”Device drivers are unforgiving – one small error can cause an entire system to fail”[35]. Debugging is an interdisciplinary task which requires broad knowledge of the hardware and the software in order to identify the origin of the failure. Hardware lacks a sufficient number of browsers/observers for the internal hardware states. Debugging of software in kernel mode is a different problem, because a debugger requires a running kernel to operate, and can therefore not break the execution of the kernel for inspection. The problems of kernel debugging are described in more detail in [95].

Additionally it seems that something like the Heisenberg uncertainty principle in hardware-near programming exists. The moment you try to debug a hardware system the behaviour changes completely, because it is modified by the hardware and software probes.

In [35] as further steps the integration into the target system and the documentation of the driver API are discussed as single topics. The misuse by the application software of the already separately tested driver, can highlight to light unrecognised errors during driver integration. This situation can require a further test, as well as debugging iteration with the involvement of more people from the application side. The lifeline of a driver is depicted in figure 2.12.

After a successful first shipment of the driver, the development process changes to an evolutionary [38] development process. The next stage is the adaption to the next release of a hardware, or the incorporation of a similar device of the same device class. The driver ”grows” with the evolvement of the system. In the Linux history this can be observed by different drivers, for example the ”tulip”driver for Ethernet cards. It can work with different network cards. The X-server ”`xfree-4`” [121] is a new release (`4.x`) which includes all previous dedicated X-server (<`4.x`) for the different video cards into one monolithic X-server.

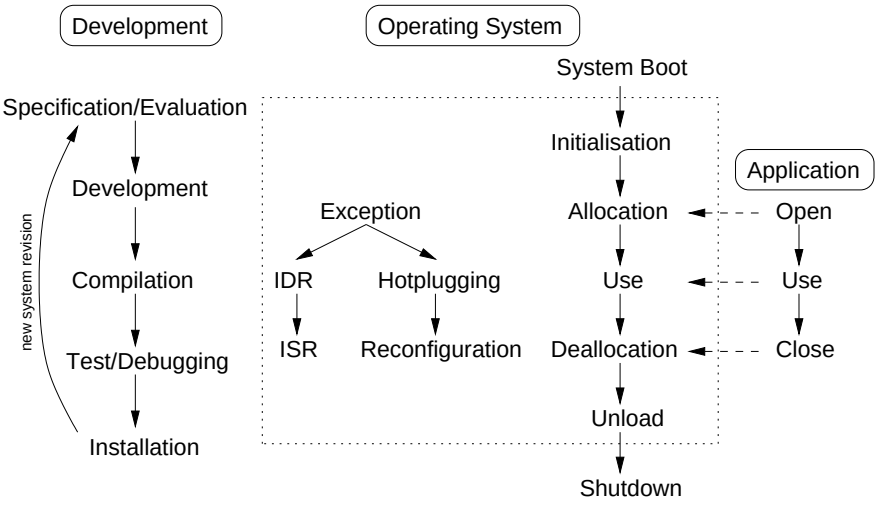



Figure 2.12: Lifeline of a device driver. Loops in the lifeline are not shown.

At some point in time the driver is installed on to the system. During runtime, the driver is loaded into the operating system and after an initialisation process the kernel/user-application can

interact with the services of the driver. The methods of the driver are invoked almost in arbitrary order until the system is shutdown or the driver is dynamically removed from the operating system.

Optimisations can be done at different stages in the lifeline; in the development process of the driver code, the compiler and the boot process in the system. Here the driver can adapt itself to the conditions of the actual system. An example is the Linux Ethernet driver ”tulip”. At first glance the source code seems to be only an if-then-else structure which branches only on the different device types. Thus during runtime only the parts dedicated to the identified card type are executed. These parts include the functionality of the driver which becomes significant on the second glance.

At the different development stages different parameters are fixed. During the development phase only those parameters can be incorporated into the design, which are fixed up to the compilation point. For instance, the number of devices and the location on the system bus must be resolved during system boot, and the driver must determine those parameters during initialisation. So parameters which are dynamic after compilation can be fixed by contract. The required assignments between major and minor number and the devices in the `/dev` directory, which are registered by the system administrator during installation, can be part of a contract in the documentation. Hence they can be resolved during compilation.

The lifeline in figure 2.12 shows on the right side the minimum sequence of method calls in the driver. The first method call is the initialisation for the basic set up of the complete driver. Afterwards the application allocates the device. Here the driver can adapt itself to the application. Then, the application can use the driver methods almost in arbitrary order; they are only restricted by the internal semantic of the services. On normal operation the application deallocates the device, and the system unloads the driver. Depending on the physical nature of the device, a hot-plugging of hardware resources can happen at nearly any time. So the driver routines must be able to handle this exception.

This analysis of the lifeline and the use of driver services has an impact on the driver design. It has to be determined when which events can happen, and which parameters are to be fixed after a set point in time. This influences the placement of code in optimisation and synthesis approaches, which will be shown throughout the next chapters.

## 2.6 Problems in Device Driver Design

The problems in device driver design are not visible at first glance, depending on the overview presented in the previous sections. A major problem is the understanding of the nature of drivers and how they interact with their environment. But even for an experienced programmer some problems are hard to cover.

The aim of the driver development is the implementation of an abstract device under the consideration of the fixed hardware part, and under the given system architecture either in hardware or in software. The requirement is to bring the device hardware into dedicated states, resulting in the required total behaviour of software being in combination with the hardware.

The first problem here is to understand the behaviour of the hardware. The device hardware is documented most of the time in written form with natural language. The description is intermixed with formal parts, simple register definitions, etc. Next to the understanding that hardware works in parallel, the behaviour of the given implementation has to be understood. It has to be in the way of what manipulations of the hardware have done, in order to reach a desired state.

Furthermore, the impacts of the system architecture have to be taken into consideration, because they influence the exchange of information between the driver software and the device hardware. Up to now, no patterns for the analysis and the resulting hints for the programming exists.

This leads to the second problem, that currently no programming model for drivers exists. One approach is a domain specific language, which provides a level of abstraction for device drivers of a special class of devices. Other abstract specification methods are the standard methods of software engineering. But it is often claimed that they can not be used for abstract specification of the driver behaviour, because the resulting overhead is too high. Nevertheless, with appropriate use of the methods, the overhead in implementation is minimal and they should be used even for this low level programming task.

In the previous sections, only the macro perspective on drivers has been shown. From this starting point the fine grained structure of the driver has to be derived. Still no patterns are known to systematically derive these structures and how to derive parts of the driver behaviour from the analysis of the given hardware architecture. Thus one problem is the specification of the internal structure by separation of the different concerns.

A further problem is the communication between the driver and the device hardware in the system. They mostly communicate by the manipulation of the content of registers in the device register file. This manipulation lacks a suitable abstraction and is directly programmed by bit field manipulations. These error prone bit operations are mostly encapsulated in function sets of a board support package, providing a higher level of abstraction, but most of the time it leads to an overhead during runtime. On the other hand, the function sets have to be generated by someone, hence the problem of programming bit operations is only shifted to the programmer of the support functions. Moreover the bit operations of the registers are hard to read, and so they are a source of errors which are hard to find. A compiler can not re-construct the purpose of the manipulation, and neither can do checks, nor optimisations of the instructions.

Furthermore, the different hardware components between the CPU and the device have an impact on the communication, and hence on the register manipulation. A change in the order of the manipulation sequence, or a manipulation of the transported data can result in a total different behaviour of the device. These side-effects have to be compensated for each CPU/device pair and the given communication architecture. This spans a new dimension in driver design where the driver not only has to be adapted to the given combination of OS, CPU/programming-language, and device; it furthermore has to be adapted to the given system architecture. So in combination with an adequate abstraction level, a systematic translation of this level to the instructions required for the given communication architecture is missing.

The last point to mention is the integration of different methods into a design flow in combination with a supporting tool suite. The different software development tools, even those which are dedicated to driver design are self-contained or very restricted to a special operation system or hardware architecture.

## 2.7 Summary

In this chapter the interior and the environment of a device driver have been shown. A device driver acts as an adaptor between the hardware of the device and the operating system components. From the perspective of the operating system, the driver in combination with a device forms an abstract device. The driver can be separated into the concerns of the driver algorithm, the driver kernel, and the communication of a hardware abstraction layer towards the device hardware. Closely related with the communication is the topic of device identification and low-level system security.

Furthermore, this chapter has introduced the different requirements on a device driver and has given an overview on the problems in analysis and specification. The inter-relationships are complex and have different impacts on the work of a driver.

The next chapter will give an overview for approaches for the design of the different driver

parts. The following chapters describe an approach for a general model of a driver structure. An approach for a generic hardware abstraction layer will be shown together with an analysis approach for the required driver kernel.

# Chapter 3

# State of the Art

This chapter gives an overview on the different approaches in the field of device driver design and hardware-near programming. The overview can not be a complete survey because this field is as old as computer architecture and operating systems itself. In respect to the interdisciplinarity and the width of the field, many references here given, can only be seen as examples of their specific area. Some approaches are not directly related to device driver design, but in combination with the directly assigned work, this chapter provides an overview on the different aspects and view points of driver design.

## 3.1 Books on Device Driver Design

Device driver architectures play only a secondary role in the literature on foundations in operating systems. Examples are [19, 106, 84, 64, 102, 70] which discuss more the topics on process and memory management, networks and security. Device access is only a brief section. The hardware is mostly described as a device controller which is accessible by the system bus.

Books on programming in the different operating system environments like UNIX, for example [104], describe the use of a device class from the users perspective. These books give hints to how the driver of a special device class is typically used.

Books on dedicated operating systems like BeOS, Windows, or Linux are more specific. They describe the operating system architecture and the embedding of the driver software. For the Linux OS in [8], the kernel architecture of the system and the interfaces to the drivers are described. The section on driver programming is only a brief summary of [95]. There the driver model and the kernel service functions are described in detail. It teaches how to use the functions, how to access hardware and what behaviour the different drivers have to provide. The author gives executable source code examples, and the book is the basic reference book for Linux driver programmers. Design rules for the internal driver structures are not given. The description of the hardware use ends with the access to the device registers.

In [6] and in [74], the driver architecture of Windows 2000 is described. Similar to the driver book for Linux, these books provide a description of the kernel service functions, how to use them and how hardware is managed by the operating system. The access to the hardware is given by the Hardware Abstraction Layer (HAL) of Windows. Under Windows, the internal driver structure has to fit to the model of the driver kernel, the hardware access via the HAL and the integration of filters between those layers.

The book on the BeOS kernel [107] describes the use of devices and the device driver model of this operating system. It is not as detailed as the books for Windows or Linux described above. But the book provides enough information to write drivers for PCI-based devices. Similar to the

previous books, hardware is accessed by the use of special functions and no design rules for the driver internal structure is given.

Books on the programming of embedded systems, for example [7] or [14], describe the use of hardware peripherals from the perspective of the used programming language. The ”driver” software is not a part of an operating system, it is more the part of a monolithic block consisting of the driver, the operating system and the application. The literature on microcontroller programming is closer to hardware. There, the programming of the integrated peripheral devices is described. Again, they lack a systematic procedure for driver design. The driver control-flow is derived by interpretation of the device documentation.

A general book on device driver design which covers the aspects of hardware, software and design could not yet be found.

## 3.2 Driver Algorithm

This section focuses on the approaches for driver wrappers, the generation of drivers by means of domain specific languages, and the synthesis of the driver kernel in the sense of adaptor synthesis.

### 3.2.1 Portable Driver Design

In the master thesis of Stein Jørgen Ryan [93], a portable device driver for a shared memory adaptor for a Scalable Coherent Interface (SCI) [100] is described. The driver kernel is wrapped by a kernel interface adaptor, a device hardware abstraction layer and a kernel services adaption layer. The driver kernel invokes different functions which have to be reprogrammed or adapted for a port to a particular operating system. Furthermore, the function calls of the kernel have to be adapted to the functions provided by the driver kernel. All adaptations have to be made by hand. With this approach the driver kernel has to be designed once, and can be used in different operating systems. An adaption to Windows NT, Solaris x86, and to Linux 2.2 has successfully been made. A systematic approach for the deriving of the driver algorithm is not given.

The “Project UDI: Uniform Driver Interface” [113] is a co-operation of different companies, for example IBM, Lynx, Sun, Hewlett-Packard, etc., for specification of a common wrapper for driver algorithms. The idea is the same as in the former approach; to provide platform and operating system independent interfaces for a driver kernel. The driver kernel can request services from the kernel and also provides callback functions to the application interface. Towards the device it can request channels for I/O transfers. The device register access is provided by channels which cover the complete communication channel up to the device, similar to the approach in [110]. The driver is fully isolated from the environment. The integration of the driver kernel into the target system is done during the compilation phase. The required communication functions of the target platform are added by the compiler and linker. The specification is at the present time in the revision 1.01 [116, 117]. The specification contains sub-specifications for different adaption layers, for example the physical I/O access [115]. The participating companies and the free software groups under Linux [114] are working on the implementation of the wrapper functions.

### 3.2.2 Domain Specific Languages

In [110] a Domain Specific Language (DSL) for video device drivers is described. The language statements are evaluated by an abstract state machine. The state machine interprets a micro-program and calls the corresponding domain specific code sequences out of a library. The language statements and the sequences are derived from an analysis for similar patterns in the drivers for the

X-window system under Linux. The domain specificity is given by similarities in the objectives the drivers have, because they all belong to the same device class. The authors identified different similarities as patterns in the drivers:

- Operation pattern which repeated code sequences that differ only by data.
- Similar sequences which slightly differ in the hardware communication and which are handled only in a small number of different ways.
- Initialisation blocks for the different devices.

These patterns are used to analyse the driver source codes and to derive the elements and the structure of the Domain Specific Language.

The devices are identified by a decision tree which is defined in the DSL as a rule system. Inside the DSL, switch-case structures allow to define device specific sections.

The different devices provide different register maps. This is covered by a *port* approach. The port covers the complete communication up to the register. The access to the register is abstracted by a register identifier and access direction (read or write). The registers are described by location and bit fields. The communication to the device is abstracted by manipulation of *logical variables* inside the device. A translation scheme from logical variables to the coding of the registers bit-fields is described. With this high level language the driver can be described and implemented. An adaption to different operating systems can be performed by adapting the abstract state machine which processes the driver code. The driver is split into an interface for the device registers and the algorithm working on them [60]. The communication with the registers has resulted in the Devil approach, which is discussed in the section on hardware abstraction layers (3.3).

### 3.2.3 Automatic Interface Adaptor Synthesis

A research group at the Royal Institute of Technology, ESDLab, Sweden, worked on automaton synthesis, and interface synthesis based on automaton descriptions. The idea is to describe the allowed access sequences to the device registers in a regular grammar. The device driver is described by its interfaces and the adaptor (automaton) is synthesised.

A component which works as an adaptor between two interface protocols has to fulfill two protocol definitions. The protocols are defined in the language ProGram [71, 75] which is used for automaton synthesis [58, 78]. Both interface specifications are interpreted as automaton descriptions. The synthesis algorithms for the adaptor component creates a product automaton [42], and withdraws all product states which can not be reached, or which can not be reached by a constraint of cause and action on the received and sent signals [72, 79]. In case of a set of solutions the algorithm chooses the one with the shortest path.

This approach can synthesise adaptors for syntactical re-ordering of information. The interface descriptions define the order of the required and provided information. The semantical mapping of the information is done by a mapping table of the information place holders. An example is the parallel to serial conversion. The semantical identity of the protocols must be chosen at the outside by assignment of a placeholder for each bit in the two protocols. The approach requires a synchronous processing model of all three automatons, the left side automaton, the adaptor, and the right side automaton.

Mattias O'Nils used this approach in his PhD thesis [75] to synthesise device drivers. The driver adapts two ports: to the application/kernel side and to the device side [79]. Special services are encapsulated by library functions and the invocation protocol is described by the ProGram language. The access to the device is covered by communication channels. In [76], a classification scheme

for interface synthesis and different tools for channel synthesis are evaluated. In the ProGram approach, a simple channel with the read/write in combination with the register and device base address is used. The channel implementation is provided by a platform dependent communication library [77].

## 3.3 Hardware Abstraction Layer

For the automatic generation of hardware abstraction layers for device access, different methods including tool integration are available.

The tool ”COMIX” [123, 80] can be used to generate a hardware abstraction layer. The idea is to generate the hardware abstraction layer together with the register file implementation in hardware, so the addresses, coding of the content and bitwidth is determined during synthesis. The register file of the device is defined with the required abstract content, similar to the Devil approach [61]. The definition is stored in a database organised as an XML-file.

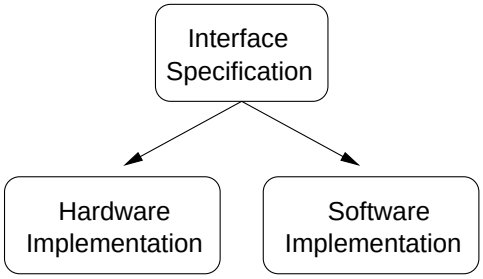



Figure 3.1: Idea of HAL synthesis in COMIX/TEMPLIX.

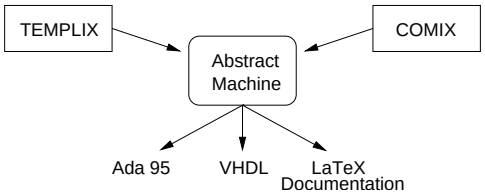



Figure 3.2: Processing of the specifications in COMIX/TEMPLIX [123].

The required synthesis steps are defined in the language ”TEMPLIX”. The statements are interpreted and work on the tree structure of the register definition file. The language ”TEMPLIX” is defined as a XML document and allows limited operations on the data-structures. Synthesis programs for a hardware abstraction layer for Ada95 are available. Ada has a well defined view regarding to memory, and the Ada compiler ensures correct bit-operations in memory space. This eases the HAL synthesis. However, the device register file in combination with the communication architecture must hold the assumptions of Ada on the behaviour as memory. So the main work in adaption of the communication channel between software and the device has be done in the hardware interface.

The system ”ElBaCo” [73] is used to synthesise device adaptors for the IPANEMA-system [41] for mechatronic systems. The IPANEMA framework requires an adaption of the real hardware components of the system, sensors and actuators, via encoders, DAC, ADC, etc. These adaptors can not be derived from the specification of a mechatronic system. Simulation models of the mechatronic design environment lack the knowledge on the adaption from abstract values to real hardware interfaces. Hence, adaptors for the calculator objects in the IPANEMA system have to hide the implementation of data capture by hardware or network access.

The objective of the ElBaCo system is to generate these adaptor objects from a further specification. This adaption scheme is specified in a hierarchical description language. The specified adaption scheme is analysed and the adaption code for the framework is generated. This approach is similar to the COMIX/TEMPLIX approach (see above), although here the transformation is not done by interpreting the TEMPLIX-description. The transformation description is compiled to an executable binary with the Eli-compiler [27]. The executable then processes the adaption description.

The ElBaCo description language is structured like a tree with special leaves. Depending on the structure, the attributes in the leaves and the nodes, code segments for the driver are collected and parameterised. Code for channel adaption by scaling is automatically generated from the description. This tool is specialised for the adaption to the requirements of mechatronic systems and the IPANEMA framework. This approach shows the description of the system in a hierarchical structure, and the processing of this structure towards a device driver by collection and parameterisation of driver sub-components.

The domain specific language (DSL) for driver synthesis (see section 3.2.2) in the Devil approach, uses ports of an device abstraction layer for the communication with the device. This approach has been extended to the ”Devil”language (DEVice Interface Language) [90, 91] for abstraction layer synthesis. The ”bit operations can represent up to 30% of driver code”[61]. As abstraction for the communication, the manipulation of *logical variables* inside the device is used. The language specifies the register set of a device, and the transformation from logical variables to bit fields in the register. Furthermore, special features like the default value and special behaviour of registers like passive, volatile or trigger are specified. The communication channel with read and write operations on a register is abstracted by ports, which can be accessed from the higher driver layer. Next to the access, pre- and post-actions for a register access can be defined, for example the indirect access to a register by manipulation of an index register.

As a first verification step, a consistency check of the register description is made. The different rules are described in [61, 90]. The language can be translated to `C` functions which provide the access to the device registers. In debug mode, assertion and log messages can automatically be included to the access stubs [92]. This approach has been compared with hand coded drivers [92]. The evaluation classifies the different errors by incorrect driver behaviour. The analysis shows a significant improvement of detecting errors in the development phase of a driver by applying this synthesis method.

## 3.4 Integrated Solutions

In this section, integrated solutions in the sense of code generators for a family of target systems and the combined synthesis of hardware and software are presented. This overview is surely not complete and is only to compare/categorise the approaches of this PhD thesis.

### 3.4.1 Hardware/Software Co-design

From the point of view of hardware software co-design (compare for example [108]), a system consisting of hardware and software is generated from a complete specification. The majority of the co-design tools focus on the partitioning between hardware and software, and are used in the field of embedded systems [18]. The system specifications can be implemented in hardware as in software. The specification is separated into a hardware part, a communication part [26, 25], and a software part. The software parts are translated to a programming language and implemented in the target platform. The hardware part is mapped to standard cells in ASIC design or Complex Logic Blocks (CLBs) in FPGA design. This is a fine grained structure in comparison to the large and relatively fixed structures of devices. The synthesis process for the communication is not simple, because of the strong dependencies to the target platform. The synthesis tools mostly focus on one part of the system and are not able to automatically establish inter-operability.

The hardware/software co-design approach can not be used for device driver synthesis. An abstract device can be specified through hardware/software co-design as an independent embedded system in the computer architecture, however, the used device hardware is coarse grained with

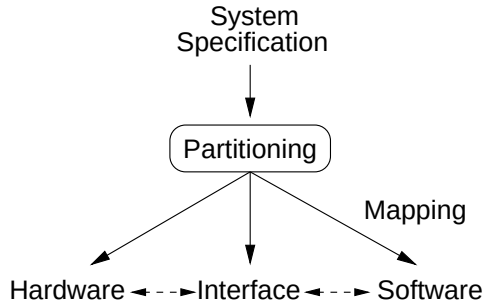



Figure 3.3:
Partitioning and mapping of a system specification to hardware, software and to the interface between them.

complex logic and automatons. It is difficult here to find a mapping between components already implemented in the device hardware and the required components in the specification.

The complexity of the communication channel is higher, because the HW/SW co-design approach is intended for embedded systems which are small in comparison to desktop computer systems. Furthermore, the approach assumes that the system is completely defined in all aspects, and that it can be broken down into a target system. A device driver is only a component of a complex system. The driver design has to deal firstly with a usage by an application, not being specified in detail a priori, secondly a more complex system architecture, and thirdly a more complex mapping problem of required behaviour to implemented control structures. In this sense a specification of a device driver is incomplete. Hence, the HW/SW co-design approach is not directly applicable to device driver design, because the abstract device is not isolated like in an embedded system.

The homogeneous refinement methodology behind SpecC [18] provides the concept of communication channels between components. They use a set of ports through which the specified behaviours can communicate. The channel has a set of functions for communication, the interface. For adaption to the channel, wrapper and transducer for channel-channel adaption are used in the specification phase. This approach can be used to specify the abstract device. During the refinement towards an implementation ”[...] the designer selects the appropriate communication protocol for the system busses during the allocation task of architecture exploration”[18]. Hence, up till now the approach can be used for specification of the communication behaviour and the behaviour itself. The adaption of both components must be done by hand during refinement towards the implementation.

### 3.4.2 Code Generators and Software Development Kits

The different chip vendors of peripheral devices and microcontrollers provide different tools. They can be classified as code generators for source code generation, library sets with parameterisers and configurators, or integrated help and wizard systems for integrated development environments. Some chip vendors provide off-the-shelf device drivers for their chip sets. Examples are drivers for graphic cards for the PC platform with Nvidia chip sets. The used design methodology is unknown.

The tool ”DAVE”(TM) from Infineon (former Siemens) belongs to the class of code generators for device drivers. The tool generates the access functions for the on-chip peripherals of the Siemens microcontroller family. The required functions for the application are selected by the developer in a wizard system of the code generator. The unwanted functions are not generated, so dead code is avoided, instead of a dead code elimination by the target compiler. The internal methods for code generation are unknown, but the author assumes that the code is generated by the use of templates and term replacement. The output language is optimised for a set of target compilers. The system is self-contained to the Infineon microcontroller family.

In [39], other tools from other vendors similar to ”DAVE” are briefly introduced. The tool

”DriveWay 3DE” is an explorer for the device and processor architecture, and can be seen as a front-end for hierarchical device documentation. The peripheral device can be explored and configuration code for the selected configuration options can be exported. The tool supports different microcontrollers. The ”ApBuilder” by Intel is similar to ”DriveWay” but covers only Intel microcontroller. The same is true for the ”MCUinit” tool by Motorola which supports only Motorola microcontrollers. All tools ”[..] can keep track of peripherals interrelations; in some cases, they can alert the developer to a conflict” [39]. The major drawback of all tools is that they only support the fixed architectures which the tool vendor has decided to support.

The chip vendor ”PLX Technology” provides a Software Development Kit (SDK) for its PCI-bridges. The development kit contains configuration tools and library functions for the access to the bridge under Windows. Applications which have to access components beyond the bridge can use the API-functions of the SDK to cross the bridge. The SDK is restricted to the supported bridges and the Windows operating system.

Other integrated development environments (IDE) like the Metrowerks `C/C++` complier [63] for PowerPC processors and microcontrollers, provide an extension in the programming language for access to the device registers. An advanced help system supports the programmer by providing detailed information on the accessed register files. As in DAVE the systems are self-contained to a microcontroller family. In both systems, the way of using the provided functions and how to use the peripherals must be derived from the device documentation.

## 3.5 Component/Object Oriented Solutions

For the design of the software side the different approaches in software development can be used. Object orientation is a methodology which is well established in the field of application design. In the field of operating systems and in hardware-near programming, the object oriented approach is not yet fully accepted. In this section the appliance of this modelling methodology and the use of the concepts is discussed.

### 3.5.1 Object Oriented Operating Systems

Modern operating systems are designed as component or object oriented systems. Examples for commercial object oriented operating systems are NextStep, BeOS [9, 107] and Windows 2000 [6, 74]. Although these systems are not purely object oriented; they still use data-structures and architectures from procedural programming, instead of full encapsulation in objects [107, 6]. Examples of research operating systems for embedded systems are PURE [10] or DReAMS [22]. The operating system is an instance of a component selection from a class library. In both operation systems the selection process is supported by configuration tools which check and resolve intercomponent constraints. A device driver is a component in the class library, like the other components in the system. The hardware abstraction layer of the driver classes is programmed by hand.

The underlying philosophy of object orientation allows polymorphism, overloading of methods, etc. This leads to an overhead due to pointer resolving during runtime and disables optimisation by in-line expansion. This requires a lightweight programming approach by the developers. Most operating systems for embedded systems are as static as possible. The required components are selected before compilation and interrelationships are statically resolved. Further appropriate programming styles, like pre-allocation of memory, reduce the overhead due to memory allocation during runtime. In the DReAMS component library [23], a preprocessor of the compiler is able to perform dead code elimination by means of static resolution.

In the PURE system, the approach of Aspect Oriented Programming [48] (AOP) is used. Management structures change dramatically in complexity depending on the cardinality of elements. It has to be distinguished between 1, a fixed number $N$ or a fully dynamic number of elements $n$ [51]. The same principle holds for polymorphism if the number of derived classes drops to 1. The tool PUMA [57] analyses the selected PURE components for the cardinal number of the used components and changes the management structures accordingly.

The TEReCS tool [11] is the configurator for the DReAMS component library with emphasis on the communication between embedded systems. The class library is extended by a dependency and relationship description. The complete solution space is included in the dependency constraints. The selection is done by specification of the communication behaviour and the components are selected in respect to the constraints. The classes required for communication and device interaction are designed and implemented by hand.

### 3.5.2 Java and JINI

The Java programming language (see [45]) includes an architecture for distributed services named JINI (see [5, 4, 46]). The services or objects with services, can register themselves at a look-up server. A user of a service sends a request to the lookup server and receives an object which provides the requested service. The minimal requirements to the runtime system are a Java virtual machine (JVM), and the Remote Method Invocation (RMI) system of Java.

The JINI system is often named in combination with device drivers. The JINI infrastructure can be used to provide a driver facade to the user. An example is a printer service. The user requests a printing service and receives an object from the look-up server which provides the printing service. This object acts as a proxy which takes the document and sends it to the printer hardware with its own proprietary protocols. The advantage is, that the device can provide the required protocol adapter component by itself. So a new device registers its driver at the look-up server as a proxy object for the device services.

JINI only provides the infrastructure for distributed services. No assumptions on the interfaces of the services towards the hardware are made. The interface towards the application can be explored with negotiation and reflection methods of component systems [40].

### 3.5.3 Programming Languages

The selection of the programming language has an impact on the design process of the software system. Nowadays `C` and `C++` are predominately used as the programming language for operating systems. Assembler sequences are still embedded in the source code as processor dependent subsystem and are highly optimised by hand. In `C/C++` machine-near operations can be expressed, but the language lacks the clear definition of bitsizes for data variables. A defined bitsize is fundamental in hardware-near programming, because the variable content must be mapped to device registers. The programming language Ada95 [15] provides a clear definition of the bitoperations in the memory. In [91] the non-adequate and inappropriate mechanisms of `C` in the field of hardware-near programming are discussed, and the DSL approach is derived (see section 3.2.2). Thus the advantage of `C` (and Assembler macros) of being hardware-near is the drawback for the programmer, because they lack abstraction and the source code becomes unreadable.

The `C++` programming language is used in some commercial operating system kernels and research kernels. The Linux developer community does not use `C++` in the monolithic kernel for performance reasons. The object orientation produces some overhead due to the look-up tables for function pointer resolving of overloaded methods. Furthermore, the dynamics in object oriented

systems leads to an more extensive use of implicit memory allocations by object instantiation (`new`-operator). In [67], an example of the incorrect use of object orientation (programming of a register as an object) in combination with a bad compiler is given. The lack of good compilers is often the disqualification criterion for the use of `C++` in embedded systems.

However, the structure of operating system source code often looks like the unrolling of object oriented `C++` code to `C`. Data-structures are passed to functions which operate on the data, which then leads to error prone pointer operations. The encapsulation of functions and data into an object reduces the risk of errors by information hiding and abstraction.

Operating systems like PURE and DReAMS show that with a appropriate use of the object oriented features, an object oriented language is suitable for operating system kernel design. Thus, when appropriately used, the high level programming languages can be used down to a level where the granularity of the language constructs is higher than the entities they are working on. Here the abstraction level must be reduced to constructs which are suitable for the given granularity. For the same reason the programmer changes from `C` to Assembler.

The compilers of different programming languages are only concerned with the mapping of instructions and data on the processor and memory architecture. They lack a knowledge of the system architecture on peripheral devices; they only mark this memory area as volatile, and only perform optimisations for the processor internal architecture. From the compilers point of view, it is not important how the information is stored in the memory. It is only required that data sent to the memory is restored in the processor register on reload without change.

This problem can be solved in two ways: information about the system architecture and the behaviour of registers over a specific period of time, must be provided to the compiler optimising methods. Here the compiler must re-construct the semantic of memory accesses or the programming language must be enriched with I/O-specific instructions that the compiler can recognise for exact manipulation of memory locations. A second way is an optimisation at preprocessor stage. Here special I/O-instructions or a domain specific language is translated to the source language of the compiler. Optimisations are performed outside the compiler which avoids cross optimisations with the non I/O instructions.

### 3.5.4 Object Orientation, UML and Design Patterns

As stated in the previous section, it has to be distinguished between a design methodology, and the implementation impacts respective of the problems with automatic translation to the target language. The object oriented approach can be used for modelling. Before implementation, the model has to be analysed if some parts have to be modified to improve the implementation quality. One approach is the use of aspect oriented programming as in PURE.

An advantage of the object oriented programming is an implicit call order: On object instantiation the constructor method is implicitly called, then the methods can be used in different order, and finally the destructor method is called before destruction. This corresponds with the initialisation and use section of the device driver lifeline.

The Unified Modeling Language (UML) [31, 30, 13] is an approach to combine the aspects of structure, data and behaviour in a joined view. The visual specification language is supported by tools [88, 33, 111], which includes a translation to OOP languages like C++ or Java. Some tools provide a re-engineering by translation of OOP source code into a UML specification. The advantage of the UML are the different views on the software project in one methodology. This includes concepts for integration of Real-Time aspects by, for instance, integration of the Real-Time Object Orient Modeling method [55]. The Real-Time aspect is given in this method by modelling the behaviour with state charts. The execution time of the automata can be determined

(compare for example [28]), and the timing behaviour of the composed system can be evaluated. Other approaches introduce the aspect of time by annotations in the sequence charts [3].

The drawback of the use of UML tools is that this modelling introduces one further translation step which can lead to further overhead by non optimised translations. For example, aggregations are translated to lists of references in both directions, even if only one direction is required. This is not a drawback in large software projects as the dual direction is often required. But device drivers must be built with high effective coding. So the UML can be used for modelling the device driver design. At present the driver designers should refrain from automatic code generation.

Design Patterns [36] are a collection of generalised solutions for problems which appear regularly in object oriented design. They are formulated as OOP models (UML) and are not in an OOP programming language. The design patterns can be used as part solutions for given specification problems. On the other hand, they can be used to describe solutions and document behaviour of the system design and architecture.

In [36], it is stated that there may be more specialised patterns for special application fields, for instance device drivers. From my point of view, no specialised design patterns for device drivers are required, because they are already in use. They are only hidden due to the use of low level programming languages. The components are implemented both in hardware and in software, so the patterns are hard to identify.

The method of object orientation, the methods of UML, and design patterns, provide a high level of abstraction. The effectiveness of the implementation depends on the synthesis of the model to target code. So the methods can be used to structure the problem and also for the design of the system architecture. So these methods will be used throughout this thesis to present and describe the approaches.

## 3.6 Summary

In this chapter an overview on the different fields and approaches on device driver programming has been given. Most synthesis methods are restricted or self-contained, in the sense that the system specification can not be expanded by own subsystems. The Devil approach is the most flexible for the generation of hardware abstraction layers. On top of that layer, an object oriented structure is suitable and feasible for device driver design.

However, a methodology for deriving the driver structure and behaviour is lacking. The literature on operating systems is too abstract on the one hand, or too concrete in the sense of describing the system API on the other hand. The behaviour of devices is described in the sense of listing registers and presenting examples of code sequences only.

In the next chapter, an approach for deriving the driver structure from the hardware topology is presented. An object oriented architecture will be introduced which can be derived from the communication model of the system components.

The channel approach of Devil is inflexible, in the sense of adaption to a new computer system architecture. Thus, the port approach is refined in this thesis to a synthesis method for a communication channel out of channel segments. The segments can be described individually by the use of attributed grammar. The adaption to a new architecture is performed by rearrangement of the segments and a synthesis of the channel.

A method for deriving the driver behaviour out of the device behaviour is also not covered by these approaches. One chapter of this thesis focuses on this topic by refinement of the device internal interaction structure. Here, the influence of the communication channel is taken into consideration and an order of the messages over a specific period of time can be suggested. In case of the interrupt system, the refinement can be proceeded to a level where software can be

synthesised and the configuration parameters can be determined. The required specification is composable.

The last chapter of this thesis discusses some issues on security and appliance of compiler techniques, in the field of device driver development.

# Chapter 4

# Approach for Device Driver Design

The aim of this approach is to provide an object oriented architecture, in combination with a structured procedure for the development of a device driver. The main focus is to provide a method to structure the problem of device driver design, thus reducing the development time and the risk of errors. The result is not optimal in the sense of speed or memory consumption, as these goals are treated as secondary aspect compared with development time and guaranteed quality.

Furthermore, this approach should enhance the understanding of device drivers and help to find a starting point for the design of a device driver. To gain a better understanding of the hardware behaviour, the similarities and analogies of hardware devices and software are discussed first. The hardware behaviour is described from the perspective of object oriented design.

The different operating system kernels use different interfaces to a driver, which interestingly enough are somehow similar [112], as well as different hardware channels from the driver to the device hardware. In the different approaches of microkernels, monolithic, and object oriented kernels, the perspective of a driver as an adaptor/mediator between application and hardware is the best abstraction.

In this approach the structure of the hardware components, depending on their communication relationship, is analysed. From the analysis results, the software structure and interfaces are derived. This leads to an object oriented architecture, a pattern, for the structuring of driver components. The structure of the software components is a mirror of the hardware structure. Hence, the hierarchy of the hardware components is taken into consideration, instead of the flat model which is used in the Uniform Driver Interface approach [113]. This leads to an coarse grained architecture for the driver.

The driver behaviour depends on the hardware behaviour and the demanded total behaviour of the combination of driver and device. This can not be derived from the hardware structure, so hints for the analysis techniques of the hardware and derivation of patterns in the driver design, are only possible at this stage. In this chapter the reflection of the hardware communication structure is used for structuring inside the driver. The access to the device registers is wrapped by an abstraction layer. The next chapter discusses the problem of channel adaption for the register access in hierarchical structured system busses. Chapter 6 returns to the driver kernel. In that chapter design hints for the driver behaviour are derived for the device internal behaviour.

The methods of object orientation and design patterns are used to structure the problem in this approach. They are only used as a model. For effective implementation purpose, in some cases the model must be left and flattened by hand.

## 4.1 Intercomponent Communication

In this section the behaviour of hardware devices from the perspective and in the terms of the object oriented design, will be discussed. Following this abstraction, the object-oriented driver architecture will be derived in the next section.

In most computer systems the devices are attached to a system bus and show some registers towards the CPU. The communication is performed only by passing information through these registers. The registers are located in the device controller, and can be classified by data-in, data-out, status, and command [84, 64, 70]. They act as a shared dual-port memory between a process on the CPU, and the device automaton (see figure 4.4). The model of the device will be extended in this section.

In this thesis, the perspective of a register as communication interface for a device is used. The registers are one subpart of the device, and they are organised in one register file. The hardware designer can provide interfaces for different peripheral busses, for example the FireWire Linklayer chip TSB12LV32 [109] provides a bus protocol acceptor for the Motorola 68000, as well as for the ColdFireE bus protocol. This feature is provided by implementation of different bus protocol accepting automatons, which control the access to the device registers.

This leads to the first level of organisation, the split of the device into the components device behaviour (device kernel), and the register file including the bus interface (see figure 4.1). The register file includes as a sub component the communication to the bus system, in most cases an accepting automaton for the bus protocol. The device behaviour, the *device kernel*, has parallel access to the register file[1].

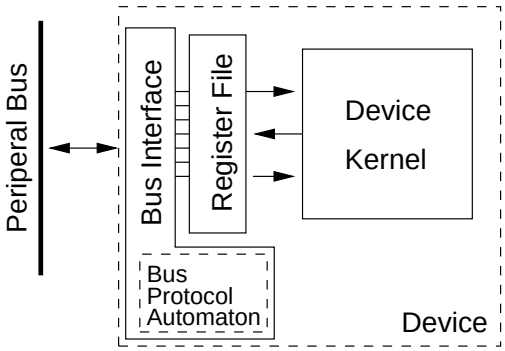



Figure 4.1: Devision of a device into the bus-interface, the bus protocol automaton and the device kernel.

In the model the information is exchanged between the register file and the device automaton by signals which carry abstract symbols. These abstract signals are named in the following as *Information Entities* (IE), whereas the abstract symbols represent the state of the IE. In the Devil-approach [110], the term *device variables* is used. The binary coding of the symbols of an IE is represented by a bit-field. The bit-fields are mapped to registers. For ease of organisation, they are not arbitrarily spread out in the registers, but are in most cases consecutively embedded in the register file. Bit-fields are split and distributed across registers if the size is larger than the register size. In most cases, the register size is a multiple of $2^n \cdot 8$ bit ($2^n$ byte), oriented to the databus width of the peripheral bus. In case of a parallel data bus, a mapping from a bit position in the register to a data bit on the bus lines can be provided. This mapping can depend on the access protocol which is actually used on the peripheral bus. A register file is a regular organisation of

[1] in some cases a data-path register is connected to a FIFO-buffer

the registers. Inside the register file each register has a unique identifier, the register address. This address corresponds with parameters on the peripheral bus, which addresses a dedicated set of bits in a set of registers.

Manipulation methods of the IEs, have to be mapped to manipulation methods of the bits in the corresponding bit-field (see UML diagram in figure 4.2). So, two translations have to be performed: A coding of the new IE state into a bit-pattern and the storage in the corresponding bit-field in the register file.

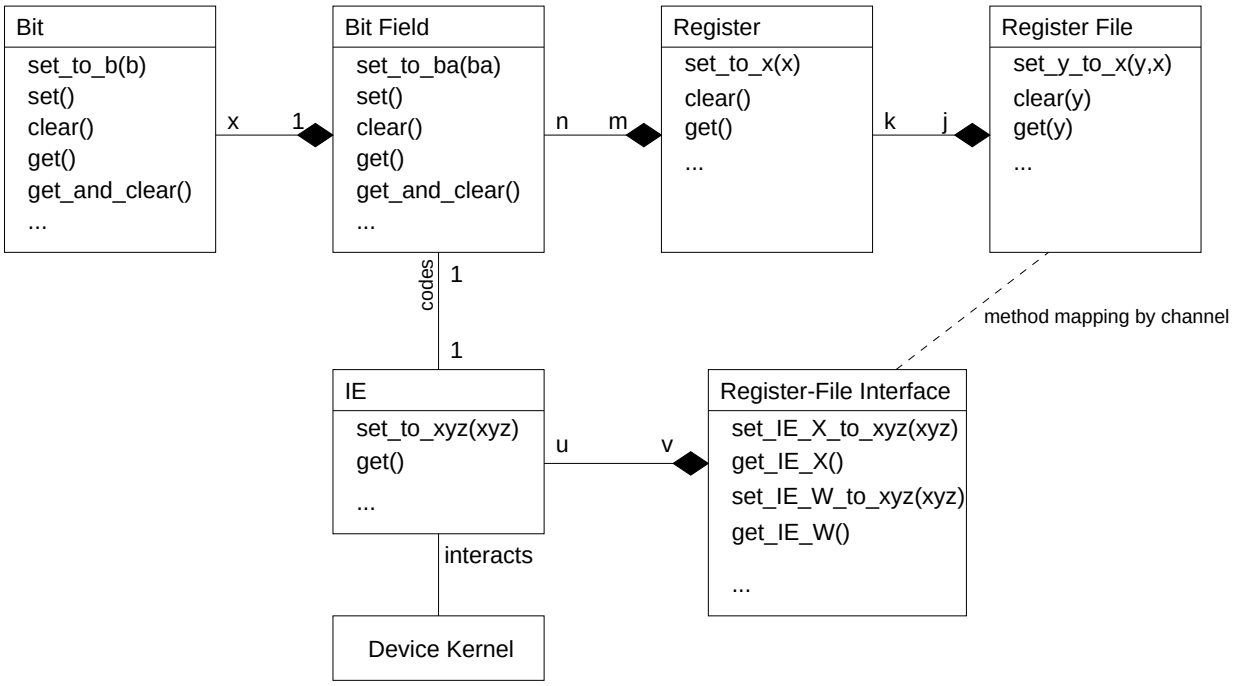



Figure 4.2: Structuring of bit fields and registers with examples of manipulation methods. The IE are encodings of the bit-fields.

The aim of each transaction between CPU and device register file is the exchange of the state of an IE, or in object orientation perspective, the call of a method of an IE. On the lowest level, the transaction is a manipulation of the content of a register, or a transfer of the register content to the CPU internal registers.

The Devil approach [61] uses a similar description of the device communication, whereas device variables and the coding of the registers are modelled [90]. In the approach presented here, the bit-fields of the registers which represent the device variables are coded and the registers are manipulated by abstract *ports*. With the port functions the register content can be read or written. The functions cover the complete channel from the driver software to the device register. For a migration to a different platform the port must be adapted to the new channel architecture.

The abstraction of the communication channel and its interaction is an important factor for the hardware abstraction layer, and will be explored in greater detail. The basic question is how the channel interface on the CPU side must be manipulated, so that it results in the correct manipulation of the register without side-effects? The answer depends on the architecture between the CPU and the device, as well as the software interface to the system hardware inside the CPU.

From the object oriented point of view, the autonomous device can be modelled as an active object. The register file acts on first glance as a shared memory of the device object, and can be modelled using public variables. The content of the variables is manipulated by the methods

provided by the system bus interface. As discussed in 2.3.2 the access to a register can produce events towards the device kernel. The events are the result of the recognition of sections of the bus protocol. One event is the access to the content of a register. So in principle, the device kernel is able to track each interaction with the register file.

In software, an access to variables can only be recognised by encapsulation of the variables as privates attributes of an object, as well as the access methods to the attributes. The access is recognised by the object, and hence by the modelled device object, by means of the method call (see sequence chart in figure 4.3). Inside these manipulation methods of the register file, the translation of the register content to the corresponding symbol of the IE is performed. This is only a model of communication and should not be intermixed with an implementation.

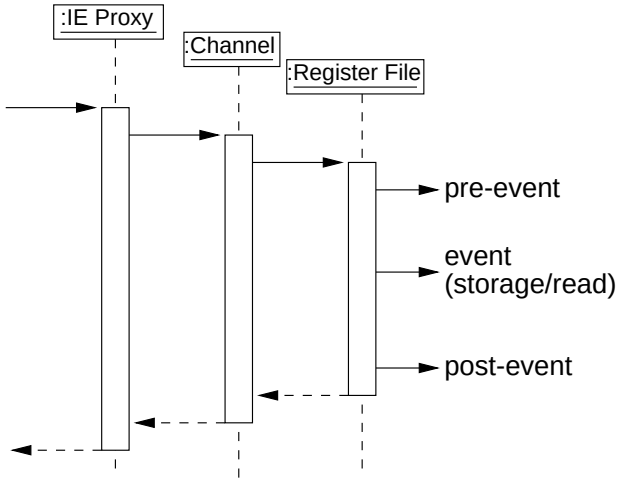



Figure 4.3: Sequence diagram of the ”method calls” in proxy, channel, and the register file. In hardware, the events are the result of the recognition of protocol sections.

The methods of the active device object are not invoked directly by the driver object. The driver calls the methods of the communication channel which invokes the method of the device object (see figure 4.3). The former question of how to manipulate the register, now becomes the question of which channel method with what parameters must be used to result in the desired manipulation of the registers.

On the lowest level, the register bit-field manipulations are implemented in the bus protocol automaton of the device, and the register method calls must be translated to protocol sequences. The protocol sequences are initiated by the CPU on an access to external resources. The register file of a memory mapped device is embedded in the system memory, so that the CPU must initiate memory access sequences by the correct sequence of software instructions. In the C programming language a data-structure can be mapped to the memory location of the register file. A memory access to elements of the data structure is de-referenced to an access to a register. The structure is designed in such a way, that the memory map in combination with the translations of manipulations in the data-structure by the C-compiler ,result in a correct manipulation of the register file. This requires correct knowledge of the complete memory architecture, the embedding of the device registers, and the translation scheme of the compiler for data-structures and types, to memory locations.

From the hardware designer’s point of view, the driver can be modelled as an automaton with its own control-path and data-path. The device automaton, the device kernel, has parallel access to the shared information in the registers. From the peripheral bus side, a non-parallel communication

with the driver automaton is enforced by the time multiplex on the bus systems. The sequential nature of the CPU results in a sequential access to the IE on the software side, the driver side.

From the object oriented point of view, the driver and the device can be modelled as active objects which communicate by the adaptor object, the register file. In both directions two types of information are exchanged: the content of a register, and implicitly the event of an access and the point in time of the accesses. The active object of the driver can explicitly be informed of a change in the register content only by a notification method, which is mapped to an interrupt request in the real system (see 2.3.2).

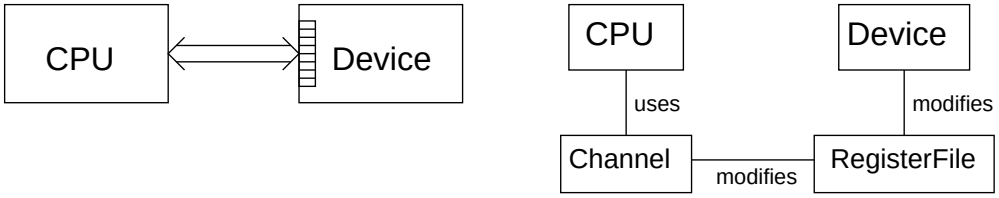



Figure 4.4: CPU, device, and communication by a register file, as depicted by a block diagram, and as active objects.

The object oriented view of the IE can be conserved on the driver side of the channel by introducing a proxy [36] for the register manipulations. The implementation of the proxy performs the register manipulation, by means of calling channel methods. Towards a more abstract level, the proxy should not provide register manipulation methods. On the contrary, it should provide methods for access to the IEs. They have to be mapped to register manipulations and hence to channel manipulations. The proxy objects for access to the registers work as adaptors for transparent communication with the dedicated device component. This model is similar to the JINI-approach [5, 4] where the application communicates with a proxy object, and the proxy communicates somehow with the target object which provides the services.

Here the structure is organised as in figure 4.5. The access to the device registers is provided by an object that implements the interface with the manipulations of the register. This interface is used by an object which communicates with the device by register manipulation. The granularity of register manipulations is bits and bytes, whilst by object oriented programming it is coarse. Hence, this model of object orientation on registers should not directly be implemented [67].

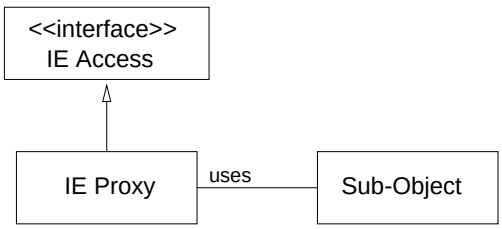



Figure 4.5: The driver object uses a proxy for the IE access. The IE proxy implements the interface with the required channel adaption for the access to the IE.

The driver on the software side, and the device on the hardware side, can be separated, analogously to the ISO/OSI reference model, into two layers: the communication with exchange of shared information, as well as the behaviour on both sides. The behaviour is encapsulated in the component driver kernel, respectively the device kernel. The communication layer encapsulates all

sub-layers required to access the physical communication layer. It provides transparent communication between both behaviour components. Without a direct channel from CPU to the component, each transformation of information by a component on the path must be taken into consideration. This is similar to the ISO/OSI reference model in the presence of bridges and gateways, where the influence of the bridges must be considered by the sender.

At this point, the driver is separated into a communication proxy object, and an object which implements the driver functionality for one hardware component. The communication channel is abstracted by an adaptor object to the device register file, which itself can be abstracted to an active device object. The next section discusses an approach to structure the driver on the communication level, between hardware components and the driver. The channel adaption will be elaborated further when the channel software synthesis is discussed in chapter 5.

## 4.2 Coarse Grained Driver Structure

The previous section has discussed the separation in the concerns of a device behaviour, the register file, and the communication channel, with a proxy as interface on the software side. This approach can be used as a pattern to structure the device driver, analogous to the structure of components in the device. Nowadays, peripheral devices are not a single component, they are a combination of different entities. The driver must cover the complete structure.

The hardware components in a computer system interact by exchanging information on a physical level. The information is transformed inside the components and routed to other components. The structure of information flow is visible from the outside on board level or in the case of communication between systems on one chip it is not visible. The structure of interacting hardware components can be reflected by a structure of interacting software components in the driver.

### 4.2.1 Driver Object Structure

The structure of the software components can be derived by reflection of the hardware components communication-graph, in combination with the communication paths. All components which contains registers have at least one corresponding object in the driver. They communicate with their dedicated hardware component by register access. The access is provided by a proxy that implements an interface for the register manipulations. Accordingly, the hardware components are sub-components of the device. The corresponding software objects are named *sub-object* of the driver. The interrelationship of the objects can be derived from the communication relationship.

Most registers are not directly accessible from the CPU. The access is routed through some intermediate components on the path between the CPU and the device. Hence, the route of information flow in the driver objects is analogous to the information flow in hardware components.

A communication proxy requires communication services from a lower-level sub-object. The lower-level sub-object is the counterpart object of the next component on the route from the hardware device towards the CPU. This set of communication methods can be modelled by an interface provided to upper level components (see figure 4.6). It can be seen as a contract between both objects.

From the perspective of a top-level refinement method like SpecC [18], the pattern of sub-component, sub-object and proxy, is a refinement of the channel between proxy and device component. In figure 4.7, the proxy adapts to the requirements of the channel, and the register file moderates between the channel and the device. The channel can be refined by a sub-object with its proxy which adapts to channel and a register file of the sub-component. The sub-component is connected by an adaptor and a further channel to the main target component.

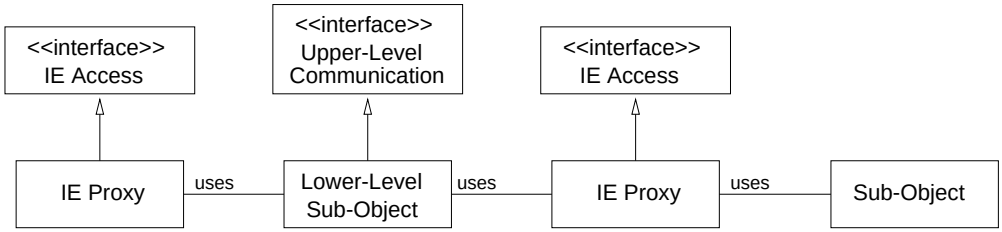



Figure 4.6: The sub-object uses communication services of a lower-level sub-object whose component lays on the route to the own component.

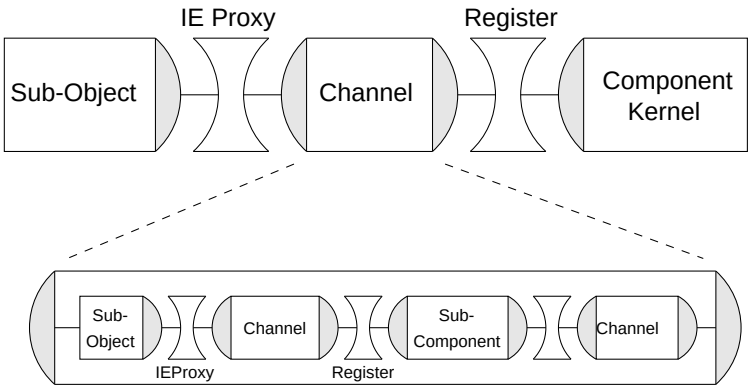



Figure 4.7: SpecC-chart [18] using a sub-level component as a refinement of the channel.

As stated above, the driver sub-objects have to provide an interface for communication with components beyond their own hardware component. They can provide different interfaces depending on the required speed and trustworthiness of the upper-level components use. One interface requires in the implementation, a test of each parameter which should be send to the component. An other interface does not carry out a test, but provides a fast and direct access by, for example, memory re-mapping. Thus, the pattern for communication with the use of sub-level sub-objects leads to a use of driver sub-objects by the proxies. The proxies implement an interface for register manipulation which is used by the driver object. This object implements an interface for a communication channel for an upper level proxy (see figure 4.8). The selection of the used channel depends on the demands of the upper level sub-object.

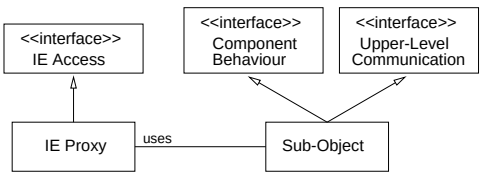



Figure 4.8:

Proxy must implement the register file access and the access to components beyond the component by the bridge interface.

From the perspective of an upper-level sub-object the lower-level sub-objects are wrappers for the communication channel to their hardware counterpart. It is an abstract device driver inside the driver, a *micro-driver*. The separation into the concerns of communication and behaviour for a micro-driver is in some cases not clear. For instance, it has to access its hardware counterpart by a protocol in order to route information to the next hardware component. This protocol automaton can be seen as part of its proxy, or as part of the behaviour. In total it is an adaptor to the channel and the hardware component, providing a transparent channel to the components beyond.

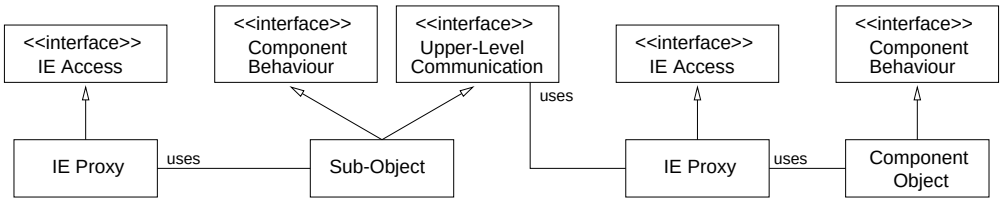



Figure 4.9: Use of the Upper-Level Communication interface by an object higher in the communication hierarchy (compare figure 4.8).

The communication channel between the proxy and the hardware component is ”stateless” in the ideal case, which means that two consecutive transactions do not influence each other, or change the channel state, in the sense that the next transaction must take the previous ones into consideration. This assumption holds on most memory bus systems. In the other cases, especially by communication over components, the transition in components states must be taken into consideration, because on the outside it is a transition in the state of the channel. This means a more complex protocol in the proxy has to track the change of state in the sub-level component. The required state for the transaction has to be re-established, or the interaction with the channel has to change according to the new behaviour. For example, a transaction to the local bus over the PCI-bridge has changed the address remapping parameter. Therefore, the next access has to change the remapping parameter before the access, if the desired access is outside the remapping window.

The model shown for the driver structure does not cover any aspects of optimality in the sense of memory consumption or speed. It is a model to start the driver design with. The approach of structural reflection provides a first design rule for the coarse grained structure of the device driver. Each sub-component has to be refined according to the behaviour of its counterpart in hardware and the desired total behaviour. The model provides a separation of concerns of the communication, the sub-object behaviours as wrapper, and the communication to other components as part of the required behaviour. The aspects of communication and total behaviour are discussed in the next chapters. However before this, the next sections discuss the influence of the operating system kernel architecture and the device identification procedure.

### 4.2.2 Device Identification and Driver Structure set up

In this section the identification process of device components and the set up of the driver counterparts are depicted. The discussion starts with the procedure for a static component structure, and is performed at boot time. Afterwards the other extreme of hot-pluggable components is evaluated. Reconfigurable devices are a middle course between static and dynamic systems. Because they do not change as suddenly as hot-pluggable devices, a different strategy for set up of the driver structure can be used. The section concludes with a brief view on further management structures.

**Static Component Structures** One objective of a device driver is the clear identification of the hardware device to guarantee compatibility with the driver code. The driver is a *facade* (compare for the pattern [36]) for an internal structure. The facade is a collection of the interfaces of the encapsulated objects. The encapsulated objects are facades of their assigned hardware components. In the architecture presented here, each object represents a hardware component, and hence, the driver structure must match component by component with the hardware component graph. Unfortunately, in most cases a match as a whole is not at once possible, and can lead to system freeze or other system crashes (compare section 2.2.1). For a match as a whole, too many assumptions

on the system hardware architecture have to be made, which may not hold and then result in the wrong, fatal behaviour of the hardware during identification test.

In most cases, a hardware device is not identified by only one characteristic. In the Devil approach [60], a hierarchy of characteristics can be defined. The hierarchy is transformed to an if-then-else tree, which is followed during the identification procedure, and a hardware pattern is tested. At the leaves of the decision tree, flags which represent the identified system are set. These flags are used within the driver to branch to the required software sections according to the identified device.

In Windows 2000 a tool tests during the installation procedure the system hardware and establishes a mirror of the hardware topology. In the registry of Windows each hardware type can be assigned a driver. After identification of the topology, the drivers and their filters are loaded and stacked according to the device topology. The identification process is not described in detail in the literature, because the driver programmer does not have to take it into consideration [74, 6].

The identification process requires for each component a reflection mechanism to determine the type of the component. This reflection mechanism must be stable, in the sense that the system does not crash or freeze through this identification procedure. The already identified components have to provide information about the availability of next level components for further identification. Similar to software, the components must provide reflection methods about the availability of the next level components, or the inner structure. The other way is to guarantee the availability and the structure by one key value. This value must be unique to that structure. This key must, furthermore, distinguish between different revisions of the component, as a different revision means different behaviour.

The identification of hardware components and the instantiation of drivers can be combined by the use of identifying objects and *object fabrics*. The fabrics are objects which delegate the instantiation of objects to sub-classes (compare for the pattern [36]). Thus, a software component, which is responsible for a dedicated hardware component, identifies all hardware elements beyond its own counterpart. Depending on the type, an object fabric instantiates the required counterpart for that sub-component, and binds it to the hardware (see figure 4.10). The process can continue recursively until a leaf in the hardware topology is found. The result is a structure of objects which reflects the topology of the hardware components. Identical components are modelled in software as classes, and are instantiated in the number of real existing components. Their communication proxies binds them to their counterparts in hardware.

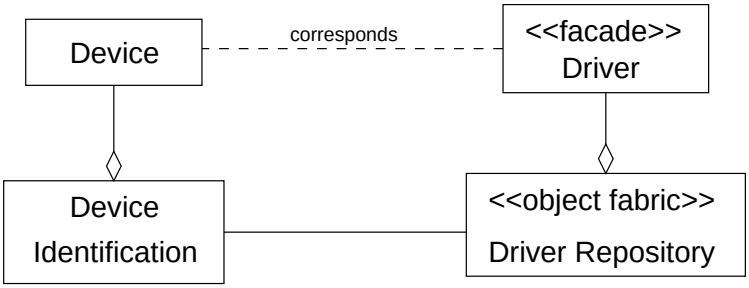



Figure 4.10: Device identification requests driver instantiation by an driver object fabric.

Still, if the topology can be identified by the root component of a structure, the recursive approach can be replaced by an object which unfolds the structure with a specialised object fabric. In this case, the expected depth of the topology is limited. The root object can act on top of that structure as a facade for the internal structure.

However, the root object of a structure has the further objective to configure the component, and to control the interaction between them. The configuration of the interaction cannot, in most cases, be initialised by each component itself.

The stepwise identification requires that software components are linked during runtime, which can lead to an overhead due to function call sequences. This can be avoided if the facade with its internal structure, is built in a monolithic and static manner allowing in-line expansion and early binding. On the other hand, it has to match the hardware structure as a whole. This leads to a separate descending identification of the hardware and on a match an instantiation of the facade as a whole.

**Hot-Plugging** In the case of hot-plugging of devices the resource assignments may have to be reorganised. The removal or insertion of a hot-pluggable device changes the topology of the hardware. Different components in the driver, inside the kernel, or in the user space, have to be informed if the change of a value or structure is visible to them. Here the observer-pattern (compare [36]) can be used to inform these components. Components have to be informed about an event register itself, as a listener to the event, by an object which receives the event as the main receiver. It distributes this event after reception to all registered listeners.

If devices are removed, sub-trees in hardware are not longer available and hence, the corresponding tree in software has to be also removed. Vice versa for the insertion of a sub-tree, the tree has to be identified and the driver structure has to be established analogously to the boot sequence.

Other driver objects have to be also informed about the change, as the binding of the proxies to the communication channel may change. Here, only the resource mapping tables have to be reorganised according to the changes in the system (addresses, interrupt lines, etc.), and not the object structure. The binding between the sub-objects and their corresponding hardware components should remain if the hardware is still available. The system reorganisation only requires the adaption of parameters for the channel to the hardware component in the proxy. Nevertheless the changes have to be identified and therefore the sub-object must re-identify its hardware component. If the structure of the system changes completely, a unique ID, for example the 48 bit MAC-address of Ethernet cards, of the device hardware can ease the reorganisation. The ID is stored in the resource management tables (cookies), and the reassignment of parts of the management structure can be performed according to the unique resource ID. The strategies of reorganisation depend on the mappings of identifiers in the system, and the interpretation in the system (topology, address space, routing, etc.).

**Integration of Reconfigurable Devices** Reconfigurable devices, or more specific reconfigurable peripheral devices, can change their behaviour in total by downloading a new configuration to the flexible hardware, for example, a FPGA. Along with the reconfigured hardware behaviour, a driver on the software side has to be established. This behaviour is the middle, between the static structure and the fully dynamic hot-plugging structure. In a hot-pluggable environment the hardware structure can suddenly change. In the case of reconfigurable devices, the change is initiated by some component which can inform affected components immediately before downloading the new configuration.

Different strategies are possible for the integration into the driver architecture. The reconfigured device must contain a protocol adaptor for the communication to the previous component. Analogous to the driver structure previously discussed the driver component of the reconfigurable device uses the services of the sub-object for the access of its hardware. The downloading software must instantiate the required software components and bind them to the appropriate sub-objects.

A reconfigurable device can be delivered as an IP[2] core in the sense of an isolated device kernel. This device kernel is embedded into a wrapper with integrated register file and bus protocol automaton before downloading. Analogously, on the driver side the proxy for the register file is instantiated. The linking of the device kernel interface to the registers is hidden by an adaptor for the driver kernel. The resulting structures are depicted in figure 4.11.

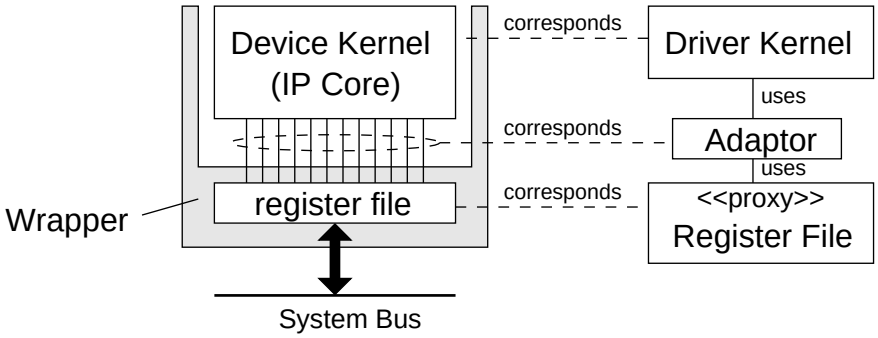



Figure 4.11: Encapsulation of an IP Core by a wrapper with integrated register file. On the software side, corresponding structures are established.

Again the reconfiguration is the same as by hot-plugging, only that the initiator can inform other driver parts about the change. From this, the requirement can be derived, that the component which is responsible for the reconfiguration provides a structure analogous to the observer pattern. It informs all listeners about the deferred reconfiguration and that the device is (temporarily) not available. Afterwards it reassembles the driver structure and notifies all listeners about the finished reconfiguration.

**Housekeeping** A device driver covers a sub-graph of hardware components. On the other side, an object has to manage all device drivers which are available in the system. In the case of many instances of boards in one computer system, a further *majordomo* object has to do the housekeeping. It handles the management of a special device class, for example the class FPGA-boards.

The majordomo object is informed by the object fabric (or is the object fabric itself) on the newly established board facade. A link to this driver is set in the majordomo resource table. Thus, the majordomo object is an interface for the kernel or for the application to a special device class. It provides reflection methods to get information on the available resources. Similar to the reflection methods of the device objects in BeOS [107], it provides reflection methods to provide device identifiers and selectors for the application.

In the second architecture, the majordomo object is an object fabric itself. The major object fabric is a decorator from subordinate object-fabrics. It delivers the request of the instantiation of a driver component to the appropriate object fabric. The majordomo object is inherited from the object fabric, because the object fabric and the majordomo object are responsible for the same device class (see figure 4.12).

Both architectures are suitable. It depends on the design philosophy of the remaining system. In both cases, an application must have access to the majordomo component, which may require a further management object for the majordomo object of the different device classes. In the BeOS system the application instantiates a proxy object for the device class. This object provides reflection methods for information of the available devices. The application can select one device and bind it to the device. The connection to the majordomo object is done automatically by the

[2]Intellectual Property

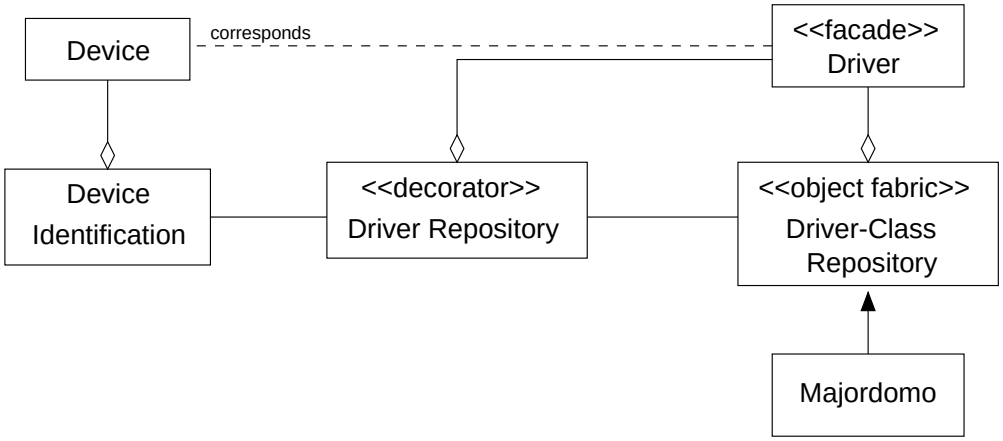



Figure 4.12: Driver request is dispatched to the suitable object fabric for the device class.

proxy [107]. The proxy is required on the user side of the architecture due to the user interface of the OS kernel. This channel inside the OS kernel is discussed in more detail in the next section.

The system of majordomo objects can be applied recursively to sub-components in the driver object hierarchy. For example, a majordomo object is the interface to the available devices of a defined device class. It can provide a reference to a selected device through a reflection method. The referenced object itself again provides reflection methods to get a reference to a sub-component. Thus, this housekeeping is not associated with only one layer in the device driver hierarchy.

### 4.2.3 Influence of the Operating System Channel

In this section, the bridging of the border between user space and kernel space in the object oriented driver architecture is discussed. Most operating systems, for instance UNIX, separate the processing and the memory access into a user space, and a kernel space or superuser space. This concept allows access to system critical operations and services, like access to hardware resources, only for processes in kernel space respectively kernel mode[3]. By gate-functionality, processes from the user space are able to call functions (system calls) in the gate, which redirect them to dedicated components in the kernel space. The access rights are checked and the function is executed with permission. This procedure is similar to Remote Procedure Calls (RPC). The procedure call is packed by a stub, sent through the network and is unpacked on the target node and the final procedure call is invoked. In the operating system, a system library function packs the call, sends it to the gate, and inside the gate the call is unpacked and executed in kernel mode.

In [112], the gates and interfaces to drivers of UNIX-like OS, VxWorks, DOS and OS/2 are compared to each other. In all operating systems, the kernel together with the driver, dispatch the system call of an application to a dedicated service routine inside the driver. The level of dispatching varies in the different kernels. The pattern used is the same in all architectures: a call of a user process is embedded into a command code and a data-structure. The system changes into kernel mode and dispatches according to the device class, the opened user channel, and the command code to an assigned callback function of the driver. Inside this callback function, the data-structure can be interpreted (for instance in `ioctl`-calls [104, 95]) again as a command and with data for further dispatching. This is similar to the command-pattern in [36], whereas here structures are used and the flexibility is much lower than in the described object oriented pattern.

[3] Another term is *superuser mode*

The kernel can be seen as an intermediate level, which helps to organise the different drivers, the assignment to devices, and the dispatching to the requested functionality and device instance. Therefore, the superuser status is required to get access to the system hardware. Hence, some parts of the driver must lay inside the kernel space. However, the driver behaviour, the driver kernel, is not required to lay inside the kernel space; it can be placed inside the user space as a driver support software.

Typically the lowest level of a device driver requires a direct access to a memory location with low-level functions in kernel mode. In Linux, for example, they are covered by system dependent assembler macros. Other kernel services like allocation of DMA-able memory, or registration of interrupt services are only available in kernel mode. On the other hand, a rule of thumb for device driver development, is to do as much as possible in user space [95], because debugging in user space is much easier than in kernel space. A normal debugger can be used and the driver works under memory protection.

As discussed above, the kernel is an intermediate structure for a command-interpreter pattern with a gate for passing the messages. All operations in kernel mode must be separated from user space calls this type of channel. The packing of commands and data (system call), and the dispatching including copy of data from user space to kernel memory space, requires much time. So the transition must be placed according to the frequency of calls and the size of information which is passed. Frequent communication should not cross the border between the kernel space and the user space. In the placement of functionality within user space and kernel space, a balance between reduction of system calls and eased debugging has to be found.

For separation of the driver in a kernel space and a user space part, the driver has to be examined for services, which are required to operate in kernel mode. These must be located on the kernel side. Between the kernel and the user part a wrapper has to be placed for the kernel call-gate channel. A proxy in user space is the stub for the kernel channel, and inside the kernel the messages are unpacked and delivered (see figure 4.13). In contemporary UNIX-like operating systems this technique is not used in that much clarity, as the user application has to pack the method calls itself, for example the `ioctl`-calls (see [104]). Hence, from the driver design point of view, the kernel becomes a provider for special services and a communication channel for method invocation in kernel mode.

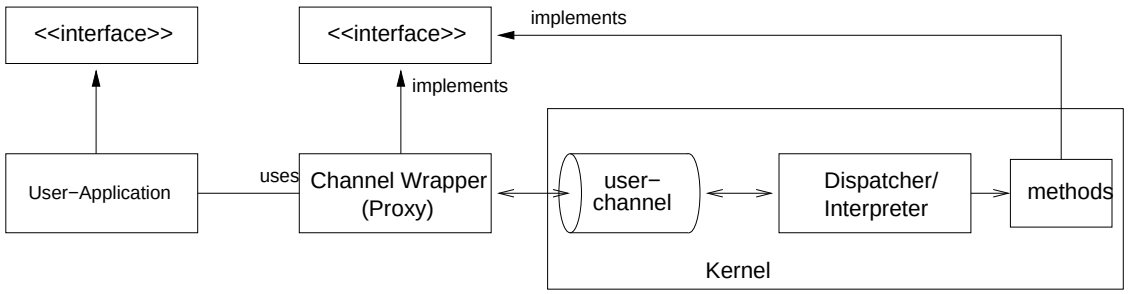



Figure 4.13: Method calls of the driver are packed by the proxy stub into a system call, passed to the kernel, and dispatched to the desired driver method.

The location of the kernel channel depends on the required kernel service routines and the frequency of system calls. Due to the command and dispatcher pattern, the user/kernel boundary can be located at different positions of the driver interface. Typically it is located at the interface to the driver facade, because in most cases it is the highest level interface with the least invocations. Thus, the effort for data copying and system call packing and unpacking at its lowest.

A further point must be taken into consideration for the partitioning decisions: if parts are located in user space, they are bound to the application. That means, the informations stored in the user space proxy are only visible inside the application and is not shared with other applications. Furthermore, the informations last as long as the applications only. If the informations have to be shared between multiple users, or if it has to be available as long as the driver is working, extra care has to be taken or the information must be placed inside the driver. The driver is only instantiated in the number of fractions of hardware resources it can handle, at minimum once only. Again, for the partitioning the question of 0,1,$N$ and the importance of the lifeline, becomes more conspicuous.

## 4.3 Example

The previous approaches for the internal object structure of a driver will be depicted in this section on an example device, an FPGA board. The object diagrams are only to illustrate the discussed approaches, and are not complete in the sense of a specification/documentation for the actually implemented driver.

The *Raptor I* FPGA-board [47], named in the following Raptor-board, is an FPGA board equipped with two Xilinx XC4062 FPGAs, a dual port SRAM and a Complex Programmable Logic Device (CPLD) for configuration of the FPGAs and bus arbitration (see figure 4.14). All components are connected to a local bus on the board which is connected to a PCI bus by a bus bridge PLX9080. The structure of interconnections is depicted in figure 4.15. All components except the FPGAs have fixed register files, which in principle can be manipulated by the CPU. The FPGA can also have registers, it depends on the actual configuration of the FPGA, the implemented hardware design. The FPGAs occupy an address space on the local bus, thus they have *hypothetical* registers. After a design is downloaded to the FPGA, they have to be mapped to the real registers by the application. The PCI bridge has a special facility of memory remapping. The PCI memory spaces `base address 0` and `base address 1` of the bridge are directly mapped into the local-bus address space. In this model the registers are all memory mapped, so a distinction between registers and memory (locations) is not made. Hence, the mapping of bridge address areas is also modelled as register files.

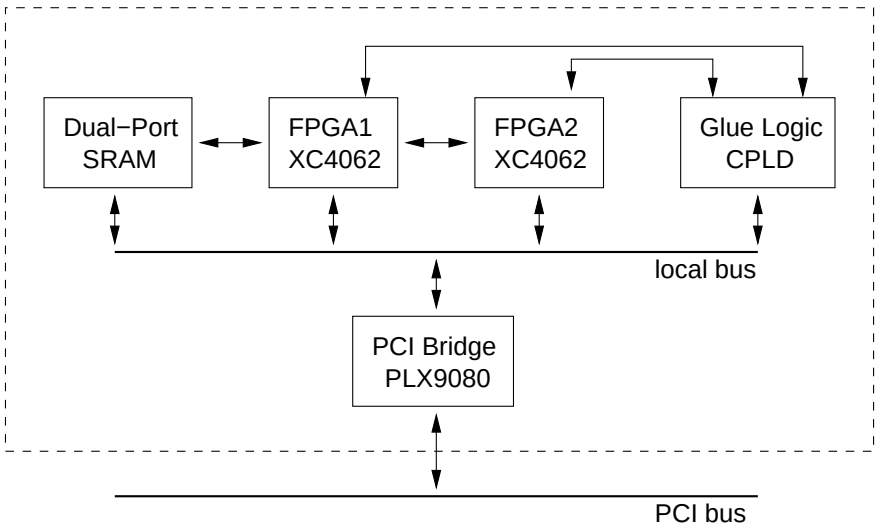



Figure 4.14: Block diagram of a Raptor-board. The register file in the FPGA depends on the implemented design.

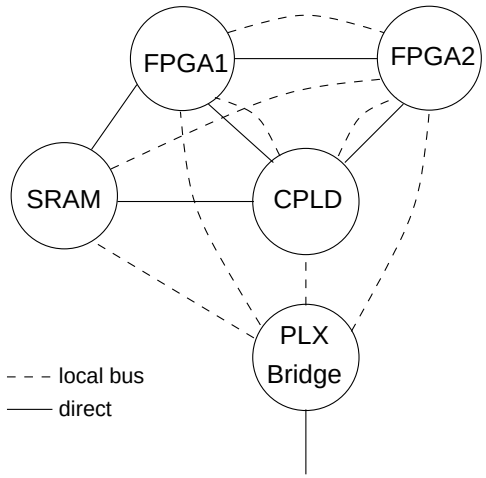



Figure 4.15: Communication interconnection graph of the Raptor-board. Compare figure 4.14.

For example, the communication proxy of the CPLD must use the communication methods provided by the PLX bridge sub-object. Another example is the configuration of the FPGAs. The bitstream follows the route over PLX bridge and CPLD to the FPGA. Hence, in the driver the bitstream is routed from the FPGA sub-object, which contains the download method, over the CPLD sub-object and the PLX bridge sub-object. The route is mapped to the software sub-objects with inverse direction. This leads to the software structure depicted in 4.16. Hence, the software components must provide a communication channel for interaction with hardware components beyond itself.

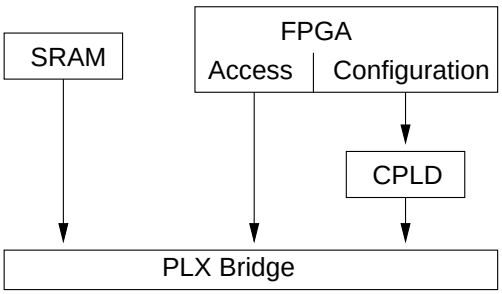



Figure 4.16: Structure of driver components by reflection of the communication routes.

In figure 4.17 the object structure of a driver for the Raptor-board is depicted. The FPGA objects have an interface for downloading a FPGA-design to the FPGAs via the CPLD. The CPLD implements an automaton for streaming the configuration data to the FPGA programming interface. The FPGA objects wrap the functionality of the CPLD. For an object which uses the programming interface of the FPGA objects, the FPGA objects act as micro-drivers to translate the configuration data into the packages for the CPLD. Here this communication behaviour is packed into the configuration concern of the FPGA objects as part of their behaviour specification.

For the identification process, the system must provide clear information on the hardware components to follow the search tree. The problem will be discussed in more detail on a second type of FPGA-board, the Spyder-board. The Spyder Virtex FPGA board (Spyder-board) [118] is similar to the Raptor-board (compare section 4.3). It has a local bus with one Xilinx Virtex FPGA, a CPLD as glue logic, and a PCI bridge by PLX Technology connected to an on board local bus. The two memory banks are only accessible through the FPGA (see figure 4.18). The PCI bridge is the PLX9080 on Spyder-boards release $< 1.3$ and PLX9054 for release $1.3$. Thus from the point of view of the PCI bus, the Spyder $< 1.3$ and the Raptor-boards are the same by bridge type.

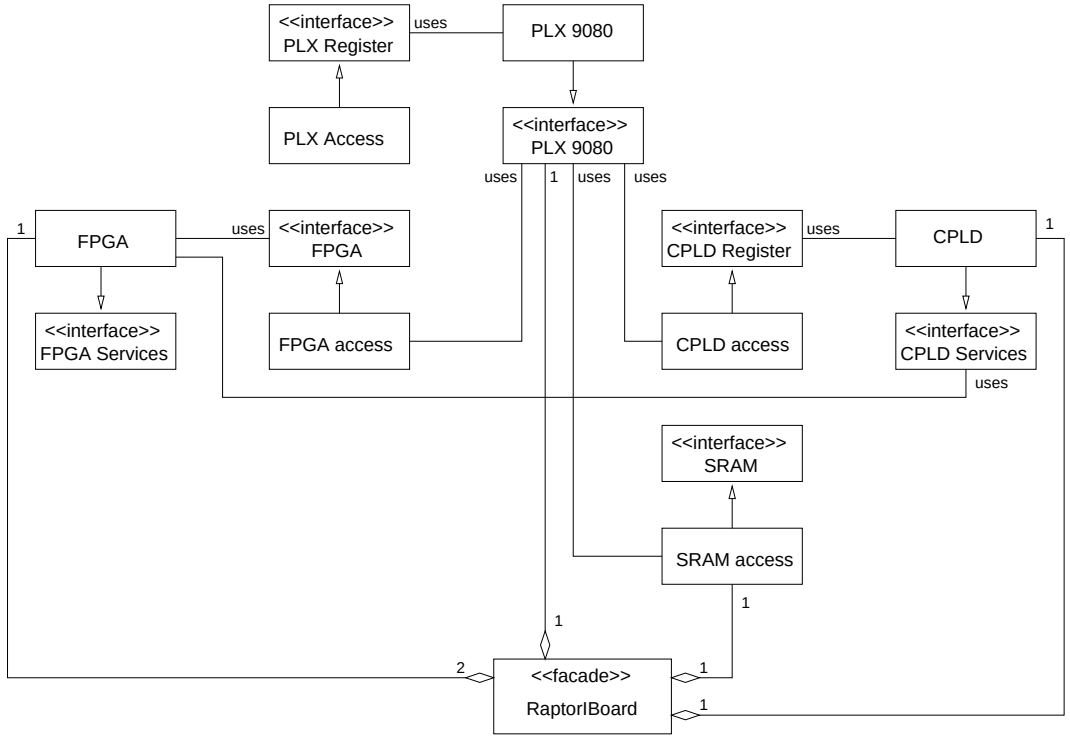



Figure 4.17: Object structure of a driver for the Raptor-board (compare communication structure in figure 4.15). The FPGA objects are interfaces for the programming of the FPGA as well as wraps the interface in the CPLD.

The different releases of the Spyder board can only be identified by a code number in a register of the CPLD, thus depending on this register it can be distinguished between a Spyder-board and a Raptor-board. Because of the different organisations of the address space on the local bus on the two boards, the CPLD can only be read out safely in the case of a Spyder board. On the Raptor-board, the try of a read out leads to a freeze of the system, because no device is answering at that location on the local bus. Thus, in the beginning the two types were distinguished by slightly different default settings in the PLX bridge registers[4].

In the example of the Spyder and Raptor FPGA-boards, the PCI-bridge should provide enough information to distinguish between the board types, and therefore, between the required driver type and structure. The key code in the PCI bridge has to guarantee, that on all board sub-types with the Spyder board-key, they have a CPLD register at the same memory location with the revision number available.

All identified FPGA boards in a computer system are collected in the majordomo object `FPGABoards` (compare UML diagram in figure 4.19). The user application can get a reference to each board by reflection methods. For the access to a dedicated FPGA on the board, or to other components, the application can again get a reference to them by management functions of the `FPGABoard`-object. The configuration bitstream for the FPGA is then directly sent to the FPGA counterpart object. So instead of using the `FPGABoard`-object as facade, the application can get direct access to the components.

A reconfigurable device has a counterpart on the software side, too. This object is responsible for the configuration of an assigned FPGA. Furthermore, it can register itself at the FPGA object, to be notified if another component reconfigures the FPGA.

[4]Later the sub vendor ID of the Raptor-board has been changed by urge of the driver programmer.

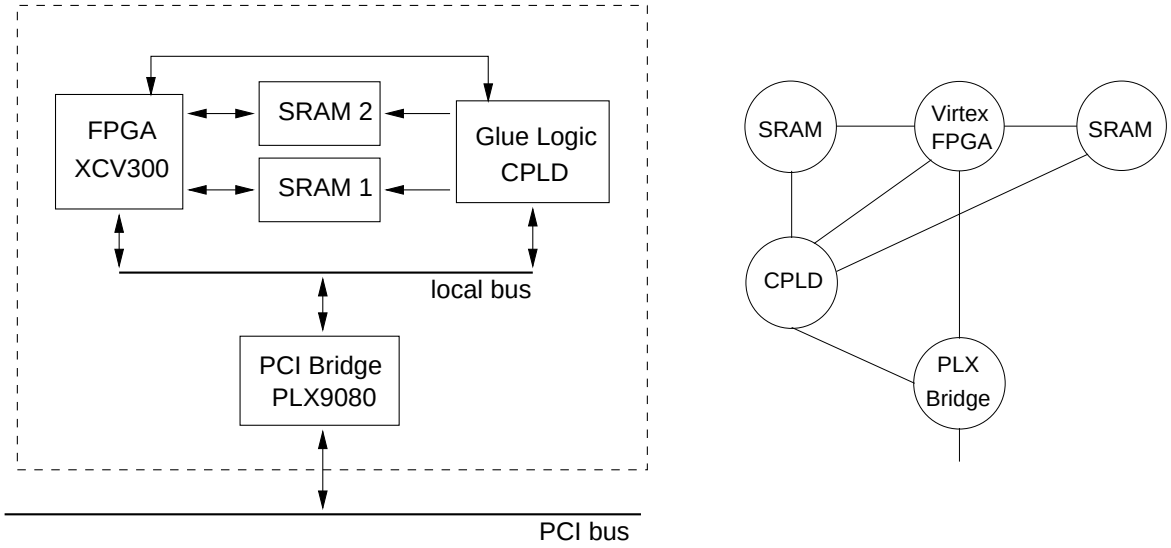



Figure 4.18: Block diagram and communication graph of the Spyder Virtex XCV300 FPGA board.

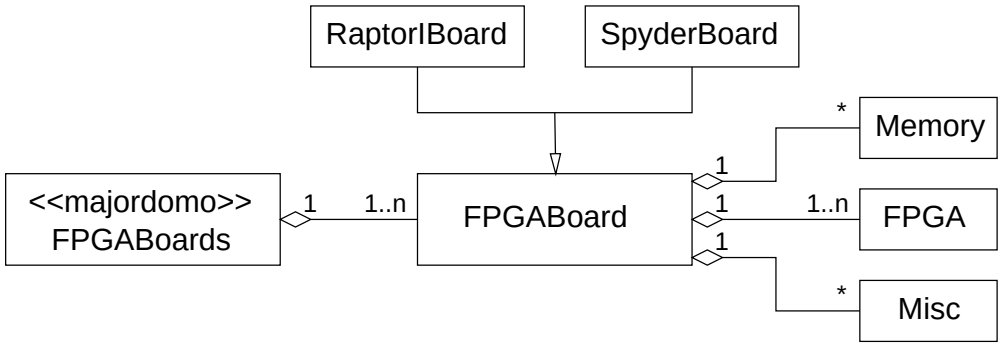



Figure 4.19: General base interface structure of a FPGA board with the on board components FPGA, memory, and further components of a different type (Misc). The dedicated board types are derived from this base structure. Proxy-objects for the communication are not shown here.

The structures discussed here only outline the complete driver architecture. But they also show how the driver structure can be derived from the hardware architecture, and how the presented patterns are integrated. The patterns can even be used as a check list during development, to check whether this feature has to be provided or not.

## 4.4 Summary

In this chapter a coarse grained structure of a device driver has been presented, which can be directly derived from the hardware structure. The software architecture provides the feature separating the concerns into behaviour and communication. Thus each sub-component of the system has been separated into a communication part to the component registers, and a sub-object part to adjust the behaviour. The communication proxy adapts to the given channel to the corresponding hardware of the driver kernel. Hence by replacement of the proxy, the driver can easily be adapted to new hardware topologies. The communication to hardware components beyond their own corresponding one, becomes a part of the desired sub-object behaviour. Other sub-object proxies use

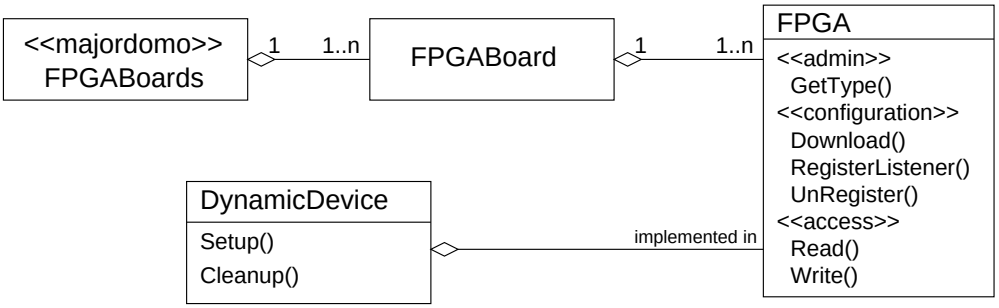



Figure 4.20: Dynamic devices (reconfigurable peripheral devices) are bound dynamically to the FPGA they are implemented in. Proxy-objects for the communication are not shown here.

these services for the communication to their hardware.

The device inter-relationship is a graph of components, and the driver covers the graph of sub-objects as a facade. The facade furthermore adapts the sub-object behaviour to the total behaviour demanded. Along with the construction of the internal structures, issues on device identification have been discussed. The possibility of multiple instances of a device led to the requirement of a higher level housekeeping object, with reflection methods to access the management structures. The influence of the kernel in communication with a user process has been discussed, and patterns for channel wrappers and partitioning have been derived.

The presented approach of separation in the concerns of behaviour and communication is similar to the Devil and the UDI approach. In those approaches, portable function sets are used instead of proxy objects, and the driver kernel is a monolithic block. Here the design is completely established by the use of high-level concepts of OOP and Design Patterns. Furthermore, a methodology for deriving the internal driver structure out of the communication interrelationship has been shown.

In the next chapter, an approach for the synthesis of proxy classes for the communication channel adaptors, respectively a hardware abstraction layer, is presented. The chapter following will discuss some methods to derive the internal structure of the sub-object behaviour from the hardware behaviour.

# Chapter 5

# Generic Hardware Abstraction Layer

In the previous chapter the differentiation between the driver algorithm and the communication with device registers by means of a proxy object, has been discussed. This chapter focuses on the synthesis of the communication layer, the proxy implementation, in the case of channels with a restricted characteristic.

In the literature for driver and interface-synthesis (see for instance [69, 75]), libraries of access functions are used for this low-level communication. The implementations can vary from assembler macros, to pointer operations, or to system calls, depending on the abstraction level where the entry point of the channel is located. In [61], the generation of the communication stub from the device description language *Devil* is described. It is used in combination with a Domain Specific Language (DSL) [110] for the driver algorithm [60]. The communication is based on the use of *ports*, which are the software endpoints of communication with the device registers.

In all these approaches, a section of software (function/port) covers the complete communication channel from the software entry point up to the device registers, and bridges the gap between hardware and software. The implementation, or the derivation of an implementation, is only sparsely discussed or is not mentioned. Due to the strong relationship to the system architecture, no general approach to cover this influences is known up to now.

The set of library functions for the access to the device hardware can be clustered to a proxy class for the access to the complete register file. Each register can be seen as an object which has own manipulation methods depending on the bus-interface. In [97], the overhead of object oriented register manipulation is discussed. Thus the object oriented design must be wisely translated into a target implementation in order to be as optimal as the hand coded implementation.

The implementation of the hardware access in OOP as well as the implementation in libraries, have the drawback that the semantics of the operations from the compiler point of view is lost. The compiler can recognise them only as operations on fixed memory locations, but cannot reconstruct the purpose of this manipulation. So the memory location is marked as volatile and non-cache-able, and no cross-method optimisation is performed after inlining. A preservation of the semantics can yield to device dependent optimisation schemes.

The programming of a hardware abstraction layer by hand is error-prone, due to the programmer unfriendly bit-operations, and the lack of an appropriate abstraction level in the programming languages [91]. In Rapid Prototyping the register set can rapidly change, which leads to a reimplementation of the access layer on each prototype iteration, with the complete effort of a proxy implementation. An automatic synthesis of the hardware abstraction layer reduces the risk of errors, and shortens the round-trip time in the development cycle.

In this chapter, a new approach for the hardware abstraction layer synthesis is shown. Therefore, the hardware abstraction layer is conceptually separated into the concerns of coding, and transportation of information between the hardware abstraction layer and the device. The transport

layer covers the influences of the system architecture.

In the first step, the influences of each node on the communication channel to the device are analysed. As a description method, the model of attributed grammars will be used. The model includes the crossing from the software instructions to the protocol level on the bus systems. Thus, with the use of algorithms for code generators from compiler construction, the attributed grammars can be evaluated to synthesise the transport layer. Start string is in this case not the result of the traversal of the intermediate syntax tree, that is the result to the input language parsing. Here the start string is generated by translation of the intended access to the register file into a set of appropriate bus transactions.

The method presented here has been evaluated in a prototype. The quality of the synthesised code is compared with hand written code. This prototype shows furthermore a concept for the integration of this method into a high-level language compiler. Optimisations of the transport layer which take the external control flow of the driver into consideration are additionally discussed. The chapter concludes with a brief summary of the method.

## 5.1 Problem Analysis

In the previous chapter, the separation of a driver into a driver kernel and a Hardware Abstraction Layer (HAL) for device access has been discussed. This separation is used in different other approaches like the Domain Specific Language approach in combination with the Devil system by INRIA, or by other systems (compare section 3.2). The HAL is responsible for the direct interaction with the device hardware and the section where the logical meets the physical. The interfaces to the device kernel are the manipulation methods of each Information Entity (IE), which are visible in the register file of the device.

The hardware abstraction layer can be conceptually separated into a layer for the coding of bit-patterns, and a transport layer for the transport of the bit-pattern to the device, or from the device to the CPU. In practice, both layers are tangled by inlining of the transport layer into the coding layer.

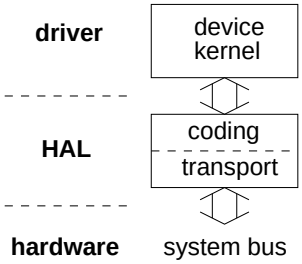



Figure 5.1: Separation of the Hardware Abstraction Layer (HAL) into a coding and a transport layer.

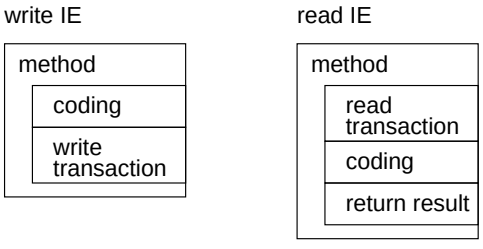



Figure 5.2: Read and write transactions separated into the coding and transport sections.

The coding layer translates abstract values to the bit-pattern used for state coding of the information entity in the register file. The manipulation methods for setting a state can use fixed or flexible parameters, which both can be translated by equations to a right-aligned bit-pattern. The size of the bit-pattern $n$ is a property of the IE, so the bit-pattern is stored in *coding layer variables* at the bit positions $n - 1$ down to 0. The parameter set can include system parameters which specify system properties. They may be only visible inside the hardware abstraction layer. The inverse functions are used for the translation from a bit-pattern back to an abstract value, and are coupled with a read transaction in the transport layer. The equations can be given in target code

or by a specification language. The former is integrated by term replacement of variable names. The latter allows direct evaluation during synthesis process by means of constant folding, or it is translated into the target language. The interface between coding layer and the transport layer is the addressed IE, the bit-pattern in coding layer variables, and the transport direction.

The transport layer is responsible for the correct exchange of bit-patterns between the coding layer variables and the device register file. This includes a bit position remapping from the right-aligned bit-pattern to the correct position in the device register, and the transport to or from the device register file. Simply speaking, it has to copy the bit-pattern from a CPU register to a device register or vice versa. This process has to be transparent for the coding layer and to be side-effect free. It requires an adequate selection of the software instruction sequence to invoke the transport in combination with the required set of parameters. The concrete instruction sequence depends on the instruction set of the processor or, in the case of the use of a high-level language, on the target language and the available operating system services. The latter may require the use of special function sets or assembler macros. The selection of instruction sequences and the parameter determination must avoid side-effects. If a side-effect free selection is not feasible, compensation must be undertaken by adaption of the software sequences, for instance, by a read-modify-write cycle. The side-effects and the parameter set depend on the system architecture and the behaviour of the device register file. Both have to be taken into consideration for compensation.

In the Devil approach [61], the same separation of coding, mapping, and bit-pattern exchange with the device registers is used. For the latter operation, a function in the high-level target language named `port()` is used. It provides a transparent exchange mechanism to the device registers and covers all influences of the system architecture between the CPU and the device. This function must be generated for each platform by hand.

The system architectures of computer systems especially in the field of embedded systems can be of great variety. The vendors of peripheral devices only know their systems and the interfaces of them. The same is true for the vendors of chip-sets for the interconnection, for instance bus bridges. In most cases, the interfaces between the chips follow a standard. So the idea is to synthesise the transport layer from a composed description of the system architecture. The descriptions specify the transport behaviour of each component. The influence can be analysed in total, depending on the results the software instructions being selected, and parameter sets being calculated.

In the next section the influences of the system architecture on the data transport between the CPU and a peripheral device is described. The feasibility of an analysis by an approach depending on the automaton theory, is briefly discussed. The remainder of this chapter focuses on the approach of this thesis which uses attributed grammars to describe the behaviour of the system components. The grammar rules are evaluated by code selection algorithms from the field of code generator construction. Adequate code sequences for the transport layer are synthesised. The results of the methods are discussed in a case study. Afterwards, optimisations under the consideration of the control-flow in the layers on top of the HAL services are discussed.

## 5.2 System Architecture Model

For the synthesis of the HALs transport layer, the impacts on the data transport of the different components in the computer system architecture has to be taken into consideration. For this purpose, the architecture can be described by a network of interacting components of the following three types:

- the main CPU, where the driver kernel and its HAL is located,
- the interconnection network, consisting of bus systems, bus bridges, and direct physical links,
- and the peripheral device with register file, accepting bus-automaton, and the device kernel.

The CPU executes the instructions of the HAL, performs bit-operations, and invokes transactions on the system bus that the CPU is connected to. Depending on the instructions, different transaction protocols with different sizes of transported data are used on the bus. In this component, the border between software and hardware is crossed.

The interconnection to the peripheral device is a network of bus systems, bus bridges, and the physical links between the components. Within this graph, routing information is required to direct the transaction data to the correct target node. The routing information is bound to the transaction and the evaluation depends on the characteristics of the channel and the components behaviour.

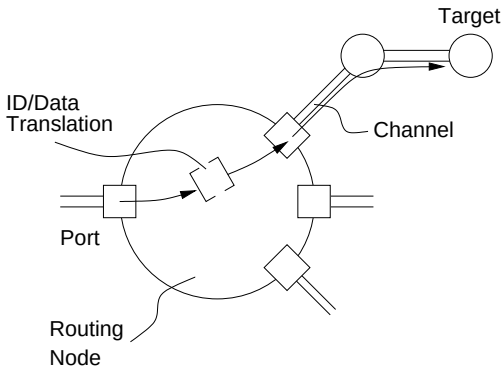



Figure 5.3: Incoming information is adapted to the requirements of the outgoing port. Parts of the incoming information is used to determine the outgoing port.

In a switched network with direct links between the nodes, depending on the target ID, a node has to decide to which port the message has to be forwarded (see figure 5.3). In homogeneous networks, like the IP-based networks of TCP/IP [85], the message does not need to be modified and the target address is not translated. In heterogeneous networks, along with the routing decision, the target ID has to be adapted to the requirements of the corresponding sub-network.

In the case of a bus system, the nodes are physically attached to a shared medium. This can be modelled in a routing node (for instance a bus bridge) by a port which broadcasts the message to the other nodes attached to the bus (one-to-many communication). The routing node forwards the message to the broadcast port. Here the target ID is translated in the sense that only the target node picks up/accepts the broadcasted message (see figure 5.4).

Each node on the route can translate the transported data for adaption to the requirements of the channel to the next node. This means re-arrangement of data sections in order, splitting of the message to different data sizes, etc.

The device register file is the endpoint of the transaction. An automaton accepts a protocol on the peripheral bus, and enables the exchange of data between the bus and the registers. The type of exchange depends on the protocol. The register and the affected bits are selected by parameters which are encapsulated in the protocol.

To sum up, due to the characteristics of system busses and the execution of instructions in the CPU, the communication can be seen as an exchange of messages with limited duration. Within

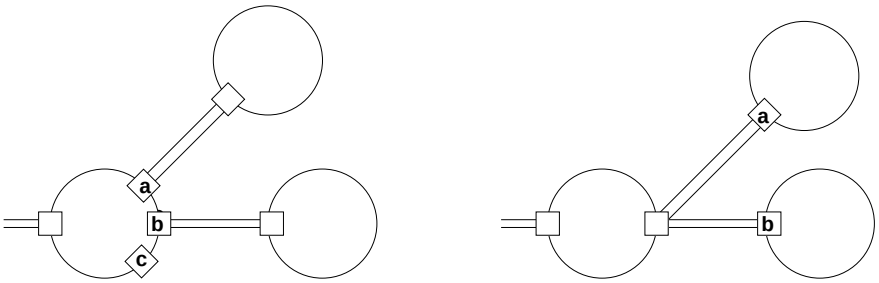



Figure 5.4: In a real switched network the port identifies the target (left), in a bus system the target port identifies the messages which are related to it (right).

the network a message is routed towards the target node of the message. The message content can be translated inside the nodes. At the endpoint, the device register file, the interaction with the information stored in the registers take place.

All these influences have to be backtracked for synthesis of a transport layer which provides transparent access to the information entities. A modelling approach describes the timing behaviour of CPU, bridges, and device by automata. Thus the communication system is a chain of interacting automata like the adaptor automaton $T$ between $A$ and $B$ shown in figure 5.5.

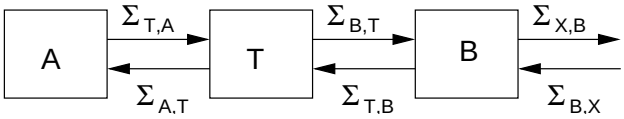



Figure 5.5: Translation between two automata $A$ and $B$ by an adaptor automaton $T$. $B$ can communicate with a further component $X$ (for instance the environment).

The protocol accepting automaton in the device, for instance the automaton $B$ in figure 5.5, and the automaton of the bridge ($T$) form a product automaton $P = T \times B$ (see figure 5.6). The accepting states of $B$ become a set of states in the product automaton $P$. This automaton represents the total behaviour of $T$ and $B$ in the perspective of $A$.

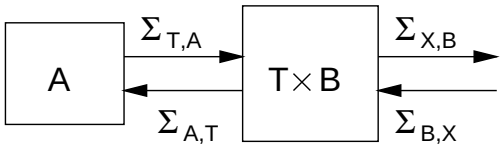



Figure 5.6: Product automaton $T \times B$ as total behaviour of $T$ and $B$ in the perspective of $A$.

In case of further components, this merge can be proceeded towards the CPU. Finally, the system can be described as one product automaton with the software instructions as the input. The question of register manipulation becomes a question of which sequence must be provided to the product automaton in order to reach one state in the set of accepting states in the register file automaton.

This analysis can not be done in practice, because of the state explosion in a product automaton [42, 50]. A system with 32 address lines leads to $2^{32}$ states to be checked. If the 32 data lines have to be additionally taken into account, the state space becomes $2^{64}$ states, not taking the states

of the accepting automaton into consideration. A more abstract perspective is required to solve the problem by state space reduction. The instruction sequence in the CPU is the initiator of a transaction, so in the following, the perspective of an information flow from the CPU towards the device is assumed.

The communication in the network is performed by protocols with limited duration. The duration of each transaction is given by the bus protocol stated in the processor/bridge documentation. On a synchronous bus a finite integer number of bus clock cycles for each protocol is given. Asynchronous busses have an idle state which separates the protocol sections. In the case of multi-master busses an arbiter keeps track on the transactions, and the idle state on the bus can be suppressed by consecutive assignment of the bus to a node. The transactions on the busses can be separated into sections which are independent of each other. These sections are in the following called a *package*. They are independent in the sense of not influencing each other, which means the packages remain independent of the previously sent package[1]. In all systems, a package transports a finite amount of information. A detailed knowledge of the sub-transactions and exchanged signals inside the package (protocol) is of interest only for the bus-interface developer. For the information transportation, the type of protocol and the exchanged information inside the package is most important.

The CPU issues, depending on the software instructions, packages for the transactions on the system bus. The role of the nodes in the network is reduced to package translation. An incoming sequence of packages with parameters is translated to a sequence on the other side. Along with the protocol translations the content of the messages is translated. The device automaton accepts the package and enables information exchange with the package. The question of register manipulation is reduced to which protocol with what parameters is required for the transaction.

The model of package and content translation can be described with the model of Attributed Grammars (AG). With the methods of code generator construction, the grammar rules can be used to select the required software instructions for the transport layer. The next sections focus on this topic. The used grammar and the attribute translation is depicted in the next section. The following one describes the code selection algorithm, a sub-algorithm which is similar to the affix-grammar method by Ganapathi and Fischer [37]. Finally, the chapter ends with a case study and a discussion on transaction optimisations.

## 5.3 Behaviour Modelling

From the perspective of the CPU, the communication network forms a tree to the peripheral devices. Each node in the tree has an individual behaviour and between the nodes packages with informations are exchanged. The impact on the information depends on the behaviour of the components on route between CPU and the device.

The influence of a component on a sequence of packages can be described by means of an attributed grammar. Each package(-type) is represented by a symbol of the grammar. The translation is described by the rewriting rules. This includes the change of order of consecutive packages. The transported content of the package can be described by binding the informations as attributes to the symbol. The transformation of the information is specified by functions assigned to each rule. Actually, in compiler construction the functions on the attributes are evaluated only in one direction, depending on whether the attribute is *inherited* or *synthesised* [49]. Here the attribute translations are evaluated in both directions. The attributes with this characteristic are denoted *derivable*. For

[1]The packages are allowed to differ in the sense of a different arbitration phase whether a bus is requested or still occupied.

backtracking from the device towards the CPU (bottom up in the tree), the derivable attributes are handled like synthesisable attributes. The other way round, for side-effect determination, the evaluation is from CPU towards the device which means down the tree. The derivable attributes behave like inherited attributes. Furthermore, the evaluation of the rules is guarded. A rule is applicable if, and only if the subject string matches, and the predicates of the guard are satisfied.

The complete grammar for the transport from CPU to the device is constructed through collection of the grammars of each component on the path (compare figure 5.7). In some cases, the grammars can be parameterised to take system settings, like path information, into consideration.

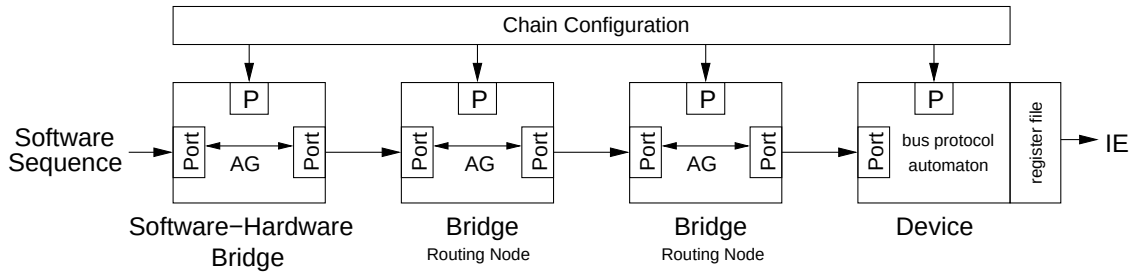



Figure 5.7: Example of a path: from software instructions on the left, via routing nodes, described by an attributed grammar, to the device register file. Each node is configured by parameters which are coupled to the path.

In the next section, the used attributed grammar and its properties are described in more detail. The attributed grammar can be used to describe the behaviour of the transition from hardware to software, and the behaviour of the nodes in the communication graph. Afterwards the construction of the complete grammar is depicted.

For the register file behaviour, a different model is used which is depicted in the following section. The mapping from packages to register file manipulation is sketched by the use of mapping functions. The inverse of the register file behaviour is used to determine start strings of packages, which can be used as input for the software synthesis algorithm. It is based on the pattern matching algorithms of code generator algorithms, and uses the attributed grammar as description for backtracking. The algorithm is described after the description of the specifications.

### 5.3.1 Attributed Grammar

For the specification of the components behaviour, a grammar described by the following tuple can be used:

$$AG = (N, T, S, \mathcal{P}, \mathcal{A}, \mathcal{F}, \mathcal{G}, C_{sys})$$

The set $N$ defines the nonterminals of this grammar; the set $T$, the terminals. The derivation can start with symbols of the set $S \subseteq N$. All symbols represent protocols which can appear on the communication link, and have to be unambiguous inside the grammar

The set $\mathcal{P}$ determines the production rules $p \in \mathcal{P}$

$$p : N+ \rightarrow N+$$

or

$$p : N+ \rightarrow T+$$

The operator $+$ denotes a string of one or more symbols of that set. The grammar is cycle free; otherwise unlimited transactions can be produced, which should not appear by the general assumptions.

The set $\mathcal{A}$ is the set of attributes which can be assigned to the symbols. The function $A_i : (N \cup T) \rightarrow \mathcal{A}$ returns the attribute $i$ of a symbol, whereas the subscription denotes which attribute. The set $\mathcal{F}$ is the set of functions for attribute translation and $\mathcal{G}$ is the set of guards. The guards assigned to each production rule must be satisfied to make the rule applicable for string rewriting. The set $C_{sys}$ represents the "external"parameters which can be used in the functions of $\mathcal{F}$ and $\mathcal{G}$. They can be used for introducing system parameters into the rule set.

Each set $\mathcal{F}$ and $\mathcal{G}$ can be separated into two disjunct sets $\mathcal{F}_\uparrow$ and $\mathcal{F}_\downarrow$, $\mathcal{F}_\uparrow \cup \mathcal{F}_\downarrow = \mathcal{F}$, respectively $\mathcal{G}_\uparrow \cup \mathcal{G}_\downarrow = \mathcal{G}$. The subscription $\uparrow$ denotes the functions for evaluation right to left, and the subscription $\downarrow$ denotes the sets for the evaluation left to right.

A function $f_{p,u,i,\uparrow} \in \mathcal{F}_\uparrow, p \in \mathcal{P}, u \in (N \cup T)$ maps all attributes of all symbols on the right side to the set of attributes, and the result is assigned to an attribute $i$ of a symbol $u$ on the left side of a rule.

The functions for attribute evaluation have to be reversible. This is sometimes tricky, for example in the case when a modulo operator is used in a function. In this case, the reverse function needs an offset, which has to be given as external parameter. Under the assumption that the translation of data only changes the position of the bit in the bitstring, or changes the temporal order of bitstrings, the transformation can be described by the use of a matrix.

Let $\vec{b}(u) = A_{data}(u), u \in (N \cup T)$ be the bit-vector of the data word of the symbol $u$. The vector has the form $\vec{b} = (x_{n-1}, x_{n-2}, \cdots, x_0)$ whereas each $x_i, 0 \leq i \leq n-1$ represents one bit. The bit-vector of a string of symbols is the concatenation of each bit-vector

$$\vec{B}(u_m, u_{m-1}, \cdots, u_0) = (\vec{b}(u_m), \vec{b}(u_{m-1}), \cdots, \vec{b}(u_0)), u_i \in (N \cup T), 0 \leq i \leq m$$

The function $A_{data}$ is unambiguous, thus from a given bit-vector $\vec{B}$ all data attributes of a symbol string can be set.

The transformation can now be expressed by

$$\begin{aligned} B^T|_R &= T_{v,w} \cdot B^T|_L, \\ v &= |B^T|_R|, w = |B^T|_L| \end{aligned} \tag{5.1}$$

whereas the subscription $R$ denotes the right side of a rule, and $L$ the left side. The transformation in (5.1) is down the tree. The transformation matrix has the size of $v$ rows and $w$ columns. The transformation matrix $T$ consists of the components $t_{i,j}$ with $\forall t_{i,j}, 1 \leq i \leq v, 1 \leq j \leq w | t_{i,j} \in \{0, 1\}$. The number of elements with the value 1 in each row and in each column must be less or equal to 1. Otherwise a bit is mirrored (number of 1 in one column greater than 1), or many bits have to be mapped to one single location (number of 1 in one row greater than 1).

With transformation matrices of the type shown above, the distribution of the bits during a translation can be expressed. For example, the matrix $T_{endian16}$ in the size $16 \times 16$ exchanges the endian of a 16 bit data word:

$$T_{endian16} = \begin{pmatrix} 0 & I_8 \\ I_8 & 0 \end{pmatrix} \qquad I_8 = \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix} \tag{5.2}$$

The inverse operation for a translation up the tree is a multiplication with the transposed matrix of $T_{v,w}$. In general $T^{-1} \neq T^T$. Due to the way the matrix expression is used here, the transposed matrix represents the inverse operation even if the matrix is not quadratic and orthogonal.

The example shown here is a special case, where the matrix $T$ is quadratic and orthogonal, thus $T^{-1}_{endian16} = T^{T}_{endian16}$.

Each production rule of $\mathcal{P}$ can be assigned an action which is executed if the rule matches. In compiler construction, this action is used to issue the machine code (compare for example [66]). The possible actions are not listed in the grammar tuple. Here the actions can be used in a different way: in the grammar, which describes the transition from software to the CPU bus, the actions can issue the required software instructions. The output can be directly translated to the code sequence, or to start symbols of a further grammar which constructs the final code sequence. In both cases, the attributes can be used by the action for parameterisation of the output.

A similar approach for code output is to process the rest string after the code generation algorithm has terminated. The rest string must contain only symbols of the start set. Each symbol represents a software sequence. Thus, further translation functions parse the symbol string and issues the software sequences.

In the production rules which represent translations in the communication network, the actions issue only a unique label of the production rule. The string of labels represent the list of rules which have been applied to the start string during rewriting. The list can be used for backtracking the translations of the attributes.

### 5.3.2 Register File Model

The register file of a device is a regular structure of bit locations. A location can be addressed by the tuple $(r, b)$ where the parameter $r \in R$ addresses a register, and the parameter $b \in B$ addresses the bitposition in the register ($B = B(r)$).

The interaction of the register file and the peripheral bus is controlled by an automaton which accepts a protocol on the bus. The protocol $P$ determines the type of interaction with the register file, and the attributes $A(P)$ of the protocol $P$ identify the involved bits. The different interactions between bus and register content can be described by the set $M$.

The impacts can now described by a function:

$$f_R : P \times A(P) \rightarrow \{R \times B \times M\} \tag{5.3}$$

An access to the register file affects a set of different bits. Many protocols can result in sets on the right side which have non-empty intersections. Hence, the inverse function $f_R^{-1}$ for a given set on the right side is ambiguous.

In most cases, the protocol carries as attributes an register address $A_{add}(P)$, and a data string of bits $A_{data}(P)$, which implicitly code the bit positions. Here the function can be split into:

$$\begin{aligned} f_r : &\quad P \times A_{add}(P) &&\rightarrow \{R\} \\ f_b : &\quad P \times A_{data}(P) &&\rightarrow \{R \times B\} \\ f_M : &\quad P &&\rightarrow M \end{aligned}$$

The reverse functions are still ambiguous, as only the number of solutions are reduced. The position of the data bits are coded implicitly in the data bit string of the protocol. For the manipulation of a set of bits in a register, the solution should affect as many bits as possible by a minimum number of transactions. However, the result of reverse evaluation is a set of transactions with the assigned attributes.

Because of the fixed size of the data bit string, more than the desired bits are affected. In the case of these side-effects, the attributes of the protocol have to be selected in such a way, that the application of the manipulation method has no effect on the value of these side-effected bits.

## 5.4 Code Generation

In this section, the evaluation of the presented grammar in the special field of software synthesis for a hardware abstraction layer is presented. The aim is the manipulation of an information entity, or the read out of its actual status. The translation instructions in the coding layer can be constructed from a specification in the device database. From the database entry, the transport direction and the mapping to the register file is determined. In combination with the system architecture the communication software can be synthesised.

Firstly, the construction of the complete grammar from of the description of each node is described. Secondly, a set of start strings is generated for the desired bit-field manipulation in the register file. These strings are matched with the rewriting rules of the grammar until a set of feasible software instructions is determined. For each set the side-effects are calculated. Different strategies for compensation are applicable, depending on the register file characteristics. From the set of side-effect free solutions, the most appropriate one is selected and target code is generated.

The code is then combined with the instructions of the coding layer by mapping it, as well as the movement of bits between the coding layer variables in the interface to the locations used by the generated instruction sequences.

### 5.4.1 Grammar Construction

The driver programmer intends to access an information entity in a dedicated device. The device is part of the system architecture which is described by a graph of components. In this graph, the route from the CPU to the device can be determined. This list of nodes is used to determine which attributed grammars have to be collected for the construction of a complete grammar. This constructed grammar describes the influences of the complete communication channel.

The route determines further system parameters which are required to route the information flow on to the desired path. If a *dedicated route* is specified, parameters are set according to the ports which have to be used. In the case of a *principle route*, which means the dedicated ports are determined at runtime, the parameters stay symbolic. Before the channel is used first during runtime, the parameters have to be determined by initialisation routines. The parameter set of the route is used to parameterise the functions and guards of the attributed grammars.

The complete grammar for the channel is constructed by combining all grammars of the nodes on the route. Each rule is assigned a unique label, for example by simply enumerating each rule. This label is used to track the application of rules during rewriting.

Two components must use the same protocol on a link in order to communicate with each other. This means that they use the same symbols for the description of the communication. Here some problems may appear:

Firstly, if a bus bridge mediates between two busses of the same type, then on both sides the same protocols and hence the same symbols are used. This is not allowed, as the grammar can not distinguish between symbols from the one side and symbols from the other side. In this case, the set of start symbols $S$ is part the terminal symbols $S \subseteq T$. The result is a loop in the grammar. Loops of this kind are avoided by using different symbols on each side for the same protocol. This requires an adaption of the symbols used by other components which are linked to that type of component during collection of the grammars. The adaption has to ensure that on both sides of a link the same symbols are used to represent the same protocol.

The second problem arises if one type of a bus system appears more than once on the path to the device. The busses use the same type of protocols, and hence, the same set of symbols for representation of the protocols. If the rule sets are simply merged, a rule of the incorrect component can be matched during translation, and the translation ”beams”to a different position on the route.

In the grammar, no rule on the order of application of rules can be defined. This problem can be avoided by a modification of the symbol sets. Each bus system with the same type on the path is assigned a unique label. This label is used as prefix for the symbols used to represent the protocols. Thus the symbol sets of the bus systems become unique and a beaming effect is avoided.

A similar problem can appear if two grammars use the same symbols which do not appear on the outside of the links. The following has to be satisfied for all components $i, j$:

$$(N_i \setminus S_i) \cap (N_j \setminus S_j) = \emptyset, \forall i, j | i \neq j \tag{5.4}$$

Again, this problem can be solved by the renaming of the symbols to disjunctive sets, for example by assigning unique component prefixes.

After building up the total behaviour description, a start string as starting point for applying the rewriting rules has to be generated for each transaction. This is performed by the bus protocol generator which evaluates the inverse function of the bus protocol accepting automaton in the device.

### 5.4.2 Start String Generation

The start-string for the rewriting algorithm is determined by reverse evaluation of the register file behaviour. The properties of the information entity in the database determines the register, the intended bit-field, and the access method to the register file. The desired access to the information entity determines the set of feasible protocols which can be used. In combination with the mapping to the register file, the attributes of each protocol are determined. This mapping is ambiguous.

The aim is the access to a bit-field which can be distributed over more than one register, if the bit-field size is larger than the register. A feasible solution is to access each bit of the bit-field by a single transaction. On the other hand, a transaction protocol may be used which covers the bit-field as a sub-string. From all feasible solutions a sub-set has to be determined by a heuristic.

As expressed in equation (5.3), a protocol with its parameters maps to a set of tuples which denote a register, a bit in the register, and the access method. Let us denote this set of tuples with $\mathcal{B} \in \{R \times B \times M\}^+$, and the protocol which maps to this set still with $p \in P$, without explicitly naming the attributes. Then equation (5.3) becomes

$$f_R(p_n) = \mathcal{B}_n \tag{5.5}$$

and a set of consecutive applied protocols $p_c = \{p_0, p_1, \ldots, p_n\}, p_i \in P, 0 \leq i \leq n$ to an only larger set of accessed bits:

$$f_R(p_c) = \mathcal{B}_c \tag{5.6}$$

Let $\mathcal{B}_{IE}$ be the set of tuples which describes the intended access to the bit-field of the information entity. The set of protocols $p_{c,IE} \in P^+$ has to be selected so that

$$\begin{aligned} f_R(p_{c,IE}) &= \mathcal{B}_{c,IE} \\ \mathcal{B}_{c,IE} \cap \mathcal{B}_{IE} &= \mathcal{B}_{IE} \end{aligned} \tag{5.7}$$

and the set $p_{c,IE}$ is minimal, which means a reduction of $p_{c,IE}$ by an element violates equation (5.7).

Still the set $p_{c,IE}$ is not unambiguous. Different sets fulfill equation (5.7). Each set is a feasible set and a start string has to be generated from each set.

In general, a more specific algorithm for the determination of the start string can not be given here, because of the variety of accepting automatons in devices, and the different organisations of the register files. Thus for each device, a specific algorithm has to generate the start string. This component is named bus protocol generator in the following. The case study (depicted later in section 5.5) has shown, that the implementation of this generator for a dedicated device is rather easy. In some cases, the bus protocol generator must furthermore, take into consideration a relationship on the order of some bit accesses. Some bit manipulations must be performed in an atomic manner, and are not allowed to be split into separate transactions. This boundary condition is not taken into consideration in the previously discussed equations.

However, the result of the translation of the access to an information entity is a set of protocol strings with assigned attributes. The protocols are represented in the terminal symbols of the component which is the predecessor of the device. These strings are matched with the rewriting rules of the grammar to determine the required software sequence, which results in the desired register-file transaction.

### 5.4.3 Grammar Evaluation

The algorithm for translation of an access to a device follows the affix-grammar method for code generation by Ganapathi and Fischer [37]. As an extension, the code generation tracks the translation to check at a later date for side-effects and may apply a special compensation. The required access is coded in the start string.

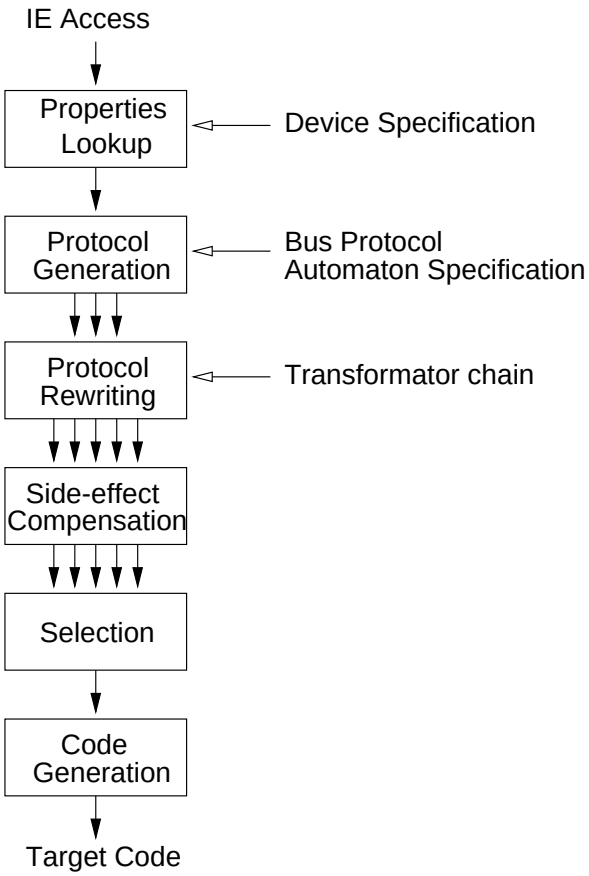



Figure 5.8:
Overview of the stages of grammar evaluation for access code generation

The attributed grammar is a set of rewriting rules. The algorithm matches the start string with the right side of the rules. If the guards are satisfied, the rule is applied and the symbols replaced

by the symbols on the left side. The attributes are translated right to left according to the assigned set of functions.

If more than one right side matches, the algorithm forks and follows all alternatives. The selection of the best solution by evaluation of a cost function, and the use of dynamic programming similar to the algorithms by Aho, Ganapathni, and Tjiang [1] is not feasible at this stage, as no local cost function can be applied. The compensation of yet unknown side-effects can also not be taken into consideration at this stage. The estimated duration of the complete transaction can be used as a cost function. The evaluation of the cost function must be delayed until the translation is side-effect free, which may require the application of compensation.

The action of each rule issues the label of the rule. The resulting string represents the history of rule application. Only the rules of the first node issue additional information for the generation of the desired software instructions.

A translation is only correct if the resulting string contains only start symbols of the grammar of the first node. The pattern matching results in a set of feasible software instructions, which vice versa, result in the intended protocol and the register file access at the device. Then possible side-effects of the data transport have to be compensated by applying different strategies.

The data attribute of a protocol symbol contains the position of bits instead of the value. This information is used to establish a mapping between the parameters of the software instructions, and the variable which is used in the interface between the coding layer and the transport layer. If the size of the parameter and the interface variable does not match, instructions for splitting, respectively merging, are inserted.

In the case of a read transaction, a side-effect is that more than the required bits are read out. The superfluous bits have to be masked and the desired bits transferred to the interface variables.

In the case of a write transaction, the superfluous bits of the interface parameter may affect additional bits in the register file. The compensation must be done by setting the superfluous bits to values which keep the register content in the same state as before. Therefore, the affected registers and bits have to be determined. The bit positions of the superfluous bits in the parameter of the software instructions is propagated forward by translation of this attribute. Therefore, the history of rules is evaluated and the attribute translation is applied to the data positions, however now in the direction towards the device. Finally, the bit positions are mapped to the register as well as bits in the register file, by evaluation of the behaviour of the bus accepting automaton. The result is a mapping table between bits of the interface variable and the affected bits in the register file.

The side-effect compensation strategy depends on the characteristic of the affected bits, or respectively, the affected information entities. If the value of the bits is statically constant, the superfluous bits can be set to this constant value. If the value is not statically constant, but can be determined by an external analysis method, the provided known value is used analogous to the statically constant value. In some cases the register file recognises special bit patterns, which have the semantic 'do not alter the content' (compare for example the device CAN 82527 [44]). These patterns are evaluated with a higher priority than the access method. These special patterns can be handled analogous to constant values.

In the other cases where the value can not be determined at compile time, the value of the superfluous bit has to be gathered at runtime by reading out the value before writing the new ones. This leads to the well known read-modify-write cycle. This cycle can be also constructed with the term rewriting method presented here.

For all affected bits, a start string for a read transaction is generated. The read transaction is constructed by applying the previously presented method. The bus protocol generator has to construct a start string for all affected information entities, but only for the parts of the bit-fields

which are affected by the write operation. After code generation, the position of the affected bit can be even tracked up to the result of the read transaction. The bit positions are now adapted to the positions of the parameter variable in the write transaction. The bit-field of the intended write transaction is incorporated into this bit-field.

The method of generating a read-modify-write cycle can be combined with the incorporation of known and constant values. This can lead to a shorter read transaction, because less bits have to be read and the remapping is reduced. Again, each solution of write transactions is combined in a cross product by the solutions of the read transactions.

The result of the method used so far is a set of side-effect free solutions for the access to an information entity. From this set the most appropriate one has to be selected according to a cost function. In each solution, the required software instructions and the the protocol transactions between the nodes are known. Each part is coupled with an execution time. For the protocols, the required time for the transaction can be expressed by an assigned attribute denoting the *estimated execution time*. Only an estimation can be used, because the model of rewriting rules can not cover interleaving effects, or the gap between two consecutive protocol sections. The same is true for the execution time of the software instructions. In both cases a more fine grained model must be used to determine a Worst Case Execution Time (WCET). Here the estimated execution time is used as a heuristic for the selection of a solution. It depends on the target system if the heuristic is enough, or if it is used for a reduction of the solution set. The final selection is done after a WCET analysis of each preselected solution.

From the selected solution the target code is generated. This can be done by further code generators which use the start symbols strings, including parameters and the inserted code for side-effect compensation, as input. The target code generators have to take the integration into the coding layer into consideration.

## 5.5 Case Study

For evaluation of the method discussed in the previous section, a prototype has been implemented. The prototype can translate an access to an information entity in a device into the instruction sequence of a high level language, in this case `C/C++`. The prototype can synthesise a simple coding layer and directly integrate the required transaction instructions, so the coding layer and the transport layer are woven to a hardware abstraction layer. The prototype has been implemented in the object oriented script language Python. The bus protocol automaton description, the grammar rules, and the attribute translation functions, are directly implemented in the script language. The rules are combined to form the rule set for a node (compare figure 5.9). The path to the device is a list of these components. All system descriptions are dynamically bound to the evaluation algorithms in the prototype during runtime, so the prototype can be seen more like a framework than an application.

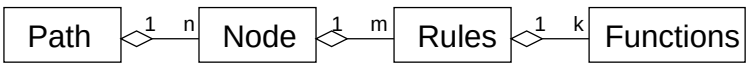



Figure 5.9: Object class structure for the system specification by the attributed grammar in the prototype.

The description of the devices information entities is stored in an database. The data structure is shown in the UML diagram in figure 5.10. For simplification of the specification, the methods which can be applied to an information entity are bound to a type of the information entity, and

the information entity has a dedicated type. This reduces the effort for specification. The methods contain the coding equations, the transport direction, etc. The mapping to a register of the register file is stored as part of the information entity description. This information is used to determine the location in the register file.

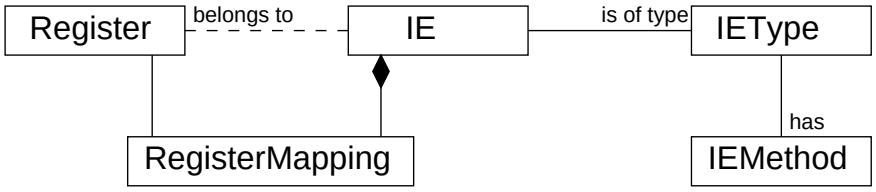



Figure 5.10: Structure of the database used in the prototype as UML class diagram.

The specification can be exported to, respectively imported from, an XML-file. Thus, the device specification is read in from the device XML-file directly after the start of the prototype.

The prototype has been designed not only as a case study for the generation of a hardware abstraction layer, furthermore it is a concept study for the integration into a compiler environment. In the first operation mode, the prototype reads the device specification and generates a hardware abstraction layer by translating each access method of all information entities to the target language. The result is a library with all, in respect to the specification, available access functions of the device.

In the second operation mode, the prototype directly integrates the access methods into a source file in the high-level target language. The intended access is placed into the source file by the use of an language extension with special access statements which are enclosed into tags. In figure 5.11, an access to the information entity '`DONE`' of the CPLD on a Spyder board [118] with the method '`set`' enclosed by the start tag '#@'and the end tag '@#' is depicted.

The first stage of the prototype parses this input source file and scans only for the tags. Then the enclosed information entity, the method and the parameter names are determined. The rest of the translation process is analogous to the generation of the functions for the HAL library. The information entity is then looked up in the database, and the specified method description is determined as well as the mapping to the register file.

From this set of properties, the set of start symbols is generated by the bus protocol automaton object. Then, according to the list of components, the start symbols are translated. All alternatives are followed. The result of the first component of the list, which represents the transition from software to a bus protocol, is a set of software symbols. The applied rules during translation are tracked, and side-effects are compensated by selection of the required constants, or generation of the additionally required software symbols for the read function.

In this prototype, each software symbol represents a software generator object, which produces the required software instructions for the transactions. The instructions are not directly issued. The generator objects create a graph of nodes which represent the software instructions. The graph leaves are connected to the nodes of other generator objects, for instance the objects of the modify-operations of a read-modify-write cycle.

The result of a transaction conversation is a set of graphs, where the nodes represent the required software instructions similar to the Directed Acyclic Graph (DAG) representation of the intermediate languages in compilers (compare for example figure 5.12). On a final run, the DAG nodes are translated into the instructions of the target language. The best solution is selected by estimation of the execution time. The DAG nodes, which represent the interaction with the device, evaluate the attribute which represents the estimated execution time given by the software symbol.

Input:

```
void Spyder_CPLD::DONE_set( u8 xxx )
{
  #@init@#

  #@DONE.set( xxx )@# ;
};
```

Translated to:

```
void Spyder_CPLD::DONE_set( u8 xxx )
{
  unsigned int    tmp378=0;
  struct ioctlstr tmp577={0};
  struct ioctlstr tmp131={0};

  tmp577.offset = 2;
  tmp577.value = 0;
  ioctl( DeviceChannel, FPGA9080_READ_PCICONFIG_BYTE , &tmp577 );
  tmp378=tmp577.data;
  tmp131.offset = 2;
  tmp131.value = (((xxx & 0x1) << 0x6) | (tmp378 & 0xba));
  ioctl( DeviceChannel, FPGA9080_WRITE_PCICONFIG_BYTE , &tmp131 );
};
```

Figure 5.11: Code example of a proxy method in annotated C-Code. The access statement is translated to a read-modify-write cycle which uses system calls as an interface to the device. The variable `DeviceChannel` has to be determined during runtime.

The DAG nodes for bit operations and variables are assigned estimated execution times during DAG construction, depending on the model of the target machine and the target compiler. For each graph the duration is estimated by summing up this property of each DAG node. This is only an estimation, because of interleaving effects on the bus systems.

Before integration of the translated access method in the surrounding source code, some nodes of the graphs have to be moved to special locations in the source code. Nodes for include-operations of header files, and declarations of temporary variables must be moved to special locations in the source code, depending on the constraints of the target language. In this prototype, the special locations are marked with hooks encapsulated in tags. Within the final translation run, the code of the special nodes is placed at these locations, instead of the location of the device access.

The described prototype has been used for the evaluation of the described synthesis method. As devices and communication networks, the following examples have been used for method verification and evaluation of the code quality:

- A CAN-Controller of the type 82527 by Intel [44] attached to the local bus of a PLX 9080 PCI bridge [86] by PLX Technologie[2]. The PCI-bus protocol is translated to the memory bus of a Pentium processor by Intel. The memory bus protocol bridges to the software side, using the Linux macros for memory access inside the Linux operating system kernel. The translation order is from Linux assembler macros to memory bus protocol, to PCI-bus, to local bus, and finally to the device register.

[2]The bus protocol accepting automaton of the device can not recognise the local bus protocol of the PLX bridge. For this evaluation the automaton has been virtually replaced by an adequate one.

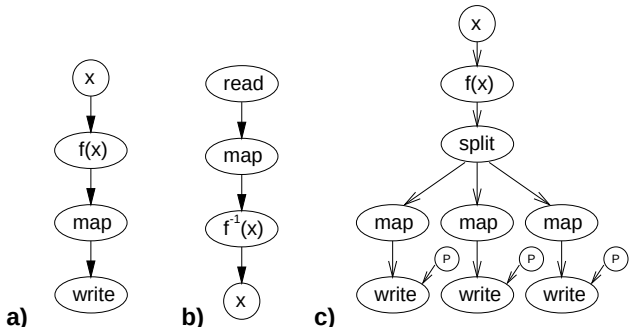



Figure 5.12: Examples for the resulting DAGs. On the left a write transaction, in the middle a read transaction, and on the right a write transaction, which is then split into three separate transactions.

- A PCI-device must provide a special register file with device specific information, the PCI-header (compare specification [83]). This header has been described in combination with a device driver, which allows access to this register set of a PCI-device from the user space.

- The main control device of a Spyder board by X2E [118] which is designed into a Complex Programmable Logic Device (CPLD). The CPLD is attached to the local bus of a PLX 9080 PCI-bridge. The translation chain is analogous to the one in the first example of the CAN-controller. Furthermore, a device driver has been described with its impacts on the local bus, respectively, the protocols which are produced on the local bus by the different user space API-functions of the driver.

For a discussion of the code quality, the latter example is used here. The results can only being compared to hand written code. Other approaches use a different structure and methods. For instance, for a comparison with the Devil approach, a comparable implementation of the `port()`-function is required, which is not available. Hence, the produced code will be compared with a version which would have been generated by hand.

As a starting point, a proxy object for the access to the main control device of the Spyder board has been created. This proxy uses the previously described embedded statements for the access to the device. These special statements are translated within minutes to the target code in `C++` (see as example 5.11). The result can be described in the following way:

1. The tool generates more temporary variables, and move-operations between variables, than in the code written by hand. This is because the tool does not track the usage of temporary variables. The target compiler has to reduce the number of variables during register allocation in combination with lifeline analysis of variables. Nowadays compilers should be able to reduce the number to the same number of variables used in hand written code.

2. Some of the used bit operations have no effect, for example bit-masking of an 8 bit variable with the operation `and 0xff`. Such unnecessary bit operations have to be also reduced by the target compiler.

3. Some bit-masks contain 'holes', all at the same bit location. For example, a bit-mask clears two bits where the information entity is only one bit in width. This is not an error, because the second bit position has not been defined in the device specification. The tool has not been able to look-up the information about this bit in the database. Because this bit position is not occupied in the register file, the hole in the bit-mask is not an error.

4. The tool has selected the transaction instructions with the smallest bitsize. All information entities are at most, 8 bits in width. The access to the register file has the same duration for 8, 16, and 32 bit accesses. The selection of the 8 bit version depends on the costs for bit-operations with a data width of 8 bit, which is less expensive than 32 bit operations.

5. The channel parameter of the interface (`ioctl()`-call) is not set automatically. This parameter is determined during runtime. The software sequence for channel selection is not incorporated into the target code by the tool, and must be added by hand. It is difficult to integrate the placement of this channel selection in the prototype, because the position in control flow of the HAL surrounding application is unknown, and is difficult to determine. In an object oriented approach, this section may be placed in the constructor of the proxy object, and the parameter is a private attribute of that object. For comparison, in the Devil approach, this parameter determination is hidden in the port function.

6. Read operations for a read-modify-write cycle are only included if they are necessary, analogous to hand written code.

This case study has shown, that the resulting code is nearly the same quality as hand written code. The translation times are within minutes for a complete hardware abstraction layer. Due to the modular approach of the method, a change of components or a new device can be integrated quickly. Error prone adaption of bit operations is not required. Furthermore, the prototype has shown, that the method can be integrated into a high-level language compiler by means of an extension of the language with special statements.

## 5.6 Optimisations

The overall execution time of a device driver can be reduced by a reduction in bus traffic. The reduction can only be performed with knowledge of the control-flow of the driver and the device. For the optimisation approaches, the control-flow is assumed to be known. The control-flow analysis is out of scope here (see [68] for control-flow analysis in software). In this section only the starting-points for optimisations will be discussed.

### 5.6.1 Combining of Accesses

The objective of the hardware abstraction layer is to provide an interface for the access to, or for the manipulation of bit-fields in a register. In a sequence of manipulations, bit-fields in the same register can be affected. The idea is to combine accesses to the same register on top of the coding layer into one single access. The pre-condition is, that a change of the access order, or the merger of two transactions, does not influence the behaviour of the device at this point. That means that the register access does not raise pre, storage, or post-events (compare figure 4.3 in section 4.1), which trigger an action in the device automaton.

So for this kind of optimisation by merging consecutive accesses to one register, the bus-interface characteristic and the device behaviour must statically permit this join. This can be expressed in the database by a relationship between the information entities which map to the same set of registers. The set of registers is determined by the bus accepting automaton, and is the collection of registers which can be affected by a transaction on the peripheral bus at once.

The function

$$join_{write} : IE \times IE \rightarrow \{\text{TRUE,FALSE}\} \tag{5.8}$$

determines whether or not, two information entities can be merged for a write transaction. The function has to be provided by the device vendor, respectively, by the device designer.

The optimisation requires information from the level above the coding layer. Therefore, the optimiser has to recognise the transactions in a code block (for code block identification compare for example [66]). Because the optimisation affects the transport layer and has to identify sections for applying the optimisations on top of the coding layer, the optimisation scheme can not be used in the case of the synthesis of a set of library functions. As discussed in the previous section, a synthesiser with access-statement expansion must be used.

One optimisation scheme can be to identify all write transactions to a device in a code block, collect the IE properties, and pass the complete set to the start string generator for the code generator. This will lead to a large collection of feasible start strings, which may be expanded during code generation by further alternatives.

Due to this explosion of alternatives by a large set of start strings, a different algorithm is suggested here. Each transaction is translated individually. Afterwards, the set of affected ($IE_{aff}$), next to the desired information entity ($IE_{des}$), for each write transaction is known. Now the intersection between two write transactions $i$ and $j$ in a code block is determined.

$$J_{i,j} = IE_{aff,i} \cap IE_{des,j}, i \neq j \tag{5.9}$$

If $J$ is not empty, the $join_{write}$ function of the desired information entities of transaction $i$ and $j$ is evaluated. If the $join_{write}$ function permits the merger of the write transactions, the start string is only generated for these IE, because they have the potential for a joined transaction with shorter execution time. This pre-selection of candidates for a merger reduces the risk of alternatives explosion. Again, the estimated execution time in combination with all bit operations has to be determined. Only if the merged transactions yield a shorter execution time, it replaces the single transactions.

The same optimisation method can be applied for read transactions. Here, the function

$$\text{join}_{read} : IE \times IE \rightarrow \{\text{TRUE,FALSE }\} \tag{5.10}$$

determines whether or not, the joined read out of two information entities, results in the same values as the consecutive transactions. Again, the access to a register must not trigger any action which changes the content of the other information entity.

Instead of the construction of newly joined write transactions, a different scheme is used here. The construction algorithm creates different alternatives with different data sizes for each transaction. Here, for each alternative of a read operation, the information entities, which are read out next to the intended one as a side effect, are determined by backtracking. This includes the read accesses introduced by the generation of read-modify-write cycles. No distinction is made whether the read access is an intended or a generated one. Again, with the calculation of the intersection of consecutive read-outs, the information entities, which are read out more than once, are determined. If the evaluation of the $join_{read}$ function permits a merger, the already gathered information by a previous read is recycled, instead of performing a further read out of the same information entity. The recycling is performed by replacing a read node in the DAG graph with a remapping node. This leads to a configuration problem, because a read value can only be recycled if a special alternative is selected. And if this recycling takes place, maybe another recycling is not longer possible, because less information entities are read out as a side-effect. The problem can be solved with dynamic programming which optimises the total execution time of the code block. For the last read out, the execution time for the read and for the version with a recycled value is determined. This number is propagated to the previous read transactions. If the value is recycled, the execution time is propagated to the read out of the information entity. If no recycling takes place, the execution

time is added to the previously performed transaction. The algorithm proceeds backwards in the code block and determines the configuration with the shortest total execution time.

The optimisation results only ín a reduction in execution time if a complete read out can be saved. A partial recycling only leads to further bit-operations, although the read out is performed for the missing information entities. This can happen in the case of read transactions generated for a read-modify-write cycle, because many information entities are read, instead of only one in an intended read. Hence, at an early stage, some recycling versions can be dropped.

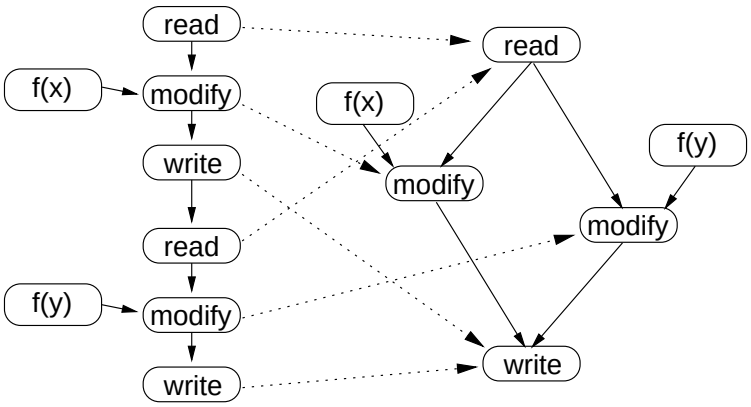



Figure 5.13: Merge of consecutive read and write transactions (left) to single accesses (right).

In all cases, for read as for write transactions, it has to be checked if the number of bus-transactions and hence the execution time is really reduced. The execution time in the bus systems is reduced by replacement of single transactions with a transaction of a larger bitsize. However, an overhead in the bit-operation for separation of the bit-fields is required, increasing the execution time on the software side. In contemporary computer systems the time on the bus-system overwhelms the duration of bit operations, so a reduction of bus traffic with an overhead on bit operations, can lead to a reduction in the overall execution time in total.

### 5.6.2 Explicitly Caching

Some registers of the device can be cached in the hardware abstraction layer. This is an extension to the previously discussed method of combining of accesses, because here the caching is not within a code block; it can be spread across many transactions distributed in the control flow.

As described in the previous section, a read value of a register can be recycled within the code block. This can be extended beyond the boundary of a code block, by a caching mechanism similar to the cache architectures in modern processors. Instead of accessing a register, the cached value is used in read operations.

The cached device register must have a special characteristic for applying a caching strategy. In processor architecture, the memory is not assumed to alter its content without direct investigation of the CPU. So the cache can track all changes by observing the write transactions to the memory. A device register can alter its content by investigation of the device kernel without notification of the processor, respectively, the caching code.

This means, if the content can not be changed by the device kernel, or between two well known points in time, the device does not change the content of a register, and a cached value can be used instead of really reading out the register. During this time, only the write transactions to the register must update the cached value.

In some cases the content of a register can not be altered by the device kernel at any time. For example, some parameters are only set by the CPU and can not be changed by the device itself. This characteristic can be the result of a detailed analysis of the device control-flow, or is directly stated in the documentation.

If this characteristic can not be statically determined, the control flow of the device has to be tracked during runtime, and on each read transaction it has to be decided whether to use the cached value, or to perform a register access. Because of the sequential nature of processors, this leads to a further read of a flag variable and a branch in the control-flow, which can result in a pipeline stall. So this kind of caching strategy can lead to more overhead than a direct read access to the device register. It depends on the target computer architecture, and a detailed Worst Case Execution Time (WCET) analysis of the code required for caching, to decide whether this scheme can be applied or not.

Still, the cached value has to be initialised before the first read access to the device. Therefore, the control-flow of the application, or the order in the use of the proxy routines in the hardware abstraction layer, has to be known. If the target language supports a constructor method, like for example `C++`, the initialisation of the cache can be placed in the constructor method. The nature of this method forces a first read of the register before a transaction to register by the proxy methods can take place. If the target language does not support such an implicit initialisation routine, a place similar to the initialisation routine has to be identified and the code placed there.

However, caching only makes sense if two conditions hold: firstly the management of cached values access to the cached values is faster than the accesses to the device register content, and secondly parts of the drivers and the devices control-flow is known. Thus in each individual case, a special caching strategie has to be selected.

## 5.7 Restrictions and Extensions of the Method

The previous described method still needs some improvements. In this section some topics are presented for future elaboration.

**Statelessness of Communication Channel** For this method of communication channel synthesis, it is assumed that the channels are stateless. The channel configuration is only determined by the independent transactions. A channel initialisation with fixed parameters can be done in the constructor method of the proxy object. Thus, before the first use, the channel is set up. Hence, no other entity is allowed to change these settings.

If the other components in the system can influence the channel and change the channel settings, the re-establishing of the required state has to be done at each transaction to guarantee the correctness of the transport. This leads to overhead. So a further enhancement of the method can be to cross check the required channel settings, and to schedule the modifications to reduce reconfiguration to a minimum.

**Symbolic Variables** From the implementation point of view, the system can not deal with symbolic indexes for registers and symbolic parameters in attribute transformation. In some cases, the access to registers, for instance data fields, is controlled by index variables as iterators. For the abstraction layer, they are symbolic without any information on the content, range, or the interaction step size. But these properties are required to check the alignment of parameters, for example addresses. Some bus protocols are only valid if the transport has a special address alignment. Techniques from the computer algebra must be incorporated here into the system, allowing the range

check, as well as other transformations on the attributes in the attributed grammar. For example, the algebra system can prove whether the alignment (modulo-operation) holds with the properties of the symbolic variable or not.

**Property Checks of IEs** The description of the IEs can be enhanced by constraints on the content which can be incorporated into the coding layer. A value check can encapsulate the write access to the register to prevent the change of the device to an invalid state (see figure 5.14). This enhancement can only be made in combination with an exception policy in the case that the transaction is permitted (compare Devil approach in [61]). Together with the constraint, a strategy for the rejection has to be specified, which is then supported by the synthesiser and the target code generators.

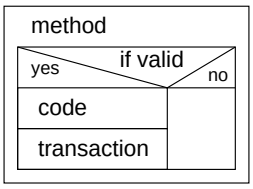



Figure 5.14: Encapsulation of the write access by a parameter check.

**Burst Transfers** At the moment a fixed size for the attributes of a protocol symbol is used. It lacks a concept for the integration of burst transfers. The data field in the attributes tracks the positions of the bits throughout the channels. In the case of a burst transfer, the protocol package transports an almost arbitrary amount of data, hence, this field has to be flexible in size. At this point the methodology has to be enhanced to cover burst-transfers.

## 5.8 Summary

In this chapter, a methodology for the synthesis of the hardware abstraction layer by establishing a communication channel to the device, has been shown through the use of partially known channels. Each channel behaviour is described by an attributted grammar. The complete path behaviour can be constructed by combining the grammar of each section.

The synthesis algorithm for the transport layer works similar to the code generator algorithms used in compiler construction. The algorithm determines the required software sequence, including the individual parameter set for each transaction. Side-effects are compensated by different strategies, including expansion to a read-modify-write cycle. From the set of feasible instruction sequences, the most appropriate one is selected for the transaction, according to the estimated execution time.

The system specification is modular in the sense that the device vendors only describe the behaviour of their systems in the aspects of register file organisation, bus protocol, and routing behaviour in the case of bridges. Along the computer system design, the interconnection of the components is specified and the topology of the elements and routing nodes are determined. This forms the system specification for communication layer synthesis. The locality of the complex specifications leads to easy re-use.

The case study has shown, that the concept of re-use is applicable, and the synthesis method can be integrated into a compiler system for high-level languages, by an extension of the language. The

synthesis of a complete hardware abstraction layer for a device in a dedicated computer architecture takes only minutes, thus the methodology can be easily incorporated into future compiler systems.

In comparison to the Devil approach, not a single channel is used where the complete route to the device with a fixed service access point is covered without havin side-effects. The method presented here can determine the channel entry point, including the required parameter set, and integrates it into an access layer. Thus the method can also be used for the synthesis of the `port()` function of the Devil approach.

In the next chapter, the layer on top of this hardware abstraction layer is looked at. From the device behaviour, hints for the fine grained structure of the driver, and the control-flow in the driver can be derived.

# Chapter 6

# Deriving the Driver Algorithm

In chapter 4, a general approach for structuring as well as the organisation of a device driver, depending on the hardware interconnection has been presented. Each driver component is separated into the concern of the driver component behaviour, and an access layer for transparent communication with the device hardware. The communication layer with its access methods can be synthesised with the method which has been presented in the previous chapter, respectively, the method of the Devil approach [61] can be applied to synthesise access stubs.

The driver behaviour has transparent access to the information entities, respectively, the device variables which control the device behaviour. Due to the hardware abstraction layer they are on the same level of communication. On this level they communicate logically and in parallel with each other. The channel characteristics leads internally to a time multiplexing. From the logical point of view the communication is directly linked.

From the driver behaviour and the desired total behaviour, the driver component on the behaviour level has to be derived. A fully automatic synthesis is not possible at the moment. Approaches for the driver kernel design are Domain Specific Languages (DSL) which shifts the behaviour programming to a more abstract level (compare section 3.2.2). If the interfaces on both sides of the driver kernel are very restricted, well specified, and the driver behaviour is simple, the driver kernel behaviour can be synthesised by the methods of adaptor automaton synthesis (compare section 3.2.3).

In the first approach, the control structures of the driver software have to be established by the programmer and mapped to the DSL. The second approach requires a detailed specification of the interfaces use over a specific period of time and a mapping of the interface ports to each other. These definitions are again provided by the programmer. The experienced driver designer has to derive specifications from the complex structure and behaviour of the device.

In this chapter, analysis methods in the sense of patterns in the device architecture are presented. The results lead to hints for the structuring and the design of the control flow in the driver. The programmer can use these hints to ease their work by knowing what to look for, as well as what to do with the given structures. Firstly, the problem of device driver design is analysed again, under the aspects of driver behaviour synthesis. Secondly, three approaches are presented to derive the behaviour for the driver.

The first approach re-uses the concept of reflecting the structural hardware relationships. With this approach, the middle grained structure and sections of the control-flow can be constructed. With a further refinement of the components semantics, the approach can be used for software synthesis. In a case study, this refinement has been applied to interrupt architectures. For the description of the fine grained structure a visual language has been developed. With the structural information and the underlying schematic of each component, a software synthesis for the interrupt dispatcher is feasible.

The other two approaches deal with the analysis of the temporal behaviour of the device, as in most cases the required access sequence to the information entities for achieving the desired behaviour is unknown. One approach derives the required input sequence by means of the counter examples of a model checker. The applicability is restricted, because of the state explosion in the model checker. Thus, analysis approaches based on the semantic re-construction of certain driver behaviour in order to determine the access order of some signals will be discussed. Main idea of all approaches is the systematic analysis of the cause-and-effect chain in the system.

## 6.1 Problem Analysis

In most cases the documentation of a device is a description in natural language. The starting point of each analysis is a formal description of the device behaviour. Hence, the informal description must be transformed into a more formal specification which can then be analysed due to its underlying semantics. To the knowledge of the author at the moment, *the* language for the description of a device does not exist.

One option would be to use the same language as description language which is used to specify the device implementation. An example of this is the hardware description language VHDL, but as some information is ”cryptic” in the language, this makes the re-construction of the semantical meaning difficult. The hardware description languages lack an appropriate level of abstraction for this analysis.

Due to the different aspects of device hardware design and the different levels of abstractions, sometimes a unified modelling language for hardware devices maybe designed. Up to now the different models which exist next to each other have to be used. Register file structures can be described by tables (see for example the Devil approach [61], or the COMIX approach [123, 80]). Logic constraints can be expressed by if-then-else statements, which can easily be derived from the documentation by translation of sentences in the form of ”When this happens, then that happens”. Ordered actions in the time can be expressed by state machines. Constraints which have to hold at every point in time can be expressed by boolean equations. Sometimes relationships can be expressed on a higher level by arithmetic equations. In some cases, like in the interrupt structure, the behaviour can be expressed by special icons of a visual language with dedicated behaviour. Currently, these different specifications, some expressed only in natural language, can be found in the device documentation. Due to the incompatibility of the formal models, special solvers and translators for each model are required. The best solver at the moment is an experienced human being.

Next to formal structuring, the exploration and translation of the documentation can be seen as a way of learning and understanding the system. The translator expresses the information in the formal model in such a way that it is understood best by themself.

Information which s sometimes hidden is the sub-structuring of the device behaviour into sub-components interacting with each other. These block diagrams are for logical ordering of the system behaviour. In the SpecC design methodology, systems are interacting components, communicating with each other by channels [18]. The logical structure does not have to match with the real structural ordering of the device, because the real structure is hidden from the driver by the facade of the register file. This structure gives some information on the structure and the control-flow in the driver, which is described in the next section.

The driver and the device can be understood as two systems of automata with assigned data-paths running in parallel. The clocking is independent of each other, so the transitions must be synchronised by exchanging of signals. In some embedded systems the CPU and device clocks are synchronous, for instance systems with one CPU and a directly coupled FPGA. The actions of

both automata are still asynchronous due to the difficult timing of sequential software on super-scalar processors. The problem of estimating the execution times of transitions of a State Chart implementation is discussed in [28]. One result of this thesis is that nearly all transitions, which are code sequences, have different execution times. This underlines the asynchronous assumptions about the communication.

The device behaviour and the driver behaviour can be seen as two automata, each controlling an assigned data-path. Together they form the desired total behaviour of the abstract device. This perspective can be reversed, so that the driver and the device are separated implementations of the total behaviour (compare figure 6.1). The separation in combination with asynchronous communication, leads to typical structures for the synchronisation of both parts at the intersection. The characteristics of these structures can be used to re-construct the counterparts on the other side and to get hints on the control-flow. Therefore, the process of splitting automata and data paths is evaluated, and hints for structuring the control-flow in the drivers are derived.

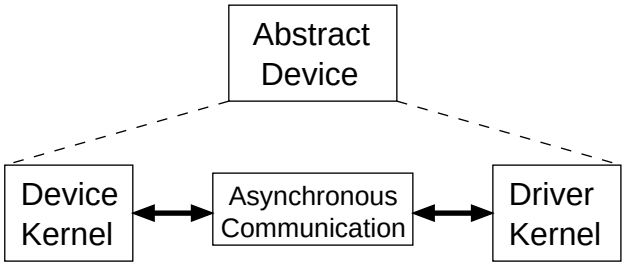



Figure 6.1: An abstract device can be seen as a formation of a driver kernel, a device kernel and an asynchronous communication system in between.

A further problem is the interaction of the parallel automata in the device. The interweaved behaviour makes it difficult to identify the required signal sequence to bring the device into a dedicated state. The sequence of transitions is analogous in order to achieve a desired behaviour of the device. The analysis is difficult due to the complexity of collective actions. A device documentation has the drawback that the information on interactions is spread throughout the manual. Thus, the links have to be re-constructed by the programmer. This process should be supported by an analysis method on the set of device automata. The analysis method has to provide the sequence of interactions with the device in the sense of an expert system. The programmer has to take the suggested sequence and integrate it into the driver control flow.

The next section starts with a derivation of the middle grained driver structure from the hardware structure of a device. Afterwards the splitting of automata and data-paths is evaluated. The last section of this chapter presents an analysis method for the interaction of device automata based on Model Checker.

## 6.2 Structural Reflections

In this section a pattern for the derivation of components in the driver kernel is presented. The method of *structural reflection* will be presented which uses the logical separation of the hardware elements to determine a component structure in the driver. The method will be discussed in the next section, and will be illustrated through two examples, including a case study for a software synthesis tool based on this method.

### 6.2.1 Modelling and Method

Structural ordering of a device behaviour into sub-components is a reduction of chunks to enhance the understanding by hierarchical ordering [124]. The complete complex interweaved behaviour is reduced to separated entities, representing the hidden inner complex behaviour. The level of complexity can differ for each entity. The understanding is eased if the entities cover functional units with well defined and understood behaviours like ”add”, ”multiplex”, etc.

As discussed in chapter 4.1, the device can be separated into a register file, a bus protocol automaton, and the device behaviour. The device behaviour can be split on the next finer level into middle grained blocks. Some blocks still have connections to the register file. The interconnection between all blocks shows the information flow between the entities. This refinement of the model is similar to the refinement strategies in the hardware design processes.

This model of interacting entities inside the device kernel can be used to derive structures in the driver kernel. This method will be related to as *structural reflection* in the following. Each component in the device has its counterpart in the driver, or the component leads to a structure in the control-flow of the driver behaviour.

The device kernel is separated into logical sub-components, with the common front-end of the register file towards the driver. The links between the sub-components show the interrelationship among them (compare delineation in figure 6.2). Some components can act independently. Furthermore, the interrelationship describes a flow of information. Some sub-components do not have a direct connection with the register file. Hence the information exchange is modified by the components on route from the register file. The modification depends on each individual behaviour of the sub-components. The information can be modified by translation of the content, the change of the chronological order, or through the use of special routes which can be taken for a dedicated message. The interaction of sub-components leads to a total behaviour from the perspective of the register file.

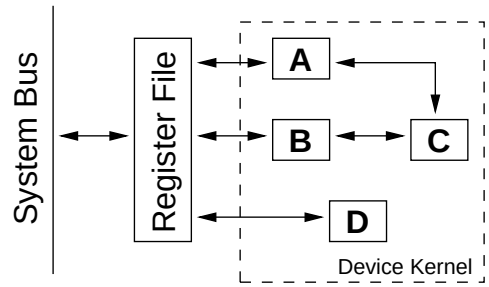



Figure 6.2:
The components A, B, and D can directly interact with the register file. The component C can only be reached via A or B.

Depending on the level of abstraction, the behaviour of each component is ambiguous and has no underlying formal semantics. The dis-entanglement provides as a result the interrelationship graph, and the number of components which interact with each other. These are unambiguous semantics, and can be used to derive structures in the device driver.

The structure of the counterpart components in the driver is a reflection of the relationship structure of the logical sub-components. The route of information flow in the devices sub-components, leads to a route of information flow in the counterparts in the driver. The transformation of messages by the behaviour of the devices sub-components in combination with their counterparts in the driver, has to be taken into consideration for the specification of the behaviour of a driver component. This requires a more detailed analysis of the devices sub-components in combination with a re-construction of the total behaviour of the causal chain. The separation into logical components eases the process, because the device kernel is unfolded and the analysis is reduced to the components which really have an impact on the dedicated component. If the sub-components have formal semantics, the ”inverse” semantic for the driver components can be derived. In most cases

no one-to-one mapping can be given, because sub-component and the counterpart have to fulfill, in total, a desired behaviour required by the target application.

An unambiguous semantic of the separation into sub-components is the number of identical components in the system. The multiple instances of the same sub-component, and hence the same behaviour, leads to special structures in the control-flow in the driver counterparts. The driver has to select which instance has to be interacted with. This leads to the control-flow patterns *for which* ($\rightarrow$ branch) and *for all* ($\rightarrow$ loop). For each instance of a sub-component, a dedicated counterpart exists in the driver structure. The layer towards the application has to direct interactions to a dedicated counterpart, or has to interact with all components, for instance during initialisation.

The method of structural reflection will be illustrated in the following section through the example of a Controller Area Network (CAN) controller. Because of the coarse grained structure and the lack of a formal semantic for each component, only hints for the driver structure can be given. The following section describes the use of this concept to derive software for the interrupt system of an embedded system on system level. Due to the more specific aim of the software, and dedicated semantics of the fine grained hardware components, a synthesis method for the interrupt dispatcher software and the configuration code can be given. In both examples, the concept is not a top-down synthesis method for the specified components, because the described behaviour is not synthesised. A somehow ”inverse” behaviour is produced instead.

### 6.2.2 Example CAN-Controller

In this example, hints for the organisation of the internals of the driver kernel will be derived by the method of structural reflection from the internal structure of the CAN-controller. The structures lead to patterns for the control flow and the required number of components in the driver kernel.

According to its documentation, the CAN controller 82527 [44] from Intel can be structured as depicted in figure 6.3. The controller interacts with the network by a transceiver and an adaptor to the bus medium. The transceiver selects a message to be sent from the message buffers 1–14, or it can store a received message according to the filter rules in the buffers 1–15. The interaction of the components is controlled by a master component. Furthermore, the device provides programmable parallel I/O ports for free use. The main control component has interconnections to nearly every other component in the device.

Analogous to the structural descriptions on system level, the interaction of the driver with a sub-component leads to a route from the register file to the sub-component. For instance, the transceiver is reachable from the register file by two major routes: via the main control, and via the message buffers and the selector. The route for the data is ambiguous because of the multiple instances of message buffers, and the multiplexing character of the selector. Furthermore, another interconnection shows that the multiplexer is also influenced by the main control. To select and configure a path for the data, the interaction of these components must be analysed and formed into a dedicated component in the driver kernel. On the other hand, the programmable parallel I/O ports are independent from other components, and can be controlled by an independent component in the driver (compare UML diagram in figure 6.4).

The structuring of the device with multiple instances leads to hints for the control flow in the driver. The block diagram 6.3 shows multiple instances of a message buffer 1–14 and a further special message buffer 15. The behaviour of the buffers 1–14 is the same. The hardware design here is similar to the use of a fixed number of class instances in software. Message buffer 15 is special as it only receives messages with the use of a shadow buffer. For the driver control-flow, the multiple instances of the same behaviour leads to structures in the control-flow. In this case the initialisation of a message buffer has to be performed *for all* message buffers. If a message has to

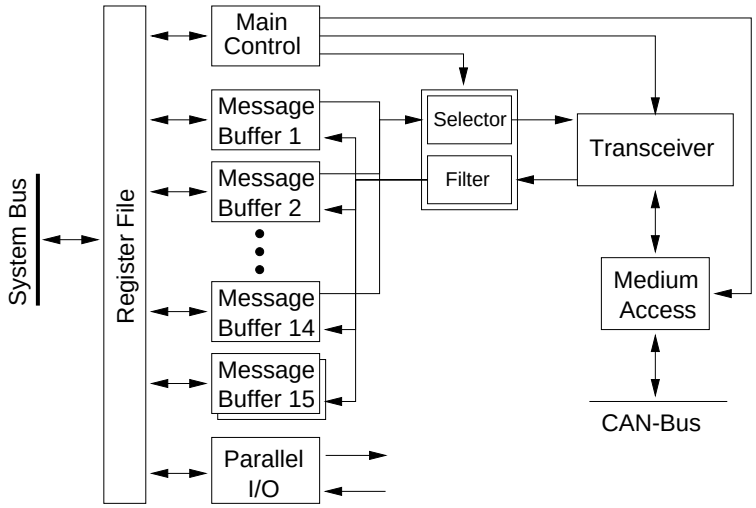



Figure 6.3: Internal logical structure of the CAN82527 network controller derived from the device documentation [44].

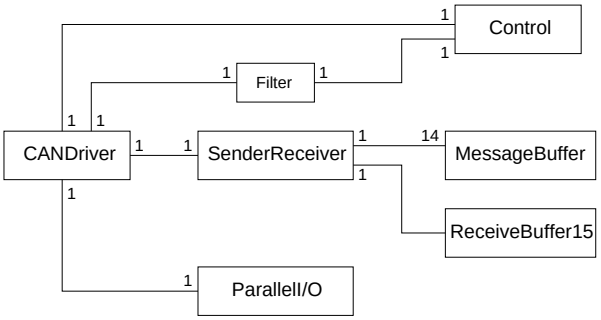



Figure 6.4:
UML diagram of the derived class structure by reflection of the hardware structure (compare figure 6.3).

be stored for sending in a message buffer, a dispatcher must decide *which* buffer should be used. On the other hand, on a receipt of a message, an identifier has to find out *which* buffer has received the message. The message must be fetched and be passed to the application. Thus, all methods of the driver which have to deal with the buffers must be able to select one or a set of buffers. This control-flow patterns can be found (or have to be placed) in the facade of the message buffers, the `SendReceiver` class in the figure 6.4. Hence multiple instances of the same sub-component type, lead to the pattern *identifier* and *dispatcher* (*for all* can be interpreted as special form of the *for which* operation) in the control-flow. The inner structures, for instance the loop bodies, are actions with a selector parameter for identification of the affected sub-component instance.

The ordering and the structural reflection has given hints for the organisation and interaction of the driver components. Due to the coarse grained level of abstraction used in this example, only suggestions can be made here. In the next section, the behaviour of the device component are more restricted, which makes software synthesis of driver counterparts feasible.

### 6.2.3 Example Interrupt Systems

In this section, the application of the method of structural reflection on the fine grained description of the interrupt system of an embedded computer system will be depicted. Interrupt systems are designed to reduce overhead in the CPU, caused by busy-waits for signals from the device. According to an event in the device, an identifying flag is set in the register file as part of the synchronisation message. The device raises an interrupt signal to inform the CPU of the changed

register content. The signal line for the interrupt request is merged in the computer system architecture with the lines of other devices. In most cases the information content of the signal at the CPU is only one bit (one line)[1]. One task after the CPU has taken the exception (after completion of the actual instruction) is to identify the origin of the signal, and branch the control-flow to the interrupt service routine assigned to this interrupt source. With a formal description of the architecture of the interrupt system, an identifier and dispatcher software can be synthesised. Furthermore, the configuration settings can be determined and initialisation code can be generated.

**Modelling** The documentation for the interrupt system of a device is a semi-formal specification in natural language intermixed with schematics in a broad outline, similar to the rest of the device descriptions. A first approach for a more formal description by means of a logic schematic can be found in the documentation of the PLX9054 PCI bridge [87]. The graphical representation of the relationship consists of sources, switches, and inclusive OR-gates for the join of signals.

This approach can be extended by further elements which can be found in the hardware implementation of interrupt systems. The elements form a graph of interacting components with specific behaviour. The method of structural reflection can be applied to synthesise the software for the determination of the origin of an interrupt request. These elements are registers of different behaviour for storage of flags, multiplexers, inclusive OR-gates with priority, OR-gates with a register for source identification, etc. A complete list of symbols and abstract behaviour is shown in appendix A.

The graph of interacting components can be seen as a visual description of the interrupt system. It can be used for analysis, software synthesis and determination of configuration settings. The temporal behaviour of each component is simple. The registers for storing the event of a request are the only elements which have a state, and therefore a changed temporal behaviour. All other elements only have a logical behaviour, and do not change their temporal behaviour (only due to the explicit re-configuration of the system settings).

**Software Synthesis** The method of structural reflection can be used to synthesise the software for determination of the origin of an interrupt request. The exception gate is the sink of the interrupt signal in hardware, and the bridge to software. On a request, it invokes the branch from the actual program flow to the interrupt dispatcher routine (IDR). The IDR identifies the active interrupt source, and invokes the assigned interrupt service routine. The synthesis of the IDR by means of the method of structural reflection will be shown here.

To reduce overhead in the IDR at runtime, the system description is reduced to the really required subset in a pre-synthesis phase. In a computer system, not all IRQ-sources are enabled and can request an interrupt, and not all possible paths are used. The synthesis of the request dispatcher should only consider the used sources and configured paths. The specification of used sources leads to routes in the graph. If some routes are determined at runtime, they are also marked as used (required). The unused parts of the system specification are erased. OR-gates with only one input are replaced by direct connections, and the same is done for multiplexers. Thus dead branches in the description tree are erased which eases the synthesis process, and dead code is avoided by construction.

The synthesis process of the IDR starts at the exception gate and translates the structure towards to the interrupt request sources. The interrupt request signal flows from the source to the exception gate. From the perspective of each exception gate, the system description has a tree structure. The trees are overlapping inside the description graph. The software for determination of the active

[1] Even a vector-oriented interrupt system provides for the invoked interrupt service routine the information content of one bit, i.e. the invocation itself.

interrupt request source has to descent the tree and follow the possible information flow in the graph of interacting components.

The synthesis algorithm starts at the exception gate and follows the tree structure in a with-first search. The behaviour of each component is replaced during the descent by a corresponding component in an intermediate representation. Here the semantic of each element of the specification is reversed, and a corresponding counterpart is instantiated in the intermediate language. If a leaf with an active source is identified, an element for the invocation of the assigned interrupt service routine is inserted. The tree structure of the components is preserved during synthesis and is transformed into a control-flow tree. Hence, it is not a direct translation of the components into other constructs with the same semantic.

To illustrate the replacement of a corresponding behaviour, the example of an OR-gate replacement is shown in figure 6.5 and 6.6. Each incoming line of the OR-gate can be active, so each line has to be tested by diving into the structure and testing of subsequent elements. Thus, the replacement leads to a software sequence of further test-components for each line which dive into the graph. If an assigned register to the OR-gate contains the active line, this information is used in a switch-case-structure. Thus, the joining of information by an OR-gate leads to a branch-structure in the dispatcher control-flow.

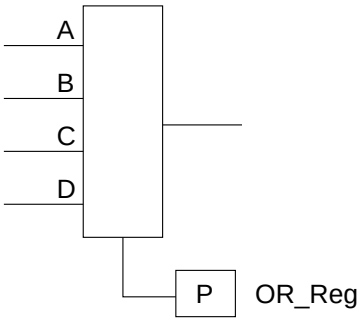



Figure 6.5: OR-Gate with an assigned register `OR_Reg` which contains the active input line.

```
...
  switch( OR_Reg.getstatus() ){
  case A:
    ...
    break;
  case B:
    ...
    break;
  case C:
    ...
    break;
  case D:
    ...
    break;
  }
...
```

Figure 6.6: Generated source code section of the IDR. Depending on the active line a subsequence is executed.

For practical reasons, the model of interacting components of the description of the interrupt system can include the concept of frames, and instances of sub-graphs which can be placed in the frames. Thus, the description effort is reused by multiple instantiation of sub-graphs in the frames. Furthermore, this model provides a dynamic exchange of subsystems by dynamic placement in the frame. This concept represents the quasi-static replacement of boards in slots, the dynamic exchange of re-configurable devices, or the assembling of systems in Digital DNA [20]. The corresponding counterparts of the interfaces at the frame borders must support the dynamic assignment of software structures for descending into the sub-graph instance.

**Determination of Configuration Settings** The presented interrupt-system description can be used to derive configuration settings for the interrupt system components. For each interrupt source

which is used during runtime, a traversable path from the source to the exception gate has to exist. In the dynamic case, respectively, from a source to an interface of a frame, or from an interface to the exception gate. All feasible paths can be found with search algorithms in graphs. Result are sets of routes for each used interrupt source. From the set of routes for each source, one is selected which best fulfills a priority scheme between all sources. The priority is determined by the elements on the used path. The path selection is done by the possible configurations of the multiplexer, so the priority relation is an argumentation on the multiplexer configurations. If a suitable configuration is found, the setup code can be generated. All switches and multiplexers on a path must be enabled, and also be set to route the signal from the source over the selected lines to the exception gate.

**Case Study** The here presented synthesis method has been evaluated in a case study. The interrupt architecture of the MPC555 has been specified according to the documentation in natural language of the interrupt behaviour in [24, 65]. The MPC555 has 120 internal interrupt sources due to the peripheral devices on-chip. Furthermore 8 external interrupt signals can be feed into the processor. The MPC555 is used as main CPU in the RABBIT computer system [122]. The standard configuration of the system with one FireWire-board and one FPGA-board, leads to further 35 external interrupt sources for the MPC555. A reduced size copy of the completely developed schematic can be found in appendix B. The hierarchy of frames and sub-graphs can also be found in the appendix.

A detail of the specification is shown in figure 6.7, which depicts the visual description of the interrupt sources in the real-time clock (RTC) of the Motorola microcontroller MPC555 [65]. The sources SEC and ALR can be enabled/disabled by switches. The registers store the event of the interrupt request, and both signals are joined by an inclusive OR-gate. The multiplexers different priority levels (not show here) can be selected.

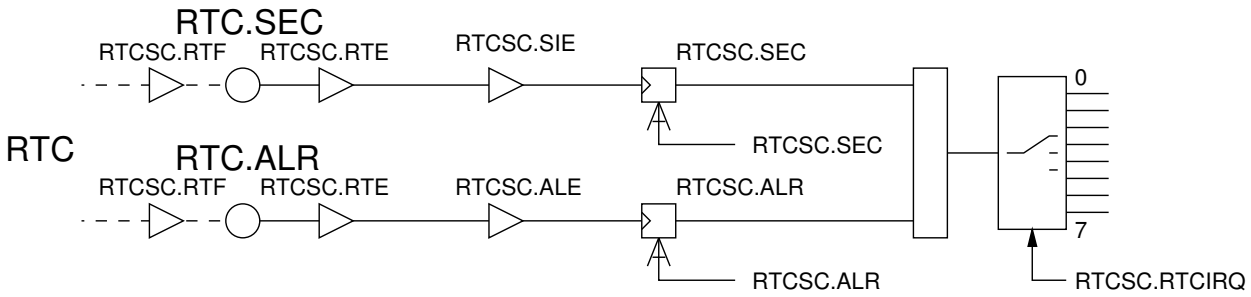



Figure 6.7: Interrupt system schematic of MPC555, a detail part real-time clock (RTC). See text for description.

For the software synthesis a prototype has been implemented which performs the synthesis of the interrupt dispatcher routine. The system description is read in, and is reduced according to the system configuration. The graph is translated to an intermediate representation with the counterpart behaviour. This intermediate representation is then translated to the target source code, and the register accesses can be resolved by means of the synthesis method presented in chapter 5. In the MPC555 documentation [24], examples for interrupt dispatchers in assembler code are presented. The previous method has been applied with the same use-specification, and the prototype produced the same dispatcher code.

The prototype has been extended to calculate the configuration settings for the different components depending on a usage specification. Again, the results were compared to the presented

examples in the documentation. Only the code order of instructions in the initialisation sequence was different.

**Summary** This section has shown, that the concept of structural reflection can be used for the synthesis of software in the field of interrupt systems. This is possible, due to the simple and easy to describe behaviour of the elements, and a simple desired total behaviour of the system. The case-study has shown that the method can be used for automatic synthesis, and can reduce the development time of efficient interrupt dispatcher in operating systems. As a trade-off, the system specification can be used to determine the configuration of the interrupt system.

A further trade-off of the translation of the documentation in the case-study was the identification of ambiguous paragraphs in the documentation, which have to be compared with the real system behaviour. Thus, this method can be used as a cross-check to the natural documentation, or as add-on to the documentation.

### 6.2.4 Summary

In this section, the structure and the behaviour of the elements of the device kernel have been used to structure the driver kernel by structural reflection. The components have an abstract behaviour and only in special cases like the interrupt systems, the behaviour is restricted in such a way, that the software of the driver can be directly derived. In the other cases, the structural patterns lead only to hints for the control-flow. On lowest level a device kernel consists of logical elements and sequential parts by means of automata. The impact of these elements on the driver software is analysed in the rest of this chapter.

## 6.3 Classification of FSM elements

In this section a manual classification of the exchanged signals is worked out, and the impacts on the device behaviour are analysed to construct parts of the driver control-flow. This classification can enhance the understanding of the hardware behaviour on the exploration of the hardware by the driver programmer and also increase the learning curve.

A device and its driver can be seen as the split of an abstract device. Depending on the splitting of the data-paths and the automata, signals are introduced into the systems which can be classified to derive hints for the point of appearance in the driver control-flow.

As a model for the device behaviour a Moore-automaton with assigned data-path is assumed. As discussed before, a driver and a device can be seen as a separation of a complete system with an asynchronous communication channel in between both parts. Depending on this segregation in the real world the classification is discussed.

### 6.3.1 Splitting of Automata

A splitting of an automaton with insertion of an asynchronous communication, introduces a set of communication states and exchanged symbols into the system. In figure 6.8–6.10 the steps of a separation are shown by an automata example. Firstly, the intersected transitions are replaced by a sender and a receiver state (6.9). Secondly, the sender states and the receiver states on each side are merged to cross product states, depending on the existence of a path in the other automaton part, from receiver to sender (compare automaton B in 6.8 $\rightarrow$ 6.10) to re-link the control-flow. This leads to a fully interlocked protocol between the automata.

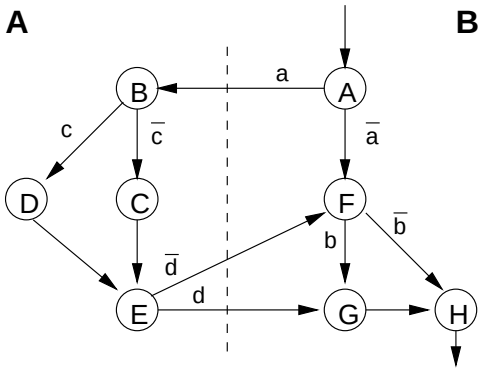



Figure 6.8: Splitting of an example automaton at the dashed line into parts A and B.

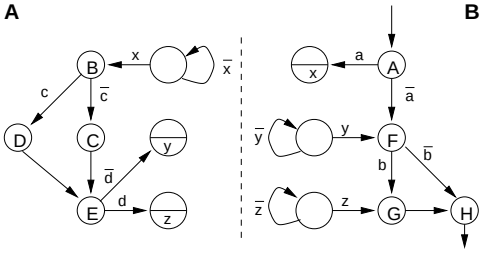



Figure 6.9: Exchange of splitted transitions by sender and receiver states.

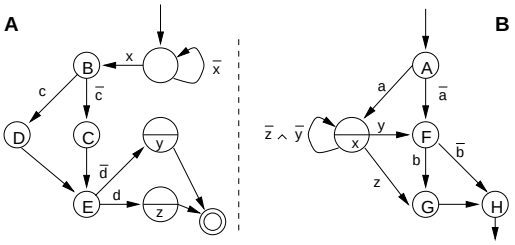



Figure 6.10: Merge of sender and receiver states on each side to re-link the control-flow.

The newly introduced signals exchanged between the automatons are denoted $\Sigma_{A,B}$ and $\Sigma_{B,A}$ here (compare figure 6.11). The automaton A can now be defined by the tuple:

$$A = (Q_A, \Sigma_A, \delta_A, q_{0,A}, F_A, O_A, \Omega_A) \tag{6.1}$$

with

- the set of final states $F_A \subset Q_A$
- the start state $q_{0,A} \in Q_A$
- the set of input symbols $\Sigma_A = \Sigma_{A,X} \cup \Sigma_{A,B}$
- the transition functions $\delta_A : Q_A \times \Sigma_A \rightarrow Q_A$

and the output $O_A$ of the automaton coupled to the states by the function set $\Omega$ defined as

$$\begin{aligned} O_A &= O_{A,X} \cup \Sigma_{B,A} \\ \Omega_A &= \Omega_{A,X} \cup \Omega_{B,A} \\ \Omega_{A,X} &: Q_A \rightarrow O_{A,X} \\ \Omega_{B,A} &: Q_A \rightarrow O_{B,A} \end{aligned}$$

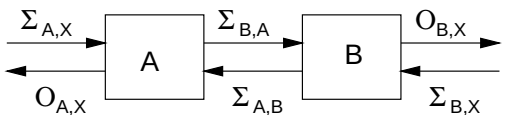



Figure 6.11: Insertion of communication symbols $\Sigma_{A,B}$ and $\Sigma_{B,A}$ between split automata into section A and B.

The automaton B can be defined analogous with indices $A$ replaced by $B$, and $B, A$ replaced by $A, B$ respectively. Let in the following discussion the automaton B be the part in the device, and A the part in the driver.

Now some patterns can be defined:

**Definition 1** *A* flag *is a signal which shows the current state of the automaton by presenting it to the other automaton. It is element of the output symbols set $\Sigma_{A,B}$, respectively, $\Sigma_{B,A}$ which is used for communication between the automata.*

A flag is part of the newly introduced signals, and shows the other automaton that the automaton now is in a state, where the other one has to proceed. In figure 6.10 the automaton A has the flags y, z and the automaton B has only the *flag* x.

On the other side the flag is evaluated by a *trigger*:

**Definition 2** *A* trigger *is a symbol $t \in \Sigma_{A,B}$ in automaton* A*, respectively $t \in \Sigma_{B,A}$ in* B*, which lets the automaton leave a state in the sense that:*

$$\begin{aligned} \delta(q_i, \bar{t}) &= q_i \\ \delta(q_i, t) &= q_j \\ q_i, q_j \in Q, q_i &\neq q_j \end{aligned}$$

In comparison, in the Devil approach [61] "a *trigger* behaviour means that a write (or read) access to the variable induces a side-effect (i.e. an action) on the controller". This can now be refined so that the device automaton leaves the state of busy-wait on receipt of a *trigger* symbol.

In practice in the context of devices, the number of *trigger* symbols is limited by the coding of the symbols exchanged via the asynchronous communication channel between the driver and the device automaton. If the coding of a *trigger* symbol exceeds the maximum size of atomic exchangeable information, the symbol has to be split into two consecutive exchanged symbols. On the sender side, the *flag* $t$ is replaced by the split symbol $t'$ and $t''$, which is presented to the other automaton in a sequence of two consecutive states. The receiver side is changed accordingly (see transformation of the states in figure 6.12). Sometimes, the split of the flags and trigger signals is performed by the hardware designer, only to provide a more convenient interface to the device with eased signal coding.

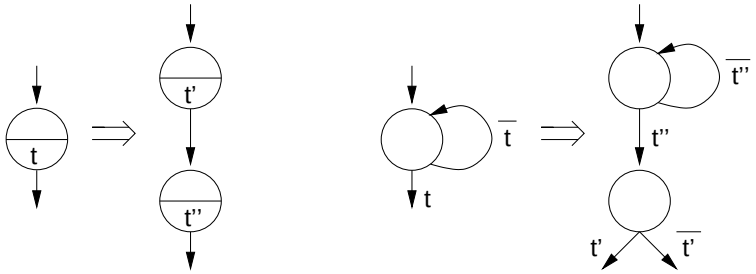



Figure 6.12: The split of the exchanged signal $t$ to $t'$ and $t''$ leads to a split of states for the sender side (left) and the receiver side (right).

One symbol of the split keeps the characteristic of a *trigger* on the receiver side (symbol $t''$ in figure 6.12). The second newly introduced symbol will be denoted as *control*, because it controls

a branch in the receiving automaton (compare figure 6.12). The *control* signal must be transported before the associated *trigger* signal is exchanged. The *control* signal is stored in the communication layer, for instance the register file of the device. If the signal is not exchanged before the *trigger* signal, the automaton may take the wrong transition, because the correct *control* signal may being exchanged too late by the asynchronously operating communication.

In the automaton path the branch is a successor of a *trigger*. A specification of the exchanged signals by simply tracing the path in an automaton, leads to a specification for synchronous systems. With a synchronous approach for signal exchange according to [82, 76], the synthesis process for an adaptor automaton generates a sequence where the *control* signal is exchanged after the *trigger* signal. Here, due to the asynchronous behaviour of the communication channel, the exchange of the *control* signal can be performed by the channel in time or too late. Thus, depending on the communication channel, the order has to be exchanged in comparison to the order in the automaton path. On the other hand, in the asynchronous world the *control* signal must not be given too early, if, due to the symbols coding, it is also used in other paths of the automaton. The analysis requires a determination of a time point (state of the device) from then on until the *trigger*, it is secure to exchange the *control* message. The point in time is in most cases signaled by a *flag* from the device. The pattern for the control-flow in the driver, is that the exchange of the *control* signals is enclosed by the exchange of *flags* and the *trigger*.

It must be taken into consideration that the *trigger* may not have an explicit value. The pre-, post-event, even the event of a register access, can act as the trigger of the device automaton. For instance, the FIFO message buffer of TSB12LV32 Linklayer chip [109] has two registers to fill the buffer. A write access to the one which is marked as ”last data”, uses the post-event to act as a trigger for the transmission start of the buffer.

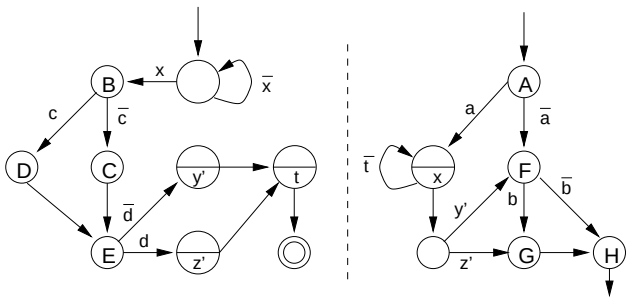



Figure 6.13: Change of message coding into *control* signals $(y', z')$ and *trigger* signal $(t)$. The signals $(y', z')$ are stored in the register file (not shown here).

So a device driver programmer, who re-constructs the automaton A in this case, has to classify the exchanged signals according to *flag*, *control*, and *trigger*. According to the classification, the required sender and receiver states in the driver have to be implemented.

### 6.3.2 Split of Data-paths

A split of a data-path with an asynchronous communication channel, is a bit more complicated due to its non-discrete timing behaviour. The logical elements of the data-path cause a delay of the data-flow. Thus, at the intersection, a flag must signal to the communication channel that the data is processed, the result is valid, and that can be transported. During transportation it must be assured that the data is still valid and coherent on the sender side. This requires storage like a buffer at the intersection, or a stall in the data processing until transport completion. On the other hand,

the communication channel must signal the completion of the transport to both sides (compare signals, data ready and transportation complete, in figure 6.14). In some cases, no processing stall, nor an endless buffer can be installed, so only a flag is installed which signals a violation of the data coherence (compare for example message buffer overflow in CAN82527 [44]).

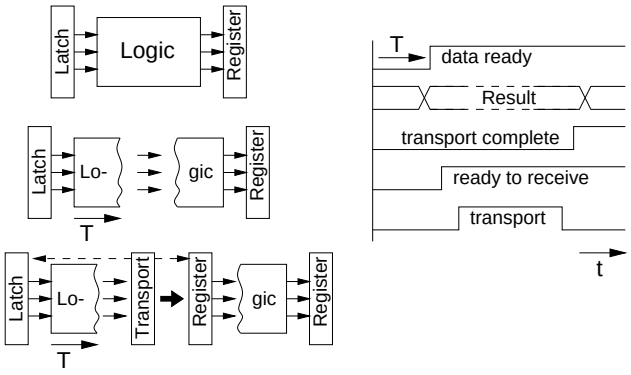



Figure 6.14: Split of logic block and insertion of register for data transportation.

On the receiver side the completion of the transport must be signaled to enable the data processing. Vice versa a flag must signal to the communication channel that it is allowed to store data in the incoming buffer[2]. Again it is a fully interlocked communication due to the asynchronous communication channel. On the receiver side, the order of the received data can have a direct impact on the result. For instance, a trigger in the automaton is released by the result of the logical operation. So on the receiver side, by means of shadow registers, it has to be assured that the result is independent of the order, or a protocol for the exchange order has to be specified which prevents incoherence.

This classification of the signals can be used to control the communication channel. In most cases the channel and the transports are initiated by the driver. Thus, the control-flow of the driver must readout the flags of the device data-path and provide the flags to the device. Some of the signals can be part of the control path in the device, because a delay is difficult to implement in hardware and is rounded up to the duration of a time unit of the implemented automaton in the control path. The identified signals have to be evaluated at the points in the driver control-flow where the exchange of data between the driver and the device is located.

### 6.3.3 Conclusions for Device Driver Design

The exchanged control information between device and driver can be classified into *triggers*, *controls*, *flags* and *raw data*. These signals can be identified in the device and with these signal classifications, suggestions for the position of interaction in the driver control flow can be made. The presentation of a *flag* in the device, signals to the driver the current state of the device. The driver can react depending on this status information. *Data* can be exchanged and *controls* can be set, with no impact to the device behaviour while the associated *trigger* is not sent. This leads to a control-flow pattern for the interaction which is shown in figure 6.15.

The list of impacts the *trigger* signals have must be provided by the device designer. The classification and the relationship can be used to derive control-flow fragments. These fragments have to be embedded into the control-flow specified by the desired total behaviour of the driver.

[2]A storage is required due to the assumed time multiplex of the communication channel to emulate the full data width.

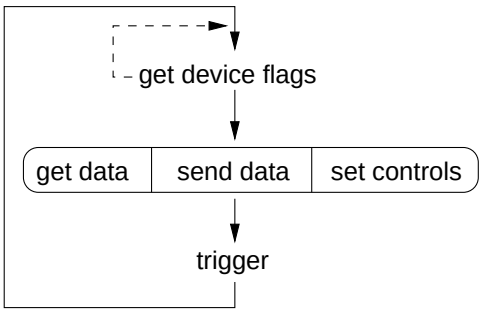



Figure 6.15:
General control-flow of exchanged signals. Data exchange, and the set of *control* signals are enclosed in the *flag* access and the *trigger* release. Branches depending on application demands are not shown.

In this section, the behaviour of a single automaton has been analysed. Due to the massive parallelism of hardware, the interaction of several automatons in a peripheral device have to be taken into consideration. An analysis method for this interaction in the context of driver design is presented in the next section.

## 6.4 Control-Flow Analysis with Model Checker

In the previous section, only single automata and a single data-path line have been discussed. A device is massively parallel hardware, and therefore, many entities with data-paths and control automata are implemented and work in parallel. In this section the analysis of the control-flow in parallel automata is discussed. The aim is to determine access sequences to get the device into a dedicated state with respect to the entangled automata structure. Again this method can be used for exploration of the device behaviour during the learning on the system.

For an automated synthesis of adaptor-automata, according to the approaches in [82, 76], a mapping between a sequence of input signals to a sequence of output signals has to be specified. The mappings, and in most cases even the involved signals, and the sequences on the hardware side, are unknown. In this section the derivation of the required signals to force a device into a dedicated state by use of counter examples of a model checker is discussed. A model checker is used because it is able to analyse parallel systems which are described as automata. The temporal behaviour of a device is implemented by the assembly of interconnected automata, which together forms a product automaton.

Model checkers are specialised for analysis of product automata by comparing the state transitions with a desired behaviour, specified as Concurrent Temporal Logic (CTL) formula, or similar specification languages [50, 16]. One typical analysis checks that the automaton can not reach a dedicated state, which models a fatal error state. If the verification fails, a counter example is generated for the formula that does not hold. The counter example shows a list of transitions to reach this state. The expert can analyse this sequence to find the cause for the misbehaviour of the analysed system.

This type of analysis can be used to find a path from the initialisation state to the dedicated state. Here the device should be set, for instance, into the state `send message`. The sequence and the involved signals (information entities) which have to be provided to the device kernel are often only partially known. The aim now is to prove how not to reach the dedicated state. So the model checker has to prove ”never reach the state `send message`”. The state is reachable, otherwise the device is not able to work, and the model checker finds a transition path in the automaton that leads to that state. The counter example is *one* valid sequence of signals/states to reach this state.

The required input signals, the manipulation of information entities, must be modelled as automata, because most model checkers use state-based automata for system specification. The transitions of those automata represent the changes of input signals required to bring the ”main” au-

tomaton into the desired state.

In this section the modelling of the register file and device/driver interaction is shown. Afterwards, results of the proposed analysis method are discussed, and the restrictions of the method are depicted. The technique of device behaviour modelling and state reduction of the system is beyond the current scope. The focus here is the special case of finding the required input sequence for a device. Only the modelling of the register file, and the emulation of the driver behaviour within the model is described.

### 6.4.1 Modelling

Most model checkers require a system specification in the model of a Kripke-structure [59, 50]. This means that the state based automata are totally described. The system itself must be closed, which means the complete environment must also be modelled as an automaton within the system specification. All parts of the system specification should be at a high-level of abstraction to avoid the risk of the state-space explosion.

The device consists of the parts device kernel and register file with the bus protocol automaton. Depending on the assumption that the communication between the driver kernel and the device is transparent up to the information entities, the system can be separated into three parts: device kernel, information entities, and the driver kernel. Due to restrictions by the communication interface (time multiplex, etc.), the limited bandwidth is included in the specification of the behaviour of the information entities.

Each information entity is modelled as its own automaton with the allowed transitions of state. The transition can be triggered from the device kernel or from the driver side. Within these restrictions, the model checker can non-deterministically select a transition, so that it fits best for the proof, or respectively, for the counter example.

The aim of the model is to find the required sequence of accesses to get the device into a dedicated state. The access can be interpreted on the driver side as a command for the information entity automata to change state. A write command with a special pattern is the same as ”set information entity to state”. In the first step the allowed commands do not have any restrictions in appearance, so they are modelled as free running automata. The transitions in this automata depend on the commands, and on the device kernel side. The command automata can stay in an idle state with the interpretation ”change nothing” (compare figure 6.16). This is required due to the sequential behaviour of software, which occasionally does something different from controlling a device, due to the time multiplex on the communication channel which allows only a small number of commands at a time.

The time at that the command automaton leaves this idle state can be non-deterministically chosen by the model checker, because the access sequence is yet unknown[3]. The model checker can now choose which command to be taken, and when it is required, depending on the transitions of the other automata.

Tests have shown that a totally free running command automaton can produce senseless commands for transitions to states which are already reached. The signal does not change, but bus-traffic is produced if such a non-filtered sequence is used in a driver. Hence the command automata are interlocked and can only invoke a change if the state is not already reached at the moment (see example in figure 6.16).

At this stage of modelling, the system specification allows that all commands are issued in parallel. The restriction of limited bandwidth by the bus-interface requires a further refinement by an automaton, which emulates this bus behaviour. Its states represent the currently accessed register

[3]Unknown, if the device documentation does not explicitly suggests a sequence.

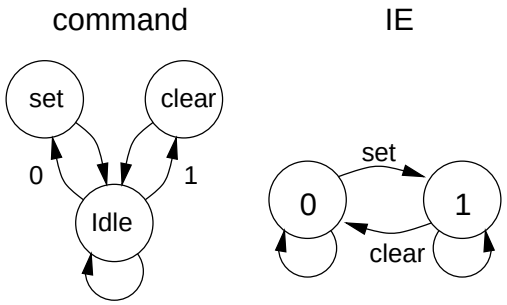



Figure 6.16: Information entity with two states (0,1), and a non-deterministic command automaton which can invoke state changes (`set`,`clear`).

with its bandwidth. This means, information entities can change state, respectively, can receive a command if their information entity belongs to the currently accessed register. How many registers can be accessed at a time depends on the bus-interface and the bus protocol accepting automaton. Again an idle state represents no access by the driver.Thus, the driver kernel in combination with the bus-interface, is modelled by two non-deterministic automata which represent the accessed register, and the issued command to change an information entity state.

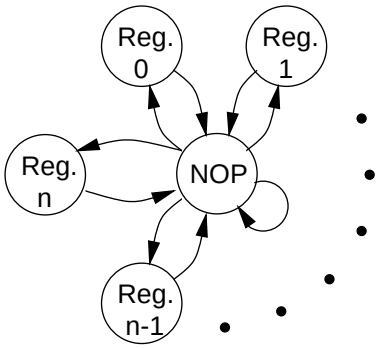



Figure 6.17: Control of the register access by a non-deterministic automaton, which states represent the accessed register.

The behaviour of the device kernel, including the environment of the device, is modelled as an own component of interacting automata. This model bases on the documentation, or is derived from a formal specification by the vendor. The modelling of the device and its environment is not elaborated on here. The model of the device kernel, the information entities as automata, and the emulation of the driver access to the register file is, used to determine the required sequence to bring a device into a dedicated state.

### 6.4.2 Use of Model Checker Results

The aim of a driver programmer is to get the device into a dedicated state. This requirement is inverted and the analysis by the model checker validates that this state is never reached. The counter example shows the state transitions in all automata which take place during the transition from the initial state to the dedicated state. Only the commands from the driver to the device are of interest, thus this list has to be filtered for the commands to each information entity which is eased by giving the automata a special prefix. The result is a sequence of required accesses to the device.

This sequence must be embedded into the control-flow of the driver, analogous to the fragments and control-flow hints in the previous sections. A dedicated driver method is responsible to bring the device into the dedicated state. The access sequence in that method, in addition to all previous accesses by the other driver methods, must be the in same sequence as presented by the model checker.

As a case study, the behaviour of the CAN controller 82527 by Intel has been specified according to its device documentation [44]. The SMV [59] model checker has been used as analysis tool. The tool is well established and has an easy input language, and an easy to parse report of the counter example. But the results can not be used. For the model checker, a validation fails if *one* counter example for the given CTL formula is found. This counter example is allowed to have a nearly infinite length, the only characteristic is that this sequence of transitions does not hold the CTL formula. So many of the reported sequences have been useless because of senseless loops, and minimal sequences are demanded to utilise only the required communication traffic to the device.

Thus, the specification of the CAN82527 has been translated to the input language of the RAVEN [96] model checker. This tool is able to handle time constraints on the transitions, and hence it can find minimum sequences of transitions between one state of the system to another one. Instead of just a counter example, a witness example of the minimum sequence is reported. The time model of RAVEN leads to a larger state space if transitions do not change within one time unit. Advantage of the system is that the specification of the device transitions can now have a duration which brings the specification closer to the real device, where some actions are of longer duration. In a model with a unified transition time, like in SMV, the modelling of this behaviour is rather difficult, because counting automata to emulate time has to be built.

From the CAN82527 the parts shown in figure 6.18 have been specified. For the test of the method some access sequences have been generated. The model checker finds the required settings in the correct order, in accordance to the complex interrelations of the automata. A disadvantage is that the sequences are intermixed with useless transitions. If an automaton is not related to the target state, it can change its state randomly. This does not lead to a longer path of the sequence, if the settings are made in parallel to really required settings. For example, the CAN-bus synchronisation automaton of the controller has been modelled with the minimum of time it takes to synchronise it to the bus under ideal circumstances. During this time, no other action in the device is required, only further settings of initialisation parameters can be made. The counter example of the model checker reports a busy alternation of some automata which are not required during this section of bus-synchronisation. This includes command automata which leads to useless bus traffic.

This phenomenon of busy automata will be elaborated on in the simple example of two state based automata depicted in figure 6.19. The task is to bring the right automaton from state $W$ into the state $Z$. Only the transition from $Y \rightarrow Z$ has the constraint that the other automaton is in the state $A$. The left automaton starts in $A$ and does not need to alter its state while the other take the transitions towards $Z$: (W,A),(X,A),(Y,A),(Z,A). But the model checker reports a sequence where the left automaton does a busy alternation of its state: (W,A),(X,B),(Y,A),(Z,B). The length of the sequences is in both cases the same. So from the point of view of the model checker, this result is as good as the former one.

The unrelated automata are not easy to identify, and require a cross check by hand with the device specification whether a relation exists or not. An analysis only on the reported sequence can not be performed because there is no hint whether a transition is required or not. Sometimes in the sequence a period of alternation can be observed, which gives hints on the transitions which have to be analysed for useless alternations.

Maybe the model checker internal algorithm can be modified in such a way that as few as

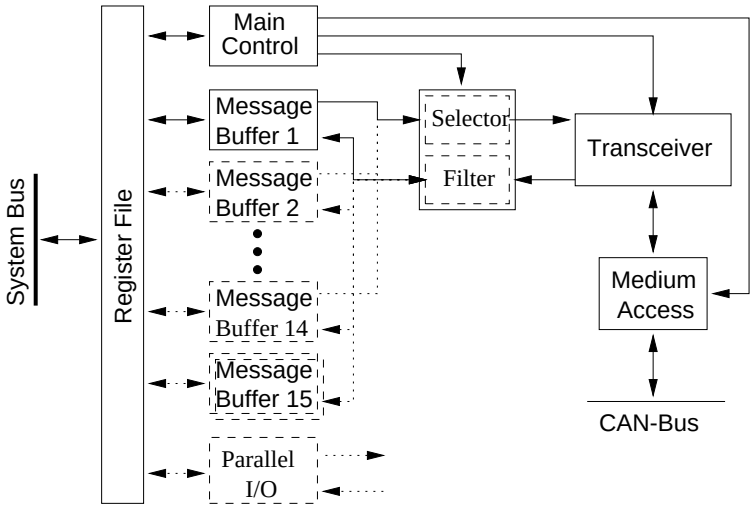



Figure 6.18: Blocks (solid) which have been specified in RAVEN. I/O parts are independent, and for the message buffer behaviour a single instance is sufficient, thus the others have not been specified.

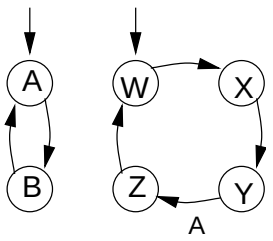



Figure 6.19: Required sequence for $W \rightarrow Z$ is $(W, A), (X, A), (Y, A), (Z, A)$. The model checker produces a busy version of $(W, A), (X, B), (Y, A), (Z, B)$ with the same length.

possible transitions are taken. The search of a path in the graph of the cross product automaton inside the model checker, has to be modified so that not only a minimal path in the sense of minimal length is found. Each edge has to be labelled with the number of automata which change state to get to the next cross product state. Now the path in the cross product automaton has to hold the constraint on the path length and a minimum sum of weight. The modification of the internal model checker algorithms is beyond the scope of this thesis, because the internally used algorithms are much more complex than the suggested path search approach.

This case study shows further that the analysis works on *synchronous* systems, even if the driver can issue commands non-deterministically, and therefore asynchronously. This leads to a problem which has been briefly mentioned before, that the *control* signals protected by a *trigger* have to be exchanged *in time*. In the previous example (figure 6.19) the automata have to meet again in the cross product state $(Y, A)$, to immediately take the transition from $Y$ to $W$. In a further example shown in figure 6.20, the sequence to get form $W \rightarrow U$ is $(W, A), (X, B)$. The automaton must synchronously step forward. If the transition $A \rightarrow B$ comes too late, the sequence becomes $(W, A), (X, A), (Y, B)$ and $U$ is missed.

A synchronous processing is possible for automata inside the device, but the exchange of signals with the system bus is asynchronous, and can not provide the signals to the exact point in time. The model of the model checker bases on synchronous systems. *Control* signals can not be identified

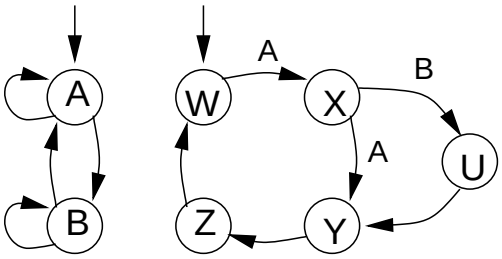



Figure 6.20: Required sequence for $W \rightarrow U$ is $(W, A), (X, B), (U, -)$. Hence, the left automaton must switch from $A$ to $B$ within one cycle.

by the model checker, so the existence of a *trigger-control* relation has to be identified by a work-around. The automaton which controls the register file access is modified in such a way, that the $idle$ state takes different fixed durations. In the first analysis step, the specification works with no extra delay in the $idle$ state. Now synchronous *trigger-control* relations can appear. In a second analysis the $idle$ delay is extended, so that the *control* signal can not be given in time. The reported sequence is now modified in such a way that the trigger and control change their order.

The case study has shown that it is possible to determine the required sequence to get a device into a dedicated state. But the method is not always applicable because of the alternations in the result sequences and the transitions in unrelated automata. The alternations can be identified by an expert because of a typical period in the sequence.

On the other hand, this method can be used for first evaluation of devices. Still the result contains the minimum set of transitions, which are required to bring the device into the dedicated state. Thus for first exploration of the system, the model checker can be used as an expert system to request sequences for getting the device into a dedicated state, in respect to the massive parallel behaviour.

## 6.5 Summary

In this chapter, the method on structural reflection of the hardware structure has been refined to derive the middle grained structure of the driver kernel. From the structure analysis, the structure of the driver kernel and fragments of the control-flow inside the driver have been constructed. Therefore, the simple semantics of component interrelationship, and the number of components have been used. Analogous to the coarse grained structuring, the communication route leads to the interaction relationship of the middle grain driver components. The pattern of multiple instances of the same behaviour, leads to structures in the control-flow level. With a more specific semantic of each component, this approach can be enhanced to build synthesis tools for the generation of the device driver counterparts. This has been evaluated in a case study for the synthesis of IRQ dispatcher software.

The second part of the chapter has focused on methods for the exploration of the device behaviour. The first method simply classifies the characteristics of the exchanged information. The classification leads patterns for the order of appearance of signal exchange in the driver control-flow. The second method analysises the parallel characteristic of the device behaviour. The aim is to determine code sequences to bring the device into a dedicated state. The method does not provide directly usable code sequences, but enhances the learning curve in the understanding of the device.

The following chapter presents a proposal for a tool chain to integrate the presented methods

into a device driver design-flow. Advanced topics in the field of device driver designs are discussed afterwards. They can be used as starting points for future research.

# Chapter 7

# Design Flow and Tool Integration

In this chapter, a concept for the integration of the different approaches is proposed. It discusses the inter-operation of the methods presented so far and the resulting design flow. Aspects of specification languages or graphical interfaces for effective use by a designer are out of scope here. The focus is on the approach integration for device driver design. Later a brief concept for integration into a software development environment is presented. Parts of this concept have been validated in the case-study presented in chapter 5.5.

## 7.1 Design Flow for Driver Design

In this section the three suggested stages of analysis, design, and code generation in the design flow are discussed.

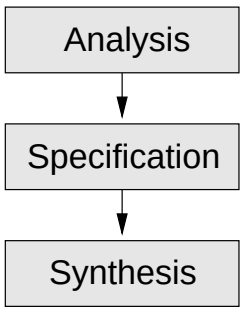



Figure 7.1:
Three stages of the design flow: system analysis, driver specification, and code synthesis.

### Device Behaviour Analysis

The analysis of the device behaviour, the classification of the device signals, and the description of the register file properties, is performed before the driver design stage (see figure 7.2). This step can be done independently of the design phase. The device vendor can do the analysis and the results can be seen as an advanced documentation, and can be shipped together with the device. There is no risk of giving away internals about the implementation details of the device, because the analysis provides only reduced information from the perspective of the register file. The analysis result is only a more formal and more specific version of the device documentation.

The analysis result is the base of a knowledge database of the device. It includes a definition of the register file with the register set, analogous to the presented properties in the Devil approach [90]. This definition of the register file structure is annotated with the analysis results presented in chapter 6.3, and the definition of methods for signal manipulation. For advanced tool support,

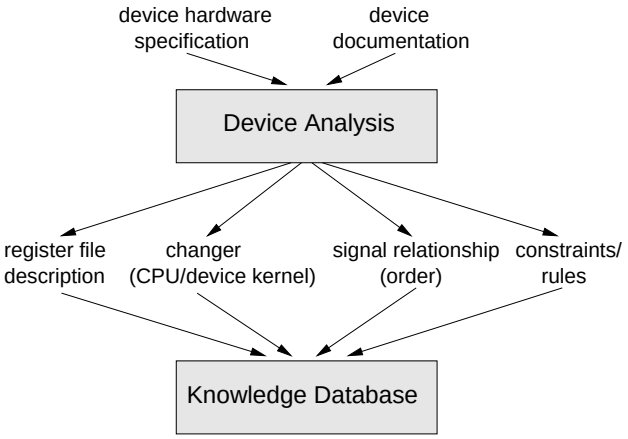



Figure 7.2: Analysis stage to build the entries for the knowledge database.

ranges for signals, rules on register content, and constraints in logical or temporal logical form can be stored in this knowledge-base. The constraint set can be used during development by the programmer, or for automatically static analysis before compilation time.

The knowledge-base is used in the next design stage. It is extended by a second knowledge-base on component communication behaviour for the system architecture specification. For each component, this database holds the compatible protocols and the translations between the ports, defined by an attributed grammar analogous to the method presented in chapter 5.

## Device Driver Design Phase

The driver design phase starts with the selection of the device out of the knowledge-base (see figure 7.3). With the device selection a proxy class or a list of methods for the device is selected. In the case of a proxy class, the object has to be instantiated, and the design phase can use this object for the communication of the driver with the device.

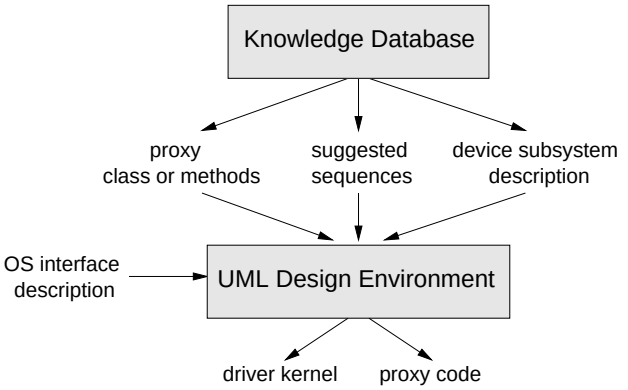



Figure 7.3: Specification stage to design the driver.

The operating system surrounding the driver or the next driver layer, defines the interfaces and the provided services. Together with the device communication proxy, the three sides of the driver kernel (kernel interface, kernel services, and HAL interface) are now defined.

The interface definitions are now embedded into a UML based software development environment. Then the internal driver structure is established by reflection of the device architecture as

presented in chapter 6. From the knowledge-base the structure of the device can be requested. The driver kernel behaviour can be specified with the methods provided in UML. Here sequence charts and state charts can be used. Depending on the signal classification and the defined relationships between the signals, the temporal order of information exchange can be provided by the knowledge-base.

Before code generation, the system architecture which connects the CPU, where the driver is located, and the device has to be specified. Out of the knowledge-base the communication components are selected and combined to form the complete grammar which describes the translator chain. Together with the abstract specification of the driver-device interaction, the code for the driver in the target language can be generated.

Alternatively to the use of the proxy object, the knowledge-base can only provide abstract instructions as an interface for the device signal manipulation. The only difference is that the access methods are explicitly inlined into the driver kernel, instead of a method call in an aggregated proxy object. In the following code generation phase a dedicated proxy object is assumed.

## Code Generation

From the device specification in UML the driver-kernel code is generated (see figure 7.4). For this part, the standard code generation out of UML specifications can be used. Beforehand, the specified access sequences can be checked with the specified access order in the device knowledge-base. On a mismatch, a warning for the designer can be generated. If possible the specified constraints on the register range and the design rules can be checked at this state.

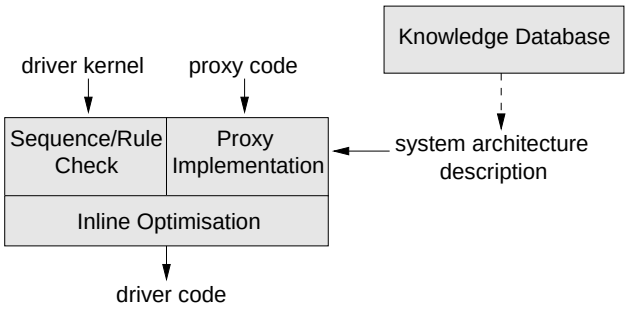



Figure 7.4: Synthesis stage to generate the communication code and integrate it into the driver code.

The proxy is exported to the tool which generates the proxy implementation according to the specified communication architecture between CPU and device. This tool can be an external tool which reads-in the proxy specification with special instructions for the register access. These statements are translated to the access code under consideration of the signal behaviour defined in the knowledge-base, and the specified system architecture with the methods described in chapter 5. The most appropriate solution is selected from the alternatives of the access implementations. The estimated access execution time can be annotated in the produced source code as a special comment.

Both code parts, the communication proxy and the driver kernel, can now be analysed in combination. The WCET can be calculated by taking into consideration the execution time of the device accesses. With methods of aspect oriented programming, the proxy code may be partially inlined, because at this stage semantical information can still be available if they are preserved by annotations in the generated source code.

The resulting source code is compiled with the standard compiler for the target system into binary code. A standard compiler can be used, because the final stage of the back-end code generators, produce source code in the language of the compiler with no extra statements. Afterwards the driver has to be tested in the same way as hand generated code. This testing task is a problem of its own, and is out of scope of this discussion.

## 7.2 Tool Integration

The previously discussed design flow, which uses the presented method, has to be integrated into a tool chain. The knowledge-base is simply a database which holds the properties of the device, and thus can be seen as a tool readable documentation of the device. The exchange format can be XML for example, whereas the tags encapsulate a database entry.

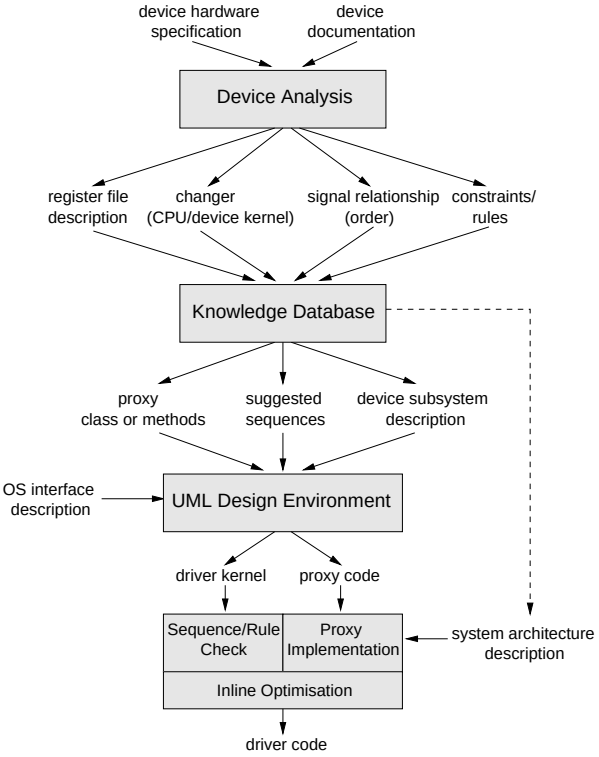



Figure 7.5:
Three stages of the design flow in the complete view.

The analysis of the device specification in order to extract the device properties, must be currently done by hand, as the meaning of some signals has to be re-constructed and described within the model of the knowledge-base.

The front-end of the knowledge-bases, the architecture specification, and the access to the code generators, can be integrated as a module into a UML design environment, which has been provided by different companies for (application) software development. The knowledge-base with the device properties can be accessed by an integrated browser, which works similar to the integrated processor documentation like in the Metrowerks-development suite [63]. From the knowledge-base the access instructions, respectively, the method collections in the form of generated proxy objects, are selected and integrated into the software development project. The UML development environment realises that these objects have to be handled by special code generators, because they have to be passed to the back-end-tool for software synthesis.

The specification of the communication architecture can be integrated into the development environment, or be specified in an external tool. The use of an external tool has the advantage that it can store different architectures, and the different target dependent implementations of the

driver can be generated by batch processing. The communication architecture is a selection of components from a graph, which stores the relationships between the components in the sense of which components can work with each other, and which protocols can be used for exchanging the data.

The back-end of the UML design environment must work in different ways. The proxies are exported in a special form, so that the hardware abstraction layer generator can handle the proxy code. After separation of the parts, the different back-end tools which postprocess the source code can be started. Thus, the stage of code generation for the proxy, the optimisation, and the integration into the driver behaviour code, is an additional sequence in the script for the back-end processing by the development environment.

Afterwards the project is ready to be compiled with the standard target compiler, and the complete system can be tested and debugged.

This tool-chain is a proposal for two reasons: firstly, a full implementation requires a large amount of effort, and is out of scope in this thesis which focuses on the methodology. Secondly, at the moment the automation of some parts of the chain is not complete. Some are still performed by hand. Further investigations in future have to solve this problem, and have to be integrated into this proposed tool-chain.

# Chapter 8

# Advanced Topics

In this chapter, the topics of application of compiler techniques, and some issues on security in the context of driver and devices are discussed. The topics are not in the major scope of this thesis, but are important enough to be mentioned and maybe a starting point for future research.

## 8.1 Applying Compiler Techniques

Nowadays compilers are not concerned with the existence of devices with special types of interfacing memory locations. Still the compiler perspective is a unique memory map. The only concession is made by marking areas as non cache-able and volatile, taking them out of the scope of optimisation. High level programming languages have few statements for I/O interaction. Most the time an architecture-specific in-line assembler is used, which by-passes the compiler and is directly integrated in the target code (see low level access functions under Linux [95]). In this chapter, the application of known compiler techniques in the special field of device driver programming is briefly discussed. As reference for compiler techniques, mainly [66] and additionally [54, 68, 81, 120, 32] have been used.

### 8.1.1 Techniques and Pre-conditions

State of the art compiler techniques are code hoisting, constant folding, control-flow analysis, dead code elimination, and code re-ordering, just to name a few. The main question is, which optimisations can safely be done in combination with devices, and which technique must be applied? The aim of the optimisation is high speed in total execution time of the driver functions.

A comparison of the different techniques results in the fundamental constraint, that the compiler assumes an exclusive and correct access to the memory. Correct means that a value is transfered from the CPU register to the main memory, and is read back without changing its value. The main memory acts as a background storage for the CPU registers. The different storage locations, identified by memory addresses, must not interfere. The real physical organisation is not important, only for explicit type casting like in `C/C++`, an order in the main memory is assumed.

Devices have a special physical organisation of the visible parts in the main memory, defined by the register file structure, the bus-interface, and the embedding in the memory map of the system. The device can alter the content in the register file. Hence, the perspective of the device for the compiler is shared memory with non-exclusive access.

One impact on the execution speed is the hierarchy of the bus systems in contemporary computer systems. Write operations can be buffered by the bus bridges. Read operations lead to a stall until the data is received. The CPU processing speed is much higher than the bus speed, so a stall

on the bus leads to idle operations of the CPU. The situation can be improved by pre-fetching like the bus-interface of the PowerPC architecture [101]. But still the order of access at the device must be preserved, because not only the exchanged values, even the event of exchange can lead to an action in the device. The objective of optimisation is to reduce the bus-traffic while keeping the correct access order and not suppressing events by access.

### 8.1.2 Traffic Optimisation

The applicability of the different compiler techniques depends on the characteristic of the device registers. If they behave exactly as the normal memory locations the known optimisations can be used. Thus reduction of traffic is based on an analysis of the register file characteristics.

In rare cases a register is marked as constant. Then, constant folding can be used to incorporate this value into the driver code. In some cases, registers are marked in the documentation as to be altered only by the CPU. Here explicit caching can be used, as discussed in section 5.6.2. If the value is written to the device, a copy can be held in a CPU register or in the main memory. It is assumed that main memory access is faster than device access. Therefore, the caching location must be a non-swappable memory location. Now the register has become a normal memory location (the caching location), and all optimisation techniques can be applied. With further analysis of the driver code it can be determined if all write accesses have the same constant content. Then a read to this location can be replaced by this constant, and folded into the driver code.

If the register location is marked as alterable by the CPU and by the device, more detailed analysis of the device behaviour is required. In this case, the register behaves like shared memory, and the CPU and the device are autonomous processes. An analysis for the complete driver and device is rather difficult, as not a single control-flow or its impact on the variables content, have to be analysed (for methods see [68]). Lines on both sides, and due to the massive parallelism of hardware, maybe multiple paths in the control flow have to be followed. This must be performed by hand, or with the use of a model checker, which creates the lifeline of an information entity.

The problem can be simplified if the analysis can be restricted to smaller sections of the system. Such sections are, for instance loop-bodies with consecutive access to the device. During execution of the loop, registers stay constant because a change of content is protected by a trigger given after the loop. Thus, settings or readouts in the loop body are quasi-constant. In this case, the technique of code hoisting can be used to reorder the code and reduce the traffic. Examples are the set up of base addresses of PCI-devices which are stored in the device registers. They have to be read out only once and the access in the loop body uses the quasi-constant base address. The code hoisting method moves the base address setting out of the loop.

This example shows the difficulty in device driver programming, as assumptions on the total behaviour must always be made to get an effective code. The driver documentation must include the assumptions which are made during programming, because all modifications must respect these assumptions.

The code hoisting approach leads to a design rule for explicitly hoisting in if-then-else structures: Check settings as early as possible in the lifeline of the driver. Thus repeated checks deeper in the control-flow structure are avoided. This leads to contracts for the use of methods or a level of trust. An isolated method must check the parameter set at each iteration. With a contract that the calling instance guarantees the range of the parameters, the check can be erased. This is more or less code hoisting beyond the boundaries of method calls.

The method of code hoisting can be enhanced to dynamic code modification during runtime. If a value is quasi-constant and determined at runtime, all equations which use this parameter can be constant-folded. On value change, an observer is immediately informed and re-modifies the

affected code sequences.

If a register produces events on accesses, the intermixed control-flow on both sides have to be analysed to identify whether the access can be avoided or not. As discussed in section 4.1 the driver and the device now behave as fully parallel objects, with intercommunication by method calls. Thus analysis techniques from the design of parallel systems have to be applied, which are beyond the scope here.

### 8.1.3 Dead Code Elimination in Hardware

Constant values allow constant folding of expressions in the source code and dead code elimination [66, 23] of the dead branches. In [56], the dead code elimination in configurable hardware devices is described. Unused elements and states of automata are erased from the device specification. The aim of this work is to create minimal configurations for reconfigurable devices, which still fulfill the desired behaviour. The specification of which parts are not used is made by hand.

If all accesses to devices are known by an analysis of the proxy use by the sub-object (lifeline analysis of the device registers), constant register settings and unused registers can be identified. Perhaps the control-flow analysis of the use of the proxy should be extended up to the use of the driver by an application.

Constant settings can be incorporated into the component hardware specification. Unused registers are erased from the specification. The reduced input logic leads to a further automatic optimisation in the hardware design by the synthesis tool. If automata in the device are affected, the optimisations can again have impacts on the driver. If flags are constant, read operations can be replaced by constants, and dead code in the driver can be erased. The depth of this fix-point iteration is a topic for future investigation. During adaption, it has to be ensured that the criteria for device type identification are not erased.

The result is a highly adapted driver to a highly (to the driver) adapted device. This method of *recursive dead element optimisation* can be used in self optimising systems for optimisation after behaviour adaption.

This reduction principle can potentially be applied at an earlier stage of the development process of the system. The system specification is analysed and unused parts can be eliminated. This leads to shorter analysis times of the synthesis tools, because the exploration space is reduced.

### 8.1.4 Summary

The main pre-condition for optimisations by a compiler is the quasi-constant of register content. The analysis problem is analogous to an analysis in parallel running tasks on an SMP (Symmetric Multi-Processor) machine. This model does not include the impacts on the communication layer. If the actions of the channel are atomic and side-effect free, the transaction is transparent and does not have an impact on the optimisations. An analysis of parallel asynchronous control flow is beyond the scope of this thesis, and an interesting field for future work. The optimisations discussed here can be used as rules of thumb for the device driver programmer who has an idea of the control-flow.

## 8.2 System Security

In the literature on fundamentals of operating systems [19, 106, 84, 64, 102, 70], security of the system is discussed from the perspective of user access, and the intrusion over networks. Secu-

rity issues in combination with device drivers are not discussed. The driver is assumed to be a trustworthy module of the operating system.

A driver itself is a risk factor for the system. Under Linux, a driver is a kernel module with full access to all kernel services. Hence, a driver can manipulate the kernel integrity. Furthermore, the driver does not provide properties of a device use to the kernel. The driver can access all devices in the system and can manipulate them. This is a problem of the monolithic architecture of the Linux operating system. Each part of the kernel has the same rights and works as system root with the highest privileges.

A protection approach can be to use a microkernel architecture which allows only limited access of the driver to memory locations. This protects the kernel itself against memory attacks, because the driver is handled as a user process with memory protection in monolithic kernels. The memory management unit is programmed by the kernel, so that the driver can only access its own memory and the peripheral bus. An attack of foreign devices is still possible, because the kernel is not able to decide for which device the driver is designed for.

Instead of attacking foreign devices, the attack against the system can be done by faking a device by the corrupted device driver. The driver can, for instance, pretend to be a secure socket connection and can attack the information send through this socket.

A security hurdle is to foist a system off from a corrupted driver. In most operating system, one needs root privilege to install a device driver. But getting this root privileges is the aim of the intruder. The intruder will not spend the effort to get the root rights by a corrupted driver, if the root privileges are required for the attack. So an intrusion of a system can be to tempt the system administrator to install the corrupted driver. For example a new driver release is faked and delivered by driver CDs of computer magazines. In the field of open source software, the users and system administrators are used to download new operating system releases from the Internet. A corrupted driver can be pushed underneath the system by faking a new driver release in the Internet.

Linux is an operating system which is delivered as source code, so a corrupt driver can be identified easily by checking the source code. This is easier than monitoring the driver behaviour in the running system. So the system is protected by peer review of the driver source, and the announcement in the net. With binary delivery of the drivers a reengineering of the code has to be done. In both cases, the analysis of driver behaviour and verification is a very difficult task, because the device behaviour must be taken into account as part of the abstract device. For example, the real-time version of Linux RTAI [94, 21] is known to have a high performance in the sense of hard real-time constraints, and is also known to be very stable. Tests have shown that the real-time behaviour can be totally ”crashed” by a harmless looking hardware device, the graphic card. This specific graphic system on the AGP bus uses the main memory as a texture buffer. During an access by a DMA-burst to its texture memory the real-time task on CPU is not able to access the peripheral devices, and thus the real-time constraints could not be met. Thus the real-time patch of the kernel is not able to prevent the system from this situation.

The complexity of the devices lead to a special form of system attacks, even if the driver is encapsulated to its memory and its peripheral device. Here DMA transfers are the main attacking mechanisms for intrusion by devices. Devices with their own DMA controller (for example the PCI bridges by PLX Technology [86, 87]) can be used to read out the kernel memory, and transfer it to locations where it can be analysed by the driver or a user application. The Memory Management Unit (MMU) controls only the access from the CPU to the memory, but not the access from a peripheral bus. This totally undermines memory protection. Such a use of a DMA-controller can be scrambled into the driver source code. The only solution of this problem is a further memory management unit in the memory bridge to also protect kernel memory sections from the peripheral

bus side.

An attack similar to DMA-transfers is the use of the multi-master capability of peripheral busses. Accesses to devices are neither authorised on the bus nor encoded. Hence, a device with master capability can access other devices directly, and pretend to be the driver running on the CPU. This feature has been used in [53], to transfer results of a calculation in an FPGA on a Raptor I-board, directly by DMA-burst to the SCI (Scalable Coherent Interface)-network card. The SCI-network card can not distinguish between an access from the CPU or from the FPGA-board. An intrusion of a device can be done by an indirect access to devices via a repeater. With this mechanism the previously discussed access control of a driver by the MMU in the CPU can be undermined.

The security and verification problem gets even worse in the presence of reconfigurable peripheral devices. Here the behaviour of the device depends on the implemented and downloaded hardware design. This implementation can carry out repeater functions, or can analyse the gathered data itself and send back identified back doors. With DMA access to network-cards it can send the information directly to the network-card of the LAN connection. The major problem here is, that the driver can be trustworthy but the bitstream for the reconfigurable device may be corrupted. The re-engineering of a bitstream to reconstruct the algorithm is nearly impossible. The FPGA vendor Xilinx offers an encryption scheme for the bitstreams analogous to PGP (Pretty Good Privacy). A system protection scheme can be to manipulate the bitstream on downloading to the FPGA, by insertion of a wrapper which prevents access to protected locations (isolation). The encapsulation is eased if the reconfigurable device consists only of the device kernel, and the register file and bus protocol acceptor is inserted by the correctly working downloader (see section 4.2.2).

Furthermore, the wrapper can reduce the risk of hardware sniffers. If reconfigurable devices are directly attached to a system bus, they can sniff the bypassed information by a modified bus-interface. Here the FPGA does not require an active access to the system bus, it can simply work as a sniffer on the bus. The attacker's main problem is to receive the gathered information back.

The complete topic of security of a computer system must be seen under the perspective of a distributed system. The same approaches of security in networks must be scaled down to the network inside a single computer. This can, or must lead, to a complete redesign of modern computer architectures. An approach can be the serial interconnection approaches like InfiniBand(R) [105, 34], or HyperTransport(R) [17] for node internal communication. Broadcasts must be explicitly implemented in the bridges. This is a starting point for security to deny access on routing level to the internal components in the host adaptor.

The points discussed here should not only be seen under the perspective of system security. Many of the points discussed under the aspect of security also enhance the robustness and the reliability of the system. The system is not only protected against corrupted components, it is even protected against malfunctioning components.

# Chapter 9

# Conclusion

In this thesis, different aspects of device driver design have been discussed. The emphasis has been placed on the synthesis of the communication channel between the driver software and its opposite hardware part, and on the systematic analysis of the device hardware. From the results, programming guidelines for the structure and templates for the order of accesses to the device have been derived.

**Summary** In the first chapter the field of device drivers has been described. The driver is a component in an operating system which interacts with hardware and other software components. They define the environment the driver is settled in. Depending on the chosen perspective, the driver acts as a mediator between these components and routes information to the dedicated opposite. From the perspective of the user application, the driver encapsulates the device with a software wrapper, and acts as an abstract device which fulfills the requirements of a device class. In combination with the concerns of the driver surroundings, aspects of the design and development process have been discussed.

The chapter "state of the art" has given a brief overview on the activities in the field of hardware-near programming and device driver design. The currently available books mostly deal with the driver architectures of the different operating systems, and how to integrate an own driver into the system architecture. Other books focus on the programming of special hardware, or on the hardware-near aspects of programming languages. The tool sets and software development kits of the chip vendors are self-contained and do not provide interfaces for the integration of own system extensions.

At first glance, high-level software development techniques seem not to be applicable in the low-level field of hardware-near programming. This is true only if the high-level methodology is intermixed with the inappropriate translations to the low-level constructs by the different high-level tools. The design methods like Design Patterns and the specification methods of UML can be applied if they are wisely translated to the target programming language. They should be applied to benefit from the advantages they have already shown in application design.

Up till now, there are only two approaches in the field of driver synthesis: the adaptor synthesis by the Royal Institute of Technology, ESDLab, Sweden, based on the specification language ProGram, and the Devil approach by Institut National de Recherche en Informatique et en Automatique. The approaches of the hardware/software co-design cannot be applied, as they still have problems with the mapping to the communication structure of the target system. The Devil approach separates the driver into the concerns of the behaviour, which can be described by a domain specific language, and the communication part to the device registers. The latter uses a communication channel which covers the complete channel from the fixed entry point in software up to the device register file. The adaptor synthesis with the ProGram approach needs a specifica-

tion of the required access sequences on the kernel interface side, and on the device interface side of the driver. The approach focuses on the synthesis of a driver in the sense of an adaptor. To my knowledge, up till now no analysis method for the device behaviour and the resulting impacts on software have been published.

This has lead to an approach presented in this thesis for the derivation of the coarse grained driver structure from the communication interrelationship of the hardware components. The communication graph has been reflected to create the relationship of the driver components. Here the direct communication to the hardware is provided by proxies, which directly access the hardware. Or they use services of other driver components, because the corresponding hardware components are on the route to their own counterpart. This leads to a requirement for the driver component to provide a communication interface to hardware components beyond their own counterpart. The coarse grained structure has to be refined in two aspects: the synthesis of the communication proxies, and the design of the driver behaviour.

The communication proxies for the driver components can be synthesised with the Devil approach. The approach uses transparent communication channels to access the registers by read and write operations. The channel is monolithic and has to be adapted to the target hardware architecture. This approach has been refined to an approach which describes the heterogeneous communication architecture in a computer system by a network of routing nodes. The behaviour of the network is described with attributed grammars. Thus, in combination with a given route, the impacts of the channel sections can be combined to a total behaviour. This description is analysed to determine the entry point of the channel in software, and the required parameters by use of a code generator from compiler construction. Furthermore, side-effects can be detected and compensated by the synthesis of a channel adaptor at the channel entry point. The side-effect analysis can be used to do optimisations by merging multiple accesses to one single transaction on the peripheral bus. The aim of the optimisation is the total execution time of the accesses to the device.

The proxy provides transparent communication up to the device kernel, which determines the device behaviour. The driver is the adaptor of the device hardware to the desired total behaviour of an abstract device. This requires an analysis of the device behaviour. The internal fine-grained structure of the driver component can again be derived from the internal organisation of the device hardware. The structure leads furthermore to some fragments for the driver control-flow, due to the number of instances of a device part. In the special case of the interrupt system of a computer architecture, the software for identifying the origin of an interrupt request can be synthesised. The synthesis can be done due to the simple interfaces and the quasi-static (atomic) temporal behaviour of this system.

The analysis of the temporal behaviour has been the emphasis of two further investigations. Firstly, the combination of the driver and the design has been seen as the separation of a total behaviour. The asynchronous communication channel between the components leads to a fully interlocked communication. Together with a classification of the exchanged signals semantic, the order of appearance in the driver code can be determined. Secondly, the hardware is massively parallel. This results in internal interaction which is difficult to overlook. The aim of a device driver designer is to bring a device into a dedicated state by a sequence of accesses to the device. For the generation of this sequence a model checker has been used. The aim of reaching a state has been reversed to the analysis aim of never reaching this state, and the model checker reports a sequence to get into the state as in the counter example of the failed verification. A case study has shown that the results are encouraging, but the algorithms of the model checker have to be modified to get rid of busy automata. Both approaches can be used for exploration of the device behaviour, and for steepening the learning curve of the device driver designer.

**Conclusion** The work presented in this thesis has shown that device drivers are partially synthesiseable. The communication part of the register access can be fully automatically generated out of an architecture description. The driver behaviour cannot be synthesised, due to the lack of a specification language for the desired total behaviour, and the missing synthesis algorithm. Approaches like the domain specific language in the Devil approach are heading towards a language for driver design.

The given hardware can be systematically analysed and the driver structure can be constructed by reflection of the hardware structure. A classification of the exchanged information leads to a position of appearance in the driver code. With the use of a model checker, access sequences for bringing a device into a dedicated state can be generated. These are only fragments for the driver design and have to be brought together by an experienced driver designer. However, they enhance the understanding of hardware and provide a structured analysis scheme for the given hardware.

Implicitly, the thesis has shown that high-level software design concepts like Design Pattern can be used to describe the driver architecture and parts of the behaviour in the design phase. This has been done by separation of the modelling concept from the implementation on the target system. Due to the different granularity, the model has to be left at some point on the way towards the implementation. But due to their abstract nature, they help to order and structure the problem of device driver design.

**Outlook/Future** One aim for future investigations can be a definition of a language for the description of a computer architecture in the concern of the communication interrelationship. Together with an extended domain specific language for driver specification, the main target of driver synthesis comes closer. These approaches may be combined with the approaches of aspect oriented programming, and the construction from component libraries in the field of componentware.

It has been shown that with the integration of hardware access statements into the programming language in combination with an architecture description, compilers are able to optimise these code sections. At the moment, they are beyond the scope of compiler construction.

The approaches of generic communication proxies can be used in the field of hardware/software co-design to solve the problem of interface synthesis. Still the partitioning can only sense the requirement of an interface, as a mapping to hardware is missing, due to the lack of an analysable description of the target system architecture. Furthermore, the approach of dead element optimisation by recursive elimination of software parts and hardware parts, can be an approach for self-optimising systems.

New hardware architectures for inter-computer communication like InfiniBand and HyperTransport, can ease the problem of proxy synthesis due to the IP based communication scheme. This can ease the synthesis methods described here.

As briefly discussed, system security is a major topic of the future and can not stop at the driver level. Different scenarios of using driver and devices for a system attack have been described. At the moment no real attack by using a driver is reported, but with the rigorous encapsulation of the systems in the future, these kinds of attacks may come into scope. The closing of these back doors is a topic of future investigations.

# Bibliography


[1] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *TOPLAS*, 11(4):491–516, 1989.

[2] Don Anderson. *FireWire System Architecture*. Mindshare Inc., Addison-Wesley, 2nd edition, 1999.

[3] C. Andre, M-A. Peraldi-Frati, and J-P. Rigault. Scenario and property checking of real-time systems using a synchronous approach. 2001. Spezifikation von Zeit in sequentiellen Ablaeufen durch Zeitangaben in Sequenc Charts.

[4] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Spezification*. Addison-Wesley, Massachusetts, 1999.

[5] Herbert Bader and Walter Huber. *Jini*. Addison-Wesley, 2000. check291001.

[6] Art Baker and Jerry Lozano. *The Windows 2000 Device Driver Book*. Microsoft Technologies Series, 2000.

[7] Michael Barr. *Programming Embedded Systems in C and C++*. O'Reilly, 1999.

[8] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner. *Linux-Kernel-Programmierung*. Addison-Wesley, 4th edition, 1997.

[9] BeOS. Internet; http://www.be.com/.

[10] Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. On the Development of Object-Oriented Operating Systems for Deeply Embedded Systems - The PURE Project. In *ECOOP Workshops*, pages 27–31, 1999.

[11] Carsten Böke. Software Synthesis of Real-Time Communication System Code for Distributed Embedded Applications. In *Proc. of the 6th Annual Australasian IFIP Conf. on Parallel and Real-Time Systems (PART'99)*, Melbourne, Australia, Dez. 1999.

[12] Karl W. Bonfig. *Feldbus-Systeme*. expert-Verlag, 1995.

[13] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley, 9th edition, 1998.

[14] John Forrest Brown. *Embedded Systems Programming in C and Assembly*. Van Nostrand Reinhold, New York, 1997.

[15] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 3rd edition, 2001.

[16] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, Mass. USA, 1999.

[17] HyperTransport Technology Consortium. HyperTransport I/O Link Specification, Revision 1.03. Technical report, 2001.

[18] Jianwen Zhu Daniel D. Gajski, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specificatin Language and Methodology*. Kluwer Academic Publisher, Boston, Dordrecht, London, 2000.

[19] Harvey M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[20] Digital DNA. http://www.digitaldna.com/.

[21] Dipartimento di Ingegneria Aerospaziale - Politecnico di Milano (DIAPM). *RTAI Programming Guide*, September 2000.

[22] Carsten Ditze. A Step towards Operating System Synthesis. In *Proc. of the 5th Annual Australasian Conf. on Parallel and Real-Time Systems (PART)*. IFIP, IEEE, 1998.

[23] Carsten Ditze. *Towards Operating System Synthesis*. PhD thesis, University of Paderborn, 2000.

[24] John Dunlop, Josef Fuchs, and Steve Mihalik. *MPC555 Interrupts*. Motorola INC., 0 edition, July 2001.

[25] M. Eisenring and J. Teich. Domain-Specific Interface Generation From Dataflow Specifications. In *Proc. of Codes/CASHE'98, the 6th Int. Workshop on Hardware/Software Codesign*, pages 43–47, Seattle, Washington (USA), March 1998.

[26] M. Eisenring and J. Teich. Interfacing Hardware and Software. In *Proc. of FPL'98, the Conf. on Field-Programmable Logic and Applications*, pages 520–524, Tallin, Estonia, September 1998. Springer Lecture Notes in Computer Science.

[27] Eli. Internet; http://www.upb.de/project-hp/eli.html.

[28] Edwin Erpenbach. *Compilation, Worst-Case Execution Times and Schedulability Analysis of Statemate Models*. PhD thesis, University Paderborn, 2000. check291001.

[29] IEEE 1394 for Linux. Internet.

[30] Martin Fowler and Kendall Scott. *UML Distilled*. Addison Wessley, 1997.

[31] Martin Fowler and Kendall Scott. *UML Konzentriert*. Addison-Wesle-Longman, 1998.

[32] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler : Design and Implementation*. 1995.

[33] Fujaba. Internet;
http://www.upb.de/fachbereich/AG/schaefer/ag_dt/PG/Fujaba/.

[34] William T. Futral. *InfiniBand Architecture Development and Deployment*. Intel, 2001.

[35] Shaul Gal-Oz and Avi Cohen. The Hazards of Device Driver Programming. *Embedded System Programming*, pages 34–46, May 1997.

[36] Erich Gamma. *Entwurfsmuster*. Addison-Wesley, 1996.

[37] Mahadevan Ganapathi and Charles N. Fischer. Affix grammar driven code generation. *TOPLAS*, 7(4):560–599, 1985.

[38] Hassan Goma. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley, 2000.

[39] Rick Grehan. Driver Assistance. *Computer Design*, 36(1):75–80, Nov. 1997.

[40] Frank Griffel. *Componentware*. dpunkt-Verlag, Heidelberg, 1998.

[41] Uwe Honekamp. *IPANEMA - verteilte Echtzeit-Informationsverarbeitung in mechatronischen Systemen*. VDI-Verl., 1998.

[42] John E. Hopcroft, Rajeev Motwani, and Jeffrez D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition, 2001.

[43] IEEE Computer Society, New York. *IEEE Standard for a High Performance Serial Bus*, 1995 edition, 1995. IEEE Std 1394-1995.

[44] Intel Corporation. *82527 Serial Communications Controller Architectural Overview Automotive*, 1996.

[45] Java technology. Internet; http://www.sun.com/java/.

[46] Jini network technology. Internet; http://www.sun.com/jini/.

[47] Heiko Kalte, Mario Pormann, and Ulrich Rückert. Rapid Prototyping System für dynamisch rekonfigurierbare Hardwarestrukturen. In *AES 2000*, pages 149–157, Karlsruhe, 2000.

[48] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP '97 - Object-Oriented Programming*, page 220ff., Berlin, 1997. Springer-Verlag.

[49] Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2:127–145, 1968.

[50] Thomas Kropf. *Introduction to Formal Hardware Verification*. Springer, Berlin, 1999.

[51] Harold W. Lawson. *Parallel Processing in Industrial Real-Time Applications*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[52] Thomas Lehmann. *Device Driver for "Spyder" and "Raptor" FPGA Boards, 1.4*. Universität Paderborn, 2000.

[53] Thomas Lehmann and Andreas Schreckenberg. Case Study of Integration of Reconfigurabel Logic as a Coprocessor into a SCI-Cluster under RT-Linux. In *Field-Programmable Logic and Applications*, Belfast, Northern Irland, August 2001. Springer.

[54] Rainer Leupers. *Code Optimization Techniques for Embedded Processors*. Kluwer Academic Publishers, 2000.

[55] Andrew Lyons. Uml for real-time overview. Technical report, Rational Rose, 1998.

[56] John MacBeth and Patrick Lysaght. Dynamically Reconfigurable Cores. In *Field-Programmable Logic and Applications*, pages 462–472, Belfast, Northern Irland, August 2001. Springer.

[57] Daniel Mahrenholz. Minimal Invasive Monitoring. In *The fourth IEEE international Symposium on Object-Oriented Real-Time Distributed Computing*, pages 243–250. IEEE Computer Society, IEEE, 2001.

[58] Axel Jantsc Mattias O'Nils. Device Driver and DMA Controller Synthesis from HW/SW Communication Protocol Specifications. *Design Automation for Embedded Systems*, 2000.

[59] K. L. McMillan. *The SMV System*, Nov 2000. check291001.

[60] Fabrice Merillon, Laurent Reveillere, Charles Consel, Robin Hansen, Renaud Marlet, and Gilles Muller. Towards Verifiable Device Drivers: An Approach based on Domain-Specific Languages. Technical report, Institut National de Recherche en Informatique et en Automatique, Nov. 1999.

[61] Fabrice Merillon, Laurent Reveillere, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An IDL for Hardware Programming. In *OSDI 2000*, pages 17–30, San Diego, Okt. 2000.

[62] Hans-Peter Messmer. *PC Hardware Aufbau Funktionsweise Programmierung*. Addison-Wesley, 1997.

[63] Metrowerks C/C++ Compiler. Internet, http://www.metrowerks.com.

[64] Milan Milenkovic. *Operating Systems*. McGraw-Hill, Inc., 2. edition, 1992.

[65] Motorola INC. *MPC555/MPC556 - User's Manual*, October 2000.

[66] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1997.

[67] Thomas Neimann. Nuts to OOP! *Embedded Systems Programming*, 12(8):16–22, August 1999.

[68] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[69] Ralf Niemann and Peter Marwedel. Synthesis of Communicating Controllers for Concurrent Hardware/Software Systems. In *DATE*, 1998.

[70] Gary Nutt. *Operating Systems*. Addison Wesley Longman, Inc., 2nd edition, 2000.

[71] J. Öberg, A. Jantsch, and A. Hemani. Validation of Interface Protocols Using Grammar-based Models. In *Proc. of the IEEE International High Level Design Validation and Test Workshop (HLDVT'98)*, pages 40 – 46, La Jolla, California, 1998. IEEE.

[72] J. Öberg, A. Kumar, and A. Hemani. Grammar-based Hardware Synthesis of Data Communication Protocols. In *The 9th International Symposium on System Synthesis*, pages 14–19, La Jolla, USA, November 1996.

[73] Oliver Oberschelp. Entwurf und Implementierung eines flexiblen Codegenerators zur Prozesskopplung für verteilte Hardware-in-the-Loop Simulation. Master's thesis, Universität Paderborn.

[74] Walter Oney, Ben Ryan, Devon Musgrave, and Robert Lyon. *Programming the Microsoft Windows Driver Model*. Microsoft Press, 1999.

[75] M. O'Nils. *Specification, Syntheisis and Validation of Hardware/Software Interfaces*. PhD thesis, Royal Institute of Technology, Department of Electronics, Stockholm Sweden, 1999. TRITA-ESD-99-04, ISSN 1104-8697, ISRN KTH/ESD/AVH–99/4–SE.

[76] Mattias O'Nils and Axel Jantsch. Communication in Hardware/Software Embedded Systems - a Taxonomy and Problem Formulation. In *IEEE NORCHIP*, Tallin, Estonia, November 1997. IEEE.

[77] Mattias O'Nils and Axel Jantsch. Operating System Sensitive Device Driver Synthesis from Implementation Independent Protocol Specification. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 562–568, Munich, Germany, 1999.

[78] Mattias O'Nils and Axel Jantsch. Synthesis of DMA Controllers from Architecture Independent Descriptions of HW/SW Communication Protocols. In *Proceedings of the 12th International IEEE Conference on VLSI Design*, pages 138–145, Goa, India, Jan 1999. IEEE.

[79] Nattias O'Nils, Johnny Öberg, and Axel Jantsch. Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces. Technical report, Royal Institute of Technology, ESDLab, Stockholm, Schweden, 1998.

[80] Frank Oppenheimer, Dongming Zhang, and Wolfgang Nebel. Modelling Communication Interfaces with COMIX. In D. Craeynest and A. Strohmeier, editors, *Reliable Software Technologies - Ada-Europe 2001*, page 337 ff, Leuven, Belgium, May 14-18 2001. Springer Verlag.

[81] Thomas W. Parsons. *An Introduction to Compiler Construction*. W.H. Freeman and Company, 1997.

[82] Roberto Passerone, James Rowson, and Alberto Sangiovanni-Vincentelli. Automatic Synthesis of Interfaces between Incompatible Protocols. In *35th Design Automation Conference*, San Francisco, CA USA, 6 1998. ACM, ACM.

[83] PCI Specification 2.2. Internet; http://www.pcisig.com.

[84] James R. Pinkert and Larry L. Wear. *Operating Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[85] David Piscitello and A. Lyman Chapin, editors. *Open Systems Networking*. Addison-Wesley, Reading, Massachusetts, 1993.

[86] PLX-Technology. *PCI 9080 Data Book*, 1.05 edition, 1998.

[87] PLX-Technology. *PCI 9054 Data Book*, 2.1 edition, 2000.

[88] Ratinoal Rose. Internet; http://www.rational.com/index.jsp.

[89] Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly, Sebastopol, CA, 1999.

[90] Laurent Reveillere, Fabrice Merillon, Charles Consel, Renaud Marlet, and Gilles Muller. The Devil Language. Technical report, Institut National de Recherche en Informatique et en Automatique, Okt. 2000.

[91] Laurent Reveillere, Fabrice Merillon, Charles Consel, Renaud Marlet, and Gilles Muller. A DSL Approach to Improve Productivity and Safety in Device Drivers Development. In *Automated Software Engeneering*, pages 101–110, Grenoble, 2000.

[92] Laurent Reveillere and Gilles Muller. Improving Driver Robustness: an Evaluation of the Devil Approach. In *DSN-2001*, Göteborg, Sweden, July 2001.

[93] Stein Jørgen Ryan. The Design and Implementation of a Portable Driver for Shared Memory Cluster Adapters. Technical report, University of Oslo, Department of Informatics, 1997.

[94] RealTime Application Interface. Internet; http://www.aero.polimi.it/projects/rtai/.

[95] Alessandro Rubini. *Linux Device Drivers*. O'Reilly, Sebastopol, 1998.

[96] Jürgen Ruf and Thomas Kropf. Formale Verifikation diskreter Echtzeitsysteme. *it+ti*, 43:39–46, 2001.

[97] Dan Saks. Function Signatures and Name Mangling. *Embedded Systems Programming*, 12(8):79–81, August 1999.

[98] Serial Bus Protocol 2 (SBP-2). Technical report, T10, May 1998.

[99] Friedhelm Schmidt. *SCSI-Bus und IDE-Schnittstelle*. Addison Wesley Longman Inc., 1998.

[100] IEEE Std 1596-1992 IEEE Standard for Scalable Coherent Interface (SCI). Technical report, IEEE, 1992.

[101] Tom Shanley. *PowerPC System Architecture*. MindShare Inc., Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[102] William Stallings. *Operating Systems*. Prentice-Hall, Inc., 3. edition, 1998.

[103] William Stallings. *Computer Organizytion and Architecture: Designing for Performance*. Prentice Hall, Upper Saddle River, New Jersey, 5th edition, 2000.

[104] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley Publishing, Massachusetts, 1992.

[105] Hermann Strass. InfiniBand – Bandbreite ohne Ende. *Elektronik*, (6):56–62, 2001.

[106] Andrew S. Tannenbaum. *Moderne Betriebssysteme*. Hanser Verlag/Prentice-Hall International, 2. edition, 1995.

[107] The BeOS Development Team. *BeOS Advanced Topics-The Official Documentation for the BeOS*. O'Reilly, 1998.

[108] Jürgen Teich. *Digitale Hardware/Software-Systeme*. Springer, Heidelberg, 1997.

[109] Texas Instruments. *TSB12LV32 Data Manual*, April 2000.

[110] Scott Thibault, Renaud Marlet, and Charles Consel. A Domain-Specific Language for Video Device Drivers: from Design to Implementation. Technical report, Institut National de Recherche en Informatique et en Automatique, 1997.

[111] Togethersoft. Internet; http://www.togethersoft.com.

[112] Edward Tuggle. Writing Device Drivers. *Embedded Systems Programming*, pages 42–65, January 1993.

[113] Uniform Driver Interface. Internet; http://www.projectudi.org/.

[114] UDI Linux Implementation. Internet; http://sourceforge.net/forum/forum.php?forum_id=103581.

[115] UDI Physical I/O Specification. Technical report, 2001.

[116] UDI Core Specification Volume I. Technical report, 2001.

[117] UDI Core Specification Volume II. Technical report, 2001.

[118] Karlheinz Weiss, Thorsten Steckstor, Carsten Ötker, Igor Katchan, Carsten Nitsch, and Joachim Philipp. *Spyder - Virtex - X2 User's Manual*, 1999.

[119] Christian L. Wies and Johannes Endres. Brandmeister Pinguin. *c't*, (3rd), 2000.

[120] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau*. Springer Verlag, Berlin, Heidelberg, New York, 2nd edition, 1997.

[121] The XFree86 Project, Inc. Internet; http://www.xfree86.org.

[122] M.C. Zanella, M. Robrecht, T. Lehmann, A. de Freitas Francisco, A. Horst, and R. Gielow. RABBIT - A Modular Rapid-Prototyping Platform for Distributed Mechatronic Systems. In *SBCCI*, Pirenópolis, Brazil, September 2001.

[123] Dong Ming Zhang. Kommunikationsmodellierung für HW/SW-Systeme. Master's thesis, Universität Oldenburg, 2001.

[124] Philip G. Zimbardo. *Psychologie*. Springer-Verlag, 5 edition, 1992.

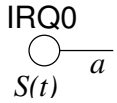
# Appendix A

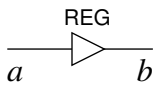
# Graphic Representations of Interrupt Systems

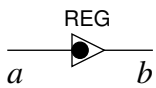
The behaviour of the hardware can be represented by the following graphical elements. All signals have a positive logic independent whether the implementation is low active or not.

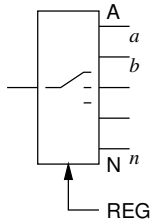
The fields of the tables are a group fo 4 fields with the following content:

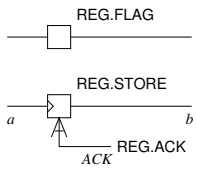
| Graphical Symbol | Description |
|---|---|
| Formal Description | Source Code |

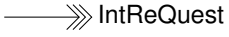
| | |
|---|---|
|  | Source of an interrupt request. This point must be assigned a service routine which handles the signal. |
| $a = S(t)$<br>$S(t) \in \{0, 1\}$ | call_ISR() |

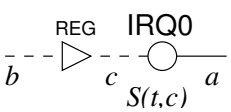
| | |
|---|---|
|  | Switch to turn this signal line on and off. The request can only be passed to the processor if the switch is on. Switch is identified by the label and must provide functions for enable and disable of the line. |
| REG.SUB.state $\in \{on, off\}$<br>$b = f$(REG.SUB.state, $a) \in \{0, 1\}$<br>$f(on, a)$=$a$<br>$f(off, a)$=0 | `/* initialisation code*/`<br>`REG.SUB.on()` |

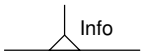
| | |
|---|---|
|  | Same like switch, but it is turned on by default (after a reset). Hence software must turn it off, if there should no interrupt request be generated. |
| | |

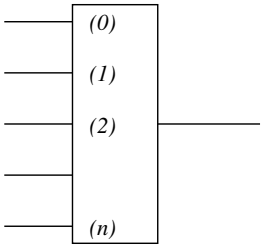
| | |
|---|---|
|  | Multiplexer. It switches the signal flow to one dedicated line. The line is named by the labels at the lines. The switch to a line can be set by the setting the label of *REG*. |
| REG.state $\in \{A, B, \cdots, N\}$<br>$a \equiv f(\text{REG.state})$<br>$f(\text{REG.state}) =$<br>$\begin{cases} a & : \text{REG.state} = A \\ b & : \text{REG.state} = B \\ \cdots & : \\ n & : \text{REG.state} = N \end{cases}$ | |

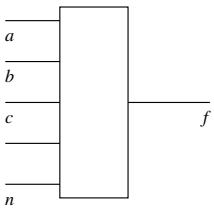
| | |
|---|---|
|  | Registers which signal that a line is active (flag). The upper one *REG.FLAG* does not store the request. The one below *REG.STORE* stores the request (rising edge). It can be cleared by acknowledge the line labeled *REG.ACK*. |
| $a$ \| ACK \| $b$<br>0 \| 0 \| $b_{t-1}$<br>1 \| 0 \| 1<br>0 \| 1 \| 0<br>1 \| 1 \| 0 | `if( REG.FLAG.is_set() ){`<br>`  REG.ACK.acknowledge();`<br>`  ...`<br>`}` |

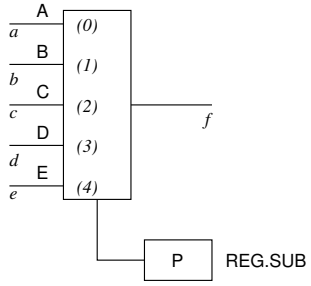
| | |
|---|---|
|  | End of the Interrupt Request System in hardware. From here on a routine in software handles the request (Processor switches to ISR-Mode). |
| | `IRD_Exceptiongate(){`<br>`...`<br>`}` |

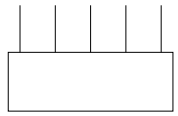
| | |
|---|---|
|  | Switch which has influence on generation of an interrupt, but not in the sense of an enable or disable. |
| $a = S(t,c), S(t) \in \{0,1\}$<br>REG.state $\in \{on, off\}$<br>$c = \begin{cases} b & : \text{REG.state} = on \\ 0 & : \text{REG.state} = off \end{cases}$ | |

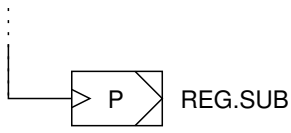
| | |
|---|---|
|  | Line with assigned information *INFO*. This information can be used by the software generator to identify the line or to provide data to the software module. |
| | |

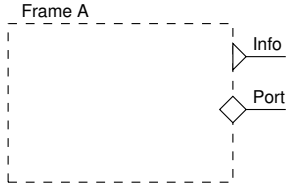
| | |
|---|---|
|  | Inclusive OR-combination of the lines. The depict priority inside the box is only for software generation. Software should check the line with the highest priority $(0)$ first. |
| $PRIOR = 0, 1, 2, \cdots, n$<br>$prior_{Gate}(y) : LINES_G \mapsto PRIOR$<br>$p_{min} = min(prior(y)) \in PRIOR. \forall y \in LINES \wedge state(y) = 1$ | `/* line A */`<br>`...`<br>`/* line B */`<br>`...`<br>`/* line C */`<br>`...` |

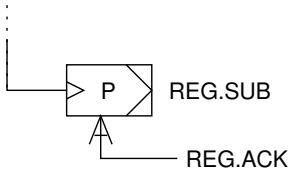
| | |
|---|---|
|  | Inclusive OR-combination of the lines. Software can check the lines in arbitrary order. |
| $f = a \vee b \vee \cdots \vee n$ | `/* line A */`<br>`...`<br>`/* line C */`<br>`...`<br>`/* line B */`<br>`...` |

| | |
|---|---|
|  | OR-combination of the lines. The label of the active line is given to the priority buffer. The priority is labeled inside the box with 0 as highest priority. The priority buffer can be read out and is identified by the label *REG.SUB*. This buffer must provide a value or signal for empty. No destructive read. As long as the line is active the value can be read out. |
| $p_{min} = min(prior(y)) \in PRIOR. \forall y \in LINES_G \wedge state(y) = 1$<br>$P = x.x \in LINES_G \wedge prior_{Gate}(x) = p_{min}$ | `switch(REG.SUB.get_status()){`<br>`case A:`<br>`...`<br>`case B:`<br>`  ...`<br>`}` |

| | |
|---|---|
|  | Wired-OR combination of the input lines. No dedicated output. Information flow depends on the final instantiation of other system components. |
| $\forall x \in LINES_G$<br>$\cdot \begin{cases} 1 & : \exists y \in LINES_G.y \neq x \wedge state(y) = 1 \\ 0 & : \text{otherwise} \end{cases}$ | depends on flow direction |

| | |
|---|---|
|  | Buffer has destructive read. The current pending IRQ is acknowledged at this OR-gate by a (read) access to the buffer. First element in priority sorted buffer $P$ is removed. |
| $p_{min} = min(prior(y)) \in PRIOR. \forall y \in LINES_G \wedge state(y) = 1$<br>$P = \{x.x \in LINES_G \wedge state(x) = 1\}$ | |

| | |
|---|---|
|  | Frame for sub system instances. The port passes the request from the inside to the outside or vice versa. The frame can provide further information *Info* to identify the frame/sub system by the software, eg. the base address. See Entities in VHDL. |
| | Vanishes during code generation |

| | |
|---|---|
|  | Pending IRQ must be acknowledged like a register. |
| $p_{min} = min(prior(y)) \in PRIOR. \forall y \in LINES_G \wedge state(y) = 1$<br>$P = \{x.x \in LINES_G \wedge state(x) = 1\}$ | `switch(REG.SUB.get_status()){`<br>`case A:`<br>`REG.ACK.acknowledge();`<br>`...`<br>`case B:`<br>`REG.ACK.acknowledge();`<br>`...`<br>`}` |

# Appendix B

# RABBIT Interrupt System

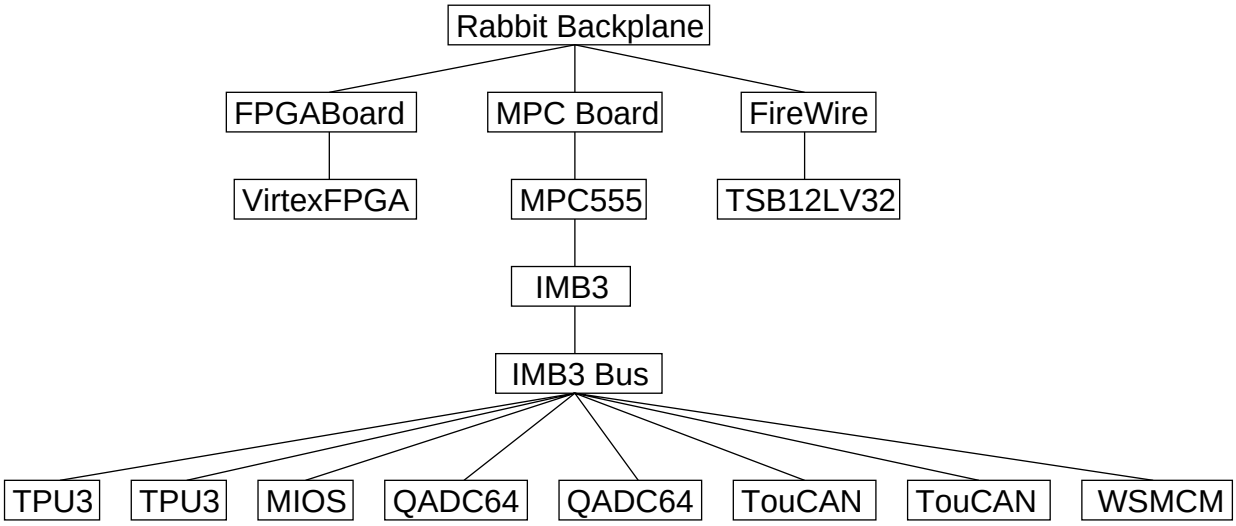



Figure B.1: Specification hierarchy of the templates. TouCAN and QADC64 are instanziated twice.

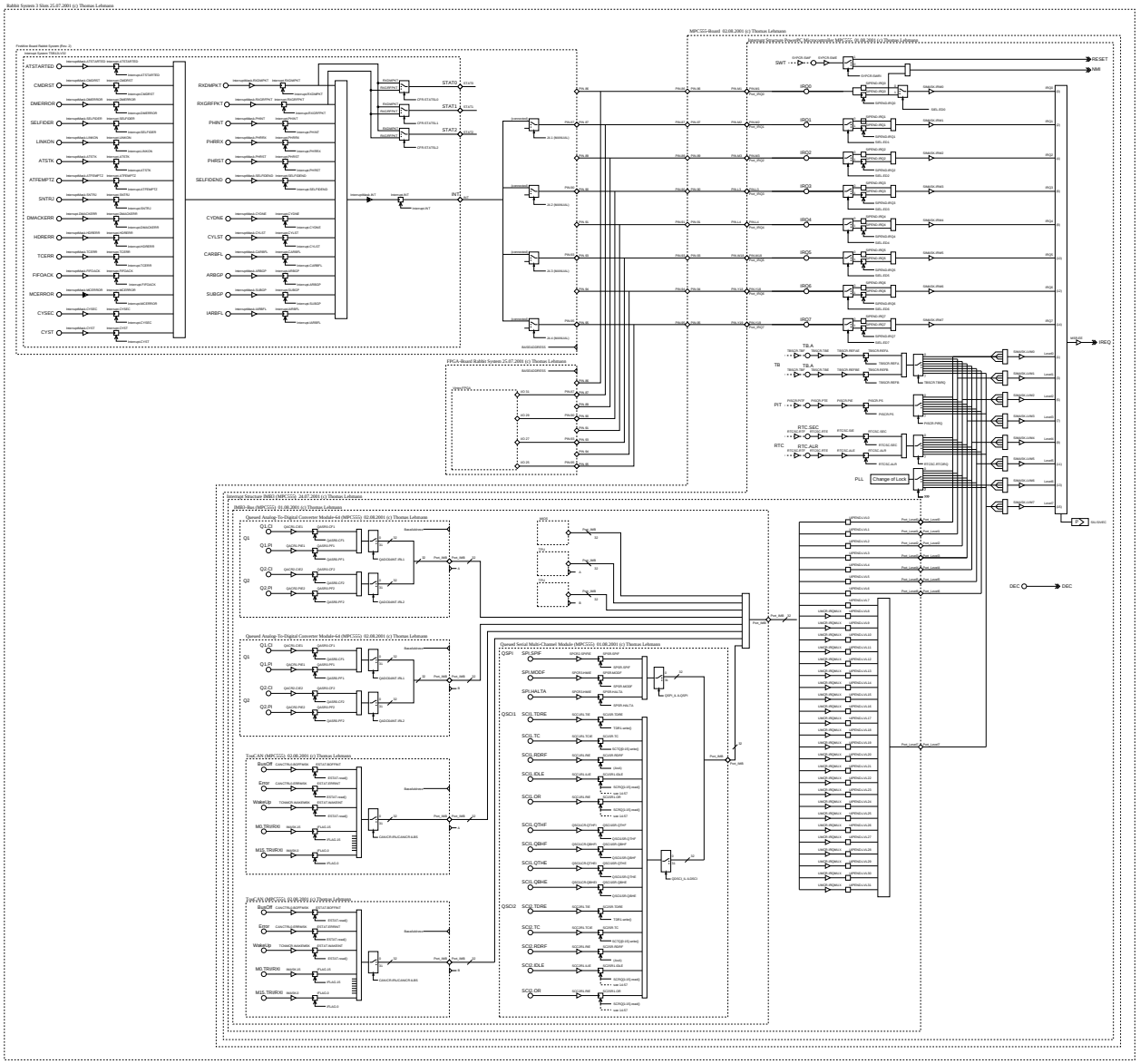


Figure B.2: Interrupt system of the RABBIT plattform.